


CUPRINS

PREFAȚĂ	5
1. INTRODUCERE ÎN PROBLEMATICA MICROARHITECTURILOR PREDICTIVE ȘI SPECULATIVE	11
2. LIMITĂRI FUNDAMENTALE ALE PARADIGMEI ILP. SOLUȚII.20	20
2.1. LIMITAREA „PRODUCĂTORULUI” (FETCH BOTTLENECK). SOLUȚII.	20
2.1.1. PROBLEMA ÎN SINE	20
2.1.2. SOLUȚIE: TRACE CACHE. TRACE – PROCESOARE.	22
2.2. LIMITAREA „CONSUMATORULUI” (ISSUE BOTTLENECK). SOLUȚII.	30
2.2.1. REUTILIZAREA DINAMICĂ A INSTRUCȚIUNILOR	35
2.2.2. PREDICȚIA DINAMICĂ A VALORILOR INSTRUCȚIUNILOR	51
2.2.3. PROBLEME DE IMPLEMENTARE A PREDICȚIEI VALORILOR ÎN ARHITECTURI CU MICROFIRE MULTIPLE DE PROCESARE	89
3. METODOLOGIA DE SIMULARE	94
3.1. BENCHMARK-URI UTILIZATE ÎN SIMULARE. CARACTERISTICI.	94
3.1.1. BENCHMARK-URILE SPEC’95.	94
3.1.2. BENCHMARK-URILE SPEC2000.	95
3.1.3. ALTE BENCHMARK-URI: SPEC JVM98, MEDIABENCH.	101
3.2. DEZVOLTAREA DE SIMULATOARE SUB MEDIUL SIMPLESCALAR UTILIZÂND BENCHMARK-URILE SPEC.	106
3.2.1. CE ESTE "SIMPLESCALAR TOOL SET" ?	107
3.2.2. AVANTAJELE UTILIZĂRII "SIMPLESCALAR TOOL SET".	108
3.2.3. DEZVANTAJE ÎN UTILIZAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR".	110
3.2.4. UTILITARE GNU.	111
3.2.5. INSTALAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR".	112
3.2.6. SIMULATOARELE SETULUI DE INSTRUMENTE "SIMPLESCALAR 3.0".	114
3.2.7. COMPILAREA ȘI EXECUȚIA PROGRAMELOR C ȘI C++ SUB LINUX. INSTALAREA ȘI COMPILAREA BENCHMARK-URILOR SPEC2000 PENTRU ARHITECTURA SIMPLESCALAR – PISA.	124
3.2.8. FOLOSIREA DEBUGGER-ULUI GNU.	132

3.3. EXTINDEREA MEDIULUI SIMPLESCALAR 3.0 CU UN MODUL PROPRIU.	137
3.3.1. DESCRIEREA SUMARĂ A CODULUI SURSĂ AFERENT SIMULATOARELOR SIMPLESCALAR EXISTENTE.	137
3.3.2. IMPLEMENTAREA SIMULATORULUI PROPRIU.	139
3.4. ARHITECTURA SIMPLESCALAR	142
4. INFLUENȚA UNOR CONCEPTE DE PROGRAMARE PROCEDURALĂ / OBIECTUALĂ ASUPRA GENERĂRII SALTURILOR INDIRECTE	145
4.1. ANALIZA LIMBAJELOR C ȘI C++ DIN PUNCT DE VEDERE AL PROCESĂRII LOR PE ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCȚIUNILOR	145
4.2. INVESTIGAȚII PRIVIND GENERAREA SALTURILOR / APELURILOR INDIRECTE PRIN APLICAȚII PROCEDURALE RESPECTIV OBIECTUALE	158
4.3. INSTRUCȚIUNI DE SALT INDIRECT CU CARACTER DINAMIC POLIMORF	180
5. CONTRIBUȚII LA PREDICȚIA SALTURILOR / APELURILOR INDIRECTE	183
5.1. PREDICȚIA DINAMICĂ A RAMIFICAȚIILOR DE PROGRAM. NECESITATE. SOLUȚII	183
5.1.1. STRUCTURI CLASICE DE PREDICȚIE A SALTURILOR CONDIȚIONATE.	187
5.1.2. DEPĂȘIREA PERFORMANȚELOR PREDICTOARELOR CLASICE PRINTR-O ABORDARE INTEGRATOARE <i>HARDWARE-SOFTWARE</i>	193
5.1.2.2. METODE NEURONALE DE PREDICȚIE. PREDICTORUL PERCEPTRON.	195
5.2. NECESITATEA PREDICȚIEI SALTURILOR / APELURILOR INDIRECTE. SOLUȚII.	213
5.2.1. PREDICTOARE ADAPTIVE PE DOUĂ NIVELURI. PREDICTORUL TARGET-CACHE.	214
5.2.2. PREDICTOARE HIBRIDE ȘI METAPREDICTOARE.	218
5.2.2.1. CU SELECȚIE BAZATĂ PE CODUL SURSĂ.	218
5.2.2.2. CU SELECȚIE BAZATĂ PE ARITATE.	221
5.2.2.3. CU SELECȚIE BAZATĂ PE CONFIDENȚĂ.	222
5.2.2.4. STRUCTURĂ DE PREDICȚIE PENTRU EXPLOATAREA CORELAȚIEI VARIABILE.	225
5.2.2.5. SELECȚIA DINAMICĂ A CONTEXTULUI RELEVANT PENTRU PREDICȚIE.	227
5.2.3. PREDICTOARE CASCADATE PE DOUĂ SAU MAI MULTE NIVELURI.	242

5.3. CERCETĂRI PROPRII PRIVITOARE LA PREDICȚIA SALTURILOR / APELURILOR INDIRECTE.	248
5.3.1. PREDICTORUL PPM COMPLET.	248
5.3.2. ÎMBUNĂTĂȚIREA PERFORMANȚEI PREDICTORULUI TARGET CACHE. IMPLEMENTAREA UNUI MECANISM DE CONFIDENȚĂ.	256
5.3.3. PREDICȚIA SALTURILOR INDIRECTE:	268
5.3.3.1. IMPLEMENTAREA PREDICTORULUI HIBRID CU SELECȚIE BAZATĂ PE ARITATE.	268
5.3.3.2. PREDICTOR HIBRID CU SELECȚIE BAZATĂ PE CONFIDENȚĂ.	272
5.3.4. JIndirSim – SIMULATOR FUNCȚIONAL PENTRU PREDICȚIA SALTURILOR INDIRECTE.	274
5.3.4.1. ARHITECTURA APLICAȚIEI.	274
5.3.4.2. JIndirSim. INTERFAȚĂ GRAFICĂ. GHID DE UTILIZARE.	277
6. CERCETĂRI CU PRIVIRE LA PREDICȚIA DINAMICĂ A VALORILOR INSTRUCȚIUNILOR	284
6.1. PREDICȚIA VALORILOR INSTRUCȚIUNILOR (LOAD, ALU).	284
6.1.1. DESCRIEREA SIMULATORULUI “VALUE PREDICTOR”.	284
6.1.2. IMPLEMENTAREA TIMING-ULUI ÎN CADRUL SIMULATORULUI.	290
6.2. VECINĂTATEA ȘI PREDICȚIA VALORILOR CENTRATĂ PE CONTEXTUL CPU.	293
6.2.1. PREDICTOARE CLASICE (LASTVALUE, INCREMENTAL, CONTEXTUAL, HIBRID)	297
6.2.2. PREDICTOR HIBRID CU SELECȚIE BAZATĂ PE METAPREDICTOARE. DESCRIERE SIMULATOR.	302
7. REZULTATE OBȚINUTE PRIN SIMULARE. INTERPRETĂRI.	318
7.1. CERCETĂRI PRIVIND VECINĂTATEA ȘI PREDICȚIA APELURILOR INDIRECTE	319
7.1.1. REZULTATE OBȚINUTE PRIN SIMULAREA BENCHMARK-URILOR SPEC	320
7.1.2. SIMULAREA PROPRIILOR PROGRAME DE TEST. REZULTATE.	338
7.2. CERCETĂRI PRIVIND VECINĂTATEA ȘI PREDICȚIA VALORILOR INSTRUCȚIUNILOR	342
7.3. PREDICȚIA VALORILOR REGIȘTRILOR CPU.	356
7.3.1. REZULTATE BAZATE PE PREDICTOARELE DE VALORI: INCREMENTAL, CONTEXTUAL ȘI HIBRID CU PRIORITIZARE STATICĂ.	356
7.3.2. EVALUAREA PREDICȚIEI NEURONALE APLICATĂ REGIȘTRILOR PROCESORULUI.	363
8. CONCLUZII ȘI CONTRIBUȚII ORIGINALE. DEZVOLTĂRI ULTERIOARE	368
 BIBLIOGRAFIE	375

ANEXA 1	388
EXEMPLE JUSTIFICATIVE PRIVIND IMPACTUL LA NIVEL MICROARHITECTURAL AL UNOR TEHNICI DE ÎMBUNĂTĂȚIRE A PERFORMANȚEI PROCESOARELOR PRIN PREDICȚIA SALTURILOR INDIRECTE	388
ANEXA 2	393
EXEMPLU PRIVIND EFICIENȚA PREDICȚIEI PE REGIȘTRII PROCESORULUI MIPS [Fl003]	393
ANEXA 3	395
LIMITAREA PREDICTIBILITĂȚII SALTURILOR PE ALGORITMI DE SORTARE RAPIDĂ	395
ANEXA 4	398
REDUCEREA COMPLEXITĂȚII UNOR STRUCTURI MICROARHITECTURALE PRIN DISPERSIA ADRESELOR	398
ANEXA 5	407
OPTIMIZAREA COLIZIUNILOR ÎN STRUCTURILE PIPELINE	407

PREFAȚĂ

Importanta rată de creștere, cu cca. 60% pe an, a performanțelor microprocesoarelor ultimilor 10 ani a fost susținută în principal prin două abordări corelate: prima vizează îmbunătățiri tehnologice (creșterea frecvenței tactului, scăderea latenței memoriilor *cache*, creșterea gradului de integrare etc.) și are o pondere de cca. 35%, iar a 2-a, mai semnificativă (65%), vizează organizări arhitecturale tot mai eficiente. Această lucrare elaborată de către dl. dr.ing. Adrian Florea se focalizează în esență pe cea de a 2-a abordare, ținând desigur cont și de caracteristicile, uneori restrictive, ale tehnologiilor moderne din domeniul microarhitecturilor de calcul.

Paradigma actuală de procesare a instrucțiunilor și datelor exploatează paralelismul la nivel de instrucțiuni prin tehnici de tip *pipelining* (paralelism temporal la nivelul fazelor de procesare ale instrucțiunilor mașină) și respectiv prin tehnici de execuții multiple ale instrucțiunilor independente (*Multiple Instruction Issue*). Desigur că aceste paralelisme sunt posibile prin utilizarea unor metode hardware-software complexe, precum cele de planificare și reorganizare a instrucțiunilor (în mod static, dinamic și hibrid) și respectiv de predicție a instrucțiunilor de ramificație urmate de procesări speculative etc. În ciuda performanțelor deosebite ale acestei paradigme arhitecturale, ea implică cel puțin două limitări fundamentale, care nu vor putea fi depășite decât printr-o nouă generație de microprocesoare de uz general, cea de a 4-a, care va îngloba structuri novatoare de prelucrare. Prima limitare, cunoscută sub denumirea de “*fetch bottleneck*” (strangularea mecanismului de aducere a instrucțiunilor), se referă în principiu la imposibilitatea actualelor microprocesoare de a aduce din memorie mai mult de o unitate secvențială de program (*basic-block*) într-un ciclu și este datorată instrucțiunilor de salt. A 2-a limitare, denumită în literatură “*dataflow bottleneck*” sau “*issue bottleneck*” (strangularea mecanismului de execuție a instrucțiunilor, datorată dependențelor reale de date între acestea), se referă la imposibilitatea rulării programului într-un timp mai mic decât latența căii critice a acestuia.

Actualitatea și oportunitatea acestei lucrări de cercetare științifică derivă tocmai din curajul autorului de a încerca să elaboreze câteva soluții originale în vederea depășirii acestor limitări fundamentale. Soluțiile propuse, încadrate în mod pragmatic într-un ansamblu al unor cercetări de prestigiu și de actualitate pe problematica anterior schițată, vor putea constitui contribuții importante, cu un puternic caracter evolutiv, în

procesul, uneori fascinant, de elaborare a viitoarei generații arhitecturale de microprocesoare.

Pe de altă parte, o provocare continuă în știința și ingineria calculatoarelor o reprezintă înțelegerea profundă a complexelor și subtilelor interacțiuni hardware-software. Optimizările interfeței hardware-software nu sunt posibile decât pe baza unui asemenea proces de înțelegere, care necesită o viziune integratoare asupra multiplelor și complicatele procese de prelucrare a informației, codificată la niveluri semantice diferite. În acest sens, consider că cercetările autorului relative la analiza legăturilor adânci dintre caracteristicile unor programe HLL (*High Level Languages*), pe de o parte, și structurile hardware de predicție a salturilor indirecte prin registru, pe de altă parte, constituie un valoros exemplu de aventură științifică formativă, dar și aplicativă.

Iată doar câteva motive care mă determină să constat că această monografie științifică este una deosebit de necesară, actuală și oportună, în mediul academic românesc. Când idealurile profesionale sunt mici, provinciale, realizările nu pot fi decât pe măsură. Nu este deloc cazul acestei cărți, care și-a propus de la început ambiții mari, în deplină sincronizare cu cele mai noi cercetări din domeniul fertil al microarhitecturilor de calcul cu paralelism la nivelul instrucțiunilor și microfiredelor de execuție.

După ce în primul capitol autorul prezintă în mod sintetic o introducere în problematica microarhitecturilor predictive cu execuții speculative ale instrucțiunilor, în capitolul al 2-lea se face o analiză utilă a limitărilor fundamentale aferente actualei generații de microprocesoare. Capitolul este deosebit de bine plasat întrucât, în continuare, pornind de la aceste limitări fundamentale, prin rafinarea soluțiilor propuse în literatura de specialitate, se dezvoltă anumite soluții originale interesante. Autorul insistă în mod justificat pe limitarea numită "*fetch bottleneck*" rezolvată prin arhitecturi de tip *Trace-Cache* (memorii *cache* care rețin pe o intrare, secvențe de *basic-block*-uri anterior procesate și deci, pretabile la re-procesare), care între timp au fost implementate în microprocesoarele comerciale (ex. Pentium IV). Apoi se prezintă limitarea căii critice de program împreună cu cele două soluții deja consacrate: reutilizarea dinamică a instrucțiunilor și respectiv predicția dinamică a valorilor instrucțiunilor. Ambele soluții se bazează pe principiul vecinătății temporale a valorilor instrucțiunilor (*value locality*), formulat prima dată de *Lipasti* și *Shen* într-un articol din 1996. Se insistă asupra celei de a 2-a soluții întrucât aici autorul determină posibile oportunități de cercetare, care vor fi prezentate pe larg în continuarea lucrării. Pe lângă analiza riguroasă a principalelor tipuri de predictoare dezvoltate în literatură, se fac considerații interesante asupra performanței globale de procesare a acestor

microarhitecturi (exprimată în instrucțiuni / ciclu), autorul elaborând și un model teoretic care corectează un model anterior, aparținând unor cunoscuți cercetători israelieni. Foarte interesant mi s-a părut a fi și paragraful 2.2.3. în care se abordează dificila și extrem de importanta problemă a arhitecturilor predictive și cu procesări multiple ale micro-firelor de execuție (*multithreaded processors*). Oarecum similar cu sistemele multiprocesor și aici se pun probleme legate de păstrarea unui model coerent și consistent al sistemului ierarhizat de memorie, din punct de vedere al variabilelor globale. Sunt prezentate în mod succint câteva soluții propuse în literatură actuală de specialitate.

În capitolul 3 autorul prezintă metodologia de simulare utilizată în cercetare. În esență aceasta se bazează pe benchmark-urile SPEC (*Standard Performance Evaluation Corporation*) și respectiv pe mediul de dezvoltare dedicat cercetării procesoarelor superscalare numit *SimpleScalar*, dezvoltat în mediile academice din SUA. Consider că prin utilizarea acestor instrumente software, lucrarea este pe deplin sincronizată la acest nivel cu cele mai noi cercetări academice și industriale din domeniu. Deși capitolul are mai degrabă un caracter tehnic decât unul științific, utilitatea sa în cadrul monografiei este indiscutabilă. Subsemnatul a fost martor la eforturile deosebite ale autorului de a înțelege mediul *SimpleScalar* de simulare în toată complexitatea sa, de a-l extinde cu noi module de program în scopul validării cercetărilor sale originale și respectiv de a compila benchmark-urile SPEC2000 și alte programe de test elaborate de către autor pentru arhitectura țintă. După știința subsemnatului dl. dr.ing. Adrian Florea este primul din România, care, bazat pe aceste instrumente software, a dezvoltat un cadru solid de investigare a microarhitecturilor cu execuții multiple și speculative ale instrucțiunilor. De altfel, această problematică a prezentat-o pe larg într-o consistentă lucrare de peste 400 de pagini, scrisă în colaborare cu subsemnatul, publicată tot în editura Matrix Rom din București.

Capitolul al 4-lea este unul oarecum inedit întrucât autorul abordează o problematică “de graniță”, relativ puțin dezbătută în literatură, și anume cea a influenței unor concepte de programare procedurală respectiv obiectuală asupra generării salturilor și apelurilor indirecte prin registru. După cum se știe, predicția adreselor destinație ale acestor salturi este o problemă extrem de dificilă (întrucât adresele țintă se modifică în mod dinamic) și actuală iar soluțiile propuse nu sunt încă mulțumitoare. Autorul a sesizat în mod corect faptul că, înainte de a aborda efectiv problema predicției ar trebui să înțeleagă în profunzime modurile în care aceste apeluri indirecte sunt generate prin compilare. Pentru aceasta, el dezvoltă programe de test proprii, atât procedurale cât și obiectuale, care conțin corpuri “suspectate” de a genera apeluri și salturi indirecte în urma

compilării. Pe baza acestei cercetări se extrag concluzii valoroase și deosebit de utile cu privire la construcțiile HLL generatoare ale unor astfel de salturi (apeluri indirecte de funcții prin pointeri, construcții *switch/case* în cazuri bine precizate de autor, prezența funcțiilor de bibliotecă, legarea dinamică realizată prin polimorfism etc.). Cred că toate aceste informații de nivel semantic superior pot fi utilizate cu succes în predicția adreselor salturilor / apelurilor indirecte care, actualmente, utilizează în majoritatea lor informații de nivel semantic mai scăzut, precum codul obiect, adrese binare etc. Mai general, pierderea semanticii codului de nivel înalt după compilare, cred că devine inacceptabilă pentru noua generație de microprocesoare. Sper că în cercetările sale viitoare dl. dr.ing. Adrian Florea va exploata mai mult și această nișă fertilă, pe care, de altfel, singur și-a creat-o.

În capitolul 5 se abordează în mod natural interesanta problemă a predicției salturilor indirecte în cadrul procesoarelor superscalare. Important de remarcat faptul că autorul prezintă mai întâi o excelentă sinteză, deosebit de actuală, asupra principalelor contribuții din literatura de specialitate. De aici, rezultă mai apoi și nișele proprii de cercetare, bazat pe îmbunătățirea unora dintre aceste scheme de predicție. O primă contribuție originală o reprezintă utilizarea unei scheme de predicție de tip *Target Cache* (o memorie cache conținând în principal ultimele instanțe ale adreselor țintă aferente unui salt / apel indirect) accesată cu o informație extinsă, mai completă, prin care acuratețea predicției poate crește. Principiul predicției are la bază un model stohastic de tip *Markov* corespunzător secvențelor de valori ale adreselor țintă aferente unui astfel de salt. Pornind de la aceeași schemă *Target Cache*, autorul o îmbunătățește prin adăugarea unei informații de confidență (încredere) a predicției, ceea ce, iarăși, poate crește performanța. În acest ultim sens se propun metrici interesante de evaluare a performanțelor. În fine, se propun apoi alte două noi predictoare de tip hibrid: unul cu selecție bazată pe aritatele adreselor destinație ale salturilor indirecte și un altul cu selecție bazată pe confidențele asociate celor două predictoare componente. Dovezile cantitative referitoare la superioritatea acestor predictoare originale vor fi aduse în capitolul 7 al lucrării, pe baza unor laborioase simulări.

Capitolul al 6-lea abordează o problemă conexasă, prezentând contribuții relative la predicția dinamică a valorilor instrucțiunilor. Pe această bază, se încearcă apoi execuția speculativă a unor secvențe de instrucțiuni aparținând căii critice de program. O contribuție deosebit de valoroasă mi s-a părut aici cea legată de centrarea predicției valorilor pe contextul CPU (registrele generale) și nu pe instrucțiuni, cum se face în toate articolele studiate până la acest moment. Avantajele sunt evidente, în primul rând reducerea drastică a numărului de predictoare implementate.

Ideea este dezvoltată în continuare și autorul propune predictoare hibride de valori (contextuale și incrementale) în care selecția predictorului curent se face prin intermediul unor selectoare dinamice de tip automat finit respectiv de tip neuronal. Desigur că selecția se va face în esență pe baza confidențelor atașate fiecărui predictor component. Cred că această contribuție este una de forță a lucrării, faptul fiind justificat și prin publicarea de către autor (în colaborare) a unui valoros articol pe această temă, în prestigioasa revistă *IEE Proceedings. Computers and Digital Techniques* (2005, acreditată *ISI Thomson*). Rezultatele cantitative vor fi prezentate și în acest caz în capitolul următor al lucrării.

Capitolul 7 prezintă într-o manieră realmente impresionantă din punct de vedere al eforturilor de programare și simulare, cu totul deosebite, modul de optimizare al schemelor propuse precum și utile comparații între acestea și cele deja publicate în literatura de specialitate. Mai întâi se abordează problema predicției salturilor indirecte. Într-un mod sistematic, autorul studiază vecinătatea temporală a valorilor (*value locality*) adreselor destinație care se dovedește a fi una extrem de semnificativă. Apoi se determină optimul *pattern*-ului de căutare pentru predictoarele contextuale (modele stohastice *Markov* simplificate). Foarte interesante mi s-au părut a fi aici: validarea ideii de extindere a informației de corelație pentru structuri de tip *Target Cache* (cu rezultate spectaculoase!), validarea schemei *Target Cache* cu mecanismul de confidență propus de autor, validarea schemelor hibride de predicție propuse, cu selecție bazată pe aritatea salturilor / apelurilor indirecte (acurateți de predicție medii cu cca. 4% mai mari față de schemele *Target Cache* respectiv contextuale!). În continuare, se prezintă rezultate extrem de interesante ale unor simulări focalizate pe predicția generalizată a valorilor instrucțiunilor. Deosebit de utile mi s-au părut a fi: optimizarea capacității tablei de predicție în cazul instrucțiunilor *Load* (512 locații) precum și optimizarea predictorului hibrid de valori. Paragraful 7.3 validează într-un mod remarcabil ideea centrării predicției valorilor pe contextul CPU. Autorul cercetează mai întâi, într-un mod sistematic, vecinătatea valorilor asociate registrelor CPU, pentru a înțelege care dintre aceste registre sunt predictibile din punct de vedere al conținutului. Mai mult, se dau și explicații calitative corecte, ca și justificare. Apoi se dezvoltă, de asemenea într-un mod sistematic, mai multe predictoare destinate registrelor respective, inclusiv predictoare hibride foarte performante. Deosebit de interesantă este cercetarea celor trei tipuri de selectoare dinamice propuse într-un capitol anterior, care cred că ar putea fi continuată și aprofundată cu succes.

În urma studierii și analizei acestei cărți de adâncă specialitate în ingineria calculatoarelor, pot concluziona următoarele aspecte:

- Lucrarea domnului dr.ing. Adrian Florea se referă la probleme actuale, extrem de importante, privind microarhitecturile predictive și speculative.
- Pe baza unei cercetări bibliografice vaste, s-a elaborat o sinteză critică valoroasă a domeniului, cu evidențierea principalelor limitări dar și a oportunităților de cercetare.
- S-au cercetat structurile de programe procedurale și obiectuale care generează salturi / apeluri indirecte prin registru.
- S-au elaborat scheme originale de predicție a salturilor / apelurilor indirecte, superioare majorității schemelor publicate în literatura de specialitate.
- S-au proiectat predictoare centrate pe contextul CPU, o idee complet originală și care oferă avantaje deosebite în raport cu deja clasicele predictoare centrate pe instrucțiune.
- S-au dezvoltat o serie de simulatoare software complexe menite să evalueze și să optimizeze microarhitecturile novatoare propuse de către autor. Astfel, în urma unor laborioase simulări realizate sub mediul *SimpleScalar* cu utilizarea benchmark-urilor SPEC, autorul dovedește în mod convingător superioritatea structurilor propuse de el, din punct de vedere al raportului performanță / cost.

Această carte reprezintă o cercetare bibliografică la zi privind aspectele actuale ale micro-arhitecturilor cu procesări speculative întrețesută cu multe completări și contribuții originale, rodul unei munci a autorului în acest domeniu de circa 8 ani. Dincolo de rezultatele științifice obținute, cred că scopul acestei monografii este unul deosebit de important și generos: anume acela de a arăta într-un mod convingător, că domeniul cercetării microarhitecturilor de mare performanță poate și trebuie să fie abordat în mod creator și în mediile academice românești. În acest sens, autorul pune la dispoziția celor interesați instrumente software utile pentru cercetare. Lucrarea este una formativă, pe baza unor experiențe aplicative autentice ale autorului. Iată de ce, o recomand în mod călduros pentru a fi utilizată efectiv în universități de studenții din anii terminali ai studiilor de licență, de studenții masteranzi și de cei doctoranzi din domeniile științei și ingineriei calculatoarelor, electronicii și domeniilor conexe; este utilă de asemenea inginerilor, cadrelor didactice universitare și cercetătorilor, tuturor profesioniștilor IT și (mai ales !) celor “foarte amatori” de aventuri în ingineria calculatoarelor.

Sibiu, 01 iunie 2005

Prof.univ.dr.ing. Lucian N. VINȚAN
Membru (c.) al Academiei de Științe Tehnice din România

1. INTRODUCERE ÎN PROBLEMATICA MICROARHITECTURILOR PREDICTIVE ȘI SPECULATIVE

Provocările domeniului arhitecturii calculatoarelor cu paralelism la nivelul instrucțiunilor, ca de altfel a multora din știința și ingineria calculatoarelor, sunt în principal conceptuale, arhitecturale, și abia în final tehnologice. În ultimii 10 ani performanța relativă a microprocesoarelor a crescut cu cca. 60% pe an. Cercetătorii susțin că aproximativ 65% din această creștere se datorează îmbunătățirilor arhitecturale și doar 35% celor de natură tehnologică. Tendințele tehnologice se referă la creșterea gradului de integrare al tranzistorilor pe cip, creșterea frecvenței ceasului procesorului, diminuarea timpul de acces la memorie, reducerea costurilor de implementare hardware la aceeași putere de calcul ori capacitate de memorare etc. Tendințele arhitecturale urmăresc exploatarea paralelismului la nivelul instrucțiunilor și microfiredelor de execuție, atât prin tehnici statice (soft), cât și dinamice (hard) sau hibride (cazul arhitecturii IA-64, procesorul *Intel Itanium*), o ierarhizare a sistemului de memorie prin utilizarea unor arhitecturi evoluate de memorii tip cache, reducerea latenței căii critice de program, utilizarea multiprocesoarelor în cadrul arhitecturilor serverelor și stațiilor grafice etc.

Microarhitecturile predictive și speculative se înscriu în domeniul procesoarelor cu paralelism la nivelul instrucțiunilor (*ILPP*). Acestea înglobează caracteristici hardware-software specifice procesoarelor viitorului apropiat fiind considerate de către cercetători ca aparținând celei de a patra generații arhitecturale (după prima generație compusă din procesoarele eminentamente seriale – von Neumann, a doua reprezentată de procesoarele pipeline scalare și a treia reprezentată de mașinile cu execuție multiplă – MEM, categorie din care fac parte procesoarele pipeline superscalare și cele de tip „Very Long Instruction Word”). Arhitecturile MEM din a treia generație arhitecturală, sunt considerate arhitecturi decuplate, în sensul că lucrează după un mecanism de tip "producător - consumator". În fiecare ciclu se desfășoară în mod simultan două procese independente: unul de aducere a instrucțiunilor din memorie în stațiile de rezervare – SR sau într-un buffer de prefetch (producătorul) și un altul de lansare în execuție a acestor instrucțiuni din respectivul buffer sau SR

(consumatorul). Mecanismul de reacție între consumator și producător este realizat prin instrucțiunile de ramificație (branch). Generația arhitecturală următoare de microprocesoare se bazează în principiu și ea pe același model doar că se vor utiliza tehnici de procesare mai agresive precum cele bazate pe reutilizarea dinamică a instrucțiunilor care au mai fost aduse sau / și executate, pe memorii de mare capacitate integrate "on chip" (trace cache), pe exploatarea paralelismului la nivel masiv de "microthread" sau pe predicția dinamică a valorilor resurselor etc. Procesoarele superscalare din generația a patra (trace procesorul, procesoare multithread, arhitectura multiscalară) extrag paralelismul dinamic prin execuție speculativă, multithreading sau trimitere spre execuție "out of order".

Din păcate procesoarele ILP sunt caracterizate de limitări atât din punct de vedere al *producătorului* cât și din punct de vedere al *consumatorului*. Rata de execuție a instrucțiunilor este fundamental limitată de către hazardurile reale de date (RAW – *read after write*) între instrucțiuni (*calea critică de program*). Cauza principală a acestei limitări o constituie natura intrinsec serială a programelor, în care se dictează ordinea secvențială de transmitere a datelor între instrucțiuni. Alte limitări sunt reprezentate de hazardurile structurale, hazardurile de ramificație etc. Aceste ultime limitări, spre deosebire de dependențele RAW pot fi depășite prin diverse tehnici software sau hardware, în cadrul paradigmei actuale.

Un alt neajuns se referă la paradigma actuală de cercetare / exploatare în domeniul arhitecturii calculatoarelor care este prea specializată, prea limitată. Instrumentele de cercetare aferente sunt destul de îmbătrânite. De asemenea, interfața "hardware-software" nu poate fi înțeleasă în profunzime, sau formalizată într-o manieră real calitativă metodele actuale de cercetare sunt bazate în principal pe simulare, benchmarking și prelucrări statistice și mai puțin pe teorie formalizată matură. În consecință, rezultatele cantitative obținute reprezintă doar efecte și nu cauze reale ale fenomenelor procesate. Metodologia actuală de investigare impune necesitatea dezvoltării unor instrumente de cercetare precum: compilatoare, asamblatoare, link-editoare, depanatoare la nivel de cod sursă, benchmark-uri reprezentative a căror execuție să fie simulată. Pe lângă acestea, cel mai important însă, trebuie dovedită abilitatea cercetătorului de a găsi soluții la dificilele provocări ale domeniului sau de a investiga probleme noi, neabordate încă.

Istoria procesoarelor ILP contrapune două paradigme pentru creșterea performanței, bazate pe software și respectiv pe hardware. În ciuda scopului comun de exploatare și creștere a paralelismului la nivelul instrucțiunilor comunitatea cercetătorilor se împarte în două entități aproximativ „disjuncte” în încercările lor de a-l îndeplini. Pe de o parte, arhitecții de calculatoare își canalizează eforturile pentru exploatarea / optimizarea

tehnicilor de procesare existente prin simulări substanțiale pe programe de test reprezentative în format cod obiect, fără a ține cont de semantica codului sursă de nivel înalt, iar pe de altă parte, autorii de compilatoare urmăresc optimizarea codului obiect, reducerea necesarului de memorie etc. Toate aceste eforturi sunt îndreptate de fapt pentru depășirea limitărilor tehnologice, dar mai ales arhitecturale, specifice procesoarelor ILP. Caracteristicile arhitecturale complexe implică tehnologii tot mai sofisticate, parte din ele încă nedisponibile. Totodată însă, îmbunătățirile aferente procesului tehnologic de fabricare, realizate prin creșterea capacității de integrare a tranzistoarelor și reducerea timpilor de comutație, determină creșterea frecvenței procesoarelor actuale și viitoare. Micșorarea continuă a perioadei de tact a acestora conduce la imposibilitatea realizării într-o singură perioadă de tact a proceselor de predicție sau accese la diverse structuri de date, cu efect imediat și asupra vitezei globale de execuție a programelor măsurat în IPC (instrucțiuni per ciclu). De exemplu, există o strânsă legătură între dimensiunea informației de corelație, capacitatea tabelii de predicție, acuratețea predicției (informații legate de arhitectură) și timpul în care se realizează predicția (informație tehnologică), în condițiile impuse de fezabilitate hardware. Pe de altă parte, performanțele arhitecturilor cresc asimptotic pe actualele modele. Totuși, schimbări fundamentale sunt mai greu de acceptat în viitorul apropiat, în primul rând datorită compilatoarelor optimizate, având drept scop exploatarea mai pronunțată a paralelismului la nivel de instrucțiuni, deoarece acestea sunt deosebit de complexe și puternic dependente de caracteristicile hardware. Soluțiile, după cum se arată și în această carte, pot veni mai ales dintr-o îmbinare a ideilor din diverse domenii științifice: arhitectura calculatoarelor și inteligență artificială.

În această lucrare este abordată o tehnică relativ nouă, *predicția valorilor resurselor*, menită să exploateze redundanța instrucțiunilor și datelor, existentă în programe. Tehnica a fost propusă pentru depășirea limitărilor fundamentale ale paradigmei procesoarelor cu paralelism la nivelul instrucțiunilor, scopul fiind de a reduce „*calea critică de program*” – efectul defavorabil provocat de dependențele reale de date asupra performanței globale de procesare. Având în vedere puternica vecinătate a valorilor asiguate unei instrucțiuni sau resurse hardware, care sugerează că valorile posibil a fi asiguate respectivei resurse să nu fie echiprobabile, ci dimpotrivă, localizate pe instrucțiune sau pe resursa hardware, face ca predicția valorilor instrucțiunilor în vederea execuției speculative a acestora să aibă șanse importante de reușită. O altă problemă atacată în lucrarea de față se referă la implementarea unor structuri moderne de predicție a salturilor codificate în moduri de adresare indirecte prin registru, pornind de

la asemănarea existentă între problema predicției valorilor și problema predicției adreselor destinație aferente instrucțiunilor de salt indirect. Structurile de date implementate în hardware pentru ambele procese de predicție au principii identice de funcționare, și anume asocierea cvasi-bijectivă a contextului de apariție al instrucțiunii respective cu data / adresa de predicționat, în mod dinamic, odată cu execuția programului. O soluție proprie, novatoare, a constituit-o extinderea predicției valorilor de la nivelul instrucțiunilor la regiștrii procesorului, cu implicații benefice asupra performanței, reducându-se totodată complexitatea și costul hardware al microarhitecturilor speculative.

Se pare că pentru a continua și în viitor creșterea exponențială a performanței microprocesoarelor, sunt necesare idei noi, revoluționare chiar, pentru depășirea limitărilor paradigmei actuale din punct de vedere conceptual. Există încă o puternică tendință de specializare îngustă care face adesea ca paradigma domeniului să fie una închisă în tipare preconceptuate. Abordări recente, ilustrate și în această lucrare, arată însă că sinergia unor instrumente aparent disjuncte ale științei calculatoarelor converge spre realizări novatoare ale unui anumit domeniu de cercetare (vezi conceptul de predictor neural spre exemplu). Alte posibile soluții constau în abordări integratoare de gen *hardware-software*, *tehnologie-arhitectură*, *algoritmi*, *concepte*, *metode*.

O primă astfel de soluție se referă la necesitatea îmbinării eficiente a tehnicilor de scheduling software cu cele dinamice, de procesare hardware. În prezent, separarea între cele 2 abordări este artificială și poate prea accentuată. În acest sens, programele ar trebui să explicitizeze paralelismul intrinsec într-un mod mai clar. Cercetări actuale arată că un program optimizat static „rulează mai prost” pe un procesor Out of Order decât pe unul In Order [Ste99]. Printre cauze se amintesc expansiunea codului după reorganizare, noile dependențe de date introduse prin execuția condiționată a instrucțiunilor, faptul că instrucțiunile gardate nu permit execuția Out of Order etc. Separarea schedulingului dinamic de cel static este o prejudecată nocivă dar care este din păcate deja consacrată în ingineria calculatoarelor unde practic nimeni nu și-a pus problema dezvoltării unui optimizator de cod dedicat unei mașini cu procesare Out of Order.

Cercetarea algoritmilor ar trebui să țină seama și de concepte precum, de exemplu, cel de cache, în vederea exploatării localităților spațiale ale datelor prin chiar algoritmul respectiv. Se cunosc, la ora actuală, relativ puține lucruri despre ce se întâmplă cu un algoritm când sunt implementate ierarhii de memorii pe mașina fizică. În general, algoritmi nu țin cont la ora actuală de caracteristicile mașinii și acest lucru nu este pozitiv pentru că algoritmul rulează întotdeauna pe o mașină fizică având limitări importante

(de ex. elementele unui tablou se pot afla parțial în cache și parțial pe disc!). Chestiuni similare sunt ilustrate în capitolul 4 al prezentei lucrări unde se arată prin exemple concrete influența polimorfismului din programele de nivel înalt asupra salturilor / apelurilor indirecte, generate la nivelul codului obiect, și greu predictibile. Cu toate acestea, nu înseamnă că programatorul va trebui să devină expert în arhitectura calculatoarelor, dar nu o va mai putea neglija total dacă va dori performanță. La momentul actual sunt realizate cercetări serioase asupra algoritmilor care vizează nu numai o îmbunătățire a performanței ci și o reducere a puterii consumate, vitală mai ales la nivelul dispozitivelor de calcul de tip “*handheld*” și al sistemelor dedicate. De asemenea, prezentul tehnologiei informației, ca să nu mai spunem de viitor, centrat pe Internet și tehnologia WWW, impun ca alături de performanța în sine, fiabilitatea, disponibilitatea și scalabilitatea să devină criterii esențiale, ceea ce implică iarăși necesitatea unei noi viziuni pentru arhitectul de calculatoare.

Cercetătorii predicționează o dezvoltare puternică în continuare a procesoarelor multimedia. Diferite de aplicațiile de uz general, cele multimedia sunt caracterizate de structuri de date regulate, de tip vectorial, cu tendințe de procesare identică a scalarilor componenți. În acest caz devine necesară procesarea și generarea răspunsurilor în timp real. Exploatarea paralelismului la nivelul microthread-urilor independente ale aplicației (codări / decodări audio, video, etc) și localizarea pronunțată a instrucțiunilor prin existența unor mici bucle de program și nuclee de execuție care domină timpul global de procesare sunt aspecte care influențează în mod direct arhitectura procesoarelor multimedia.

Pe scurt, lucrarea este structurată astfel:

În capitolul 2 sunt prezentate pe larg limitările fundamentale ale paradigmei ILP: „*fetch bottleneck*” (limitarea producătorului) și „*issue bottleneck*” (limitarea consumatorului), cauze și soluții în depășirea lor. În ceea ce privește soluțiile la limitarea producătorului se insistă asupra conceptului de Trace Cache cu toate mecanismele pe care le implică (predictor multiplu de salturi, unitate de umplere, logică de selecție), inclusiv pe implementarea comercială existentă la procesorul Intel Pentium4. O mare parte a acestui capitol este concentrată în jurul celor două tehnici novatoare (una speculativă - *predicția valorilor* și una non-speculativă - *reutilizarea dinamică a instrucțiunilor*) menite să reducă efectele defavorabile provocate de dependențele reale de date dintre instrucțiuni (limitare fundamentală a consumatorului). Sunt trecute în revistă o serie de predictoare de valori, de la cele mai simple până la cele mai complexe și actuale, analizate din punct de vedere calitativ și cantitativ performanțele și limitările acestora.

Capitolul 3 ilustrează programele de test standardizate (suita SPEC), metodologia de simulare, instrumentele software utilizate – baza de cercetare de la care s-a pornit în exploatarea schemelor de predicție propuse. Pentru compararea rezultatelor simulărilor cu cele obținute de ceilalți cercetători din domeniu pe plan internațional se impune standardizarea procesului de simulare. În acest sens a fost utilizat și descris setul SimpleScalar 3.0 – o colecție de instrumente software, pusă la dispoziția cercetătorilor în arhitecturi moderne de calcul, care cuprinde: compilatoare, asamblatoare, link-editoare, simulatoare și instrumente de vizualizare a unei arhitecturi (super)scalare generice. Pentru generarea codului la nivel limbaj de asamblare MIPS și a codului obiect specific arhitecturii virtuale SimpleScalar 3.0. a fost nevoie de recompilarea instrumentelor setului (utilitarele GNU, compilatorul Gcc). Cu ajutorul acestora au fost recompilate benchmark-urile SPEC2000 și propriile programe de test folosite în vederea studierii legăturii calitative și cantitative existente între paradigmele actuale de programare (programe procedurale vs. obiectuale) și respectiv generarea valorilor de anumite tipuri de instrucțiuni (salturi indirecte, instrucțiuni Load, ALU etc).

În capitolul 4 s-a încercat investigarea legăturii existente între moștenire, polimorfism și alocarea dinamică a memoriei pe de o parte, și comportamentul salturilor indirecte (JR *reg*) de cealaltă parte, fiind cunoscută ca o problemă dificilă predicția acestora. Se urmărește practic transmiterea de informații de la nivel software către proiectanții de arhitecturi (hardware). S-a realizat o analiză comparativă a limbajelor C și C++ din punct de vedere al procesării lor pe arhitecturi cu paralelism la nivelul instrucțiunilor și s-au evidențiat diferențele dintre acestea. Comportamentul diferit al celor două tipuri de aplicații și penuria de programe obiectuale de test standardizate au impus generarea unor programe proprii de test relativ simple, prin intermediul cărora s-a arătat că cele două "emisfere" software și hardware sunt doar în aparență „disjuncte”. Cele 2 programe obiectuale C++ și 3 procedurale C propuse evidențiază corpuri și construcții de program prezente în sursele de nivel înalt procedurale, respectiv concepte ale programării obiectuale care generează la nivel *low* salturi / apeluri indirecte. Finalul capitolului ilustrează câteva exemple de instrucțiuni de salt indirect cu caracter dinamic polimorf (care generează trei sau mai multe target-uri distincte) din benchmark-urile SPEC simulate, pentru a se evidenția dificultatea predicției acestora.

În capitolul 5 sunt prezentate cercetări ale autorului cu privire la dificila problemă a predicției branch-urilor în cadrul arhitecturilor superscalare de procesare, condiționate, dar mai ales cele codificate în moduri de adresare indirecte. Sunt aduse argumente privind necesitatea

predicției salturilor condiționate și indirecte, utilitatea păstrării unei informații de corelație cât mai bogate, eventual variabile ca lungime în funcție de fiecare salt. Capitolul se constituie într-un adevărat “*state of the art*” a structurilor de predicție dedicate atât salturilor condiționate (de la predictoare simple de tip BTB, corelate pe două niveluri până la cele mai complexe markoviene, neurale, bazate pe arbori de decizie) cât și celor indirecte (Target Cache, structuri hibride, cascade pe mai multe niveluri, structuri preluate din predicția valorilor). În ce privește cercetările proprii privind predicția salturilor / apelurilor indirecte, s-a început cu arhitectura cea mai simplă, predictorul de tip “*last value*” și s-a continuat cu predictoare contextuale – PPM complet, de tip Target Cache, hibride etc. Cu ajutorul predictorului contextual de tip PPM complet s-a încercat determinarea pattern-ului optim de căutare, stabilirea corelației existente între salturi în funcție de context. În vederea îmbunătățirii acurateții predicției aferente instrucțiunilor de salt indirect au fost aduse câteva modificări structurii de predicție Target Cache originare. Mai întâi a fost studiată influența istoriei globale a salturilor condiționate asupra predicției, urmată de extinderea informației de corelație. Un ultim experiment privitor la această structură l-a constituit încercarea de îmbunătățire a acurateții de predicție printr-o ignorare selectivă a efectuării unor predicții. A fost de asemenea exploatată și o schemă de predicție hibridă cu selecție bazată pe aritate.

În capitolul 6 se prezintă contribuțiile originale ale autorului în analiza de performanță și respectiv determinarea unor parametri optimali de proiectare, pentru diferite structuri de predicție a valorilor instrucțiunilor. Au fost implementate cele mai multe din schemele prezentate în capitolul 2 (predictorul LastValue, Incremental, Contextual, Hibrid) și s-au studiat probleme legate de vecinătatea valorilor și predicția valorilor instrucțiunilor cu consecința execuției speculative a instrucțiunilor având influențe benefice asupra timpului de procesare. De asemenea, în acest capitol s-a pus accentul pe o nouă contribuție originală care pune în evidență conceptul de vecinătate a valorilor asociate regiștrilor generali aferenți CPU. Ideea asocierii câte unui predictor de valori pentru anumiți regiștri – predictoare centrate pe regiștri și nu pe instrucțiuni, ar putea implica tehnici arhitecturale novatoare – structuri de predicție mult mai simple, și, în consecință, performanțe îmbunătățite, complexitate și costuri mai reduse ale microarhitecturilor speculative. În continuare au fost propuse soluții de înlăturare a neajunsului provocat de predictorul hibrid (de departe cel mai bun) cu prioritizare statică, fixă, în alegerea tipului de predictor component ce urmează a fi folosit în procesul de predicție și care conduce la o soluție neoptimală. Astfel, au fost descrise structurile de metapredicție implementate, care selectează dinamic, bazat pe diverse grade de încredere, structura care să fie

utilizată la un moment dat pentru predicție. Au fost propuse două tipuri de metapredictoare ne-adaptive, prin atașarea unui automat de confidență sau registru binar de deplasare fiecărui predictor component, și respectiv unul adaptiv, care utilizează o rețea neurală de tip feedforward MultiLayerPerceptron cu algoritm de învățare *backpropagation*.

Capitolul 7 prezintă cele mai elocvente rezultate cantitative și interpretarea acestora din punct de vedere calitativ. S-a realizat o structurare a rezultatelor pe trei categorii: prima dintre acestea evidențiază acuratețea predicției salturilor indirecte obținută pe diverse scheme (Target Cache, predictoare PPM, hibride). A doua categorie analizează probleme legate de vecinătatea și predicția valorilor instrucțiunilor de tip Load și aritmetico-logice. De asemenea, au fost repetate experimentele pentru locațiile memoriei de date. Ultima categorie de rezultate încearcă să dovedească fezabilitatea conceptului novator de predictor de valori centrat pe regiștrii procesorului. Simulările au fost efectuate pe procesoare Pentium III la 500 MHz parțial efectuate sub sistem de operare Microsoft Windows98, 2000, sau NT având la dispoziție emulatorul Cygwin și parțial sub sistemul Linux RedHat 7.3. Evaluările arhitecturilor propuse au fost făcute folosind o colecție de simulatoare *execution-driven* specifice arhitecturilor ILP, originale și puternic parametrizabile (dezvoltate din setul de instrumente SimpleScalar 3.0). Simularea a fost realizată pe cele două versiuni ale benchmark-urilor SPEC ('95 și 2000).

În capitolul 8 sunt trecute succint în revistă contribuțiile științifice ale acestei lucrări, este evidențiat câștigul cantitativ al fiecărei tehnici introduse, sunt realizate comparații între acestea și arătate câteva dintre direcțiile viitoare de cercetare. Concluziile acestei lucrări sugerează necesitatea unor analize teoretice mai profunde, mai generale și mai sistematizate în acest domeniu, dublate și verificate prin simulări laborioase. Cercetările abordate în această lucrare trebuie continuate în scopul rezolvării altor probleme ale domeniului ILP rămase deschise, interesante și conexe cu cele prezentate aici.

Lucrarea se încheie cu o lista bibliografică a peste 120 de lucrări utilizate pe parcursul cercetării și conține peste 130 de figuri și 20 de tabele, dintre care peste 60% conținând rezultatele cercetărilor efectuate de către autor și respectiv 30 de relații analitice. Anexa 1 prezintă două exemple justificative privind impactul la nivel microarhitectural al unor tehnici de îmbunătățire a performanței procesoarelor prin predicția cu acuratețe a salturilor indirecte. Prima aplicație, simplă la nivel high-level, dar mai complexă și dificil de urmărit la nivel asamblare urmărește să evidențieze situații în care extinderea informației de corelație pentru instrucțiunile de salt indirect are sens, contribuind la creșterea acurateții predicției acestora.

Cea de-a doua aplicație evidențiază limitarea avantajului introdus de tehnica de extindere a informației de corelație pentru pattern-uri de salturi condiționate de istorie redusă. Exemplul simplu prezentat în Anexa 2 ilustrează la un nivel redus necesitatea predicției valorilor centrată pe regiștrii procesorului MIPS. Anexele 3, 4 și 5 reprezintă exemple concrete ce demonstrează necesitatea unor abordări integratoare de gen *hardware-software*, *tehnologie-arhitectură*, *algoritmi*, *concepte*, *metode*, cunoscut fiind faptul că, în proiectarea procesoarelor noilor generații, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsă legătură cu aplicațiile potențiale. În Anexa 3 se demonstrează că predictibilitatea salturilor în unele programe poate fi analizată exact, furnizându-se o limită superioară a predictibilității pe algoritmi de sortare rapidă. În Anexa 4 se evidențiază prin intermediul a două aplicații practice cum dispersia adreselor ajută la reducerea complexității unor structuri microarhitecturale. Anexa 5 descrie posibilitatea utilizării algoritmilor *greedy* în optimizarea coliziunilor din structurile pipeline. CD-ul care însoțește această lucrare cuprinde simulatoarele (sursele, executabilele, bibliotecile necesare) dezvoltate pentru exploatarea ideilor de cercetare expuse anterior și prezentate mai amplu pe parcursul fiecărui capitol (structuri și mecanisme de predicție).

În finalul acestei introduceri doresc să mulțumesc atât conducătorului meu de doctorat, prof.dr.ing. Mircea Petrescu cât și domnului prof.dr.ing. Lucian Vințan pentru sprijinul lor profesional continuu și de o înaltă ținută științifică precum și pentru încrederea pe care mi-au acordat-o pe toată perioada pregătirii prin doctorat. De asemenea, țin să mulțumesc domnilor prof.dr.ing. Adrian Petrescu și prof.dr.ing. Vladimir Crețu pentru analiza competentă și amabilitatea de a recenza această lucrare într-o versiune anterioară. Cuvinte de recunoștință vreau să transmit și colegilor mei sibieni din catedra de Calculatoare, în special d-lui conf.dr.ing. Macarie Breazu pentru discuțiile profesionale extrem de fecunde cu privire la paradigmele actuale de programare și posibilele lor implicații în hardware. Din același colectiv doresc să mulțumesc unui tânăr cercetător de perspectivă, prietenul meu Arpad Gellert cu care de multe ori am depanat, compilat și simulat „cot la cot” o parte din surse. Un gând bun se îndreaptă și spre personalul Editurii Matrix Rom care s-a implicat cu generozitate și profesionalism în editarea acestei lucrări. În final, și întotdeauna pe nedrept la final, deși cuvintele nu pot să exprime cu adevărat atât cât ar trebui, vreau să mulțumesc familiei – soției Delilah și copilașilor mei Adrian, Albert și Deborah, pentru înțelegerea, răbdarea și dragostea cu care m-au înconjurat în toți acești ani.

2. LIMITĂRI FUNDAMENTALE ALE PARADIGMEI ILP. SOLUȚII.

2.1. LIMITAREA „PRODUCĂTORULUI” (FETCH BOTTLENECK). SOLUȚII.

2.1.1. PROBLEMA ÎN SINE

Din punct de vedere funcțional procesoarele superscalare de înaltă performanță sunt compuse din 2 mecanisme decuplate: mecanismul de aducere (**fetch**) a instrucțiunilor pe post de *producător* și respectiv mecanismul de execuție a instrucțiunilor pe post de *consumator*. Separarea între cele 2 mecanisme (arhitectură decuplată) se face prin bufferele de prefetch și stațiile de rezervare, ca în figura 2.1. Instrucțiunile de ramificație și predictoarele hardware aferente acționează printr-un mecanism de reacție între consumator și producător. Astfel, în cazul unei predicții eronate, bufferul de prefetch trebuie să fie golit măcar parțial iar adresa de acces la cache-ul de instrucțiuni trebuie să fie modificată în concordanță cu adresa la care se face saltul.

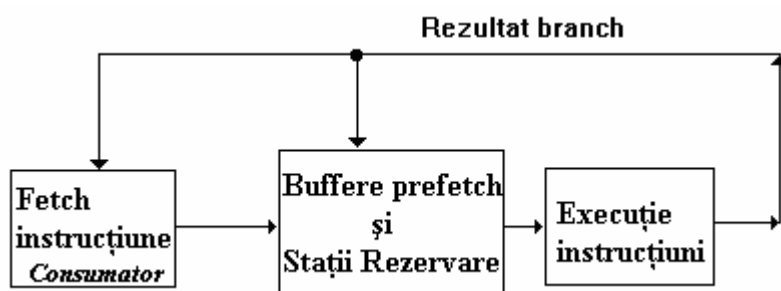


Figura 2.1. Arhitectură superscalară decuplată

Rezultate statistice bazate pe simulări laborioase pe benchmark-uri reprezentative (SPEC'95, 2000) arată că o instrucțiune de salt apare la fiecare 5÷8 instrucțiuni dinamice executate, ceea ce înseamnă că *producătorul*, prin rata de aducere a instrucțiunilor (*fetch rate* – FR) este limitat la cel mult 8, aducerea simultană a mai multor instrucțiuni fiind inutilă (*fetch bottleneck*). Această limitare fundamentală ar avea consecințe defavorabile și asupra *consumatorului* cuantificat prin rata medie de execuție a instrucțiunilor (*issue rate* – IR) întrucât $IR \leq FR$. În subcapitolul 2.2.2.6 sunt prezentate informații cantitative și calitative care evidențiază influența defavorabilă a lărgimii reduse de bandă a mecanismului de aducere a instrucțiunilor asupra eficacității unor tehnici de comprimare a căii critice de program, cum ar fi predicția valorilor. Pentru creșterea gradului de paralelism la nivelul instrucțiunilor este necesară dezvoltarea și implementarea de noi tehnici care să reducă întârzierile în procesare (*hazarduri*) pe oricare din cele două fluxuri: de instrucțiuni (*control-flow*) și respectiv de date (*data-flow*).

Progresele semnificative în algoritmi de lansare în execuție impun însă depășirea acestei bariere. În acest sens, cercetările actuale insistă pe îmbunătățirea mecanismelor de aducere a instrucțiunilor (*fetch*) prin următoarele tehnici:

- o predicția simultană a mai multor ramificații / tact rezultând rate de procesare (IR) sporite.
- o posibilitatea accesării și aducerii simultane a mai multor basic-block-uri din cache, chiar dacă acestea sunt nealiniat, prin utilizarea unor cache-uri multiport (*trace-cache*).
- o păstrarea unei latențe reduse a procesului de aducere a instrucțiunilor, în contradicție cu cele 2 cerințe anterioare.

Latența unității de aducere a instrucțiunilor, pipeline-izată, are un impact profund asupra performanței procesorului, în primul rând datorită costului de reumplere a structurii pipeline, în cazul unui salt greșit predicționat, cu instrucțiuni de la adresa corectă. Evident că, necesitatea predicției mai multor salturi simultan, precum și aliniamentul necontiguu în memorie al instrucțiunilor contigue din punct de vedere al execuției, determină o creștere a latenței unității de aducere a instrucțiunilor (*fetch*). Alți factori care determină limitarea ratei de *fetch* a instrucțiunilor (FR - *Fetch Rate*) sunt: lărgimea de bandă limitată a interfeței procesor - cache, capacitatea redusă a buffer-ului de *prefetch*, accesele cu *miss* în cache, acuratețea de predicție nesatisfăcătoare (procentajul cel mai ridicat obținut de cercetători este de 98.29%, realizabil printr-un predictor neural de tip *Perceptron* [Jim02]) și respectiv latența ridicată de refacere a contextului în

cazul unei predicții eronate. De asemenea, un alt factor care poate limita rata de fetch a instrucțiunilor îl constituie instrucțiunile de salt indirect, revenirile din proceduri și întreruperile software. Păstrând contextul, o altă limitare poate fi reprezentată de nealinierea așa numitelor „*blocuri atomice*” [Pat98]. Unitatea de umplere – *fill unit* (vezi subcapitolul 2.1.2) este forțată să creeze un segment de instrucțiuni („*bloc atomic*”) mai mic decât dimensiunea maximă a liniei din trace cache deoarece basic-block-ul următor din șirul dinamic de instrucțiuni care se execută este mai mare decât spațiul rămas disponibil în linia din trace cache.

Întrucât asupra conceptului Trace Cache se va insista în subcapitolul următor (2.1.2), în continuare sunt descrise foarte pe scurt alte câteva soluții alternative de extindere a lărgimii de bandă aferente mecanismului de fetch.

Mecanismul de predicție *Branch Address Cache*, propus în [Yeh93] ca o extensie a structurii Branch Target Buffer [Smi84] este capabil să predicționeze mai multe salturi simultan, determinând adresele de început a basic-blocurilor care se vor executa. Toate aceste target-uri multiple vor fi trimise către un cache de instrucțiuni cu grad ridicat de întrețesere pentru a fi efectuat procesul de fetch simultan într-un singur ciclu de tact.

O altă schemă, propusă în [Con95] permite extragerea a două linii necontigue din cache-ul de instrucțiuni. Suplimentar este utilizat un *buffer de colapsare* pentru detecția branch-urilor scurte din interiorul unei linii de cache și evacuarea instrucțiunilor dintre branch și target-ul său.

2.1.2. SOLUȚIE: TRACE CACHE. TRACE – PROCESOARE.

O paradigmă menită să extindă conceptul de superscalaritate și care poate constitui o soluție interesantă față de limitările mai sus menționate, o constituie trace - procesorul, adică un procesor superscalar având o **memorie trace - cache** (TC) – vezi figura 2.6. Ca și cache-urile de instrucțiuni (IC), TC este accesată cu adresa de început a noului bloc de instrucțiuni ce trebuie executat, în paralel cu IC. În caz de miss în TC, instrucțiunea va fi adusă din IC sau - în caz de miss și aici - din memoria principală. Spre deosebire însă de IC, TC memorează instrucțiuni contigue din punct de vedere al secvenței lor de execuție, în locații contigue de memorie. O linie din TC memorează un segment de instrucțiuni executate dinamic și secvențial în program (trace - segment). Un trace poate conține mai multe basic-block-uri (unități secvențiale de program). Așadar, o linie TC poate conține N instrucțiuni sau M basic - block-uri, $N > M$, înscrise pe parcursul execuției lor.

Memoria TC este accesată cu adresa de început a basic - block-ului A, în paralel cu predictorul multiplu de salturi (vezi figura 2.2). Acesta, spre deosebire de un predictor simplu, predicționează nu doar adresa de început a următorului basic - block ce trebuie executat ci toate cele $(M - 1)$ adrese de început aferente următoarelor $(M - 1)$ basic - block-uri care urmează după A. Cei $(M - 1)$ biți generați de către predictorul multiplu (taken/not taken) selectează spre logica de execuție doar acele blocuri din linia TC care sunt predicționate că se vor executa (în cazul acesta doar blocurile A și B întrucât predictorul a selectat blocurile ABD că se vor executa, în timp ce în linia TC erau memorate blocurile ABC).

O linie din TC conține:

- N instrucțiuni în formă decodificată, fiecare având specificat blocul căreia îi aparține.
- cele 2^{M-1} posibile adrese destinație aferente celor M blocuri stocate în linia TC.
- un câmp care codifică numărul și "direcțiile" salturilor memorate în linia TC.

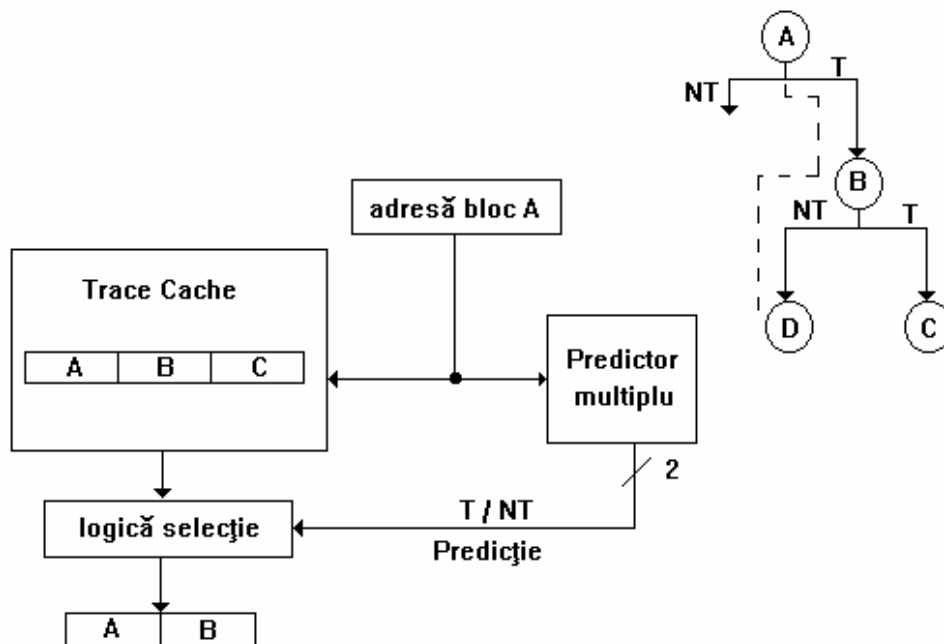


Figura 2.2. Ansamblul trace-cache respectiv predictor multiplu

Înainte de a fi memorate în TC, instrucțiunile pot fi predecodificate în scopul înscrierii în TC a unor informații legate de dependențele de date ce

caracterizează instrucțiunile din linia TC curentă. Aceste informații vor facilita procese precum bypassing-ul datelor între unitățile de execuție, redenumirea dinamică a regiștrilor cauzatori de dependențe WAR (Write After Read) sau WAW (Write After Write) între instrucțiuni etc., utile în vederea procesării Out of Order a instrucțiunilor. În [Lee02] este propusă o structură care extinde Trace Cache-ul cu un predictor hibrid de valori pentru instrucțiunile cauzatoare de dependențe. Prin predicția selectivă a valorilor instrucțiunilor este redus numărul de accese la tabela de predicție, rezultând o utilizare mult mai eficientă resurselor, permițându-se astfel „reducerea căii critice de program” și implicit creșterea ratei de execuție.

O linie din TC poate avea diferite grade de asociativitate în sensul în care ea poate conține mai multe pattern-uri de blocuri, toate având desigur aceeași adresă de început (A), ca în figura 2.3.

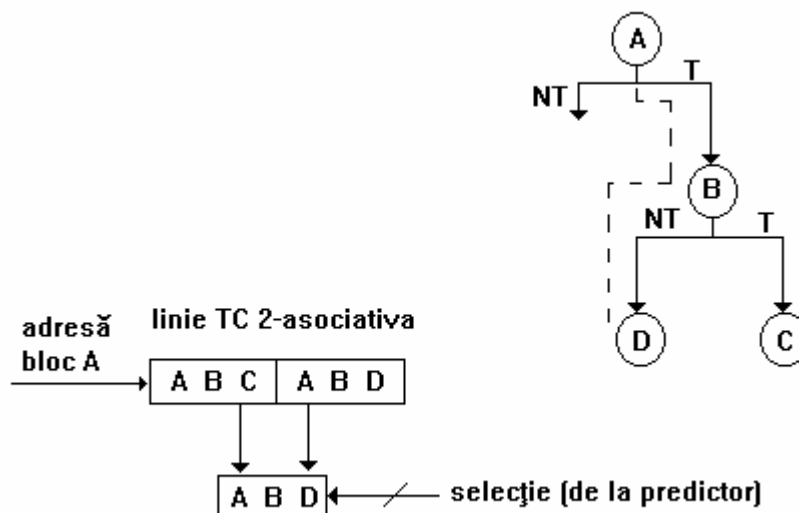


Figura 2.3. Selecția dintr-o linie trace-cache asociativă

Așadar, segmentele începând de la aceeași adresă (A), sunt memorate în aceeași linie asociativă din TC. Ca și în structurile TC neasociative, verificarea validității liniei selectate se face prin compararea (căutarea) după tag. Deosebirea de esență constă în faptul că aici este necesară selectarea - în conformitate cu pattern-ul generat de către predictorul multiplu - trace-ul cel mai lung dintre cele conținute în linia respectivă. Este posibil ca această selecție complexă să dureze mai mult decât în cazul neasociativ și prin urmare să se repercuteze negativ asupra duratei procesului de aducere a instrucțiunilor (fetch). Avantajul principal însă, după cum se observă și în

figură, constă în faptul că este probabil să se furnizeze procesorului un număr de blocuri "mai lung" decât un TC simplu. Astfel de exemplu, dacă pattern-ul real de blocuri executate este ABD, structura TC îl va furniza fără probleme, în schimb o structură TC neasociativă ce conține doar pattern-ul ABC, evident va furniza în această situație doar blocurile AB.

Pe măsură ce un grup de instrucțiuni este procesat, el este încărcat într-o așa-numită "fill unit" (FU - unitate de pregătire), după cum poate fi văzut în figura 2.6. Rolul FU este de a asambla instrucțiunile dinamice, pe măsură ce acestea sunt executate, într-un trace - segment. Segmentele astfel obținute sunt memorate în TC. Este posibil ca înainte de scrierea segmentului în TC, FU să analizeze instrucțiunile din cadrul unui segment spre a marca explicit dependențele dintre ele. Acest lucru va ușura mai apoi lansarea în execuție a acestor instrucțiuni întrucât ele vor fi aduse din TC și introduse direct în stațiile de rezervare aferente unităților funcționale. Unitatea FU se ocupă deci de colectarea instrucțiunilor lansate în execuție, asamblarea lor într-un grup de N instrucțiuni (sau M blocuri) și înscrierea unui asemenea grup într-o anumită linie din TC. Există desigur cazuri când FU poate crea copii multiple ale unor blocuri în TC (vezi figura 2.4). Această redundanță informațională poate implica degradări ale performanței (înlocuirea unor linii din trace cache care conțin informație utilă cu blocurile redundante determină creșterea ratei de miss și implicit diminuarea ratei globale de procesare), dar pe de altă parte, lipsa redundanței ar degrada valoarea ratei de fetch a instrucțiunilor deci și performanța globală.

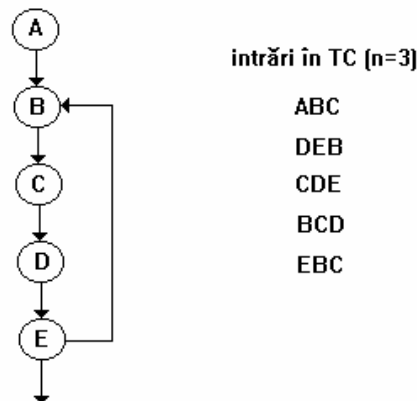


Figura 2.4. Segmente asamblate pe timpul execuției unei bucle de program

Se poate deci afirma că un TC exploatează reutilizarea eficientă a secvențelor dinamice de instrucțiuni, reprocesate frecvent în baza a 2 motive de principiu: localizarea temporală a trace-ului și respectiv comportarea predictibilă a salturilor în virtutea comportării lor anterioare. Așadar, TC

memorează trace-uri în scopul eficientizării execuției programului și nu doar în scopul eficientizării procesului de aducere al instrucțiunilor. Aceasta, pe motiv că un segment din trace conține numai instrucțiuni care se vor executa. În cazul IC, dacă într-un bloc există o ramificație efectivă, instrucțiunile următoare se aduceau inutil întrucât nu s-ar fi executat.

Cum TC trebuie să lucreze într-o strânsă dependență cu predictorul de salturi, se impune îmbunătățirea performanțelor acestor predictoare. O soluție de viitor ar consta într-un predictor multiplu de salturi, al cărui rol principal constă în predicția simultană a următoarelor $(M - 1)$ salturi asociate celor maximum M blocuri stocabile în linia TC. De exemplu, pentru a predicționa simultan 3 salturi printr-o schemă de predicție corelată pe 2 nivele, trebuie expandată fiecare intrare din structura de predicție PHT (Pattern History Table), de la un singur numărător saturat pe 2 biți, la 7 astfel de automate de predicție, ca în figura 2.5. Predicția generată de către primul predictor (taken/not taken) va multiplexa rezultatele celor 2 predictoare asociate celui de al doilea salt posibil a fi stocat în linia curentă din TC. Ambele predicții aferente primelor 2 salturi vor selecta la rândul lor unul dintre cele 4 predictoare posibile pentru cel de-al treilea salt ce ar putea fi rezident în linia TC, predicționându-se astfel simultan mai multe salturi. Dacă predictorul multiplu furnizează simultan mai multe PC-uri, TC rezolvă elegant și problema aducerii simultane a instrucțiunilor pointate de aceste PC-uri, fără multiportarea pe care un cache convențional ar fi implicat-o.

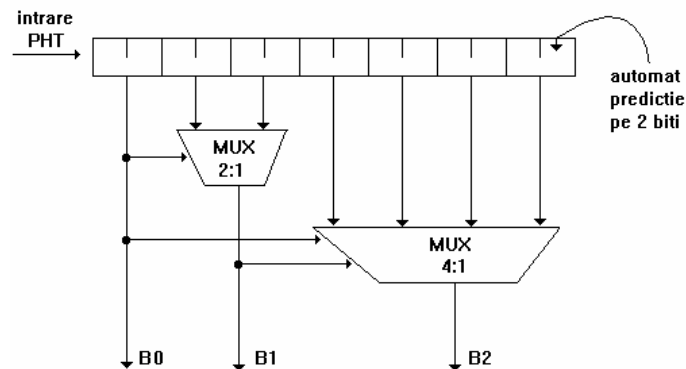


Figura 2.5. Predictor a 3 salturi succesive

Asemenea predictoare multiple în conjuncție cu structuri de tip TC conduc practic la o nouă paradigmă a procesării unui program mașină numită "*multiflow*", caracterizată prin procesarea în paralel a mai multor basic-block-uri dintr-un program. Cercetări bazate pe simulare asupra conceptelor novatoare de TC și predictor multiplu, integrate într-o

arhitectură superscalară extrem de agresivă dezvoltată la Universitatea din Michigan, SUA evidențiază următoarele aspecte [Pat97]:

- ◆ creșterea gradului de asociativitate a TC de la 0 (mapare directă) la 4 (asociativitate în blocuri de 4 intrări/ bloc) poate duce la creșteri ale ratei medii de procesare a instrucțiunilor de până la 15%.
- ◆ capacități egale ale TC și respectiv memoriei cache de instrucțiuni (64 ko, 128 ko) conduc la performanțe cvasioptimale.
- ◆ asociativitatea liniei TC nu pare a conduce la creșteri spectaculoase de performanță.
- ◆ performanța globală față de o arhitectură echivalentă, dar fără TC, crește cu circa 24%, iar rata de fetch a instrucțiunilor în medie cu 92%.

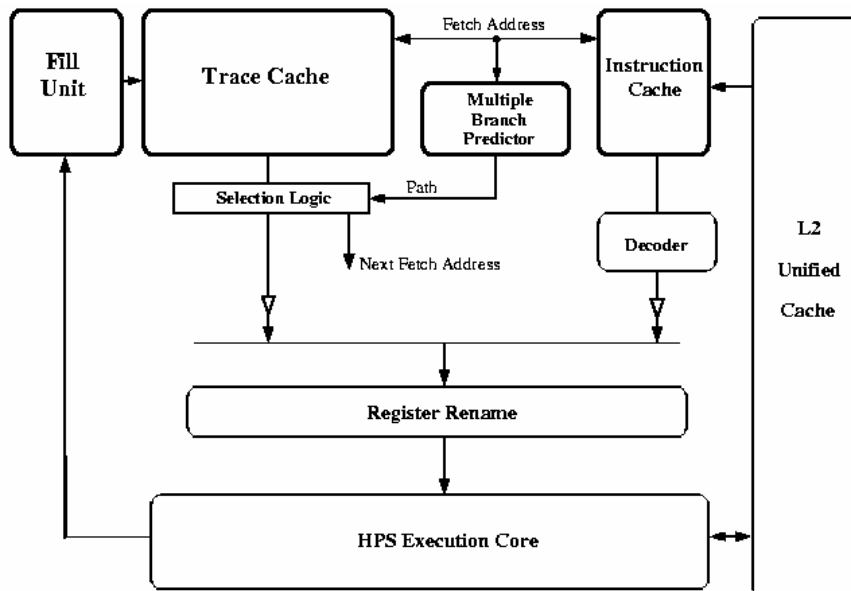


Figura 2.6. Microarhitectură de procesare care înglobează un *TraceCache*

În [Jac99] sunt propuse trei optimizări ale trace-urilor de instrucțiuni în cadrul unui trace procesor: *scheduling-ul instrucțiunilor*, *propagarea constantelor* și *colapsarea instrucțiunilor dependente de date*. Preprocesarea instrucțiunilor se face înainte de plasarea acestora în Trace Cache. Scopul preprocesării este de a completa prin transformări adiționale aplicabile în momentul execuției, și nu de a substitui optimizările compilatoarelor, pentru o mai bună utilizare a resurselor hardware (unități de execuție și lărgime de bandă a mecanismului de *issue* limitate). Scheduling-ul instrucțiunilor este unul dinamic. Practic instrucțiunile nu

sunt mutate în altă ordine ci le sunt asignate priorități care vor fi utilizate de logica de execuție *out-of-order*. Dacă o instrucțiune este identificată pentru a-i fi crescută prioritatea, atunci și lanțului de instrucțiuni dependente de aceasta le va fi sporită prioritatea. De asemenea, instrucțiunilor care folosesc valorile generate de trace-uri (basic-blocuri) anterioare le va fi decrementată prioritatea, iar instrucțiunilor care vor fi „*producătoare*” de valori pentru alte trace-uri le va fi incrementată prioritatea. Colapsarea transformă un lanț de instrucțiuni dependente de date într-o singură instrucțiune, mai complexă, cu mai mult de doi operanzi sursă. Exploatat în conjuncție cu colapsarea dependențelor de date, beneficiul scheduling-ului dinamic este mai pronunțat. Câștigul de performanță măsurat pe benchmark-urile SPEC'95 este între 4% și 24%.

La ora actuală, procesorul Intel Pentium IV reprezintă primul procesor comercial care înlocuiește nivelul L1 de cache clasic cu un *Execution Trace Cache*. Decât să memoreze instrucțiunile standard x86, acest Trace Cache reține instrucțiunile după ce tocmai au fost decodificate în instrucțiuni specifice RISC, numite microoperații - μops . În fiecare linie a Trace Cache-ului Intel stochează 6 μops . Un predictor de tip BTB cu 4096 de intrări aferent trace-cache-ului (Trace Branch Predictor) este menit să sprijine așa numitul „*Execution Trace Cache*” – 8 way asociativ, cu mecanism de evacuare de tipul Least Recently Used, care poate înmagazina până la 12000 de microoperații la versiunea Northwood a procesorului Intel realizat în tehnologie de 130nm, și respectiv până la 16000 de μops la versiunea Intel Prescott implementat în tehnologie de 90nm [Intel03]. Dacă predictorul dă greș trebuie așteptat 7 cicli până când este adusă instrucțiunea din nivelul L2 de cache sau mai mult dacă trebuie accesată memoria centrală. Dacă predicția este corectă atunci trace-cache-ul poate furniza 3 μops per ciclu de tact scheduler-ului de execuție. Întrucât memoria TC stochează doar instrucțiunile contigue din punct de vedere al execuției în zone contigue, rezultă că spațiul de cache, destul de limitat, este astfel mult mai eficient folosit. Adresarea Execution Trace Cache se face cu adrese virtuale, nefiind nevoie de conversie în adrese fizice până la accesul spre nivelul L2 de cache. Decodificatorul aferent procesorului Intel Pentium4 poate converti cel mult o instrucțiune x86 per ciclu de tact, mai puțin decât celelalte arhitecturi. Cu toate acestea, întrucât microoperațiile sunt stocate în TC și posibil refolosite parte din ele, lărgimea de bandă a unității de decodificare poate asigura o rată de 3 μops per ciclu de tact, cerut de unitatea de pre-execuție (*issue*). Dacă o instrucțiune x86 necesită mai mult de 4 μops (instrucțiuni complexe, consumatoare de timp), atunci decodificatorul extrage microoperațiile suplimentare dintr-o memorie ROM dedicată.

În final sunt subliniate câteva aspecte suplimentare referitoare la memoria Trace Cache și structurile de predicție ajutătoare, care intervin la unul din cele mai recente procesoare produse de Intel. Pentium4 Hyper-Thread conține două „*procesoare logice*” (execută practic două fire în mod concurrent) și arbitrează în fiecare ciclu de tact accesul la Trace Cache. În fiecare ciclu de tact este deservit câte un microthread. În cazul în care unul din firele de execuție este blocat atunci celălalt va putea accesa Trace Cache-ul în fiecare ciclu. Intrările din TC sunt extinse cu câte un câmp de *Tag* având informații despre fire alocarea făcându-se dinamic după necesități. Partajarea TC se poate face inegal între cele două fire după cum este nevoie. Structurile de predicție pot fi și ele partajate sau duplicate. Buffer-ul *Return Stack* care prezice destinația instrucțiunii de revenire din proceduri este duplicat deoarece este o structură foarte mică și perechile *Call / Return* sunt predicționate mai bine pentru fiecare fir de execuție în parte. Câte un *registru de istorie globală* este păstrat independent pentru fiecare thread în parte. Cu toate acestea, registrul de istorie globală aferent întregului procesor reprezintă o structură partajată având intrări cu *Tag* pentru identificarea fiecărui *thread* în parte (*processor logic*).

O tehnică de creștere a lărgimi de bandă a mecanismului de aducere prin „*sporirea*” unității atomice de instrucțiuni o reprezintă „*promovarea branch-urilor*”. Prin această tehnică, salturile condiționate, puternic polarizate (spre *taken* – T sau *not taken* – NT) sunt convertite dinamic în salturi predictibile static. Astfel, numărul salturilor prezise dinamic (chiar simultan) va scădea, iar predictorul va suferi mai puțin datorită interferențelor. Unitatea de umplere a Trace Cache-ului este responsabilă cu conversia (filtrarea) branch-urilor. Întrucât anumite salturi condiționate sunt puternic polarizate spre *not taken* iar altele spre *taken* rezultă că, prin „*promovarea*” acestora, instrucțiunile din interiorul unei unități atomice din Trace Cache sunt garantat executate toate sau neexecutate toate. Branch-urile candidate la „*promovare*” sunt detectate printr-un mecanism hardware bazat pe o tabelă de salturi care conțin rezultatul fiecărui salt (T / NT) și un automat de confidență reprezentat printr-un numărător saturat. Valoarea numărătorului reprezintă de câte ori consecutiv branch-ul a avut același rezultat. Unitatea de umplere indexează această tabelă de câte ori branch-ul este adăugat segmentului de instrucțiuni care va fi inserat în Trace Cache. Dacă valoarea numărătorului este mai mare decât un *threshold* impus, atunci saltul va fi „*promovat*”. În cazul în care un branch „*promovat*” se dovedește a fi greșit predicționat, procesarea instrucțiunilor este reluată din anteriorul „*punct de verificare*” – sfârșitul basic-block-ului anterior. Prin tehnica de *promovare* a branch-urilor, un segment de trace (un basic block) va avea mai puțini succesori, astfel că predictorul dinamic trebuie să selecteze între

mai puține „*tinte*” posibile (numărul predicțiilor necesare în fiecare ciclu de tact scade).

Simulări laborioase realizate pe benchmark-urile SPEC'95 au arătat că pentru un *threshold* de 64 este îmbunătățită rata de fetch cu 7%. De asemenea, folosind același *threshold*, necesitatea de a efectua 2 predicții per ciclu de tact este de 12%, iar necesitatea de a efectua 3 predicții per ciclu de tact este de 3%, pentru a umple o linie de 16 instrucțiuni a Trace Cache-ului.

Tehnica de „*promovare*” poate fi realizată și *static* prin extinderea arhitecturii setului de instrucțiuni (ISA) cu un câmp suplimentar de biți pentru comunicarea salturilor puternic polarizate (T/NT) procesorului, în momentul execuției. Cu toate acestea, salturile care își schimbă rezultatul pe perioada execuției (pe termen lung), dar rămân puternic polarizate (pe termen scurt) sau sunt sensibile la datele de intrare pot fi „*scăpate din vedere*” în timpul analizei statice. Există însă avantajul că salturile nu trebuie să treacă printr-o fază de „încălzire” (warm-up) până să fie detectate ca și *promovabile* [Pat98].

2.2. LIMITAREA „CONSUMATORULUI” (ISSUE BOTTLENECK). SOLUȚII.

Rata de execuție a instrucțiunilor este **fundamental limitată** de către **hazardurile RAW** între instrucțiuni (*calea critică de program*). Cauza principală a acestei limitări o constituie natura intrinsec serială a programelor, în care se dictează ordinea secvențială de transmitere a datelor între instrucțiuni. Alte limitări – **nefundamentale**, le reprezintă **hazardurile structurale** și **hazardurile de ramificație**. Aceste ultime limitări **nu** constituie o limită superioară de paralelism obținabil la nivelul instrucțiunilor întrucât pot fi depășite prin diverse tehnici software sau hardware. **Hazardurile structurale** sunt determinate de conflictele la resurse comune, adică atunci când mai multe procese simultane aferente mai multor instrucțiuni în curs de procesare, accesează o resursă comună (principală cauză a conflictelor reprezentând-o deci centralizarea resurselor). Pentru a le elimina prin hardware, se impune de obicei multiplicarea acestor resurse. De exemplu, un procesor care are un set de regiștri generali de tip uniport și în anumite situații există posibilitatea ca 2 procese să dorească să scrie în acest set simultan. O altă situație de acest fel poate consta în accesul simultan la memorie a 2 procese distincte: unul de aducere a instrucțiunii

(IF), iar celălalt de aducere a operandului sau scriere a rezultatului în cazul unei instrucțiuni LOAD / STORE (nivelul MEM). Această situație se rezolvă în general printr-o arhitectură Harvard a bus-urilor și cache-urilor (spații și bus-uri separate pe instrucțiuni și date).

O idee interesantă bazată pe descentralizarea resurselor [Fra93] are în vedere implementarea mai multor așa numite "Instruction Windows" (IW)- un fel de buffere de prefetch multiple în locul unuia singur și respectiv pe conceptul de multithreading. Lansarea în execuție a instrucțiunilor se face pe baza determinării celor independente din fiecare IW. De asemenea, trebuie determinate și dependențele inter- IW- uri. Ideea principală constă în execuția paralelă a mai multor secvențe de program aflate în IW- uri diferite, bazat pe mai multe unități funcționale (multithreading). Astfel de exemplu, 2 iterații succesive aferente unei bucle de program pot fi procesate în paralel dacă sunt memorate în IW- uri distincte. O asemenea idee facilitează implementarea conceptelor de expandabilitate și scalabilitate, deosebit de utile în dezvoltarea viitoare a arhitecturii.

Hazardurile de ramificație sunt generate de către instrucțiunile de ramificație (branch). Cauzează pierderi de performanță în general mai importante decât hazardurile structurale și de date, mai ales la procesoarele superscalare. Efectele defavorabile ale instrucțiunilor de ramificație pot fi reduse prin metode soft (**reorganizarea programului sursă**), sau prin metode hard care determină în avans dacă saltul se va face sau nu (**branch prediction**) și calculează în avans noul PC (program counter) – vezi pe larg în capitolul 5.

În momentul de față se disting mai multe abordări moderne de exploatare și creștere a paralelismului la nivelul instrucțiunii (ILP):

- ⇒ Procesoarele superscalare extrag paralelismul dinamic prin execuție speculativă, multithreading sau trimitere spre procesare "**out of order**": trace procesorul [Rot97], procesoare multithread [Wall99, Mar00], arhitectura multiscalară [Fra93]. Trace procesorul și procesorul superspeculativ speculează atât dependențele de date cât și cele de control, în timp ce arhitectura multiscalară este susținătoarea unei abordări multithread cu expediere vastă de fire spre execuție.
- ⇒ Eforturile de îmbunătățire tehnologică și arhitecturală sunt canalizate spre reducerea decalajului tehnologic dintre un procesor avansat și sistemul ierarhic de memorie (selective victim cache) [Sti94].
- ⇒ Alți factori care determină limitarea ratei de issue a instrucțiunilor sunt: lărgimea de bandă limitată a interfeței procesor - cache, miss-urile în cache-ul de date, alias-urile de memorie. Pentru a contracara efectul acestor factori se poate utiliza cu succes mecanismul **Data Write Buffer**

(DWB). DWB reprezintă un mic procesor de ieșire care lucrează în paralel cu CPU degrevându-l pe acesta de sarcina scrierii în cache. DWB oferă porturi de scriere virtuale multiple spre deosebire de DataCache care conține un număr limitat de porturi (LOAD/STORE). DWB rezolvă prin "bypassing" elegant hazardurile de tip "LOAD after STORE" cu adrese identice, deseori nemaifiind deci necesară accesarea sistemului de memorie de către instrucțiunea LOAD.

- ⇒ Prin colapsarea dependențelor reale de date între instrucțiuni se urmărește eliminarea / reducerea parțială a efectelor defavorabile cauzate de hazardurile RAW (deblocarea instrucțiunilor dependente aflate în așteptare), folosind dacă este posibil unități aritmetico-logice cu mai mult de două intrări. Tehnicile hardware recente de **reutilizare dinamică a instrucțiunilor** și **predicția valorilor**, menite să exploateze redundanța existentă în programe, reducând timpul de execuție al acestora prin colapsarea dinamică a dependențelor de date, sunt reprezentative în acest sens [Lip96, Sod00].
- ⇒ Schedulingul de instrucțiuni exploatează paralelismul în momentul compilării prin rearanjarea codului sursă, dezambiguizarea referințelor la memorie, metode de *in lining* aplicate procedurilor, tehnici de optimizare locală și/sau globală (loop unrolling, list/trace scheduling, software pipelining), etc. Procesoarele EPIC [Vin00a], cu paralelism explicit la nivelul instrucțiunilor (vezi Intel Itanium 2), prin execuția speculativă și predicativă urmăresc creșterea abilității compilatoarelor în exploatarea paralelismului în programele cu procentaj ridicat de instrucțiuni de ramificații [Sias04].
- ⇒ Dezvoltarea de noi instrumente de cercetare care aparțin și altor domenii (inteligentă artificială, algoritmi genetici etc) pentru creșterea acurateții de predicție a ramificațiilor de program - predictoare neuronale, genetice [Vin00a].

În continuare se va insista foarte pe scurt asupra ultimelor două idei, anterior enunțate. Dacă până recent, majoritatea compilatoarelor urmau un stil evoluționist de dezvoltare, bazat pe îmbunătățirea metodelor tradiționale de scheduling (global / local) pentru creșterea paralelismului la nivelul instrucțiunilor, o ultimă abordare realizată de compilatorul IMPACT [Sias04], "structurală" și bazată pe transformări radicale la nivelul "controlului" programului (execuție predicativă și speculativă, replicare de cod), încearcă o exploatare cât mai eficientă a caracteristicilor procesoarelor EPIC, având același deziderat de performanță.

În calea optimizărilor compilatorului pentru creșterea paralelismului la nivelul instrucțiunilor se remarcă câteva obstacole:

- **instrucțiunile de salt**, care descriu modul de traversare al grafului de control. Efectul defavorabil este redus prin execuție predicativă și folosirea regiștrilor booleeni de gardă. Dependențele de control sunt transformate în dependențe de date.
- **false dependențe**: accesul la memorie și apelurile de subrutine reprezintă bariere în calea „mișcării” codului („*percolation*” [Vin00a]), blocând atât scheduling-ul cât și optimizarea acestuia. IMPACT încearcă eliminarea acestor false dependențe prin algoritmi complecși de analiză interprocedurală [Sias04].
- **dependențele ocazionale**, provocate de execuția out-of-order a instrucțiunilor *load* / *store*, nu pot fi înlăturate static fără suport suplimentar hardware pentru execuție speculativă (analiza antialias dinamică ce presupune utilizarea de cod și regiștrii suplimentari) nerezolvat încă de IMPACT.
- **nedeterminismul**, poate fi introdus, spre exemplu, de accesul cu miss la memoria cache de date a instrucțiunilor cu referire la memorie.

Compilerul dezvoltat în laboratoarele de cercetare ale universității din Illinois (IMPACT) și dedicat versiunii de procesor EPIC pe 64 biți (Intel Itanium 2), dezvoltat de Intel în colaborare cu Hewlett-Packard, realizează o analiză interprocedurală, *inlining* aplicat procedurilor, dar și anumite modificări impuse în urma obținerii informațiilor de profil. Un exemplu de astfel de optimizare o reprezintă „*type feedback*” (vezi detalii în subcapitolul 4.1), care, bazat pe informații de profil, transformă apelurile indirecte de funcții în apel direct și cărora li se poate aplica apoi tehnica de „*inlining*”. Abordarea „structurală” se referă la procesul de simplificare a grafului de control, generarea de zone de cod largi, stabile din punct de vedere al execuției (trace-uri) și reorganizarea acestora. Dintre avantajele compilerului IMPACT relativ la arhitectura Intel Itanium 2 se remarcă speed-up-ul obținut față de compilerul GNU *gcc* (până la 2.3 [Sias04]), reducerea numărului de salturi cu 27%, reducerea numărului de cicluri de penalizare în cazul unei predicții eronate cu 22%, îmbunătățirea eficienței procesului *fetch* – instrucțiune și diminuarea cu 15% a stagnărilor datorate miss-urilor în cache-ul de instrucțiuni. Ca și efecte secundare, ocazional, prin execuția speculativă a instrucțiunilor *load* pot apărea întârzieri suplimentare datorate miss-urilor în cache-ul de date.

Datorită învechirii paradigmei actuale de cercetare, cercetătorii [Vin01] opinează că, pentru continuarea creșterii exponențiale a performanței microprocesoarelor, sunt necesare idei noi, revoluționare chiar, bazate pe o *abordare integratoare*, care să îmbine eficient tehnicile de scheduling software cu cele dinamice, de procesare hardware. În prezent, separarea între cele 2 abordări este destul de accentuată. În acest sens,

programele ar trebui să explicitizeze paralelismul intrinsec într-un mod mai clar. Cercetări actuale arată că un program optimizat static merge mai prost pe un procesor Out of Order decât pe unul In Order [Tat00]. Printre cauze se amintesc expansiunea codului după reorganizare, noile dependențe de date introduse prin execuția condiționată a instrucțiunilor, faptul că instrucțiunile gardate nu permit execuția Out of Order etc.

Cercetarea algoritmilor ar trebui să țină seama și de concepte precum, cel de cache, în vederea exploatarei localităților spațiale ale datelor prin chiar algoritmul respectiv. În general, algoritmii nu țin cont la ora actuală de caracteristicile mașinii și acest lucru nu este bun pentru că algoritmul nu rulează într-un "eter ideal"[Vin00a] ci, întotdeauna pe o mașină fizică având limitări importante (de ex. elementele unui tablou se pot afla parțial în cache și parțial pe disc!). Astfel, dihotomia teorie - practică devine una artificială și cu implicații negative asupra performanței globale a mașinii.

Realizatorii aplicațiilor obiectuale și vizuale trebuie să țină cont de faptul că polimorfismul generează apeluri indirecte de funcții, greu predictibile la nivel hardware [Flo04]. Preocupările programatorilor nu trebuie să vizeze doar interfața care atrage sau diversele artificii care fac din utilizator un simplu robot ci și implicațiile pe care aplicația creată o are asupra microarhitecturii. Scopul aplicației trebuie să fie utilizarea optimă atât a resurselor software (biblioteci, elemente de interfață) avute la dispoziție cât și a algoritmilor / conceptelor de programare cunoscute (declarații de funcții virtuale, apeluri de funcții prin pointer chiar și acolo unde nu este cazul). În caz contrar, "răul" (a se citi în primul rând salturi indirecte, cod obiect masiv, resurse hardware suplimentare) se răsfrânge asupra performanțelor arhitecturii. În ce-i privește pe proiectanții de arhitecturi, schemele propuse de aceștia ar putea fi mai eficiente dacă nu ar analiza numai codul obiect al benchmark-urilor avute la dispoziție (dezbrăcat de orice semantică) ci ar privi "mai sus" spre sursa de nivel înalt a programelor simulate.

Abordarea strict convențională, situată doar la nivelul "arhitecturilor de calcul", pare să fie insuficientă pentru îndeplinirea dezideratului de performanță ridicată. O abordare mai neconvențională, care să utilizeze concepte ale unor domenii considerate până în prezent a nu avea legătură cu arhitectura sistemelor de calcul (arbori de decizie, rețele neuronale, algoritmi genetici, algoritmi de predicție PPM) poate genera rezultate surprinzătoare, precum și o îmbunătățire a paradigmei arhitecturilor avansate. În [Vin99b, Jim02] sunt propuse, cu mare succes, structuri de predicție alternative care fac o legătură neașteptată între domeniul arhitecturii procesoarelor avansate și cel al recunoașterii formelor. Predictoarele neurale de tip Perceptron și MultiLayerPerceptron constituie

soluții fezabile hardware și cu acurateți de predicție cel puțin de nivelul predictorilor corelate pe două niveluri. În [Vin00b] este descris modul de determinare automată cu ajutorul arborilor binari a unor noi scheme de predicție a salturilor pe baza unor algoritmi genetici, pornind de la o populație inițială de predictoare cunoscute. Există cercetări cu rezultate remarcabile în domeniul predicției salturilor [Vin99a] sau cel al predicției valorilor [Saz99], care utilizează lanțuri Markov și algoritmul de predicție bazat pe *potrivire parțială* (PPM – complet), folosit cu succes în compresia datelor. În subcapitolul 5.3 am descris câteva cercetări proprii privind predicția target-urilor salturilor indirecte folosind predictoare bazate pe context și algoritmul PPM-complet. Dintre cele mai recente cercetări, se poate ilustra ca exemplu în sprijinul ideii de abordare neconvențională, folosirea arborilor de decizie pentru selecția celor mai relevante caracteristici necesare procesului de predicție [Fern03].

2.2.1. REUTILIZAREA DINAMICĂ A INSTRUCȚIUNILOR

Reutilizarea dinamică a codului se înscrie în domeniul optimizărilor arhitecturilor de calcul și s-a manifestat pentru prima dată la nivel software, prin tehnica de *programare dinamică* – metodă de rezolvare a problemelor a căror soluție se construiește dinamic în timp. Introdusă încă din 1957 de către matematicianul american Richard Bellman [Bell57], programarea dinamică operează într-o manieră „*bottom-up*”, nerecursiv și presupune cunoașterea exactă, de la început, a subproblemelor – care nu sunt independente – apărute în descompunerea problemei inițiale. Pentru a fi *eficientă*, metoda programării dinamice trebuie să *rezolve fiecare subproblemă o singură dată și să memoreze soluția* acesteia pentru a o putea utiliza în cazul reapariției aceleiași subprobleme în cadrul unei alte subprobleme. Fazele rezolvării unei probleme prin metoda programării dinamice sunt:

- rezolvarea subproblemelor de dimensiunile cele mai mici care apar în descompunerea problemei inițiale și memorarea soluțiilor acestora;
- rezolvarea treptată a subproblemelor de dimensiuni din ce în ce mai mari prin combinarea soluțiilor subproblemelor de dimensiuni mai mici și memorarea soluțiilor acestora până la obținerea rezultatului final.

Reutilizarea dinamică a instrucțiunilor (reutilizare de tip „*fine grain*”) este o tehnică non-speculativă menită să exploateze fenomenul de repetiție

dinamică a instrucțiunilor, reducând cantitatea de cod - mașină necesar a fi executat și care prin colapsarea dependențelor de date determină îmbunătățirea timpului de execuție al instrucțiunilor crescând gradul de paralelism al arhitecturii. Ideea originală aparține cercetătorilor *A. Sodani și G. Sohi* și a fost introdusă în 1997, la conferința *ISCA '97* ținută la *Denver, SUA*. În [Sod97] se arată că reutilizarea unor instrucțiuni sau secvențe de instrucțiuni este relativ frecventă și se datorează modului compact de scriere a programelor precum și caracteristicilor intrinseci ale structurilor de date prelucrate. O instrucțiune dinamică este reutilizabilă dacă ea operează asupra acelorași intrări și produce aceleași rezultate precum o instanță anterioară a aceleiași instrucțiuni. Ideea de bază este că dacă o secvență de instrucțiuni se reia în același “*context de intrare*”, atunci execuția sa nu mai are sens fiind suficientă o simplă actualizare a “*contextului de ieșire*”, în concordanță cu unul precedent memorat. Se reduce astfel numărul de instrucțiuni executate dinamic, acționându-se direct asupra dependențelor de date între instrucțiuni. Instrucțiunile reutilizate nu se vor mai executa din nou, contextul procesorului fiind actualizat în conformitate cu acțiunea acestor instrucțiuni, bazat pe istoria lor memorată.

În [Sod98] se analizează mai întâi dacă gradul de reutilizare a instrucțiunilor dinamice este semnificativ și se arată că răspunsul este unul afirmativ. Mai puțin de 20% din numărul instrucțiunilor statice care sunt repetate implică o repetabilitate de peste 90% a instrucțiunilor dinamice. În medie armonică, măsurat pe benchmarkurile SPEC'95, 26% dintre instrucțiunile dinamice sunt reutilizabile. Există în acest sens 2 cauze calitative: în primul rând faptul că programele sunt scrise în mod generic, ele operând asupra unei varietăți de date de intrare, iar în al doilea rând, aceste programe sunt scrise într-un mod concis – aceasta semnificând menținerea unei reprezentări statice compacte a unei secvențe dinamice de operații – în vederea obținerii rezultatelor dorite (în acest sens structurile de tip recursiv, “*buclele*” de program etc. sunt reprezentative).

Pentru o mai bună înțelegere a fenomenului de repetiție a instrucțiunilor, execuția dinamică a programelor este analizată pe trei niveluri: global, de funcție și local (în interiorul funcției). În analiza globală, pattern-urile de date utilizate în programe sunt reținute ca entități întregi și determinate sursele de repetiție ale instrucțiunilor (intrări externe, inițializări globale de date sau valori interne ale programelor). Întrucât repetiția instrucțiunilor se datorează în mare măsură ultimelor două surse de repetiție, se impune concluzia că fenomenul de repetiție este mai mult o proprietate a modului în care calculul este exprimat în program și mai puțin o proprietate a datelor de intrare. Concluziile generate în urma analizei la nivel de funcție sunt că de foarte multe ori funcțiile sunt invocate repetat cu exact aceleași

valori ale parametrilor de intrare și că relativ puține apeluri de funcții nu au argumente repetate. Chiar și în cazul unor apeluri repetate ale unei funcții cu parametri de intrare diferiți, procentajul de instrucțiuni dinamice reutilizabile poate fi semnificativ. La nivelul analizei locale, instrucțiunile funcțiilor/procedurilor sunt clasificate în funcție de sursa valorilor folosite (exemplu: argumentele funcției, date globale, valori returnate de alte funcții etc.) și funcție de sarcina realizată (exemplu: salvare - restaurare regiștri, prolog - epilog, calcul adrese globale etc.). Majoritatea repetiției instrucțiunilor se datorează valorilor globale sau argumentelor funcției dar și funcțiilor prolog și epilog.

Preluat din [Sod97], se prezintă în continuare un exemplu sugestiv în care apare fenomenul de reutilizare dinamică a instrucțiunilor. Funcția *func* (figura 2.7.a) caută o valoare x în tabloul *list* de dimensiunea *size*. Funcția principală *main_func* (figura 2.7.c) apelează funcția *func* de mai multe ori, căutând câte un alt element în același tablou la fiecare apel. La apelul funcției *func* tabloul este parcurs element cu element în mod iterativ, căutându-se valoarea până la capătul tabloului, condiția de încheiere a căutării reprezentând-o găsirea elementului. Expandarea buclei din interiorul funcției *func* corespundente unei iterații este prezentată în figura 2.7.b. Instanțele dinamice ale instrucțiunilor generate de primul apel *func* sunt descrise în figura 2.7.d. În fiecare iterație a buclei, instrucțiunea 2 este dependentă de parametrul *size*, instrucțiunile 3 și 4 sunt dependente de parametrul *list*, instrucțiunea 5 este dependentă atât de *list* cât și de valoarea căutată în tablou, iar instrucțiunea 6 este dependentă de contorul i . Dacă *func* e apelată din nou în același tablou *list* (de aceeași dimensiune *size*), dar cu alt parametru de căutare, atunci toate instanțele dinamice ale instrucțiunilor 1 ÷ 4 și 6 vor produce aceleași rezultate pe care le-au produs la apelul anterior al funcției *func*. Doar instanțele dinamice ale instrucțiunii 5 produc rezultate care ar putea diferi de apelurile anterioare ale funcției *func*. Repetarea rezultatelor instanțelor dinamice ale instrucțiunilor 1 ÷ 4 și 6 este direct atribuită faptului că *func* a fost scrisă ca o funcție generică de căutare într-un tablou, dar în acest caz particular, doar unul din parametri se modifică între apeluri diferite. Chiar dacă *func* ar fi apelată cu toți parametri diferiți pentru fiecare apel în parte, instanțele dinamice diferite ale instrucțiunii 6 ($i = 0, i = 1, i = 2, \dots$) vor produce aceleași valori generate în primul apel al funcției, consecință a utilizării buclelor pentru a exprima calculele dorite într-o manieră concisă. Dacă parametrul *size* ar fi diferit la un al doilea apel al funcției *func*, atunci doar $\min(size1, size2)$ instanțe dinamice ale instrucțiunii 6 vor produce aceleași rezultate. Prin urmare, acest exemplu sugestiv arată faptul că repetabilitatea instrucțiunilor

dinamice este considerabilă și în consecință reutilizarea instrucțiunilor este posibilă. Instanțele dinamice marcate cu "*" vor realiza aceleași operații pentru ambele apeluri ale funcției *func*.

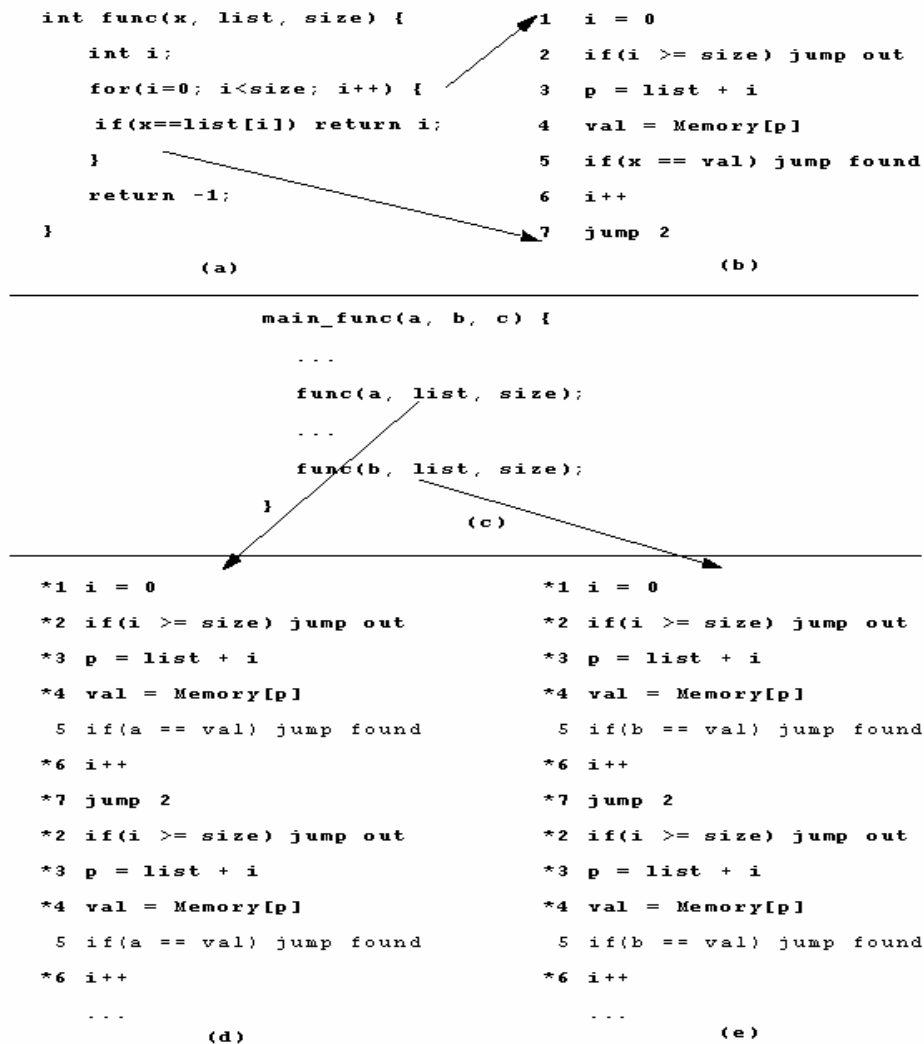


Figura 2.7. Exemplu ilustrând repetabilitatea instrucțiunilor

Pentru o înțelegere mai profundă a conceptului de reutilizare dinamică a instrucțiunilor se impune cunoașterea răspunsului la unele întrebări: *căror caracteristici ale structurilor de programe scrise în limbaje de nivel înalt li se datorează gradele ridicate de reutilizabilitate? Care sunt relațiile între caracteristicile programării obiectuale și reutilizarea dinamică ori în ce fel contribuie tehnicile de optimizare locale și globale aferente programelor*

asupra reutilizabilității instrucțiunilor mașină ? [Vin02] În ciuda cercetărilor din domeniu aceste probleme rămân încă neelucidate.

Bazat în principal pe premisele anterior expuse, *Sodani* și *Sohi* dezvoltă 3 scheme de reutilizare dinamică a instrucțiunilor, primele două la nivel de instrucțiune iar ultima, la nivel de lanț de instrucțiuni dependente RAW [Sod97]. Instrucțiunile deja executate, se memorează într-un mic cache numit buffer de reutilizare (*Reuse Buffer* - RB). Acesta poate fi adresat cu PC-ul pe timpul fazei de aducere a instrucțiunii, având și un mecanism pentru invalidarea selectivă a unor intrări bazat pe acțiunile anumitor evenimente (vezi figura 2.8). RB trebuie să permită și un mecanism de testare a reutilizabilității instrucțiunii selectate. Testul de reutilizare verifică dacă informația accesată din RB reprezintă un rezultat reutilizabil sau nu. Detaliile de implementare ale testului depind de fiecare schemă de reutilizare folosită. Trebuie tratate două aspecte privind managementul RB: stabilirea instrucțiunii care va fi plasată în buffer și menținerea consistenței bufferului de reutilizare. Decizia privind modul de inserare a instrucțiunilor în RB poate varia de la una nerestrictivă ("*no policy*"), care plasează toate instrucțiunile în buffer, în cazul în care nu sunt deja prezente, la una mai selectivă, care filtrează instrucțiunile ce vor fi inserate după probabilitatea statistică de a fi reutilizate. Problema consistenței are în vedere garantarea corectitudinii rezultatului instrucțiunii reutilizate din RB. Menținerea consistenței informațiilor în RB depinde de fiecare schemă de reutilizare în parte după cum se va putea constata în cele ce urmează.

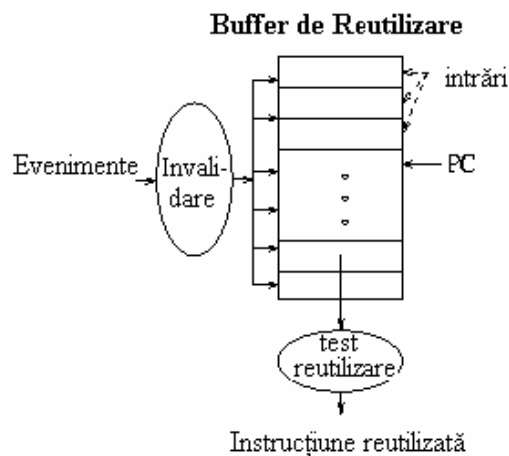


Figura 2.8. Structura hardware a bufferului de reutilizare

În vederea compatibilizării cu modelul superscalar care lansează în execuție mai multe instrucțiuni simultan, RB este în general multiport

pentru a putea permite reutilizarea mai multor instrucțiuni de execuție curentă. Este evident că gradul de multiportare al RB-ului nu are sens a fi mai mare decât fereastra maximă de execuție a instrucțiunilor.

În cazul reutilizării la nivel de instrucțiune, o intrare în RB ar putea avea următorul format:

Tag	Op1	Op2	Adr	Rez	Rez valid	Mem Valid
------------	------------	------------	------------	------------	------------------	------------------

Tag – ar putea fi reprezentat de către PC-ul instrucțiunii într-o implementare asociativă.

Op1, Op2 – reprezintă numele regiștrilor utilizați de către instrucțiune.

Rez – reprezintă rezultatul actual al instrucțiunii, cel care va fi reutilizat în caz de “hit” în bufferul RB.

Rez_Valid – indică, în cazul instrucțiunilor aritmetico-logice, dacă rezultatul “Rez” este valid sau nu. În cazul instrucțiunilor Load și Store, dacă e setat, arată că adresa instrucțiunii este validă în RB și poate fi deci reutilizată. Este setat odată cu introducerea instrucțiunii în RB. Este resetat automat de către orice instrucțiune care scrie într-unul din regiștrii sursă (Op1, Op2).

Adr – este adresa (reutilizabilă) de memorie în cazul unei instrucțiuni Load/Store.

Mem_Valid – indică dacă valoarea din câmpul “Rez” este reutilizabilă în cazul unei instrucțiuni Load. Bitul este setat la înscrierea instrucțiunii Load în RB. Resetarea bitului se face prin orice instrucțiune Store având aceeași adresă de acces.

Rezultă că pentru instrucțiunile aritmetico-logice reutilizarea este asigurată dacă bitul de stare *Rez_Valid* = 1. De asemenea, *Rez_Valid* = 1 garantează adresa corectă pentru orice instrucțiune Load/Store și scutește procesorul de calculul ei (adresare indexată). În schimb, rezultatul unei instrucțiuni Load nu poate fi reutilizat decât dacă *Mem_Valid* = 1 și *Rez_Valid* = 1. Plusul de performanță datorat reutilizării dinamice a instrucțiunilor se datorează atât scurtcircuitării unor nivele din structura “pipe” cât și reducerii hazardurilor structurale și deci a presiunii asupra diverselor resurse hardware. Prin reutilizarea instrucțiunilor se evită stagnarea în stațiile de rezervare (*Instruction Window*) și timpul de execuție, rezultatele instrucțiunilor reutilizate fiind scrise mai repede în bufferul de reordonare. Rezultă o disponibilizare a unităților funcționale de execuție care nu vor mai avea de procesat instrucțiunile reutilizate și o deblocare mai rapidă a instrucțiunilor dependente RAW de cea reutilizată. De remarcat că

evacuarea din RB trebuie să țină cont de faptul că instrucțiunile invalidate trebuie să aibă prioritate în acest proces.

În cazul unei scheme care reutilizează un întreg lanț de instrucțiuni dependente, structura unei intrări RB este aceeași cu cea precedentă, doar că aici apar două noi subcâmpuri, asociate operanzilor sursă, notate *SrcIndex1* respectiv *SrcIndex2*. Acestea pointează spre adresele instrucțiunilor din RB care au produs operandul sursă 1 respectiv operandul sursă 2, aferenți instrucțiunii curente memorate în RB. Aici, instrucțiunile sunt clasificate în 3 categorii: sursă – care produc rezultate pentru alte instrucțiuni din lanț numite dependente și respectiv independente – a căror operanzi sursă nu sunt produși în cadrul lanțului de instrucțiuni considerat. O altă diferență esențială față de schema anterioară constă în faptul că pentru schema de reutilizare la nivel de lanț de instrucțiuni (dependente), în cazul modificării unui operand sursă, sunt invalidate doar instrucțiunile independente care conțin acest operand. Pentru a înțelege mai bine beneficiile acestei reguli selective de invalidare, se consideră următoarea secvență de instrucțiuni dependente RAW [Vin02]:

```
I: R1 <- 0  
J: R2 <- R1 + 5  
K: R3 <- R1 + R2  
.....  
R: R1 <- 6
```

În acest caz, procesarea instrucțiunii R nu va invalida instrucțiunile J și K pentru că acestea sunt dependente de instrucțiunea I. Mai mult, instrucțiunea R nu va invalida nici chiar instrucțiunea independentă I, pentru simplul motiv că registrul R1 nu este sursă în această instrucțiune. Astfel, la o nouă instanțiere a lanțului IJK, rezultatele acestuia ($R1 = 0$, $R2 = R3 = 5$) vor fi reutilizate, nemaifiind necesară procesarea efectivă a instrucțiunilor respective. Din păcate, în cazul schemei anterioare acesteia (cea cu reutilizare la nivelul unei singure instrucțiuni), instrucțiunea R ar fi invalidat, în mod conservator și inutil, instrucțiunile J și K, nepermițând astfel reutilizarea acestora deși ele sunt, evident, reutilizabile.

O altă problemă care se pune [Vin02] se referă la integrarea instrucțiunilor de ramificație (*branch*) în schemele de reutilizare dinamică. Considerarea acestor instrucțiuni conduce la restricții în politica de introducere a instrucțiunilor în RB. Pentru a analiza această problemă, se consideră următoarea secvență de instrucțiuni:

I1: R1 <- 1
I2: BRANCH <Cond>, I4; If <Cond>=True, salt la I4
I3: R1 <- 0
I4: R2 <- R1+4

În cazul execuției speculative a instrucțiunilor, politica de introducere a instrucțiunilor în RB, așa cum a fost ea descrisă anterior, poate fi una inadecvată. Mai întâi, se va considera schema de reutilizare la nivel de instrucțiune, anterior descrisă. Se presupune că I2 este inițial predicționată ca *Not Taken*. Ar rezulta că instrucțiunile I3 și I4 se introduc în RB. În cazul în care, ulterior, se constată că I2 a fost greșit predicționată, este necesară refacerea contextului și execuția căii *Taken*. În consecință, este posibilă reutilizarea instrucțiunii I4, care va genera un rezultat eronat întrucât operandul sursă R1=0, așa cum l-a modificat I3. Pentru rezolvarea acestei anomalii se impune ca o instrucțiune speculativă să fie inserată în RB numai dacă instrucțiunile sale sursă sunt nespeculative. Astfel, în exemplul considerat, cum I3 este speculativă, rezultă că I4 nu se va introduce în RB și deci eroarea anterior semnalată nu mai apare.

În cazul schemei cu reutilizare la nivel de lanț de instrucțiuni dependente, situația este mai complicată. În acest caz, după introducerea instrucțiunilor I3 și I4 în RB, va exista un pointer de la I4 la I3 semnificând faptul că I3 este sursă. Așadar, când datorită predicției eronate se va procesa ramura alternativă, I4 nu se va mai reutiliza pentru că legătura sa cu I3 va dispărea, în acest caz instrucțiunea sa sursă fiind I1. Totuși, aceeași problemă care a apărut la schema anterioară va apărea și în acest caz dacă I4 ar fi fost inserată în RB ca instrucțiune independentă (spre exemplu, dacă din anumite motive, I3 nu ar mai fi în RB). Prin urmare, în cazul acestei scheme, se impune ca o instrucțiune speculativă să fie inserată în RB numai dacă instrucțiunile sale sursă sunt nespeculative sau dacă toate instrucțiunile sursă sunt prezente în RB.

Preluat din [Con99] se prezintă o idee relativ nouă pentru exploatarea repetiției dinamice a instrucțiunilor bazată pe integrarea unor tehnici hardware-software. Mai precis, în această abordare compilatorul analizează codul în scopul identificării acelor “regiuni de program” care ar putea fi reutilizate pe timpul execuției dinamice a instrucțiunilor. Compilatorul comunică codului obiect prin intermediul unei interfețe simple, special concepute în acest scop, regiunea de cod reutilizabil, specificând desigur intrările și ieșirile aferente acesteia. Pe timpul execuției programului, rezultatele acestor regiuni reutilizabile sunt memorate în buffer-e hardware pentru a fi potențial reutilizate. Spre exemplificarea acestor idei se consideră o macro-definiție preluată din cadrul benchmarkului *008.espresso*,

aparținând setului SPEC. Aceasta calculează numărul biților setați pe unu logic din cadrul unui cuvânt v pe 32 de biți. În acest scop se împarte cuvântul v în 4 octeți iar după prelucrarea lor, fiecare octet este utilizat pe post de index în tabela *bit_count*. Apoi, rezultatele obținute pentru fiecare octet al cuvântului, sunt însumate, rezultând astfel numărul de biți de unu logic din cuvântul v .

```
#define count_ones(v)\  
    (bit_count[v&255] + bit_count[(v>>8)&255]\  
    + bit_count[(v>>16)&255]  
    + bit_count[(v>>24)&255])
```

Considerând: A- instrucțiuni aritmetico-logice, L- Load, R – deplasare la dreapta, S – deplasare la stânga, graful dependențelor de date aferent secvenței anterioare scrise în limbajul C, este prezentat în figura 2.9. Se observă în mod clar că întregul graf conține o singură intrare ($r3$ – cu valoarea v) și respectiv generează o singură ieșire ($r26$ – numărul biților setați pe unu logic). Compilatorul poate determina prin analiză *anti-alias* că tabelul *bit_count* este static și prin urmare nu se schimbă pe durata execuției. Astfel, devine evident faptul că întregul graf de instrucțiuni ar fi reutilizabil în ipoteza în care valoarea introdusă în $r3$ este aceeași. Schemele “pur hardware” de reutilizare vor fi practic incapabile să sesizeze structura acestei secvențe și faptul că singura sa ieșire este $r26$. În plus, memorarea tuturor acestor instrucțiuni implică un consum mare de resurse. În schimb, la nivelul compilatorului, graful este vizibil de vreme ce chiar acest compilator l-a construit. În consecință, compilatorul poate construi un graf alternativ celui din figura 2.9, utilizând de exemplu o instrucțiune specială numită REUSE [Con99]. Aceasta comunică cu buffer-ele hardware pentru a decide dacă $r26$ poate să fie reutilizat sau nu. Dacă da, instrucțiunea REUSE va actualiza doar $r26$ cu valoarea respectivă și va trece la următoarea instrucțiune (vezi figura 2.9). Rezultă o schemă hibridă hardware-software de reutilizare care are avantajul față de schemele pur hardware că oferă o viziune superioară asupra structurii programului și deci, implică performanțe mai bune. În plus, această abordare exploatează o “redundanță semantică” superioară [Vin02], existentă la nivelul programelor scrise în limbaje de nivel înalt și invizibilă în mod normal programelor obiect. Pentru a fi fezabilă o astfel de idee, setul de instrucțiuni al mașinii va trebui în acest caz să fie îmbogățit cu o interfață corespunzătoare, prin intermediul căreia soft-ul va comunica cu logica hardware de reutilizare. Creșterile de performanță raportate pentru o astfel de arhitectură hibridă față de una

superscalară echivalentă sunt de cca. 30% [Con99], ceea ce dovedește eficiența acestei abordări hibride.

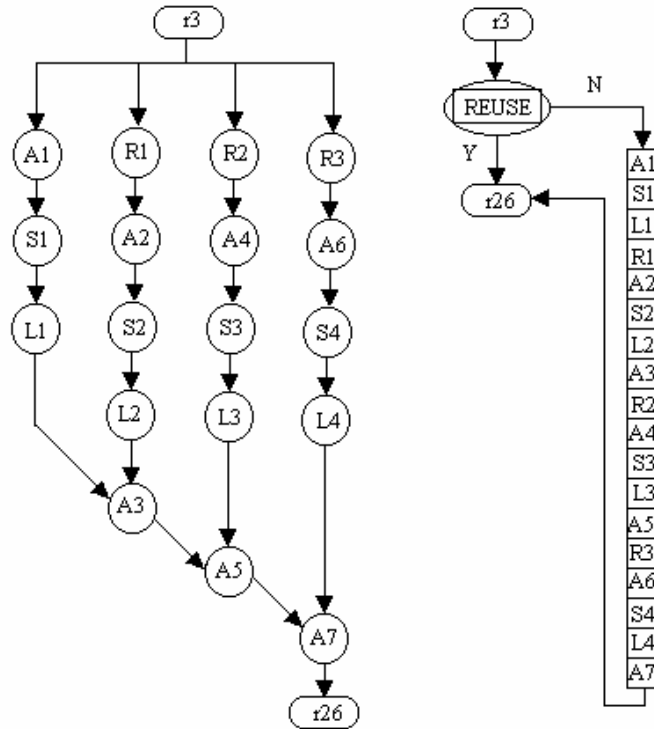


Figura 2.9 Graful dependențelor de date și secvența reutilizată

Figura 2.10 ilustrează o microarhitectură tipică cu reutilizare a instrucțiunilor. Singura modificare de principiu față de modelul superscalar este dată de apariția buffer-ului de reutilizare. În faza *fetch instrucțiune* sunt extrase din cache-ul de instrucțiuni sau memoria principală instrucțiunile și plasate în bufferul de prefetch (*Instruction Queue*). Urmează apoi faza de *decodificare* a instrucțiunilor și *redenumire* a regiștrilor în vederea eliminării conflictelor de nume (dependențe WAR și WAW). În faza de *citire operand* valorile operandilor aferenți instrucțiunilor sunt citite fie din setul de regiștri generali fie din buffer-ul de reordonare, funcție de structura care conține ultima versiune a regiștrilor. Accesul la buffer-ul de reutilizare poate fi *pipeline*-izat și suprapus cu faza de aducere a instrucțiunii. Imediat după decodificarea instrucțiunii, în timpul fazei de citire operandi, se realizează testul de reutilizare asupra intrărilor citite din RB pentru a ști dacă rezultatele instrucțiunilor sunt, sau nu, reutilizabile. Dacă este găsit un rezultat reutilizabil instrucțiunea aferentă nu mai trebuie procesată în

continuare, acesta fiind transmis direct buffer-ului de reordonare. Instrucțiunile Load evită fereastra *Instruction Window* doar dacă rezultatele ambelor micro-operații (calculul adresei și accesarea memoriei) sunt reutilizabile. Testarea reutilizării poate dura unul sau mai mulți cicli, având în vedere că este un proces secvențial ce depinde de numărul de instrucțiuni dependente din lanț.

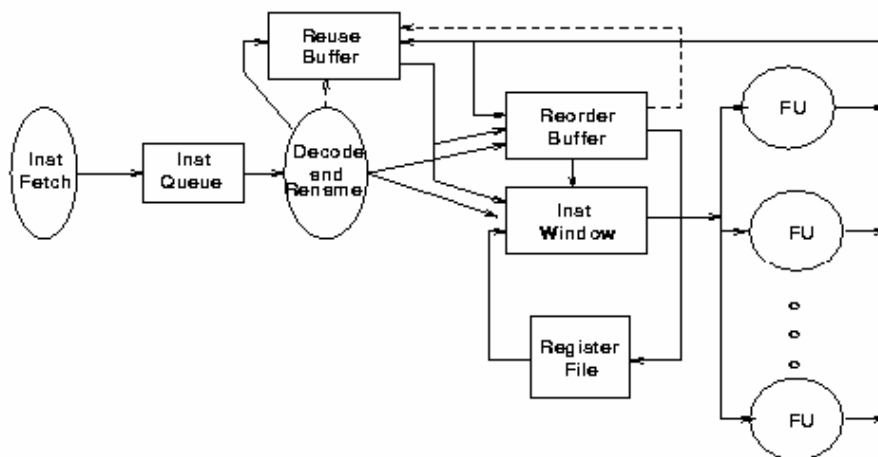


Figura 2.10. Microarhitectură superscalară generică cu buffer de reutilizare

În cazul predicției eronate a unei instrucțiuni de ramificație, mecanismul de refacere a contextului va trebui să fie suficient de selectiv astfel încât să nu invalideze în RB instrucțiunile situate imediat după posibilul punct de convergență al ramificației și care ar putea fi reutilizate. Reutilizarea este exploatată la maxim și în acest caz, cu beneficii evidente asupra performanței.

Rezumând, se desprind câteva avantaje introduse de tehnica de reutilizare dinamică a instrucțiunilor și anume:

- Scurtcircuitarea unor nivele din structura pipe de către instrucțiunile reutilizate, reducând presiunea asupra resurselor (stații de rezervare, unități funcționale, porturi ale cache-urilor de date etc.) necesare altor instrucțiuni aflate în așteptare.
- La reutilizarea unei instrucțiuni rezultatul său devine cunoscut mai devreme decât în situația în care s-ar procesa normal, permițând în consecință altor instrucțiuni dependente de aceste rezultate să fie executate mai rapid.
- Reduce penalitatea datorată predicției eronate a adreselor destinație în cazul instrucțiunilor de salt, prin reutilizarea, fie și parțială, codului succesor punctului de convergență.

- Colapsarea dependențelor de date determină îmbunătățirea timpului de execuție al instrucțiunilor crescând gradul de paralelism al arhitecturii.
- Procentajul de reutilizare al instrucțiunilor dinamice calculat pe benchmark-urile SPEC'95 este semnificativ, ajungându-se la valori maxime de 76% [Sod00].
- Accelerarea obținută față de modelul superscalar pe aceleași programe de test nu este la fel de pronunțată ca și procentajul de reutilizare (medii de 7 - 15%), valoarea maximă atinsă fiind de 43% [Sod00].

Reutilizarea dinamică a instrucțiunilor (DIR) este o tehnică nespeculativă care recunoaște un lanț de instrucțiuni dependente executat anterior și nu-l mai execută din nou - *early validation* - actualizând doar diferite date (rezultatele) în tabelele hardware aferente. DIR comprimă un lanț de instrucțiuni din calea critică de execuție a programului.

Figura 2.11 prezintă implementarea în structura pipeline a unei microarhitecturi a mecanismului de reutilizare dinamică a instrucțiunilor.

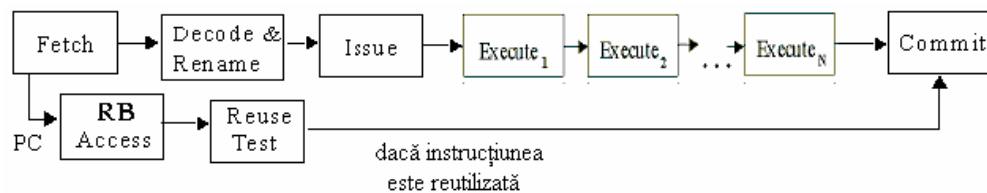


Figura 2.11. Pipeline cu DIR

Deoarece DIR validează rezultatele devreme în structura *pipe*, bazat pe intrări, pot apare următoarele situații dezavantajoase: dacă intrările unei instrucțiuni nu sunt disponibile în momentul realizării testului de reutilizare atunci respectiva instrucțiune nu va fi reutilizată. O instrucțiune care produce un rezultat identic cu unul anterior, dar cu intrări diferite (operații logice, instrucțiuni Load), nu va fi reutilizată.

DIR reduce penalitatea datorată unei predicții greșite a salturilor din două motive. În primul rând, când un branch (salt condiționat) predicționat greșit este reutilizat, predicția eronată este detectată mai devreme (în faza de decodificare) decât s-ar realiza dacă saltul s-ar executa. Al doilea motiv îl constituie posibila convergență a codului în programe. Astfel prin posibila reutilizare a codului existent după punctul de convergență a căilor de execuție, și care este evacuat în cazul unei predicții eronate, tehnica DIR îmbunătățește timpul de execuție al programelor.

DIR influențează concurența asupra resurselor prin schimbarea atât a pattern-ului în care resursele sunt folosite cât și a cererii efectuate (prin colapsarea dependențelor de date, determină execuția mai devreme a instrucțiunilor). Întrucât o instrucțiune reutilizată nu se execută, DIR tinde

să reducă concurența la resurse. Sunt eliberate astfel resurse și puse la dispoziția altor instrucțiuni concurente. DIR scade latența de execuție a operațiilor individuale de la mai mulți cicli la doar un singur ciclu (latența de reutilizare a unei instrucțiuni).

Este evident faptul că, cu cât reutilizarea se face la un grad de granularitate mai mare (instrucțiuni multiple reutilizate simultan) cu atât este mai eficient pentru performanța globală a sistemului. Pentru identificarea lanțurilor de instrucțiuni reutilizabile (dependente sau independente) este necesară o vedere globală a întregului program, dificil de îndeplinit numai printr-o abordare la nivel hardware [Sas00, Con99, Sod00]. Literatura de specialitate enumeră câteva cercetări bazate pe tehnici hardware de identificare și exploatare a reutilizării de tip masiv (*coarse grain*): *reutilizarea basic-block-urilor* [Hua99] și respectiv a *trace-urilor de instrucțiuni* [Gon99]. Studiile cercetătorilor arată că, utilizând și suportul compilatorului pentru extinderea reutilizării dinamice la nivel de *basic-block* se obține o îmbunătățire substanțială a performanței procesoarelor (de la 1% la 14%, teste realizate pe benchmark-urile SPEC) [Hua99]. Dezavantajul în cazul acestei abordări constă în complexitatea schemei datorită numărului ridicat de intrări care trebuie reținute (valorile regiștrilor „în viață” în momentul intrării în *basic block*) dar și ieșirile care pot fi apoi folosite (reutilizate). Abordarea din [Con99] reprezintă o tehnică hibridă, prin care compilatorul identifică “regiunile” reutilizabile bazat pe consultarea unor informații de profil.

În general, tehnicile software urmăresc identificarea variabilelor de intrare invariante pentru anumite “regiuni de cod” (funcții, proceduri, “bucle de program”) și exploatarea acestui fenomen prin reutilizare. O abordare strict software [Ding04] presupune existența obligatorie a etapelor:

- Identificarea zonelor de cod candidate (*bucle, rutine* etc și eliminarea zonelor infrecvent executate) pentru reducerea costului de „*Value Profiling*”.
- Determinarea intrărilor și ieșirilor pe baza unei analize a fluxului de date (introducerea în cod a secțiunilor de tipul “*profiling code stubs*”).
- Calculul granularității pe baza informațiilor de profil privind frecvența de execuție și setul de valori care se repetă pentru fiecare zonă de cod selectată.

În [Ding04] bazat pe rata de repetiție a valorilor (*value profiling*), gradul de granularitate și de reutilizare al (secvențelor de) instrucțiunilor și complexitatea dispersiei instrucțiunilor reutilizate este propus un algoritm euristic prin care se estimează dacă procesul de căutare în tabele hardware (buffer-e de reutilizare) este mai *ieftin* decât reexecuția instrucțiunilor. Rezultatele raportate în urma îmbunătățirii compilatorului GNU gcc cu acest

algoritm și simulării benchmark-urilor din colecția Mediabench [Lee97] și pe jocul GNU *go* evidențiază nu numai o îmbunătățire a performanței ci și o reducere a puterii consumate, vitală mai ales la nivelul dispozitivelor de calcul de tip „*handheld*” [Ding04]. Dezavantajele abordărilor strict software îl reprezintă însă axarea principală pe informații de profil, care pot diferi în funcție de fișierele de intrare folosite sau estimările făcute în timpul analizei codului să nu fie cele mai corecte. În [Con00] se urmărește înlocuirea rolului compilatorului care determină prin „*value profiling*” regiunile cele mai reutilizabile cu un mecanism hardware de reutilizare selectivă. Câștigul constă într-o utilizare mai eficientă a resurselor hardware.

Un exemplu care justifică potențialul existent în programe, de reutilizare masivă (a *regiunilor* de instrucțiuni) îl reprezintă conținutul unei funcții care caută o adresă de întrerupere într-o tabelă. Exemplul este extras din benchmark-ul SPEC'95 – *m88ksim*, însă poate apărea în orice aplicație amplă care se respectă.

```

int GetBreakpoint (Address addr)
{
    BreakPoint *bp = breakPoints;
    for (int i = 0; i < n; i++, bp++) {
        if (bp->code && (bp->addr == addr))
            return bp;
    }
    return NULL;
}

```

Region

Figura 2.12. Reutilizare de tip coarse-grain

În figura 2.12, prin intermediul buclei *for* este căutată într-o tabelă de adrese de întrerupere (*breakPoints*) adresa *addr*. Presupunând că, conținutul tablei nu se modifică după faza de inițializare, rezultă că valoarea adresei *addr* este singura care se poate schimba de-a lungul multiplelor apeluri ale funcției *GetBreakpoint*. În consecință, toate instrucțiunile buclei exceptând *bp->addr == addr* sunt reutilizabile. Cu toate că, din punct de vedere static instrucțiunile reutilizabile sunt contigue, din punct de vedere dinamic secvența de instrucțiuni reutilizabilă este necontiguă, întreruptă de instrucțiunea de testare a egalității („*==*”).

În [Sas00] este definită noțiunea de „*regiune*” ca fiind un set de instrucțiuni dependente care pot fi simultan reutilizate, dar care nu trebuie să fie neapărat contigue (un subgraf, fără reacție inversă, al grafului dependențelor de date aferent trace-ului programului). Două regiuni

distincte, R_1 și respectiv R_2 se consideră a executa operații identice dacă acestea conțin un set identic de instrucțiuni și un set identic de dependențe de date (subgraf-urile dependențelor de date aferente sunt „izomorfe”). Algoritmul propus în [Sas00] pentru detecția regiunilor reutilizabile urmărește identificarea subgrafurilor identice din graful dependențelor de date aferent trace-ului programului.

Algoritmul dinamic de construire a regiunilor reutilizabile este de tip „greedy” datorită complexității ridicate a problemei găsirii unui subgraf de dependențe de lungime maximă și respectiv a constrângerilor legate de spațiu și mai ales timp, care apar în momentul implementării algoritmului. Întrucât faza de analiză nu poate fi făcută asupra întregului trace, algoritmul este divizat și aplicat la nivelul „ferestrelor de instrucțiuni active”. Cu cât aceasta are dimensiunea mai mare, un număr mai mare de regiuni poate fi identificat [Sas00].

Trebuie realizat un compromis între *un grad ridicat de reutilizare* și identificarea de „regiuni” cu număr mare de instrucțiuni reutilizate. Simulări laborioase [Sas00] pe benchmark-urile SPEC'95 au demonstrat că 29% din totalul instrucțiunilor dinamice reutilizate provin din regiuni de 8 sau mai multe instrucțiuni. Pentru acestea costul accesului la buffer-ul de reutilizare este mult inferior avantajului obținut prin bypassing-ul instrucțiunilor reutilizate, rezultând o îmbunătățire a performanței globale de procesare. De asemenea se remarcă că procentajul instrucțiunilor dinamice conținute în regiuni reutilizabile (*instruction coverage*) nu depinde semnificativ de *reuse threshold* – τ (necesitatea ca o regiune reutilizabilă să apară de τ ori în execuție).

O tehnică relativ recentă (propusă de Kavi la conferința ADCOM din decembrie 2003 ținută în India [Kavi03]) urmărește extinderea conceptului de reutilizare la un nivel de granularitate extrem de masiv: *reutilizarea dinamică a rezultatelor funcțiilor*, bazat pe observația că de multe ori o funcție este re-executată cu aceeași parametri (vezi concepte gen *recursivitate*, *progamare dinamică*). Optimizările compilatorului influențează însă semnificativ câștigul de performanță introdus de tehnica de reutilizare „la nivel de funcție”. Tehnicile de compilare încearcă prin metode statice eliminarea (sau diminuarea) calculului redundant (dinamic), reducerea presiunii asupra memoriei prin tehnici de alocare a variabilelor în regiștri (*register allocation*), ceea ce conduce la scăderea potențialului de reutilizare din programele de calcul. Pentru ca reutilizarea să fie eficientă este indicată realizarea unei analize în momentul compilării și o edificare asupra profilului de execuție al programelor. De asemenea, în exploatarea

tehnicii de reutilizare la nivel de funcție se impune tratarea următoarelor aspecte:

- ✓ determinarea influenței optimizărilor de compilare asupra câștigului de performanță.
- ✓ cunoașterea numărului de parametri al funcției (cu cât numărul este mai mic crește probabilitatea de a fi reutilizată respectiva funcție).
- ✓ influența conceptelor de programare obiectuală (moștenire, polimorfism) asupra reutilizării.
- ✓ realizarea unui studiu comparativ între câștigul de performanță obținut prin reutilizarea *la nivel de funcție* (dependent atât de frecvența de execuție a respectivei funcții cât și de numărul de instrucțiuni a respectivei funcții) versus reutilizarea *la nivel de instrucțiune* (lanț de instrucțiuni dependente).

Kavi [Kavi03] în urma a laborioase simulări pe benchmark-urile SPEC'95 – *m88ksim*, *go*, *vortex*, *jpeg* și *perl*, respectiv SPEC2000 – *197.perser*, *176.gcc*, precum și pe un program de test ce presupune calculul celui de-al n -lea termen din șirul lui Fibonacci, a concluzionat că potențialul de reutilizare este distribuit uniform de-a lungul tuturor fazelor unui program (inițializare, execuție, etc.). Simulările au fost efectuate folosind instrumentul ATOM [ATOM95], pe o arhitectură DEC/HP Alpha iar benchmark-urile au fost compilate folosind GNU gcc (versiunea 2.6.3) cu opțiunea de optimizare (-O3). De asemenea, rezultatele simulărilor au evidențiat că pentru 20% dintre cele mai apelate funcții se obține un potențial de reutilizare (invocare cu aceeași parametri de intrare) variabil între 10% și până la 67.7% cu implicații extrem de benefice asupra timpului de execuție al programelor și implicit asupra ratei globale de procesare. Tabelul următor (vezi tabelul 2.1), preluat din [Kavi03], ilustrează pentru valori diferite ale parametrului de intrare pe programul Fibonacci câștigul obținut din punct de vedere al timpului de execuție (reducerea cu cel puțin un ordin de mărime al acestuia).

Funcția	Timp execuție (fără reutilizare)	Timp execuție (cu reutilizare)	Total apeluri de funcție	Funcții apelate cel puțin odată cu aceeași parametri
Fib(10)	106.742	100.655	1.090	730
Fib(20)	3.611.713	263.757	135.290	87.938
Fib(30)	454.250.016	33.173.110	1.329.000	890.430
Fib(40)	53.933.197.903	3.940.000.000	127.278.861	86.167.788

Tabelul 2.1. Reutilizarea la nivel de funcție pe programul Fibonacci

Referitor la gradul de reutilizare al funcțiilor variind numărul de argumente, în [Kavi03] se arată că pentru funcțiile cu 0 sau 1 parametrii acesta este de 61%, pentru cele cu 2 parametrii procentul este de aproximativ 22% iar pentru cele cu 3 sau mai mulți parametrii procentul scade sub 10%.

2.2.2. PREDICȚIA DINAMICĂ A VALORILOR INSTRUCȚIUNILOR

2.2.2.1. VECINĂTATEA VALORII (VALUE LOCALITY) ȘI IMPLICAȚIILE ACESTEIA ÎN PREDICȚIE

Odată cu creșterea în complexitate a procesoarelor superscalare moderne (*ferestre de instrucțiuni* largi, *pipeline* cu un număr ridicat de niveluri) cantitatea de cod executată speculativ crește și ea. Predicția valorilor (VP) produse de instrucțiuni, o tehnică speculativă și relativ recent apărută (1996), a fost sugerată ca o modalitate de comprimare a căii critice de program în cadrul procesoarelor cu execuții multiple [Vin02]. Considerând un procesor cu regiștri generali pe 32 de biți, aparent, probabilitatea de a predicționa o valoare este de $1/2^{32}$, practic nulă. În realitate lucrurile stau cu totul altfel, având în vedere puternica vecinătate a valorilor asigurate unei resurse hardware și care determină ca valorile posibil a fi asigurate unei resurse să nu fie echiprobabile, ci dimpotrivă, localizate pe producător sau pe resursa hardware. Din acest motiv, predicția valorilor instrucțiunilor în vederea execuției speculative a acestora are șanse importante de reușită.

Vecinătatea (localitatea) **valorii** reprezintă o a treia dimensiune a conceptului de *localitate* (pe lângă cea *temporală* și respectiv *spațială*, frecvent întâlnite în programele de uz general), descriind probabilitatea statistică de referire a unei valori anterior folosite și stocată în aceeași locație de memorie sau registru. Conceptul de localitate a valorii a fost introdus în premieră în mod independent de 4 grupuri de cercetare: L. Widigen și E. Sowadsky de la firma AMD, F. Gabbay and A. Mendelsohn de la Universitatea Tehnion din Israel, M. Lipasti, C. Wilkerson, J. Shen de la Universitatea Carnegie Melon SUA [Lip96] și Y. Sazeides, J. Smith de la universitatea Wisconsin, SUA. Vecinătatea valorii este strâns legată de calculul redundant (repetarea execuției unei operații cu aceiași operanzi).

Diferența de esență între localitățile temporale și spațiale și respectiv localitatea valorilor constă în faptul că primele două sunt focalizate pe adrese, în timp ce ultima este centrată pe rezultatele produse. Mai precis, localitatea temporală se referă la probabilitatea ca o anumită **adresă** – conținând o “instrucțiune” sau o “dată” – să fie referită din nou în viitorul “apropiat”, în timp ce localitatea valorii se referă la faptul că **rezultatul** unei instrucțiuni care este din nou procesată, să se repete. Exploatarea localităților spațiale și temporale se face în principal prin sisteme ierarhizate de memorii cache care reduc latența memoriilor principale, în timp ce localitatea valorilor implică predicția acestora în vederea execuțiilor speculative a instrucțiunilor.

Localitatea valorilor este justificată de câteva observații empirice desprinse din programele de uz general, medii și sisteme de operare diverse:

- ☐ *Redundanța datelor* – seturile de intrări de date pentru programele de uz general suferă mici modificări (Exemple: matrici rare, fișiere text cu spații goale, celule libere în foi de calcul tabelar).
- ☐ *Verificarea erorilor* – tehnica LVP poate fi benefică în gestionarea tabelelor de erori ale compilatoarelor, în cazul apariției unor erori repetate.
- ☐ *Constante în program* – este mult mai eficient ca programele să încarce constante situate în structuri de date din memorie, ceea ce este exploatat favorabil de tehnica LVP.
- ☐ *Calculul adreselor instrucțiunilor de salt* – în situația instrucțiunilor *case* (*switch* în C) compilatorul trebuie să genereze cod care încarcă într-un registru adresa de bază pentru branch, care este o constantă (predicția adreselor destinație pentru instrucțiunile de salt).
- ☐ *Apelul funcțiilor virtuale* – în acest caz compilatorul trebuie să genereze cod care încarcă un pointer de funcție, care este o constantă în momentul rulării.

Localitatea valorii aferentă unor instrucțiuni Load statice dintr-un program, poate fi afectată semnificativ de optimizările compilatoarelor: *loop unrolling*, *software pipelining*, *tail replication* [Vin00a] etc., întrucât aceste optimizări creează instanțe multiple ale instrucțiunilor Load.

Convingerea că "*localitatea valorilor*" există, are la bază rezultate statistice obținute prin simulare la nivel de execuție a instrucțiunilor pe benchmark-uri SPEC'95, SPEC2000. Ca și metrică de evaluare, localitatea valorii pentru un benchmark este calculată ca raport dintre numărul de instrucțiuni Load *dinamice* care regăsesc o aceeași valoare în memorie ca și precedentele *k* accese și respectiv numărul total de instrucțiuni Load *dinamice* existente în benchmark-ul respectiv. O istorie de localizare pe *k* biți semnifică faptul că o instrucțiune Load verifică dacă valoarea citită din

memorie se regăsește printre ultimele k valori anterior încărcate. O problemă importantă de proiectare care se pune la ora actuală este: cât de "multă istorie" să fie folosită în predicție? Înainte de a răspunde însă la această întrebare poate ar trebui determinat *Câtă localitate a valorii exprimă programele și cum variază acest grad de localitate în funcție de istorie?* Pentru instrucțiunile de tip Load, cu o "adâncime" a istoriei de predicție de 1 (regăsirea aceleiași valori în resursa asignată ca și în cazul precedentului acces), programele de test exprimă o localitate a valorii de 50% în timp ce extinzând verificarea în spațiul valorilor aferente ultimelor 16 accese la memorie, se obține o localitate de 80% [Lip96]. Rezultatele subliniază că majoritatea instrucțiunilor Load statice aferente unui program exprimă o variație redusă a valorilor pe care le încarcă pe parcursul execuției. Compromisurile actuale oscilează între o istorie redusă – reprezentând o acuratețe de predicție joasă dar cost scăzut sau – o istorie bogată de predicție – acuratețe ridicată de predicție dar costuri și complexitate hardware ridicate. De asemenea, statistici bazate pe simulare arată că între 15 ÷ 45% dintre instrucțiuni produc o singură valoare în ultimele lor 16 instanțe succesive și respectiv între 28 ÷ 67% dintre instrucțiuni produc maximum 4 valori distincte în ultimele lor 16 instanțe dinamice de apariție [Wang97]. Informația este deosebit de prețioasă în tentativa de reducere a dimensiunii tabelelor de predicție și implicit a costului de implementare. Încercări în acest sens cu rezultate remarcabile atât din punct de vedere al predicției valorilor cât și din punct de vedere al fezabilității hardware, și care vor fi detaliate pe parcursul capitolului curent sunt predictorul adaptiv pe două niveluri [Wang97] și perceptronul introdus în premieră în predicția valorilor de [Tho04].

Exploatarea conceptului de localitate a valorilor se face prin tehnici de predicție a acestor valori. O altă problemă care se pune se referă la *Informația cu care se va adresa structura de predicție (adresa instrucțiunii sau adresa datei)* având implicații directe asupra *implementării la nivel de ciclu "pipe" a tehnicilor de predicția valorilor*. În abordarea propusă de Lipasti [Lip96] structura de predicție este indexată cu PC-ul instrucțiunii cu referire la memorie. În implementările proprii am extins acest lucru adresând structura de predicție pentru instrucțiunile Load și cu adresa datei. Evident că, în cadrul conceptului novator de predicție focalizată pe regiștrii procesorului, propus în capitolul 6.2 și publicat în [Vin05], adresarea structurii de predicție se face cu indicele registrului supus predicției.

În [Lep00] este abordată problematica localității valorilor în contextul predicției instrucțiunilor de tip Store, cu implicații deosebit de favorabile asupra performanței sistemelor uniprocessor dar mai ales multimicroprocesor (în special prin reducerea traficului prin rețeaua comună de interconectare).

Localitatea valorilor pentru instrucțiunile de tip Store s-a măsurat pe două niveluri: la nivelul instrucțiunii (PC) și respectiv la nivelul adresei memoriei de date. Rezultatele sunt optimiste, în ambele cazuri gradul de localitate fiind cuprins între 30% și 70%. Se definește chiar noțiunea de Store “*silencios*” pentru acele instrucțiuni care scriu în memorie o aceeași valoare ca și precedenta, deci care nu modifică starea sistemului. Măsurătorile arată că între 34% și 68% din instrucțiunile Store sunt silențioase. Aceste caracteristici pot fi exploatate practic, prin anularea execuției acestor instrucțiuni silențioase, cu beneficii asupra performanței sistemului de calcul. Prin înlăturarea acestor store-uri, fie static, în momentul compilării, fie dinamic, în momentul execuției se obține un câștig potențial atât din punct de vedere al dimensiunii codului, și evident a ratei de hit în cache-ul de instrucțiuni cât și din punct de vedere al timpului de execuție. De asemenea, se reduce presiunea asupra porturilor de scriere ale cache-ului de date, asupra cozii cu instrucțiuni store (*DataWriteBuffer*) și asupra traficului pe busul de date dintre procesor și memorie. Conceptul de evacuare “fără costuri” (*free*) a store-urilor silențioase [Lep00a] se bazează pe captarea porturilor libere de citire pentru faza de verificare (în avans) și utilizarea unui mecanism agresiv de tip *DataWriteBuffer* care exploatează localitatea spațială și temporală în vederea evacuării instrucțiunilor store.

Tehnica **Load Value Prediction (LVP)**, prima implementată de către cercetători [Lip96], predicționează rezultatele instrucțiunilor Load la expedierea spre unitățile funcționale de execuție exploatând corelația dintre adresele respectivelor instrucțiuni și valorile citite din memorie de către acestea, permițând deci instrucțiunilor Load să se execute înainte de calculul adresei și îmbunătățind astfel performanța. Conceptul de localizare a valorilor se referă practic la o **corelație dinamică** între numele unei resurse (registru, locație de memorie, port I/O) și valoarea stocată în acea resursă. Dacă memoriile cache convenționale se bazează pe localitatea temporală și spațială a datelor pentru a reduce timpul mediu de acces la memorie, tehnica LVP exploatează localitatea valorii prin predicția acesteia reducând atât timpul mediu de acces la memorie cât și necesarul de lărgime bandă al memoriei (se efectuează mai puține accese la memoria centrală), asigurând astfel un câștig de performanță considerabil. Toate aceste avantaje se obțin simultan cu reducerea considerabilă a presiunii asupra memoriilor cache. Ca și consecință a predicției valorilor se reduc și efectele defavorabile ale dependențelor RAW, prin reducerea așteptărilor instrucțiunilor dependente ulterioare. Dacă instrucțiunile predicționate se află pe calea critică a programului, execuția acestuia se comprimă în mod considerabil.

În figura 2.13 este descrisă implementarea tehnicii de predicția valorilor instrucțiunilor în structura pipeline a unei microarhitecturi

generale. La citirea unei instrucțiuni - din cache sau memoria centrală - cu cei mai puțin semnificativi biți ai adresei instrucțiunii (PC_{LOW}) se adresează în tabela de predicție (VPT). Întrucât avantajul predicției valorilor apare în momentul unei predicții corecte, în principiu structura VPT trebuie să includă pe lângă valorile care vor fi prezise și un mecanism de clasificare a predicției (încredere sau nu în valoarea care se prezice la un moment dat). Prin tratarea separată a fiecărui grup de instrucțiuni în parte este posibilă exploatarea avantajului maxim în fiecare caz: se poate evita costul unei predicții greșite prin identificarea instrucțiunilor nepredictibile sau se poate evita timpul necesar accesului la memorie (pentru instrucțiunile de tip Load) identificându-le și verificându-le pe cele puternic predictibile. Mecanismul respectiv, care poate fi implementat și sub forma unui numărator saturat pe doi biți (vezi figura 5.28), determină dacă va fi făcută sau nu o predicție (bazat pe gradul de încredere al automatului corespunzător), în timp ce VPT înaintează valoarea prezisă. Aceasta va fi preluată prin *bypassing* de către instrucțiunile dependente aflate în așteptare în stațiile de rezervare. La obținerea rezultatului real după faza de execuție (din cache dacă a fost instrucțiune Load), aceasta este comparată cu valoarea prezisă, instrucțiunile dependente executate speculativ fie urmează parcursul normal - nivelul *Write Back* al structurii *pipe* - fie sunt retrimise spre execuție, iar structura VPT și automatul de clasificare sunt actualizate în consecință. Statistici bazate pe simulare realizate pe sistemele *Power PC* și *Alpha AXP*, utilizând benchmark-urile SPEC'95 arată că peste 80% din instrucțiunile Load predictibile / nepredictibile sunt clasificate corect de către un numărator saturat pe doi biți (în concordanță cu simulările proprii din subcapitolul 7.2).

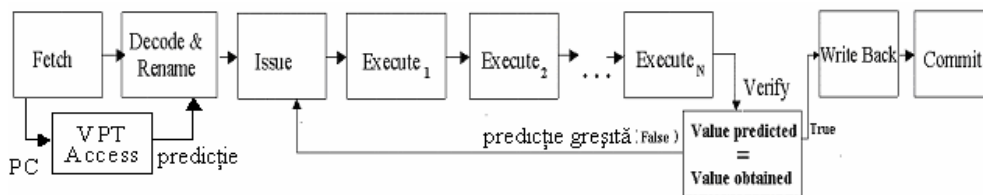


Figura 2.13. Pipeline cu VP

Disponibilitatea foarte devreme (la începutul fazei de aducere a instrucțiunii – Fetch) a indexului de accesare a tabelii de predicție VPT (accesul putând fi *pipeline*-izat peste două sau mai multe niveluri), complexitatea relativ redusă în proiectare și realizarea de tabele relativ mari fără a afecta perioada de tact a procesorului, sunt caracteristici care fac tehnica VP atractivă pentru proiectanții de viitoare microarhitecturi. Primele cercetări au arătat o creștere de performanță medie de cca. 6% datorată

implementării acestei tehnici. În subcapitolul 2.2.2.6 sunt prezentate informații cantitative și calitative care evidențiază câștigul de performanță obținut prin implementarea predicției valorilor dar și influența defavorabilă a lărgimii reduse de bandă a mecanismului de aducere a instrucțiunilor asupra eficacității VP. Pentru îmbunătățirea acurateții de predicție a valorilor au fost propuse câteva tehnici (vezi subcapitolele 2.2.2.2 ÷ 2.2.2.5). Acestea includ predictoare computaționale (ex. incrementale) respectiv contextuale (lucrează la nivel de o singură instrucțiune și încearcă să prezică viitoarea valoare care va fi produsă de instrucțiune bazându-se pe valorile anterior generate). Întrucât respectivele scheme încearcă să înmagazineze o cât mai mare istorie pentru predicție, aceasta implică tabele hardware de mari dimensiuni și costuri ridicate de implementare.

Predicția și consecința ei, execuția speculativă, vor fi procese esențiale în cadrul microarhitecturilor viitoare. Cercetările din domeniul microarhitecturilor cu paralelism la nivel de instrucțiuni s-au concentrat până de curând în vederea depășirii hazardurilor structurale, a dependențelor de date de tip WAR (*Write After Read*) și WAW (*Write After Write*) și respectiv a limitărilor impuse de către instrucțiunile de ramificație (*branch*). Acestea din urmă se rezolvă în principal prin predicție dinamică, proces care asigură execuția speculativă a instrucțiunilor prin paralelizarea unor *basic-block*-uri distincte. Dependențele de date de tip RAW (*Read After Write*), erau considerate ca reprezentând o limitare fundamentală, ce nu poate fi depășită, datorată unei secvențialități intrinseci a programului și care reducea dramatic paralelismul. Graful dependențelor de date asociat unui program, constituia – prin calea sa critică – o barieră de netrecut în calea procesării paralele. Predicția dinamică a valorilor instrucțiunilor reprezintă o tehnică relativ recentă, care permite execuția speculativă a instrucțiunilor dependente RAW, prin predicția rezultatelor acestora, reducându-se astfel în mod semnificativ latența de execuție a căii critice a programului.

O similitudine tot mai puternică se remarcă între problema predicției valorilor și problema predicției adreselor destinație aferente instrucțiunilor de salt indirect. Liantul dintre cele două tehnici de predicție l-ar putea constitui aflarea legăturii calitative și cantitative care există între paradigmele actuale de programare (procedurală și obiectuală) și respectiv generarea valorilor de către anumite tipuri de instrucțiuni. Mai precis, investigarea legăturii existente între moștenire, polimorfism și alocarea dinamică a memoriei pe de o parte, și comportamentul salturilor indirecte. Structurile de date implementate în hardware pentru ambele procese de predicție, au același principiu de funcționare și anume: asocierea cvasibijectivă a contextului de apariție al instrucțiunii respective (ALU, Load sau Branch) cu registrul / data / adresa de predicționat, în mod

dinamic, odată cu procesarea programului. Se poate constata că problematica predicției în microprocesoarele avansate, tinde să devină una generală și ca urmare implementată pe baza unor principii teoretice avansate și mai generale. Scopul principal și imediat este execuția speculativă agresivă a instrucțiunilor, cu beneficii evidente în creșterea gradului mediu de paralelism. O dovadă în acest sens îl reprezintă migrarea ideilor de predictor markovian și neural de tip Perceptron de la nivelul salturilor [Chen96, Jim02] (vezi subcapitolul 5.1.2) la nivelul predicției valorilor instrucțiunilor și regiștrilor [Tho04, Seng04].

2.2.2.2. PREDICTOARE COMPUTAȚIONALE

Una dintre potențialele dificultăți în exploatarea predictibilității valorilor, o constituie alegerea tipului potrivit de predictor pentru o anumită instrucțiune. În [Saz97] se încearcă o clasificare empirică a celor mai frecvente tipuri de secvențe de valori generate de către programele cele mai uzuale. Se disting astfel cel puțin 3 categorii primitive de secvențe de valori și anume: secvențe constante, secvențe incrementale (*Stride*) și respectiv non – incrementale. Secvențele constante (5, 5, 5, ...) sunt cele mai simple și rezultă în urma execuției unor instrucțiuni care produc în mod repetat același rezultat. Apar frecvent în programe. Secvențele incrementale (1, 4, 7, 10, ...) se caracterizează prin faptul că diferența între două elemente succesive este o constantă numită și pas. Pasul poate fi pozitiv sau negativ. Rezultă imediat că o secvență constantă este o secvență incrementală cu pasul 0. Astfel de secvențe incrementale pot apărea în cazul contorilor/indecșilor buclelor de program sau în cazul accesării “regulate” a unor structuri de date de tip șir, matrice etc. Secvențele non – incrementale reprezintă orice altă secvență de valori în afara celor constante ori incrementale (ex. 28, 13, 99, 107, 23, 456 ...). Spre exemplu, traversarea unei liste înlănțuite ar putea genera relativ frecvent valori de adrese care să fie încadrabile în această categorie. Desigur că pot exista secvențe de valori hibride, obținute prin combinarea acestor 3 tipuri de *pattern*-uri. De asemenea s-ar putea considera și secvențe de valori repetitive (incrementale – 1, 3, 5, 1, 3, 5, ... respectiv non – incrementale – 1, 13, 88, 7, 1, 13, 99, 7 ...).

Aceste tipuri frecvente de secvențe de valori conduc la cel puțin 2 categorii primitive de predictoare de valori: computaționale și respectiv bazate pe context. Predictoarele computaționale generează predicția pe baza istoriei valorilor memorate, prelucrându-le după un anumit algoritm stabilit. Un predictor incremental ori unul care predicționează valoarea unui șir

recurent, se încadrează în categoria celor computaționale. De observat că, spre deosebire de predicția contextuală, predicția computațională nu provine strict din definiția localității valorilor. Predictoarele bazate pe context predicționează valoarea următoare corespunzătoare unui anumit context particular memorat. Printr-un context se înțelege o secvență finită de valori cu apariție repetată. Aceste predictoare bazate pe context permit predicția oricărei secvențe repetitive de valori, incrementale sau non – incrementale. Un predictor contextual este de ordinul k dacă informația sa de context include ultimele k valori iar căutările se fac cu acest *pattern* de k valori. Principala limitare a acestor predictoare constă în faptul că pentru a predicționa corect o anumită valoare, aceasta trebuie să fi urmat același context cel puțin o dată. În subcapitolul curent vor fi descrise două structuri computaționale de predicție (în ordinea în care au fost propuse în literatura de specialitate – *LastValue* [Lip96] și respectiv *Incremental* [Lip96b, Gab98]). Din clasa predictoarelor contextuale sunt prezentate un predictor de tip PPM complet de ordin k compus din $(k+1)$ predictoare markoviene [Saz97] și un predictor adaptiv pe două niveluri [Wang97], ca un caz particular de predictor contextual (vezi subcapitolul 2.2.2.3). Datorită imposibilității de exploatare optimă cu un singur tip de predictor a patternurilor de predictibilitate existente în programe, în subcapitolul 2.2.2.4 s-a propus un predictor hibrid compus dintr-un predictor adaptiv pe două niveluri și un predictor incremental. Fiecare din anterior enunțatele structuri de predicție și dedicate diverselor resurse (instrucțiuni, locații de memorie, regiștrii procesorului MIPS) au fost implementate în cadrul cercetărilor proprii din capitolul 6 (vezi și [Flo02, Vin04, Vin05]). În finalul subcapitolului 2.2.2.5 sunt prezentate două abordări novatoare, extrem de recente [Tho04, Seng04], bazate pe utilizarea în premieră a *perceptronului* în predicția valorilor instrucțiunilor și regiștrilor.

A. Predictoare “*Last Value*”

Un tip relativ întâlnit de predictoare computaționale sunt așa numitele predictoare “*last value*”, caracterizate prin faptul că predicționează noua valoare ca fiind aceeași cu ultima valoare produsă de către instrucțiunea respectivă. Totuși există și variante care schimbă strategia de modificare a valorii, bazat pe histerezis. Un exemplu de mecanism de histerezis constă într-un numărator saturat asociat fiecărei intrări în tabela de predicție. Acesta este incrementat/decrementat atunci când predicția este corectă/incorectă, respectiv. Valoarea memorată în tabelă este evacuată numai când valoarea indicată de către număratorul asociat este sub un anumit prag prestabilit. Un alt mecanism de histerezis nu modifică valoarea

predicționată din tabelă până când noua valoare nu a apărut, în mod repetitiv, de un anumit număr de ori.

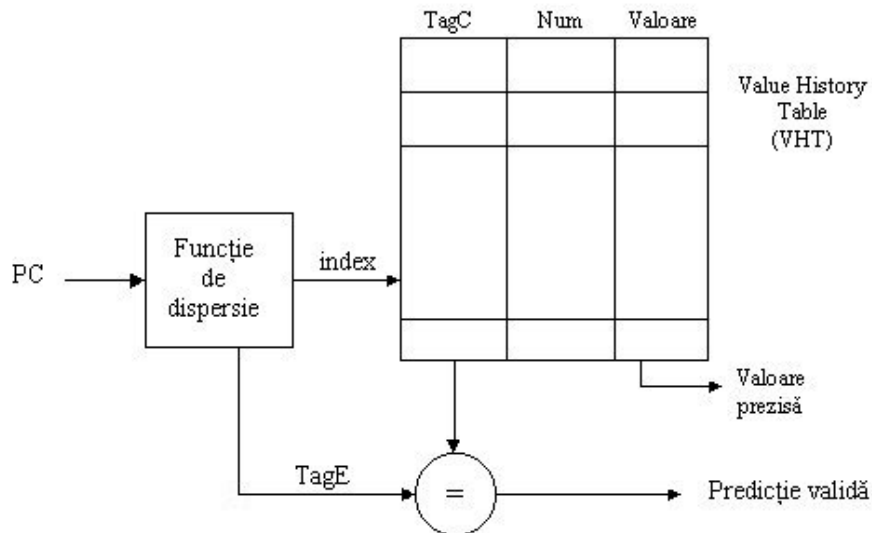


Figura 2.14. Schemă bloc pentru un predictor de tip "last value"

În schema predictorului de tip "last value", câmpul "Num" implementează mecanismul de histerzis printr-un numărator, în concordanță cu algoritmul de modificare a valorii care a fost implementat. Funcția de dispersie determină un index de accesare a tabelii VHT, pe post de adresă, printr-un mecanism de comprimare a intrării. Desigur că VHT are o capacitate limitată, în principal funcție de tehnologia de integrare folosită.

În [Lip96] se arată că acuratețea predicției obținută printr-o astfel de schemă de tip "last value" este în medie de 49%, măsurat pe o serie de benchmark-uri SPEC. Considerând că se memorează ultimele 4 valori produse de către o anumită instrucțiune și că abilitatea predictorului de a alege valoarea corectă este perfectă, s-a obținut o acuratețe de predicție medie de 61%.

B. Predictoare incrementale

Predictoarele incrementale reprezintă o generalizare a predictorului de tip *Last Value* (echivalent cu un predictor incremental cu pas 0). În acest caz, considerând că v_{n-1} și v_{n-2} sunt cele mai recente valori produse, noua valoare v_n va fi calculată după formula de recurență: $v_n = v_{n-1} + (v_{n-1} - v_{n-2})$,

unde $(v_{n-1} - v_{n-2})$ este pasul secvenței. Pasul poate fi și variabil, nu neapărat constant tot timpul ci constant doar pe anumite intervale de timp (de exemplu: iterațiile de parcurgere a unei matrici bidimensionale). În această idee se propun și în acest caz scheme de actualizare a pasului bazate pe histerezis. Astfel, pasul memorat în tabelele de predicție este modificat numai atunci când numărătorul saturat asociat, memorează o valoare situată peste un prag stabilit. Firește, acest numărător este incrementat/decrementat în cazul unei predicții corecte/incorecte, respectiv. O altă strategie cu histerezis este așa numita “2 - delta” (vezi figura 2.17). În cazul acesteia sunt memorați doi pași (s_1 și s_2), $s_1 = v_n - v_{n-1}$. Pasul s_2 este cel folosit în procesul de predicție. Numai atunci când aceiași valoare s_1 a apărut de două ori consecutiv, se face transferul $s_2 \leftarrow s_1$. Ambele metode cu histerezis reduc numărul predicțiilor eronate de la două, la doar una, în cazul secvențelor incrementale repetitive. În figura următoare (figura 2.15) se prezintă o structură tipică de predictor incremental (pas constant) iar figura 2.17 ilustrează un predictor incremental de tip “2-delta”.

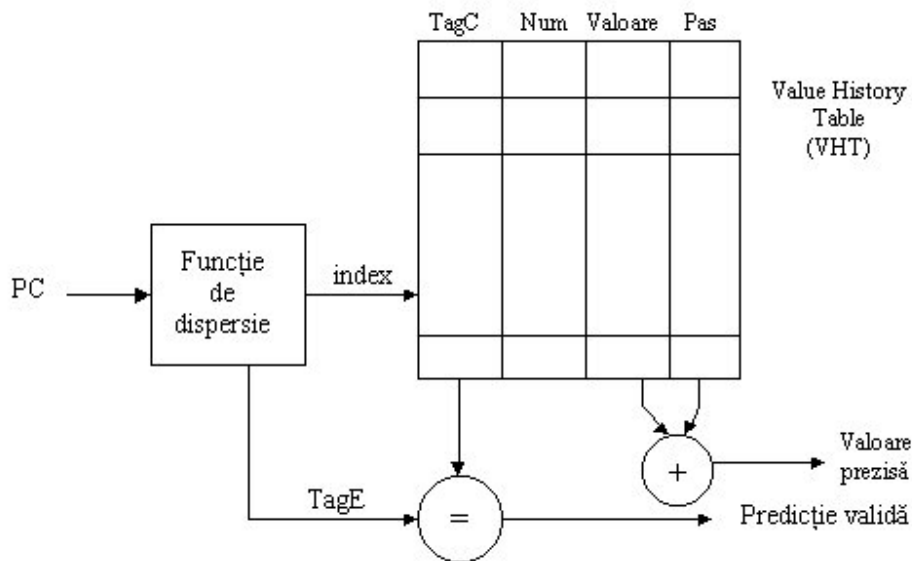


Figura 2.15. Structură de predicție incrementală

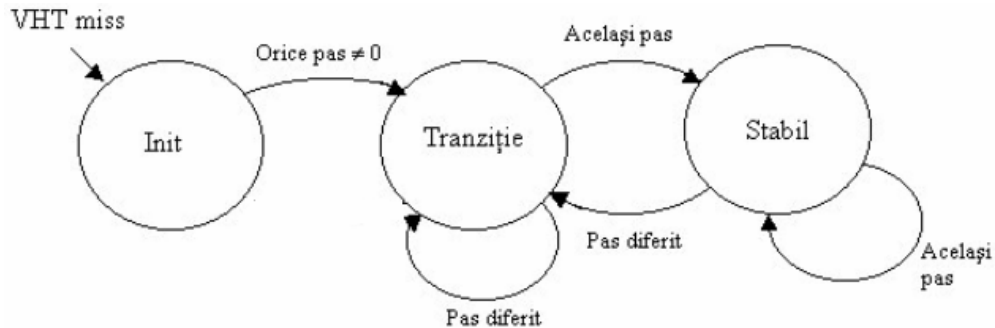


Figura 2.16. Automatul de stare asociat

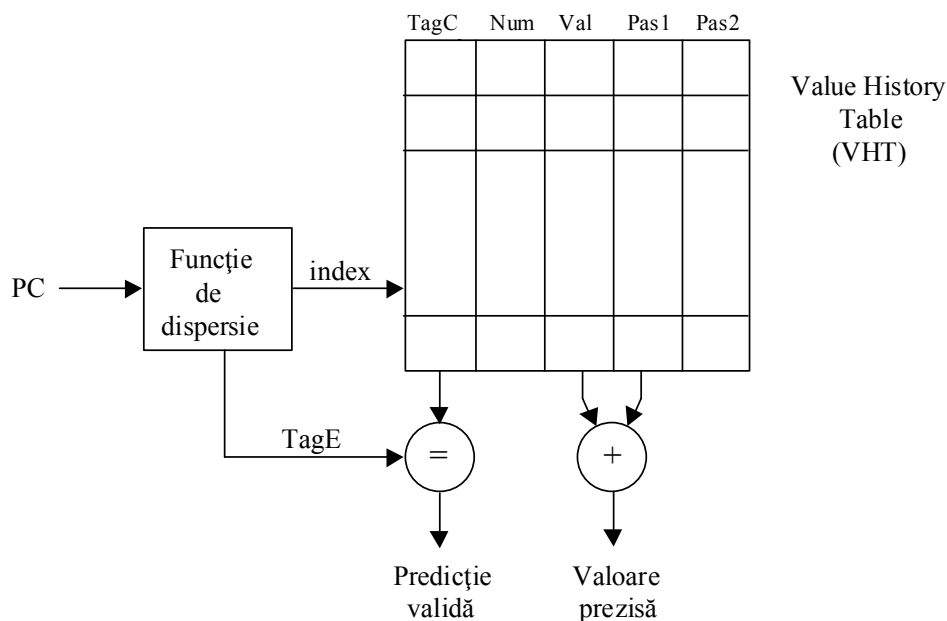


Figura 2.17. Structura predictorului incremental de tip “2-delta”

O acțiune importantă în cadrul unui predictor incremental este constituită de detecția pasului. În continuare este descris modul de funcționare al predictorului incremental de tip 2-delta și al automatului de stare asociat (câmpul *Num*). Prima dată când o instrucțiune va fi procesată, va rezulta un *miss* în tabela de predicție VHT și evident că nu se va face nici o predicție. Când o instrucțiune produce un rezultat atunci: (1) rezultatul este memorat în câmpul “Val” din VHT și (2) automatul trece în starea inițială “Init” (vezi figura 2.16). Atât timp cât eventuale următoare instanțe succesive ale aceleiași instrucțiuni produc același rezultat ($\text{pas}=0$),

automatul va rămâne în starea inițială. Dacă însă o instanță următoare a instrucțiunii produce un rezultat (R_1) diferit de precedentul, atunci se calculează pasul $S_1=R_1-Val \rightarrow VHT$ și $R_1 \rightarrow VHT$. Totodată automatul trece în starea intermediară “Tranziție”. Dacă în această stare apare o nouă instanță dinamică a instrucțiunii, nu se va genera nici o predicție. În schimb $S_2=R_2-Val \rightarrow VHT$ și $R_2 \rightarrow VHT$. Dacă $S_1 = S_2$ atunci automatul trece în starea “Stabil”, altfel rămâne în starea “Tranziție”. În starea stabil, valoarea prezisă = $R_2 + S_2$. Cu alte cuvinte, acest automat de stare implementează un anumit “grad de încredere” asociat predicției. Predicția nu se face decât în cazul în care pragul de încredere (*Confidence Threshold*) depășește o anumită valoare, apriori determinată.

Schemele anterior prezentate nu sunt singurele predictoare computaționale existente. Pot fi implementate și alte variații pe această temă, alegerea soluției optime fiind determinată prin simulare. Pentru o acuratețe ridicată a predicției trebuie anticipat tipul calculului pe care programul îl execută, în caz contrar rolul predictoarelor ar fi total inefficient. Realizarea acestui lucru se poate face doar în urma unei analize a caracteristicilor de profil a programului sursă din limbajul de nivel înalt [Vin02]. Spre exemplu, pentru o buclă care calculează un șir recurent liniar omogen de ordinul 2 ($v_n = \alpha v_{n-1} + \beta v_{n-2}$), ar trebui activat un predictor echivalent, special croit pentru acest scop. Acest fapt conduce la o limitare fundamentală a procesului de predicție a valorilor prin predictoare computaționale. Soluția care se impune în acest caz o reprezintă implementarea unui predictor hibrid.

2.2.2.3. PREDICTOARE CONTEXTUALE

În cazul predicției contextuale valoarea următoare se determină pe baza unui context anterior înregistrat și a valorilor ce au urmat imediat acestui context în istoria memorată. O clasă importantă a predictoarelor contextuale sunt cele care implementează algoritmul “*Prediction by Partial Matching*” (PPM), care conține un set de predictoare markoviene ca în figura 5.26. Ideea originală aparține lui *Trevor Mudge* cel care a folosit-o pentru prima dată în scopul dezvoltării unor predictoare de branch-uri care generalizează predictoarele de tip adaptiv pe două niveluri (*Two Level Adaptive Branch Predictors*) și care sunt prezentate în subcapitolul 5.1.1 [Mud96].

Valoarea predicționată este aceea care a urmat cu cea mai mare frecvență contextului considerat. După cum se observă în figura 5.26, ea este funcție și de contextul considerat, un context mai “bogat” (“lung”)

conducând adeseori la o acuratețe mai ridicată a predicției (nu întotdeauna însă: câteodată contextul se poate comporta ca “zgomot” – fapt ce poate fi observat și din rezultatele grafice obținute în urma simulărilor proprii din subcapitolul 7). Ca și în cazul predicției branch-urilor și aici un predictor PPM complet de ordin N trebuie să conțină $N+1$ predictoare *Markov*, de la ordinul 0 până la N . Dacă predictorul *Markov* de ordinul N produce o predicție (context găsit în secvența de valori) procesul se termină, dacă nu atunci se activează predictorul *Markov* de ordinul $(N-1)$ ș.a.m.d. Un exemplu de predictor Markov de ordin k , utilizat în predicția target-urilor salturilor indirecte este ilustrat în figura 5.27. La nivelul tehnologiei actuale, implementarea unui asemenea predictor PPM complet ar putea fi prohibită.

Bazat pe clasificarea propusă, Sazeides [Saz97] introduce două caracteristici importante în înțelegerea comportamentului fiecărui tip de predictor anterior prezentat. Prima, este perioada de învățare (*Learning Time* - LT), care reprezintă numărul de valori din secvența de intrare, generate înaintea primei predicții corecte. A 2-a este dată de gradul de învățare sau acuratețea predicției (*Learning Degree* - LD), care reprezintă procentajul de predicții corecte generate după perioada de învățare a predictorului. Pentru *secvențele constante* gradul de învățare devine 100% după un timp de învățare unitar pentru predictoarele computaționale și respectiv după un timp de învățare egal cu ordinul predictorului PPM (O) în cazul folosirii celui din urmă. O *secvență incrementală* este prezisă cu acuratețe de 100% doar de către un predictor incremental (cu pas constant) în timp ce *secvențele non-incrementale fără repetiție periodică* nu pot fi predicționate de nici una din structurile de predicție anterior prezentate. Practic orice secvență non-incrementale fără repetiție și generabilă printr-un algoritm determinist, poate fi exploatată printr-un predictor particular care să o poată predicționa satisfăcător. Provocarea cercetătorilor constă în acest caz în construirea unui predictor general care să predicționeze cu succes clase cât mai largi din categoria acestor secvențe de instrucțiuni. O propunere enunțată doar și nu rezolvată în [Vin02] se referă la implementarea unui **predictor semantic** care să utilizeze informații relative la structurile programelor HLL (*High Level Languages*) procesate. În acest scop, compilatorul ar trebui să descopere și să transmită predictorului, prin intermediul “codului obiect” generat, aceste informații de semantică structurală a aplicației HLL. Prin predictoarele actuale, această informație semantică se pierde complet în momentul predicției. O *secvență incrementală cu repetiție periodică* este prezisă cu o acuratețe de $(T-1)/T$ (unde T reprezintă perioada unei secvențe de valori cu repetiție) după același timp de învățare ca și în cazul unei secvențe incrementale fără repetiție ($LT=2$). Un predictor PPM obține o acuratețe de 100% după un

timp de învățare egal cu $T+O$. O *secvență de valori non-incrementală dar periodică* nu este predicționată corect decât de către predictorul PPM complet. Acuratețea de 100% este dobândită tot după perioada de învățare mai mare decât în cazul predictorilor incrementale ($LD=T+O$).

În figura 2.18 se prezintă schema bloc a unui predictor contextual de valori de tip PPM. Selecția atât a intrării corespundente din tabela de predicție cât și a contextului se face cu adresa instrucțiunii în faza de aducere a acesteia. Fiecare intrare din tabela de predicție VHT are asociat câte un număr saturat (*Num*). Acesta este incrementat atunci când predicția este corectă, respectiv decrementat atunci când predicția este incorectă. În câmpurile V_1, V_2, \dots, V_4 se memorează ultimele patru valori pentru fiecare instrucțiune din tabela VHT. Dacă automatul se află în starea *predictibil*, pe baza secvenței de valori memorate se generează predicția, valoarea prezisă fiind aceea care a urmat cu cea mai mare frecvență contextului. Blocul PPM reține contextul – o istorie comprimată prin intermediul unei funcții de dispersie, a ultimelor valori generate de către respectiva instrucțiune, și eventual și alte informații considerate relevante și determină care valoare s-a repetat de cele mai multe ori după pattern-ul format din ultimele cele mai recente patru valori (V_1, V_2, V_3, V_4), în istoria de valori avute la dispoziție. O implementare simplificată și fezabilă tehnologică a predictorului contextual de tip PPM o constituie predictorul adaptiv pe două niveluri.

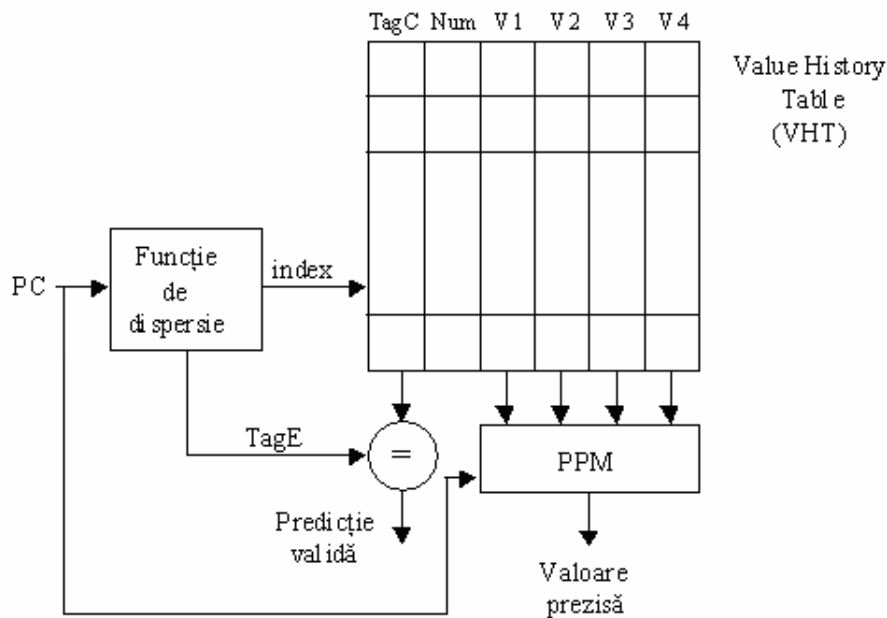


Figura 2.18 Schema generică a unui predictor contextual de tip PPM

Predictoare adaptive pe 2 niveluri

În [Wang97] se propune, prin analogie cu predictoarele de branch-uri de tip “*Two Level Adaptive*”, un predictor de valori adaptiv pe două niveluri. În esență, o astfel de schemă va memora pentru fiecare instrucțiune, ultimele 4 valori produse. Un mecanism de selecție va determina alegerea uneia dintre aceste valori ca fiind valoarea prezisă. În figura 2.19, se prezintă principiul unei astfel de scheme de predicție a valorilor.

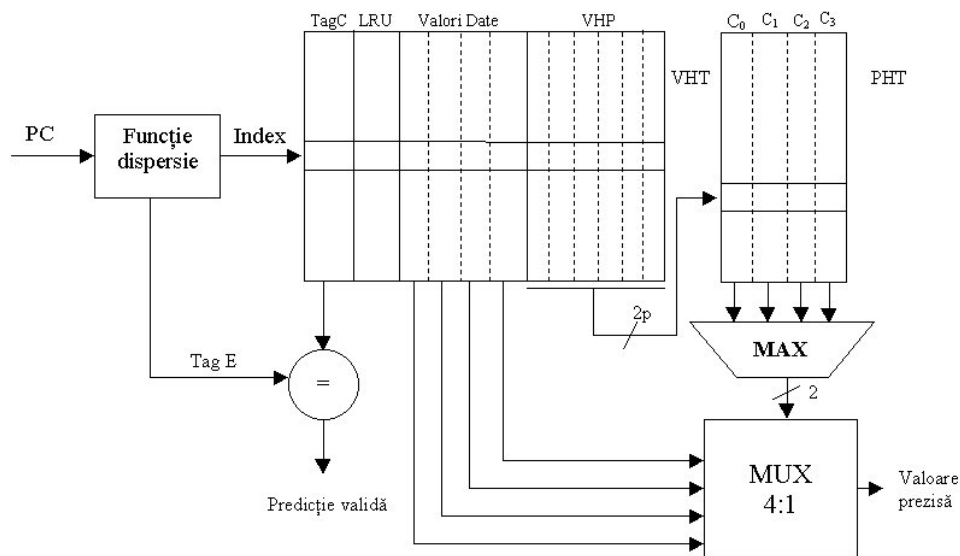


Figura 2.19 Schema bloc a unui predictor de valori pe 2 niveluri

Câmpul “Valori Date” memorează ultimele 4 valori produse de către respectiva instrucțiune. Aceste 4 valori sunt codificate binar {00, 01, 10, 11}. Când o instrucțiune produce o valoare aparținând mulțimii “Valori Date”, una dintre aceste valori va fi (eventual) selectată spre a fi prezisă. Dacă o instanță a instrucțiunii produce o valoare nouă, aceasta se introduce în tabela VHT în detrimentul, datei cel mai de demult nepredicționate. Acest fapt se stabilește cu ajutorul informației din câmpul LRU (*Least Recently Used*) care teoretic poate conține câte un contor LRU pentru fiecare dintre cele 4 date memorate. Câmpul VHP (*Value History Pattern*) din tabela VHT, memorează un *pattern* binar pe $2p$ biți care semnifică ultimele p rezultate generate de către instrucțiunea respectivă. Dacă vreuna din ultimele p instanțe generează o altă valoare decât una din cele 4 memorate,

respectiva valoare va fi înlocuită în context cu cea mai recent anterior generată valoare din șirul celor 4, întrucât predictoarele contextuale predicționează corect o anumită valoare, doar dacă aceasta a urmat același context cel puțin o dată. Se reamintește că un rezultat al unei instrucțiuni este codificat binar pe 2 biți. Acest câmp VHP adresează al doilea nivel de memorie (PHT – *Pattern History Table*). O linie PHT conține 4 numărătoare (C_0, C_1, C_2, C_3) saturate independente. Circuitul MAX din figură lucrează astfel: dacă $MAX(C_0, C_1, C_2, C_3) = C_K$ și dacă $C_K \geq A$, unde A este o valoare de prag prestabilită, atunci circuitul MAX generează la ieșire codificarea binară a lui K , pe doi biți. Dacă însă $C_K < A$, atunci nu se face nici o predicție considerându-se că valoarea C_K asociată *pattern*-ului VHP respectiv nu a apărut suficient de frecvent. Dacă valoarea asociată unui numărător C_K din linia PHT selectată este produsă de către instrucțiunea în curs, atunci numărătorul C_K este incrementat (+3) iar celelalte 3 numărătoare din linia respectivă sunt decrementate (-1). Se poate observa astfel că, prin codificarea binară a ultimelor 4 valori produse, indexul VHP reprezintă un “context comprimat” al CPU.

Informația VHP reprezintă astfel un mod ingenios de dispersie a câmpului valorilor de date (*hashing*), în scopul compresiei acestora și deci, a reducerii capacității tabelii PHT. Evident că există posibilitatea unor interferențe nedorite în tabela PHT. O soluție interesantă de reducere a proceselor de interferență cu mari beneficii asupra acurateții predicției valorilor este dată în [Des02] unde se propune utilizarea a încă unei funcții de dispersie, independentă de prima (*hash2*). Referitor la setul de numărătoare $C_0 - C_3$, acestea reprezintă practic un grad de încredere asociat predicției. Dacă acest grad este sub o anumită valoare prestabilită, se renunță la predicție, cu beneficii asupra performanței globale (în cazul unei predicții greșite e necesară refacerea contextului procesorului și reluarea unei secvențe de instrucțiuni, ceea ce induce penalități). Firește, există o multitudine de posibilități de calcul al gradului de încredere asociat predicției (*tag*-uri, numărătoare saturate, recunoaștere de „*pattern-uri*” folosind numărătoare nesaturate – procese Markov) [Des02].

În acest moment se înțelege de ce schema predictorului adaptiv pe două niveluri (vezi figura 2.19) poate reprezenta o implementare simplificată și fezabilă tehnologic a predictorului contextual de tip PPM (figura 2.18). În fond, fiecărei instrucțiuni memorate în VHT i se asociază un context reprezentând practic secvența ultimelor p valori produse de către aceasta (*pattern*-ul VHP). În cadrul acestui context se calculează prin intermediul numărătoarelor $C_0 - C_3$, valoarea cea mai frecvent generată de către respectiva instrucțiune în istoria ultimelor sale p instanțe. Dacă această

valoare a apărut suficient de frecvent, ea este predicționată ca fiind valoarea următoare.

2.2.2.4. PREDICTOARE HIBRIDE

Una dintre potențialele dificultăți în exploatarea predictibilității valorilor, o constituie alegerea tipului potrivit de predictor pentru o anumită instrucțiune. Cercetările arată în mod clar că un singur tip de predictor (computațional, contextual etc.) nu dă în general rezultatele cele mai bune. Este evident că anumite tipuri de secvențe de valori generate prin program sunt prezise mai bine de un anumit predictor, iar altele, de către un alt tip de predictor particular. Din acest motiv a apărut firească ideea predicției hibride, adică două sau mai multe predictoare de valori să conlucreze în mod dinamic în procesul de predicție. Spre exemplificare, în [Wang97] se prezintă un interesant predictor hibrid compus dintr-un predictor pe 2 niveluri și respectiv un predictor incremental (vezi figura 2.20).

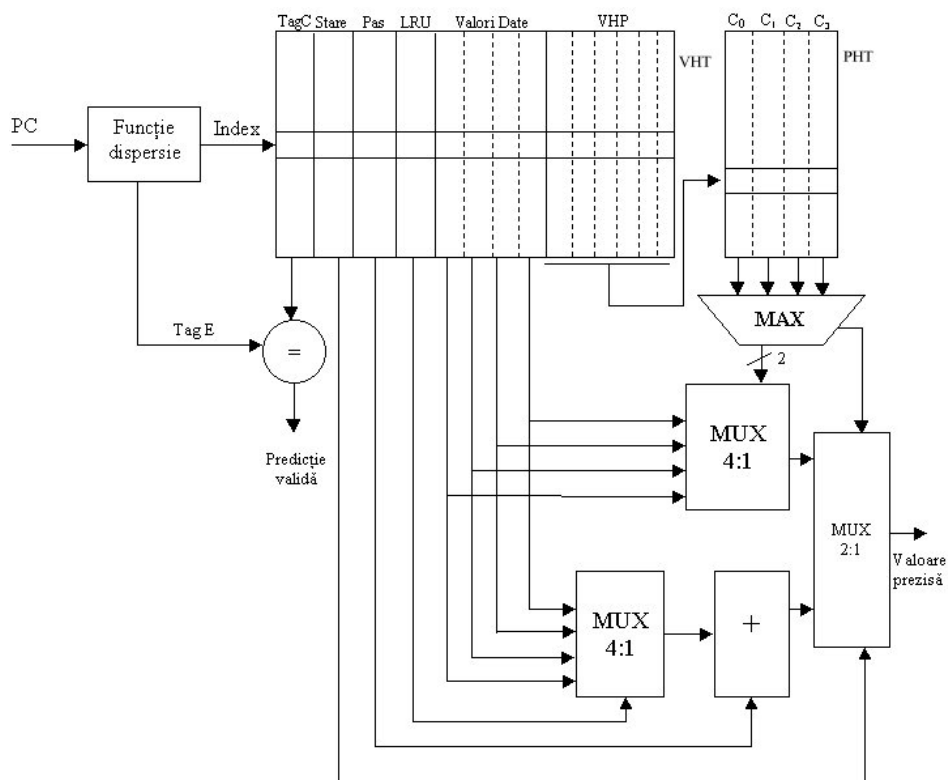


Figura 2.20 Schema bloc a unui predictor hibrid (2 – level, incremental)

Ideea de bază este simplă: dacă predictorul pe 2 niveluri generează o predicție, aceasta este cea generată de către predictorul hibrid, în caz contrar, se selectează valoarea generată de către predictorul incremental (dacă acesta generează vreuna). Spre exemplu, măsurat pe benchmark-ul *Eqntott* din seria SPEC'92, un predictor incremental a obținut o acuratețe medie de cca. 63%, un predictor pe 2 niveluri cu valoarea de prag $A=6$ a obținut cca. 67% iar predictorul hibrid corespunzător a obținut o acuratețe de cca. 81%, observându-se astfel foarte clar avantajul schemei hibride [Wang97]. Totuși, o politică mai rafinată bazată pe gradul de încredere cel mai mare, corespunzător predictoarelor componente, ar fi de dorit în locul acestei prioritizări simpliste, neadaptive. În acest sens, în cadrul cercetărilor proprii privitoare la predicția valorilor centrată pe regiștrii procesorului, pe lângă implementarea unui predictor hibrid cu selecție bazată pe prioritizare statică au fost propuse trei scheme de *metapredicție*, două statice (non-adaptive) bazate pe confidențe și una dinamică (adaptivă), în care selecția predictorului folosit se face printr-o rețea neurală de tip *MultiLayerPerceptron* (vezi subcapitolul 6.2).

Zhao [Zha00] propune o schemă de clasificare statică cu ajutorul compilatorului care grupează toate instrucțiunile unui program în câteva clase, fiecareia fiindu-i asociată un tip de predictibilitate a valorii (*pattern*). Acest *pattern* de predictibilitate este codificat în corpul instrucțiunii pentru a identifica tipul de predictor care se potrivește cel mai bine pentru predicția valorilor care sunt probabil a fi produse de fiecare instrucțiune în timpul execuției sale. Rezultatele simulărilor arată că prin această abordare se poate reduce substanțial numărul de porturi de citire/scriere ale predictorului pentru un anumit nivel de performanță. O altă concluzie ce se desprinde vizează legătura dintre vecinătatea valorii și structurile de la nivelul programului sursă pentru o mai bună exploatare a vecinătății prin predicția valorilor. Pe de altă parte, la implementarea predictorului de valori într-un procesor superscalar trebuie ținut cont și de următorul aspect: *întrucât mai multe instrucțiuni doresc să acceseze simultan predictorul în fiecare ciclu acesta trebuie să fie multiport* (să permită canale largi de citiri și scrieri).

Măsurători efectuate pe benchmark-urile SPEC'95 (programe de numere întregi) arată că un predictor de valori simplu [Zha00] trebuie să suporte în medie 5 accese per ciclu pentru un procesor cu factorul superscalar de 8, crescând la 7 accese per ciclu pentru un procesor care lansează 16 instrucțiuni simultan în execuție (vezi tabelul 2.2. Dacă predictorul de valori nu poate suporta lărgimea de bandă cerută, predicția valorilor nu poate fi aplicată tuturor instrucțiunilor din „*instruction window*”, și atunci nu toate posibilele dependențe RAW pot fi înlăturate, cu

repercusiuni negative asupra ratei de procesare. Mai mult, intrările predictorului nu vor fi actualizate cu noile valori generate la timp pentru următoarea cerere de predicție, determinând o creștere corespunzătoare a ratei de miss a predicției.

issue width	benchmarks								
	go	m88ksim	gcc	compress	li	jpeg	perl	vortex	AVG
8 issue	5.2	5.5	4.6	5.2	4.5	6.1	4.7	4.2	5.0
16 issue	6.1	10.7	6.3	6.7	6.3	11.0	6.1	6.0	7.0

Tabelul 2.2. Numărul mediu de cereri per ciclu adresate unui “predictor de valori simplu” într-un procesor superscalar

O abordare menită să rezolve necesarul de lărgime de bandă în cadrul tehnicii de predicția valorilor constă în creșterea numărului de porturi de citire/scriere aferente unui predictor de valori, cu repercusiuni negative însă asupra complexității de proiectare, densității de cablare, practic a costului de implementare. O altă soluție propusă de cercetători [Lee99] se bazează pe un predictor de valori decuplat (*hibrid*) menit să clasifice dinamic instrucțiunile și să le aloce unui tip de predictor predefinit (uzual) din cadrul mai multor predictoare existente. Un dezavantaj al acestei scheme este că necesită un trace cache modificat pentru a păstra informația de clasificare aferentă fiecărei instrucțiuni, și mai mult se consumă un timp suplimentar necesar clasificării instrucțiunilor. De asemenea, datorită modificărilor în comportamentul de predictibilitate al unor instrucțiuni în timpul execuției programului, aceste instrucțiuni migrează între predictorarele componente (De ex: un predictor hibrid alcătuit dintr-un predictor incremental și unul adaptiv pe două niveluri, pentru o aceeași instrucțiune la un moment se va predicționa cu ajutorul predictorului incremental, iar ulterior se va genera valoarea produsă de instrucțiune cu ajutorul celui alt predictor). Aceste modificări frecvente vor introduce mai multe stări tranzitorii în cadrul predictoarelor, scăzând eficiența metodei.

Două tehnici de predicție, extrem de recente [Tho04, Seng04] susținute la workshop-ul dedicat predicției valorilor din Boston, octombrie 2004, introduc în premieră conceptul de predictor neural la nivelul predicției valorilor. Datorită complexității structurilor propuse în prima lucrare [Tho04] și întrucât cea de-a doua cercetare „*predicția valorilor prin regiștri folosind perceptroane*” (pe scurt perceptronul RVP) [Seng04] se bazează pe structuri hardware existente (set de regiștrii generali ai procesorului) și presupune doar o simplă inovație arhitecturală (introducerea unei tabele de perceptroane cu algoritmi de predicție și învățare propuși de Jimenez

[Jim02] în cazul instrucțiunilor de salt), asupra lucrării lui Seng se va insista în subcapitolul 2.2.2.6, unde se va pune accentul mai mult pe câștigul de performanță introdus de perceptronul RVP într-o microarhitectură speculativă.

Predictorul *Perceptron* (cel mai simplu model de predictor neural) s-a dovedit la nivelul salturilor condiționate, nu numai extrem de eficient din punct de vedere al acurateții predicției ci și fezabil de implementat hardware [Jim02a, Sez04, Tar04]. Avantajul principal al acestuia îl constituie creșterea liniară a costului de implementare funcție de istoria memorată și nu exponențial ca în cazul predictorilor corelate pe două niveluri, ceea ce permite perceptronului utilizarea unei istorii foarte mari în procesul de predicție pentru observarea corelațiilor cât mai îndepărtate dintre saltul curent supus predicției și cele anterioare. Funcția de bază a perceptronului este de clasificare liniar separabilă. Argumentele funcției sunt n biți de istorie, utilizați pentru simplificarea calculelor în scopul fezabilității hardware ca valori bipolare (-1 și +1). Fiecare intrare este ponderată și reprezintă gradul de corelație al fiecărei intrări cu ieșirea. Pe lângă cele n ponderi mai există o pondere (*termen liber*) care reprezintă tendința (influența) intrinsecă a ieșirii independent de intrări.

În [Tho04] sunt prezentate două structuri (complexe) de predicție bazate pe perceptron: *Last-2 Value* și respectiv un predictor incremental (vezi figurile 2.21 respectiv 2.23). Structura predictorului de valori *Last-2 Value*, asemănătoare cu cea a predictorului de valori pe două niveluri anterior prezentat [Wang97], se bazează pe un prim nivel cu ultimele valori generate (VHT – *value history table*) și un al doilea nivel – o tabelă de perceptroane, ambele de capacitate 4096 intrări. Pentru reducerea gradului de interferență, tabela VHT este implementată 4-way asociativ, și este adresată cu PC-ul instrucțiunii. Pentru procesul de predicție a valorilor, fiecare instrucțiune reține cele mai recente două valori generate, evacuabile după principiul LRU (cel mai puțin recent folosit). De asemenea, VHT cuprinde și un *pattern* al comportărilor anterioare (*vhp*). Tabela de perceptroane alocă câte un perceptron pentru fiecare instrucțiune (deși era sufficient probabil, în opinia autorului, un singur perceptron și o tabelă de ponderi), indexată cu o funcție de dispersie XOR aplicată *pattern*-ului de istorie *vhp* și celor mai puțini semnificativi 12 biți ai PC-ului instrucțiunii.

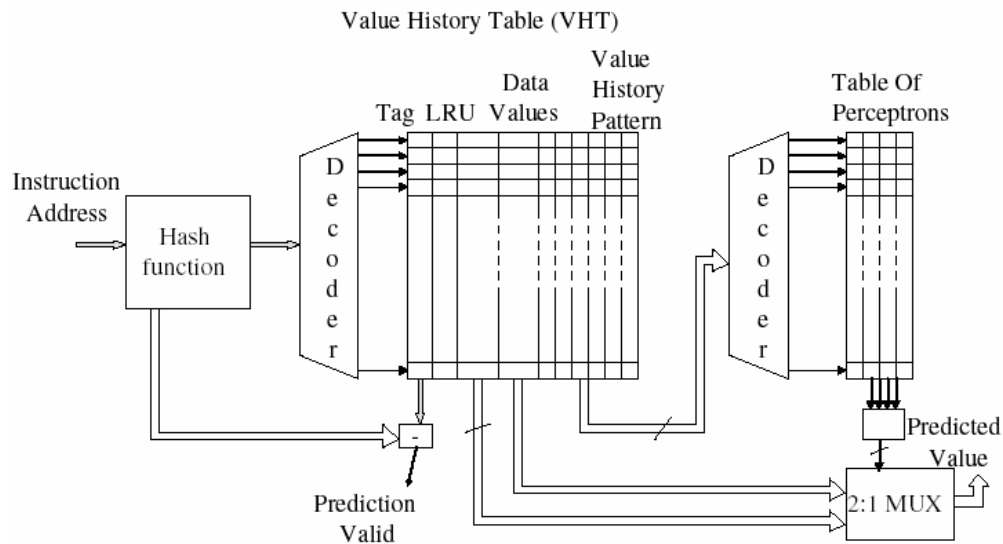


Figura 2.21 Predictor de tipul Last-2 Value cu perceptron

Vectorul vhp reține istoria codificată a valorilor (0, 1), unde 0 arată că la momentul respectiv s-a generat $val[0]$, iar 1 semnifică generarea valorii $val[1]$ din câmpul $DataValues$ (vezi figura 2.21). Pattern-ul de istorie vhp servește ca intrare a perceptronului. Algoritmul de **predicție** implementat de perceptron (vezi în continuare secvența de instrucțiuni în pseudocod apropiat de limbajul C) se bazează pe însumarea ponderilor aferente biților corelați cu $val[1]$ și scăderea ponderilor aferente biților corelați cu $val[0]$. În funcție de ieșirea obținută, dacă $y \geq 0$ este prezisă $val[1]$, altfel $val[0]$. Aceasta este înaintată instrucțiunilor dependente, doar dacă ieșirea obținută depășește un anumit prag impus (*threshold*).

```

y=w[0]
for i=1 to histlen // histlen - dimensiunea
                    pattern-ului vhp
    if (vhp[i]==1)
        y += w[i]
    else // if (vhp[i] == 0)
        y+ = -w[i]
    end if
end for
(y>=0) ? prediction = 1 : prediction = 0

```

Jimenez a arătat în [Jim02] că un parametru important în proiectarea perceptronului îl reprezintă lungimea istoriei. Pentru o dimensiune dată a acesteia (fie h), s-a observat că cel mai bun prag folosit în decizia binară

este dat de formula $\theta = 1.93 \cdot h + 14$. Ponderile perceptronului sunt întregi cu semn reprezentate pe $\log_2 \theta + 1$ biți. În implementarea din [Tho04] istoria folosită în simulare este $h=32$, $\theta=75$ și ponderile sunt reprezentate de întregi cu semn pe 8 biți.

Învățarea (actualizarea) perceptronului este efectuată în cazurile în care *predicția este greșită* respectiv *corectă dar ieșirea se află sub respectivul prag*. Se disting, de asemenea, două subcazuri în funcție de apartenența sau nu la câmpul *DataValues* din structura VHT a valorii generate de către instrucțiunea curentă. Astfel, în situația unei *predicții corecte dar ieșirea sub prag*, dacă rezultatul instrucțiunii se află printre cele două stocate în structură (*val[0]* și *val[1]*), atunci algoritmul de învățare (vezi în continuare secvența de instrucțiuni în pseudocod apropiat de limbajul C) presupune incrementarea ponderilor pentru care bitul de istorie este în corelație pozitivă cu valoarea generată și decrementarea ponderilor în cazul unei corelații negative. De asemenea, și pattern-ul de istorie *vhp* este actualizat.

```

If (Resolved_Output == val[0])
    w[0]--;
Else if (Resolved_Output == val[1])
    w[0]++;
End if
For i = 1 to histlen
    If (vhp[i] == 0 && Resolved_Output == val[0])
        w[i]++;
    If (vhp[i] == 1 && Resolved_Output == val[1])
        w[i]++;
    If (vhp[i] == 0 && Resolved_Output == val[1])
        w[i]--;
    If (vhp[i] == 1 && Resolved_Output == val[0])
        w[i]--;
    End if
End for
vhp <<= 1; vhp |= resolved_index; histlen++;

```

În cazul în care *predicția este greșită, dar valoarea generată este una din cele două stocate în VHT* (se prezice una din cele două valori iar instrucțiunea generează de fapt cealaltă valoare) ponderile sunt incrementate / decrementate cu următorul factor: $Change = Lrate \cdot (T - Y) \cdot vhp[i]$ [Min88] unde *Lrate* reprezintă rata de învățare, egală cu 2 în [Tho04] iar *T* și *Y*, cu valori bipolare (-1, 1) reprezintă ieșirea dorită și respectiv cea obținută. Valorile lui *vhp* sunt considerate bipolare pentru determinarea factorului *Change*.

În cazul unei *predicții greșite cauzate de un miss în cache-ul de valori* (valoarea generată nu se află între cele deja stocate) sunt resetate toate ponderile și sterși toți biții invalizi din *vhp* (cei corelați cu valoarea care tocmai se evacuează). Ponderea termenului liber este resetată doar dacă tendința ei (corelația) este spre valoarea care se evacuează. În continuare este descris algoritmul de actualizare pentru acest din urmă caz.

```

If (LRU == 0 && w[0] < 0) || (LRU ==1 && w[0] > 0)
    w[0] = 0;
Else if (LRU == 0 && w[0] >= 0)
    w[0]--;
Else if (LRU ==1 && w[0] < 0)
    w[0]++;
End if

For i = 1 to histlen
    w[i] = 0
End for

vhp_temp = vhp; hist_temp = histlen; vhp = 0;
histlen = 0
For i = 1 to hist_temp
    If (vhp_temp[i] != LRU)
        vhp <<= 1
        vhp | = vhp_temp[i]
        histlen++
    End if
End for
vhp <<=1; vhp | = LRU; histlen++

```

Dacă structurile de predicție incrementale de tip *2-delta* curente rețineau doar o valoare și ultimii doi pași (incrementul), și tranzitau într-o stare stabilă (predictibilă) doar dacă cei doi pași coincideau (vezi figura 2.16), predictorul incremental cu perceptron reține ultima valoare generată și cei mai recentți 2 pași (nu neapărat identici), iar tranziția automatului de clasificare în starea predictibilă se face doar dacă pasul selectat este mai mare decât pragul perceptronului (vezi figura 2.22).

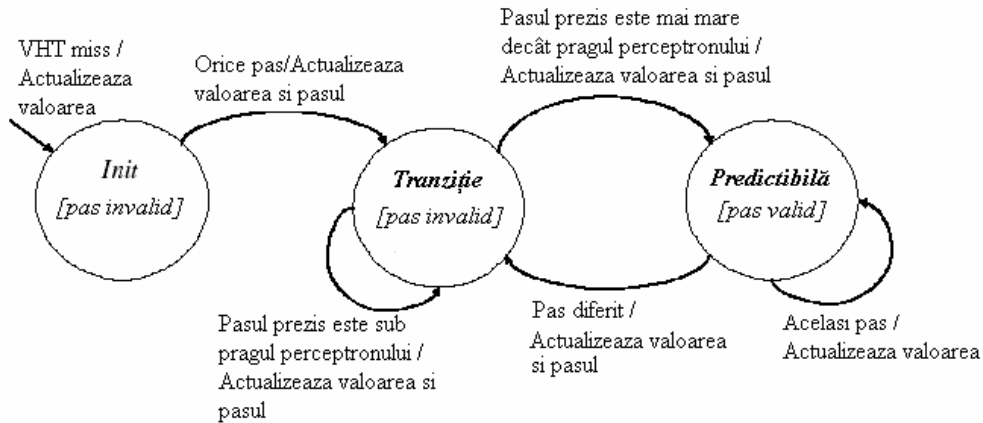


Figura 2.22 Automatul de stare asociat predictorului incremental cu perceptron

Structura predictorului incremental cu perceptron, asemănătoare cu cea a predictorului *Last-2 Value* cu perceptron, este compusă dintr-o tabelă de prim nivel (*Stride History Table*) și o tabelă de perceptrone pe al doilea nivel, ambele de 4096 intrări (vezi figura 2.23).

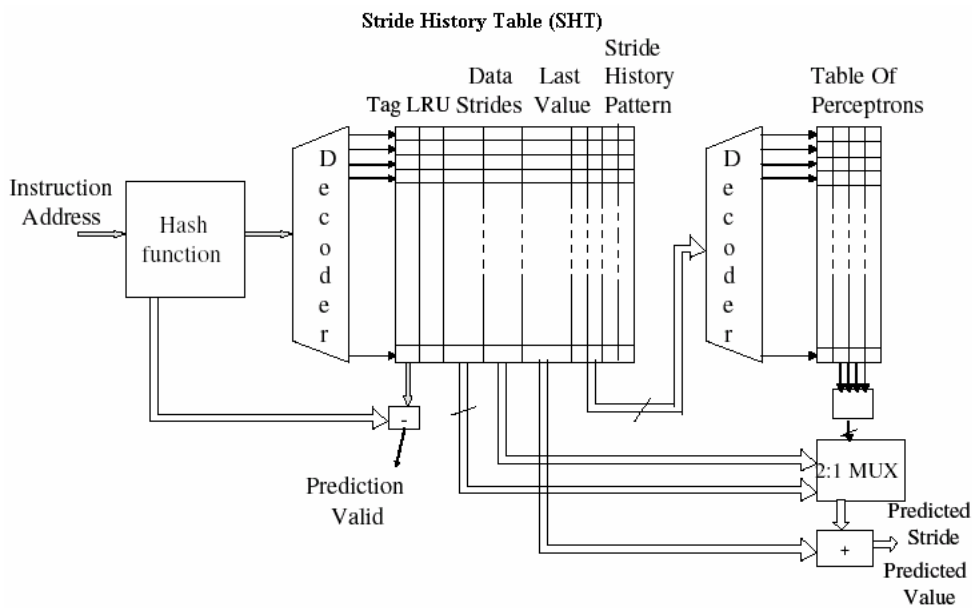


Figura 2.23 Predictor incremental cu perceptron

SHT, indexată tot cu adresa instrucțiunii reține ultima valoare generată, cei mai recentți 2 pași rezultați și un *pattern* asociat acestor pași (*shp*) care reflectă corelația pasului care va fi prezis cu unul din cei doi pași

memorați. Perceptronul practic va prezice incrementul care va fi adunat la valoarea stocată în SHT.

Pentru generarea rezultatelor și exploatarea eficientă a schemelor de predicție implementate Thomas [Tho04] a introdus o structură hibridă, având drept componente cele două predictoare anterior descrise (*Last-2 Value Perceptron* și *incremental cu perceptron*), cu selecție bazată pe numărătoare saturate. Aceasta a fost comparată cu un predictor hibrid [Wang97] compus dintr-unul incremental de tip 2-delta și unul pe două niveluri, dovedit până atunci optim din punct de vedere al acurateții predicției. Complexitatea căutării în tabela de ponderi și stabilirii predicției și respectiv antrenarea perceptronului sunt compensate de o acuratețe de predicție superioară care implică mai puține penalizări și mai puține instrucțiuni care sunt retrimise spre execuție. Simulări realizate pe SPEC2000 au arătat că acuratețea predicției perceptronului variază între 93.47% și 93.55% (oarecum asimptotic), pentru bugete hardware situate între 240KB și respective 930KB. Acuratețea predicției generată de un predictor hibrid clasic [Wang97], în condiții echivalente de cost hardware variază între 87.73% și 88.41%. Câștigul în performanța globală de procesare este nesemnificativ însă (IPC=1.006 pentru predictorul perceptron respectiv IPC=0.998 pentru predictorul hibrid bazat pe numărătoare saturate) [Tho04].

În ciuda rezultatelor promițătoare, operațiile implicate de predicția și actualizarea perceptronului (timpul de *overhead*) fac încă nefezabilă hardware soluția propusă, dar deschid calea ideilor de *crossfertilizare* a arhitecturii calculatoarelor cu inteligența artificială și în domeniul predicției valorilor. Dintre limitările propunerii din [Tho04] se menționează:

- implementarea predicției valorilor doar la nivelul instrucțiunilor Load;
- baza de selecție (valorile care sunt propuse spre predicție) este compusă din doar două valori (datorită mecanismului de clasificare binară al perceptronului), cu implicație directă asupra generalizării ideii. Aceasta nu poate fi realizată printr-un simplu perceptron care împarte spațiul stărilor posibile în două.

Rezultă că printre intențiile de viitor ale cercetătorilor trebuie încercat depășirea acestor limitări și extinderea ideii și la alte tipuri de resurse. Se poate avea în vedere și pipeline-izarea operațiilor de predicție / învățare similar ca în predicția salturilor [Jim02a] în scopul reducerii timpului de overhead.

2.2.2.5. EXPLOATAREA CORELAȚIEI GLOBALE PRIN SCHEME DE PREDICȚIE COMPUTAȚIONALE

Oarecum într-o manieră similară cu predicția salturilor, exploatarea localității din șirurile de valori produse de instrucțiuni prin predicția valorilor se poate face ținând cont de o istorie fie locală sau globală a respectivelor valori. Schemele de predicție existente, computaționale și bazate pe context [Saz99], exploatează o istorie locală – secvență de valori produse de către o aceeași instrucțiune (cea căreia i se predicționează rezultatul). Recent a fost propusă o schemă nouă, *predictorul gDiff* [Zhou03], destinat să exploateze localitatea computațională extrasă dintr-o istorie globală de valori, produse de către toate instrucțiunile dinamice conform ordinii lor de execuție. Aplicabilitatea predictorului gDiff se regăsește în instrucțiunile care concură la calculul adreselor de date/instrucțiuni urmată de citirea valorilor din memorie din cadrul programelor de uz general. Predictorul bazat pe instrucțiunea anterioară (PI), propus în [Nak99] pentru a exploata corelația dintre două instrucțiuni învecinate din secvența dinamică de instrucțiuni, poate fi considerat ca un predictor bazat pe context, global, de ordinul întâi.

Deși rezultatele simulărilor au evidențiat o acuratețe de predicție a schemei bazate pe istoria globală superioară schemelor locale - contextuale și incrementale [Zhou03], o problemă ce trebuie rezolvată și care limitează performanța o reprezintă întârzierea cu care este furnizată o anumită valoare (adresă) necesară unei instrucțiuni dependente aflată în așteptare care se dorește a fi predicționată. Se întâmplă în cazul procesoarelor superscalare cu execuție Out of Order și cu structuri pipeline complexe ca valorile corelate (după care se așteaptă) să fie indisponibile în momentul predicției (distanța cauzată de dependență este prea mică). O soluție de a reduce această întârziere (*value delay*) într-o arhitectură pipeline constă în folosirea speculativă a valorilor imediat după determinarea lor în faza de execuție, fără a se mai aștepta înscrierea lor în setul de regiștri generali. În ciuda acestui avantaj, secvența globală de valori este generată *out of order* și variază în execuție datorită acceselor cu miss în cache respectiv predicțiilor greșite ale instrucțiunilor de salt, diminuându-se astfel avantajul introdus de gradul ridicat de localitate al respectivelor valori.

Deși pentru exploatarea localității globale a valorii prin scheme de predicție computaționale inițial s-a propus o formulă analitică în care valoarea de prezis a unei instrucțiuni reprezintă o medie ponderată (combinație liniară) a celor N valori produse de anterioarele N instanțe dinamice ale respectivei instrucțiuni (vezi figura 2.24), datorită complexității de natură matematică a problemei și implicit hardware,

concentrarea s-a axat pe cazurile speciale dar de uz general. Un astfel de caz, des întâlnit în aplicații, este ilustrat în figura 2.25.

$$x_N = a_{N-1}x_{N-1} + a_{N-2}x_{N-2} + \dots + a_1x_1 + a_0x_0 \quad (1)$$

unde:

$x_N, x_{N-1}, x_{N-2}, \dots, x_1, x_0$ – valorile produse de o secvență de $N+1$ instrucțiuni dinamice $D+N, D+N-1, \dots, D+1, D$, presupunând că fiecare instanță produce o valoare

Figura 2.24. Exploatarea *localității globale a valorii* prin scheme de predicție computaționale - expresie analitică

$$x_N = x_{N-k} + D \quad (2),$$

unde: D reprezintă pasul de incrementare iar x_N – suma dintre valoarea anterior calculată în istoria globală de instrucțiuni (x_{N-k}) și pasul de incrementare.

Figura 2.25. Caz generic de *localitate globală incrementală* desprins din programele de calcul

Sunt cunoscute două cazuri care pun în valoare localitatea incrementală globală: primul reprezintă o succesiune de apeluri implicite sau explicite de instrucțiuni dependente RAW (nu neapărat adiacente) – vezi figura 2.26, iar al doilea se referă la structurile de date dinamice (cu legături) în care câmpurile componente sunt alocate în aceeași ordine în care sunt referite (secvențial).

```

...
Define ( load ra,rb,rc )           // valoarea extrasă din memorie de către instrucțiunea load
                                  este dificil de predicționat.
...
Explicit Use ( add rx, ra, #constant ) //destinația instrucțiunii de adunare (add) poate fi
                                  predicționată cu succes folosind ecuația 2.
...
Explicit Use ( sub rx, ra, rd)     // destinația instrucțiunii de scădere poate fi predicționată
                                  dacă rd respectă anumite pattern-uri repetitive.
...
Implicit Use (load rx, ry, rz)

```

Figura 2.26. Secvență de instrucțiuni cu localitate globală incrementală

În figura 2.26 instrucțiunea “*Define*” este dificil de predicționat însă ajută la predicționarea cu succes a instrucțiunilor (viitoare) subsecvente “*Use*”, altfel dificil de predicționat folosind doar istoria locală a respectivelor instrucțiuni.

Predictorul global de valori **gDiff** se bazează în principal pe două structuri hardware (vezi figura 2.27): tabela de predicție (**PT**) respectiv lista globală de valori (**GVQ**) [Zhou03]. Cea de-a doua GVQ (global value queue) este implementată sub forma unei structuri de date de tip coadă (FIFO) care reține valorile produse de secvența dinamică de instrucțiuni în vederea exploatării localității globale a valorii. Înscrierea rezultatelor instrucțiunilor în GVQ se face *in-order* imediat ce procesarea acestora s-a încheiat.

Tabela de predicție, indexată cu PC-ul instrucțiunii de predicționat reține pentru fiecare locație în parte (instrucțiune statică): *distanța selectată* (dintre instrucțiunile dependente x_N și x_{N-k}) – și *diferențele (incrementul posibil) dintre rezultatele instrucțiunii căreia i se prezice valoarea și rezultatele celor n instrucțiuni dinamice imediat anterioare și care tocmai și-au încheiat execuția* (fie $x_N - x_{N-i}$, cu $i \in 1, \dots, n$).

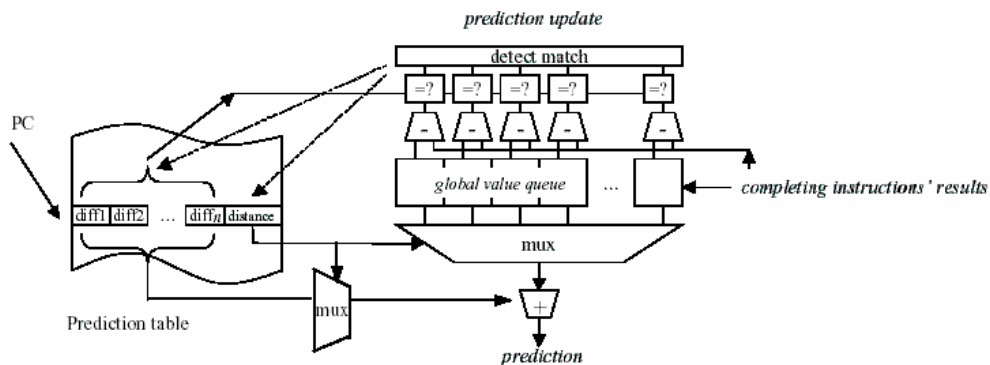


Figura 2.27. Structura predictorului gDiff de ordinul n

Așa cum se poate observa din figura 2.28, se disting două faze în funcționarea predictorului gDiff: *predicția* valorii respectiv *actualizarea* structurii de predicție. La finele fazei *fetch instrucțiune* cu PC-ul instrucțiunii de predicționat este adresată tabela de predicție. Câmpul *distance* al locației accesate (fie $distance=k$) va selecta prin intermediul celor două multiplexoare valoarea stocată la indexul k în lista globală de valori (GVQ) respectiv incrementul corespunzător din tabela de predicție PT. Suma celor două informații selectate constituie rezultatul prezis al instrucțiunii în cauză. Deși potențial realizabilă în doi cicli de procesare

predicția poate fi mascată în prima ei parte prin procesarea în paralel a nivelului de decodificare / dispatch.

Faza de actualizare debutează de îndată ce execuția instrucțiunii (producătoare de valori) s-a încheiat. Cu ajutorul rezultatului acesteia sunt calculate diferențele dintre valoarea reală și cele n valori existente în lista globală, operație efectuată în paralel. Apoi cele n diferențe calculate (pentru un predictor global de ordinul n) sunt comparate cu diferențele deja stocate în intrarea corespundătoare (indicată de PC) din tabela de predicție. Dacă s-a găsit egalitate pentru una din valorile din GVQ atunci distanța (indexul corespunzător) este stocată în câmpul “*distance*” aferent intrării corespundente din PT. Dacă însă în urma celor n comparații nici una din cele n diferențe calculate nu coincide cu cele stocate în PT, diferențele calculate sunt memorate în tabela de predicție, dar distanța rămâne cea anterioară.

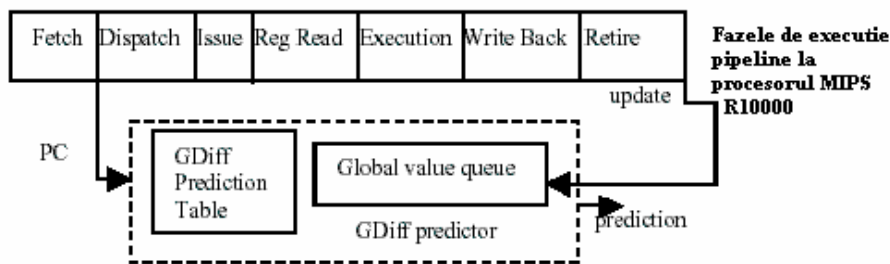
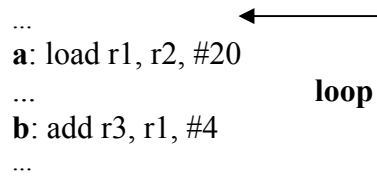


Figura 2.28. Integrarea predictorului gDiff în structura pipeline a unui procesor

După cum se va vedea și în exemplul următor, preluat din [Zhou03], un dezavantaj al schemei de predicție gDiff îl constituie *timpul de învățare* care este de 2 predicții dinamice de valori (practic abia la a treia iterație a secvenței dinamice de instrucțiuni dependente se va putea face o predicție). Perioada de învățare – numărul de valori din secvența de intrare generate înaintea primei predicții corecte – coincide cu cea a predictorului incremental local [Saz99].

Spre exemplificare se consideră secvența dinamică de instrucțiuni generată de bucla de program următoare și bazată în principal pe cele două instrucțiuni statice dependente.



Se presupune că valorile produse de instrucțiunea **a** sunt (1, 8, 3, 2, ...) iar instrucțiunea **b** generează secvența de valori (5, 12, 7, 6, ...). De asemenea, se presupune că între instrucțiunile **a** și **b** mai există alte două instrucțiuni producătoare de valori dar care nu alterează valoarea registrului r1 (cel care cauzează dependența între a și b) și nici nu au vreo corelație cu secvența de valori produsă de instrucțiunea a.

Figura 2.29 ilustrează cum predictorul **gDiff** va învăța gradual să predicționeze rezultatele instrucțiunii **b** pornind de la valorile produse de instrucțiunea **a**.

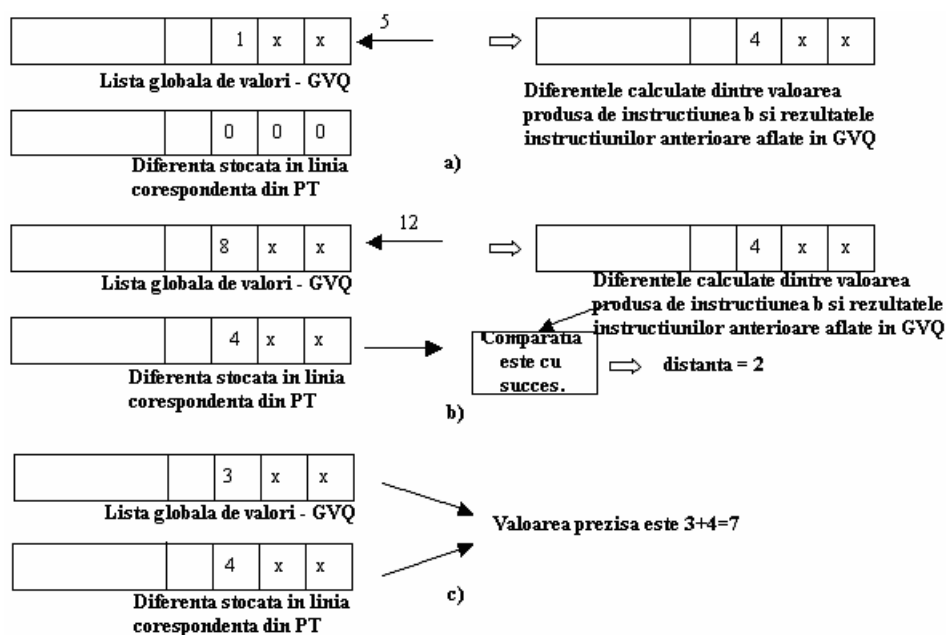


Figura 2.29. Funcționarea predictorului **gDiff** într-o secvență dinamică de instrucțiuni

La finele execuției instrucțiunii **b** sunt calculate diferențele dintre 5 (valoarea furnizată de instrucțiunea b) și valorile din coada globală de valori GVQ. Diferențele astfel calculate sunt comparate cu diferențele anterior stocate în tabela de predicție la adresa indicată de PC. Se presupune că valoarea inițială a diferențelor din tabela PT este 0 (vezi figura 2.29a). Întrucât nici una din comparații nu este cu succes diferențele calculate sunt stocate în tabela de predicție prin suprascrierea celor existente. La a doua iterație a buclei când instrucțiunea b se încheie și produce valoarea 12 (figura 2.29b) sunt calculate noile diferențe care de această dată vor determina o potrivire cu diferențele stocate pe indexul 2 în PT. Astfel, indexul rezultat va fi stocat în câmpul „*distance*” din linia corespondentă din

PT necesar pentru predicția următoarelor apariții ale respectivei instrucțiuni. Ulterior (figura 2.29c), când instrucțiunea b se va afla în faza *dispatch*, predictorul gDiff poate genera o valoare, aceasta fiind suma dintre pasul de incrementare stocat pe indexul specificat de „*distance*” (diff2 în acest exemplu) și valoarea din GVQ de pe poziția indicată de câmpul „*distance*”. Valoarea 3 din GVQ însumată cu valoarea 4 din lista diferențelor memorate va determina în acest caz rezultatul corect produs de instrucțiunea b.

2.2.2.6. CONSIDERAȚII PRIVIND PERFORMANȚA GLOBALĂ A MICROARHITECTURILOR SPECULATIVE

Înainte de a exprima considerațiile privind performanța globală a microarhitecturilor cu execuție speculativă trebuie remarcate câteva aspecte privitoare la instrumentele care fac posibilă cuantizarea și exploatarea acesteia. Din nefericire, paradigma actuală de cercetare în domeniul arhitecturii calculatoarelor este prea specializată, prea limitată. Instrumentele de cercetare corespondente sunt destul de îmbătrânite. De asemenea, interfața „*hardware-software*” nu poate fi înțeleasă în profunzime, sau formalizată într-o manieră real calitativă din mai multe motive:

- Principala abordare a arhitecturilor de calcul se bazează mai degrabă pe *benchmarking* și *simulare* decât pe modele matematice, iar rezultatele cantitative obținute reprezintă doar efecte și nu cauze reale ale fenomenelor procesate.
- Teoria matură bazată pe logică este înlocuită printr-o mulțime de metode *euristice*, empirisme, principii și reguli de natură statistică (vezi regula „90/10”).

Ținând cont de caracteristicile procesului de evaluare al arhitecturilor moderne de procesare, în continuare vor fi prezentate atât din punct de vedere cantitativ cât și calitativ rezultate statistice bazate pe simulări laborioase ale cercetărilor, urmate de descrierea unui model analitic care, deși nu poate înlocui tehnica de benchmarking, generează concluzii în concordanță cu aceasta.

O primă cercetare care demonstrează îmbunătățirea performanței unei microarhitecturi standard, echivalentă PowerPC 620, prin înglobarea unui predictor de valori, de tip „*Last Value*”, aparține lui Lipasti [Lip96b]. Simulări realizate pe benchmark-urile SPEC'92 au arătat un speed-up mediu de 5 procente în favoarea microarhitecturii speculative. Considerând un model architectural „*infini*” (resurse hardware nelimitate) dar nu „ideal” (fiind limitat de acuratețea de predicție diferită de 100% și de lărgimea de

bandă a mecanismului de fetch – un singur branch realizat per ciclu de tact), Lipasti raportează un câștig de performanță de 30% prin implementarea unui predictor de valori de tip “Last Value”.

Impactul introdus de predicția valorilor asupra performanței procesoarelor este cu atât mai mare cu cât factorul superscalar al arhitecturii este mai mare (într-o „fereastră de instrucțiuni” mai mare este posibil să apară mai multe dependențe RAW care pot fi înlăturate prin predicția valorilor instrucțiunilor). Folosind un predictor incremental de valori și realizând simulări pe benchmark-urile SPEC’95 în [Gab98] se determină câteva cauze ale limitărilor câștigului de performanță obținut de către o microarhitectură speculativă. *Potențialul predicției valorilor este cel mai bine valorificat, când procesorul dispune de un mecanism de aducere a instrucțiunilor de bandă largă.* Pentru o rată de fetch de 4 instrucțiuni per ciclu de tact, *speed-up*-ul obținut prin predicția valorilor este sub 4%, în timp ce extinzând rata de aducere a instrucțiunilor la 8, 16, 32 și 40 (valori nerealiste fără Trace Cache), câștigul mediu de performanță variază între 8% și 80%.

Numărul salturilor predicționate simultan are o influență directă asupra câștigului de performanță realizat prin predicția valorilor. Astfel, presupunând un predictor perfect de salturi, prin predicția unui singur branch per ciclu de tact, câștigul prin predicția valorilor este insignifiant (3%), în timp ce, prin predicția simultană a 4 salturi per ciclu de tact, se obține un câștig mediu de performanță de 50%. Repetând experimentul, dar implementând un predictor real de tip PAp (vezi subcapitolul 5.1.1) cu 2048 intrări, 2-way asociativ și un registru de istorie globală pe 4 biți, cu o acuratețe de predicție de 86%; câștigul de performanță prin predicția valorilor este mai moderat, de doar 20%.

Un ultim experiment referitor la aceeași sursă [Gab98], constă în implementarea unei memorii trace cache, mapate direct de capacitate 64 intrări, care poate reține până la 32 de instrucțiuni sau până la 6 basic-blocuri. Prin asocierea trace cache-ului cu predictorul PAp anterior, câștigul de performanță introdus de predictorul incremental este de doar 10%, spre deosebire de 40%, cât s-ar fi obținut dacă în locul predictorului de salturi PAp s-ar fi folosit predictorul ideal.

În [Cal99] se dezvoltă tehnici interesante de reducere a presiunii asupra tabelelor de predicție prin filtrarea instrucțiunilor care vor accesa aceste tabele. În esență, autorii demonstrează pe baza unor simulări laborioase, faptul că dacă se selectează spre a fi predicționate instrucțiunile aparținând căii critice a programului, performanța globală va crește în mod semnificativ. În practică, determinarea pe durata procesării a acestor instrucțiuni mari consumatoare de timp necesită utilizarea unor informații

tip *profilings* precum și ajutorul compilatorului. Din acest motiv, autorii selectează instrucțiunile predictibile într-un mod mai pragmatic și mai facil de implementat. Astfel se vor selecta acele instrucțiuni aflate în curs de procesare în diferite faze și care aparțin lanțului cel mai lung de dependențe RAW. Pentru tabele de 1024 de intrări se raportează o creștere medie a performanței față de o mașină superscalară clasică de cca. 11% prin utilizarea unor asemenea tehnici de predicție selectivă a instrucțiunilor.

Tullsen și Seng [Tull99] au propus o metodă intitulată “*predicția valorilor prin regiștri*” (pe scurt predictorul RVP) care identifică instrucțiunile care produc valori, deja aflate în setul de regiștri generali. Autorii au constatat că 75% din timp, valorile încărcate din memorie fie se află în setul de regiștri generali, fie au existat recent. Prin această tehnică se încearcă exploatarea reutilizării valorii regiștrilor (o formă de localitate). Astfel, rezultatul unei instrucțiuni este prezis ca fiind chiar valoarea stocată deja în actualul registru destinație. Predictorul RVP, indexat cu cei mai puțin semnificativi biți ai adresei instrucțiunii, este reprezentat printr-o tabelă de numărătoare saturate pe 3 biți, incrementate ori de câte ori o instrucțiune produce o aceeași valoare ca cea existentă deja în registrul său destinație. Contorul este resetat în momentul în care rezultatul produs de instrucțiune diferă de cel aflat în registrul destinație. Pragul impus (*threshold*) este 6, astfel încât o instrucțiune este predicționată doar dacă numărătorul depășește respectivul prag. Pe lângă câștigul de performanță raportat de 11% folosind benchmark-urile SPECint95 și respectiv 13% folosind testele SPECfp95, prin această tehnică este eliminată necesitatea unor structuri hardware de predicție reducându-se costul de implementare. Suportul compilatorului este folosit pentru a crește oportunitățile oferite de această tehnică de predicție. Cu toate că regiștri procesorului joacă un rol important în procesul de predicție, tehnica propusă de Tullsen este centrată pe instrucțiuni, numărătoarele saturate folosite în implementare fiind asociate mai degrabă instrucțiunilor decât regiștrilor. În subcapitolul 6.2 s-a propus o tehnică originală alternativă de predicția valorilor, centrată pe contextul procesorului, care prezice noua valoare a unui registru bazat pe valorile anterioare stocate în respectivul registru (*contribuție originală*).

În [Seng04] este propusă o inovație arhitecturală – introducerea unei tabele de perceptroane (*Perceptronul RVP*), având ca scop eficientizarea procesului de predicție introdus de Tullsen (RVP) prin folosirea informațiilor de redundanță aferente ultimelor N instrucțiuni executate. Perceptronul RVP, schemă globală de predicție, este indexată cu cei mai puțin semnificativi biți ai PC-ului instrucțiunii. Intrarea în perceptronul accesat sunt biții registrului de istorie globală – comportamentul ultimelor N instrucțiuni executate: *redundante* (+1) sau *neredundante* (-1). În

implementarea propusă, Seng nu folosește ponderea termen liber (*bias*) preferând utilizarea unui bit suplimentar de istorie. De fiecare dată când se execută o instrucțiune, perceptronul prezice redundanța sau neredundanța respectivei instrucțiuni. Predicția efectuată se compară cu rezultatul real al instrucțiunii, perceptronul primind printr-o reacție inversă, pozitivă sau negativă, informația de actualizare a ponderilor sale. Algoritmul de predicție (suma ponderată a intrărilor comparată cu un prag $-\theta$), precum și stabilirea pragului optim, a numărului de biți pe care sunt reprezentate ponderile sunt conform propunerilor lui Jimenez, utilizat în predicția salturilor condiționate [Jim02].

Algoritmul de învățare al perceptronului selectat se bazează pe tabelul următor (vezi tabelul 2.3 și este aplicat în situația unei predicții greșite sau dacă ieșirea perceptronului este sub pragul θ).

	A fost instrucțiunea corespunzătoare bitului i din registrul de istorie global, redundanță ?	
Este instrucțiunea redundanță ?	Da	Nu
Da	$w[i]++$	$w[i]--$
Nu	$w[i]--$	$w[i]++$

Tabelul 2.3 Algoritmul de învățare în cazul Perceptronului RVP

În implementarea realizată de Seng, având ca bază o microarhitectură speculativă cu factor superscalar de 8 instrucțiuni per ciclu, pot fi supuse predicției toate instrucțiunile care au ca destinație un registru întreg sau flotant (*load, aritmetice întregi sau flotante*). Istoria maximă considerată este de 60 de biți iar ponderile sunt numere întregi cu semn reprezentate pe cel mult 7 biți. Simulările realizate arată necesitatea suportării de către setul de regiștri generali, simultan a unei rate de citire de 2.28 accese per ciclu. De asemenea, se observă că schema de predicție neurală are un comportament contradictoriu. Deși în medie acuratețea predicției este superioară celei obținute cu predictorul RVP clasic [Tull99] există benchmark-uri pentru care structura clasică se comportă mai bine. O explicație ar putea fi faptul că predictorul RVP tinde să detecteze predictibilitatea *locală* în timpul execuției individuale a fiecărei instrucțiuni pe când perceptronul RVP se bazează pe comportamentul *global* al instrucțiunilor anterioare.

Rezultatele simulărilor pe benchmark-urile SPEC2000 conduc la concluzia că cu cât crește dimensiunea istoriei memorate cu atât se îmbunătățește performanța, cu toate că, aceasta devine asimptotică de la o

anumită dimensiune a istoriei pentru unele benchmark-uri – *mcf*, iar pentru altele, creșterea continuă și pentru istorii mai mari de 60 de biți – *mgrid*. Pentru o microarhitectură speculativă care încorporează un Perceptron RVP de 8192 intrări câștigul de performanță globală de procesare introdus este de 8.1%, ajungându-se în cazuri particulare și la 45.2% [Seng04]. Practic 80% din speed-up-ul realizat cu o istorie de 60 biți poate fi atins cu o istorie restrânsă de numai 16 biți. Procentajul instrucțiunilor redundante predicționate corect din totalul instrucțiunilor redundante, crește de la 85.6% pentru o istorie egală cu 4 la 87.8% pentru o istorie de 32 de biți. Acuratețea predicției crește de la 94.2% pentru o istorie de 4 biți până la 97.7% pentru o istorie de 16 biți ajungând la 98.6% pentru maximul de istorie considerat (60).

În vederea îndeplinirii dezideratului de fezabilitate hardware și a creșterii acurateții predicției, printre intențiile de viitor ale cercetătorilor, pe lângă optimizările legate de timpul de overhead, trebuie încercat limitarea procesului de predicție (și reținerea comportamentului redundant) la doar câteva tipuri de instrucțiuni (load / mult / div). De asemenea, se poate studia influența istoriei comportamentului salturilor (HRg), pe lângă, sau în locul, informațiilor de redundanță a valorilor. Soluția presupune economisirea de spațiu hardware deoarece această istorie (HRg) aferentă instrucțiunilor de ramificație este deja pastrată în procesor [Seng04].

Pe lângă considerațiile privind performanța globală a microarhitecturilor speculative rezultate în urma simulărilor laborioase a unor benchmark-uri reprezentative (SPEC'95, SPEC2000), în continuare, preluat din [Vin02], este prezentată o abordare analitică care evidențiază potențialul tehnicii de predicție a valorilor de a crește paralelismul la nivelul instrucțiunilor.

Inițial în [Gab98a] se propune un model analitic elaborat în vederea determinării creșterii de viteză aduse de o arhitectură cu predicție a valorilor față de una superscalară clasică. Deși modelul conține câteva greșeli și omisiuni, eliminate în [Vin02], el este în principiu corect și merită să fie analizat. Se consideră o mașină abstractă cu o infinitate de regiștri generali și unități de execuție. De asemenea, se consideră că probabilitatea ca o anumită instrucțiune să se execute speculativ și deci să fie corect predicționată, este p , fiind egală cu acuratețea medie de predicție a valorilor instrucțiunilor. Pentru simplificare, o instrucțiune predicționată corect se execută în mod instantaneu, în caz contrar ea executându-se în timpul T . Modelul ia în considerare calea critică a programului și consideră, pentru simplitate, că întregul program se află memorat în resursele interne ale procesorului (deci un *instruction window* infinit). De remarcat faptul că, având în vedere că modelul are resurse infinite și deci este posibilă o

redenumire perfectă a resurselor dependente WAR și WAW, instrucțiunile din afara căii critice se vor procesa în paralel cu cele aparținând acesteia. O reprezentare grafică simplificată aferentă procesării instrucțiunilor pe modelul propus este dată în figura 2.30.

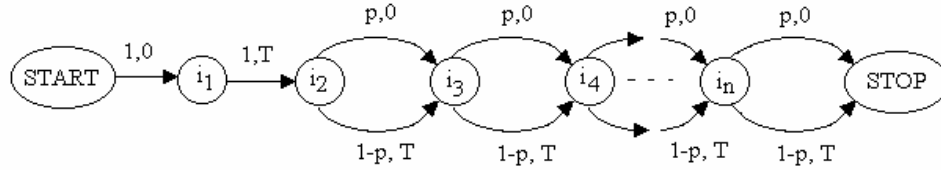


Figura 2.30 Model speculativ de procesare a instrucțiunilor

Semantica figurii 2.30 este următoarea: dacă instrucțiunea I_k este predicționată corect (p) atunci execuția ei este practic instantanee, altfel ($1-p$) aceasta se execută în timpul T . Așadar, orice arc are asignate două componente: o probabilitate de execuție speculativă și respectiv un timp de execuție a instrucțiunii (nespeculative).

Timpul total de execuție al programului este:

$$ET(n) = T + T_{n-1}, \quad (2.1)$$

unde T_{n-1} reprezintă timpul de execuție al celor $(n-1)$ instrucțiuni următoare instrucțiunii I_1 , deci timpul aferent instrucțiunilor I_1, I_2, \dots, I_n . Se poate scrie:

$$\text{Prob}(T_{n-1} = kT) = C_{n-1}^k (1-p)^k p^{n-1-k}, \quad \forall k \in \{0, 1, 2, \dots, n-1\} \quad (2.2)$$

Firește că este îndeplinită și relația de normalizare:

$$\sum_{k=0}^{n-1} \text{Prob}(T_{n-1} = kT) = 1 \quad (2.3)$$

Rezultă că timpul total de execuție este:

$$ET(n) = T + T \sum_{k=1}^{n-1} k \text{Prob}(T_{n-1} = kT) = T + T \sum_{k=1}^{n-1} k C_{n-1}^k (1-p)^k p^{(n-1-k)} \quad (2.4)$$

Pe baza identității ce rezultă din chiar definiția combinărilor și anume:

$$C_n^k = \frac{n!}{(n-k)!k!} \quad (2.5)$$

se arată imediat că este îndeplinită egalitatea:

$$kC_{n-1}^k = (n-1)C_{n-2}^{k-1} \quad (2.5')$$

Luând în considerare această ultimă identitate și ținând cont de ultima expresie a lui $ET(n)$, putem scrie succesiv egalitățile:

$$\begin{aligned} T_{n-1} &= T \sum_{k=1}^{n-1} (n-1)C_{n-2}^{k-1} (1-p)^k p^{n-1-k} = T(n-1)(1-p) \sum_{k=1}^{n-1} C_{n-2}^{k-1} (1-p)^{k-1} p^{n-1-k} = \\ &= T(n-1)(1-p)(p + (1-p))^{n-2} = T(n-1)(1-p) \end{aligned} \quad (2.6)$$

În concluzie, timpul total de execuție este:

$$ET(n) = T + T(n-1)(1-p) \quad (2.7)$$

Rezultă că accelerarea $S(n)$ față de un procesor echivalent, dar fără predictor de valori, este:

$$S(n) = \frac{nT}{ET(n)} = \frac{n}{1 + (n-1)(1-p)} \quad (2.8)$$

Ținând cont de faptul că numărul de instrucțiuni dinamice executate este practic infinit, accelerarea este:

$$S(\infty) = \lim_{n \rightarrow \infty} S(n) = \frac{1}{1-p} \quad (2.9)$$

Având în vedere multiplele simplificări operate, în realitate accelerarea este mai mică decât $S(\infty)$ obținut, dar și așa rezultatul este unul important și deosebit de sugestiv.

Se poate deduce o expresie analitică a creșterii de performanță și în cazul, mai realist, în care se consideră o fereastră finită de instrucțiuni. Notând cu w ($w > 1$) capacitatea acestei ferestre de instrucțiuni, probabilitatea de a predicționa corect k instrucțiuni consecutive este:

$$\Pr ob(L = k) = \begin{cases} (1-p)p^{k-1}, 1 \leq k \leq w-1 \\ p^{w-1}, k = w \end{cases} \quad (2.10)$$

S-a notat cu L o variabilă aleatoare care semnifică numărul de instrucțiuni evacuate din fereastra de instrucțiuni în fiecare ciclu. Numărul de instrucțiuni evacuate în fiecare ciclu L este dat de instrucțiunile din lanțul de instrucțiuni dependente RAW, situate până la prima instrucțiune predicționată greșit. Se notează $E(L)$ o medie ponderată a instrucțiunilor evacuate în fiecare stare a modelului.

$$\text{Ținând cont de faptul că: } \sum_{k=1}^w p(L = k) = 1 \Rightarrow \quad (2.10')$$

$$E(L) = 1 \cdot P(L=1) + 2 \cdot P(L=2) + \dots + (w-1) \cdot P(L=w-1) + w \cdot P(L=w) \Leftrightarrow$$

$$E(L) = \sum_{k=1}^w k \cdot p(L = k) \quad (2.11)$$

$$E(L) = \sum_{k=1}^{w-1} k \cdot p(L = k) + w \cdot p(L = w) = \sum_{k=1}^{w-1} k \cdot (1-p)p^{k-1} + w \cdot p^{w-1} = (1-p) \left(\sum_{k=1}^{w-1} k \cdot p^{k-1} \right) + w \cdot p^{w-1}$$

Întrucât:

$$\begin{aligned} \sum_{k=1}^{w-1} k \cdot p^{k-1} &= \sum_{k=1}^{w-1} (p^k)' = \left(\sum_{k=1}^{w-1} p^k \right)' = \left(p \cdot \frac{1-p^{w-1}}{1-p} \right)' = \left(\frac{p-p^w}{1-p} \right)' = \frac{(1-w \cdot p^{w-1})(1-p) + (p-p^w)}{(1-p)^2} = \\ &= \frac{1-w \cdot p^{w-1} - p + w \cdot p^w + p - p^w}{(1-p)^2} \end{aligned}$$

Rezultă că,

$$\begin{aligned} E(L) &= \frac{1-w \cdot p^{w-1} + p^w(w-1)}{(1-p)} + w \cdot p^{w-1} = \frac{1-w \cdot p^{w-1} + p^w(w-1) + w \cdot p^{w-1} - w \cdot p^w}{1-p} \\ &= \frac{1-p^w}{1-p} = \frac{1}{1-p} - \frac{p^w}{1-p} \end{aligned} \quad (2.12)$$

Valoarea obținută astfel este chiar accelerarea reală, întrucât execuția căii critice pe procesorul convențional este redusă la 1 instr./ciclu. Speed-up-ul obținut de o arhitectură speculativă cu o fereastră de w instrucțiuni

este dat de *accelerarea ideală* (obținută pentru o fereastră infinită) din care se scade *un termen dependent de lungimea ferestrei*.

Ca și contribuție proprie, în subcapitolul 6.1.2 se propune dezvoltarea unui model teoretico-practic de evaluare a performanței arhitecturilor pipeline cu execuții multiple, bazat pe implementarea schemelor de predicția valorilor. Practic se urmărește determinarea câștigului de performanță obținut ("*speed-up*") de către o microarhitectură speculativă care înglobează tehnica de predicție a valorilor instrucțiunilor de tip Load, datorită latenței ridicate de execuție a acestora, în condițiile accesului la memoria principală.

2.2.3. PROBLEME DE IMPLEMENTARE A PREDICȚIEI VALORILOR ÎN ARHITECTURI CU MICROFIRE MULTIPLE DE PROCESARE

Paralelismul speculativ la nivel de fir de execuție (*coarse grain*) a fost recent propus ca o sursă alternativă de paralelism (celui de tip *fine grain* – la nivel de instrucțiune) care poate spori performanțele aplicațiilor acolo unde sunt greu de găsit instrucțiuni independente. O abordare inedită care integrează predicția valorilor în arhitecturile de tip multithread este propusă de Marcuelo [Mar99]. Ideea de bază constă în execuția concurentă a unor *thread*-uri speculative din cadrul programului, determinate pe timpul rulării acestuia cu ajutorul predictorului de branch-uri. Firele sunt executate speculativ deoarece prezintă, în general, atât dependențe de date cât și de control față de cele anterioare, fire independente fiind greu de găsit în aplicații non-numerice. Pentru execuția *thread*-urilor speculative împreună cu cele nespeculative, este necesar ca procesorul să ofere contexte hardware multiple precum și mecanisme pentru a înainta sau a prezice valori produse de un fir și folosite de altul. Fiecare *thread* va avea propria sa fereastră de instrucțiuni. Există diferențe între microarhitecturile care exploatează paralelismul speculativ la nivel de fir de execuție, în funcție de modul în care programele sunt împărțite în fire: de către compilator sau prin tehnici hardware – o problemă realmente esențială. Spre exemplu, în cazul unei bucle de program, *thread*-urile speculative ar putea fi constituite din chiar iterațiile buclei respective (strategie numită *loop-iteration* [Mar00]). În literatura de specialitate [Mar00] sunt propuse mai multe strategii de determinare efectivă a firelor de execuție: *loop-continuation* (crearea unui thread speculativ din instrucțiunile aflate imediat după branch-ul static care determină re-execuția buclei) și respectiv *subroutine continuation* (generarea unui fir speculativ începând cu prima instrucțiune statică aflată

după cea de apel a subrutinei). O altă soluție, mai complexă însă, de care tot compilatorul este responsabil, se bazează pe euristici menite să minimizeze dependențele dintre firele de execuție active. O rezolvare simplă a interdependențelor existente între *thread*-urile componente unor programe non-numeric, caracterizate prin grade limitate de paralelism, ar consta în serializarea dependențelor de date. Predicția valorilor ar putea comprima lanțurile de instrucțiuni dependente, îmbunătățind semnificativ performanța acestor arhitecturi multifir. Cercetări laborioase au arătat că o asemenea arhitectură hibridă a implicat o creștere de performanță de 40% față de o arhitectură superscalară monofir și respectiv de 9% față de o arhitectură superscalară monofir având înglobat un predictor hibrid de valori [Vin02].

Implementarea predicției valorilor în procesoarele care suportă multithreading sau în sisteme multiprocesor determină apariția unor probleme de **corectitudine** (coerență a sistemului ierarhic de memorie și păstrarea unui mecanism precis de tratare a întreruperilor și excepțiilor). Studiile cercetătorilor [Lip01] au reliefat că în sistemele cu paralelism la nivelul *thread*-urilor simpla predicție a unei valori urmată de verificarea ulterioară (valoarea prezisă este egală cu valoarea calculată, rezultată în urma execuției) nu este întotdeauna suficientă.

Într-un sistem monoprosesor, monofir corectitudinea se traduce prin ordinea de execuție a instrucțiunilor. În sistemele cu paralelism la nivelul firelor de execuție, corectitudinea este definită prin consistența modelului de memorie (interfața hardware-software care definește ordinea legală de execuție a instrucțiunilor cu referire la memorie load / store). Un model de memorie consistent trebuie să răspundă la următoarele întrebări:

- Q1:** "*Dacă un fir scrie în două locații diferite de memorie în ce ordine vor vedea alte fire sau dispozitive ale sistemului aceste scrieri ?*"
- Q2:** "*Vor observa toate firele din sistem cele două scrieri în aceeași ordine ?*"

Întrucât predicția valorilor acționează asupra consistenței modelului de memorie prin permiterea instrucțiunilor dependente de date să fie reordonate (predicția rezultatului unei instrucțiuni *load* urmată de un *store* și apoi de re-execuția *load*-ului ca urmare a predicției greșite), trebuie verificat ca prin implementarea acestei tehnici în sistemele cu paralelism la nivelul *thread*-urilor să nu fie încălcat modelul consistent de memorie. Spre deosebire de procesoarele monofir, în sistemele cu paralelism la nivelul *thread*-urilor, care implementează tehnica de predicția valorilor este posibil ca valoarea să fie predicționată greșit în momentul predicției dar în momentul verificării să fie cea corectă, întrucât un alt fir (procesor, dispozitiv periferic) a modificat valoarea în intervalul dintre predicție și verificare.

Lamport [Lam79] a *formalizat* noțiunea de model consistent de memorie prin definirea unui sistem **consistent secvențial** (SC) în condițiile: (1) dacă rezultatul execuției oricărei instrucțiuni este același ca și cum operațiile tuturor procesoarelor (firelor) au fost executate într-o anumită ordine secvențială și (2) operațiile fiecărui proces (fir) individual apar în această secvență în ordinea stabilită prin program după compilare. SC este cel mai restrictiv model de consistență a memoriei care a fost implementat comercial în sisteme cu procesor MIPS R10000 și HP PA-8000.

Când se aplică tehnica de predicție a valorii unei instrucțiuni, procesorul prezice rezultatul respectivei instrucțiuni, și continuă speculativ execuția instrucțiunilor din cadrul acelui thread (incluzându-le și pe cele dependente). Din motive de simplitate, în unele implementări [Lip01] se așteaptă verificarea oricărei predicții de valori efectuate, înainte de execuția vreunei instrucțiuni *store* întâlnite pe perioada rulării speculative a instrucțiunilor din fir. În momentul încheierii execuției instrucțiunii al cărei rezultat a fost predicționat, posibil după mai mulți cicli datorită acceselor cu miss în cache sau altor întârzieri, procesorul compară valoarea prezisă cu cea reală rezultată. În caz de egalitate, predicția a fost cu succes, s-a câștigat timp prin execuția speculativă a instrucțiunilor, altfel se vor relua instrucțiunile thread-ului folosind un mecanism de “*recovery*” similar celui utilizat în cazul predicției greșite a branch-urilor.

Există câteva modalități de implementare corectă a predicției valorilor într-un sistem consistent secvențial. Detecția *bazată pe adresă* și respectiv *detecția bazată pe valoare* preîntâmpină încălcarea ordinii de execuție a operațiilor cauzate de predicția valorilor, însă adaugă complexitate și crește costul de implementare. În prima variantă un procesor trebuie să detecteze când un alt thread, procesor sau dispozitiv periferic scrie la o adresă care a fost citită speculativ din cache de către o instrucțiune nefinalizată încă (care nu a efectuat faza *commit / writeback*). În momentul observării unei violări a ordinii operațiilor load/store procesorul reia execuția thread-ului speculativ de la o stare consistentă cunoscută. Procesorul MIPS R10000 pune în practică această abordare prin suplimentarea cozii load/store pentru: (1) a reține adresele care au fost citite (încărcate) speculativ până la retragerea (executarea fazei *commit/writeback*) load-urilor și (2) pentru a compara toate aceste adrese cu adresele scrise de către alte procesoare (thread-uri). Aceste scrieri externe sunt observate prin sosirea unor mesaje de invalidare a acestor adrese în cadrul unui protocol de coerență. Pentru implementarea predicției valorilor în procesoarele multithread viitoare trebuie păstrată câte o tabelă pentru fiecare thread cu adresele load-urilor executate speculativ și verificate toate scrierile (store) efectuate de alte fire, aferente oricărui procesor, în respectiva tabelă. Schema însă, este extrem de conservativă în

sensul că determină uneori evacuări din tabelă și declanșează mecanismul de refacere a corectitudinii chiar și în cazul unor scrieri silențioase (acele instrucțiuni care scriu în memorie o aceeași valoare ca și precedenta, deci care nu modifică starea sistemului) [Lep00]. Rezultatele simulărilor pe benchmark-urile SPEC'95 au arătat că între 34% și 68% din instrucțiunile Store sunt silențioase, fapt ce poate fi exploatat practic, prin anularea execuției acestor instrucțiuni, cu beneficii asupra performanței sistemului de calcul [Lep00].

Detecția bazată pe valoare presupune execuția speculativă a load-urilor și a celorlalte instrucțiuni dependente din thread, urmate de reexecuția instrucțiunii load în momentul în care operanzii săi devin nespeculativi (cunoscuți). Această abordare înlătură din conservatorismul schemei anterioare însă, prin reexecuția instrucțiunilor load conduce la creșterea concurenței asupra porturilor cache-ului de date. O abordare recentă în sprijinul detecției bazată pe valoare a consistenței memoriei se referă la verificarea dinamică folosind o arhitectură decuplată (nucleu separat de procesor dotat cu cache de date propriu) care efectuează verificările în paralel cu execuția procesorului.

În extinderea conceptului de predicție a valorilor instrucțiunilor în arhitecturi cu microfibre multiple de procesare se află și *predicția la nivel de funcție* [Kavi03]. Astfel, ar putea exista un *thread* care folosește rezultatul prezis al funcției sporind gradul de paralelism al aplicației și un alt *thread* care să calculeze funcția și în cazul unei predicții greșite să refacă operațiile / contextul necesare (procesul de *recovery*).

Foarte succint, în cadrul acestui capitol, concluzia ar fi că, dintre constrângerile care limitează paralelismul la nivelul instrucțiunilor în procesoarele superscalare, singura fundamentală, și care nu poate fi eliminată prin metode hard-soft de tipul: *multiplicări de resurse, predicții de salturi, mecanisme de redenumire a regiștrilor* etc. o constituie dependențele reale de date. Începând cu anul 1996, două tehnici au fost dezvoltate pentru reducerea efectului negativ introdus de dependențele de date de tip *read after write*: reutilizarea dinamică a instrucțiunilor și predicția valorilor. Este posibil ca în viitor să se evolueze spre tehnici automate de reutilizare a codului executat dinamic atât în hardware cât și în software, combinându-se puterea de reutilizare prin transmiterea informațiilor de semantică structurală de la nivel *high* mecanismelor hardware de la nivel *low*. De asemenea, se poate spune că predicția, o problemă științifică de mare interes (în domenii din cele mai diverse: seismologie, meteorologie, astronomie etc.), care în cazul general necesită folosirea unor instrumente matematice foarte puternice precum procesele *Markov* ori seriile de timp [Vin02], va migra și în cadrul viitoarelor

generații de microprocesoare avansate. Pentru aceasta, este necesar ca arhitecții de microprocesoare să acorde o mai mare atenție problemei generale a predicției, așa cum apare ea în știința actuală. În fond, valorile generate de instrucțiuni pot fi asimilate cu niște **serii de timp** discrete. De multe ori, în cazul contextual, aceste șiruri de valori generate prin programe pot fi asimilate cu procesele Markov care reprezintă o succesiune de stări, trecerea de la una la alta efectuându-se cu o probabilitate condiționată proprie procesului, funcție de stările anterioare. Problema predictibilității este strâns legată de cea a staționarității seriilor de timp respective. O serie este staționară dacă media și dispersia sunt finite și constante în timp iar funcția de autocorelație depinde numai de valoarea intervalului pe care s-a calculat [Pop00]. Studiul seriilor de timp nu poate fi decât benefic în scopul construirii unor predictoare cât mai eficiente, favorizând execuția super-speculativă a instrucțiunilor și în consecință, viteze de procesare tot mai mari.

3. METODOLOGIA DE SIMULARE

3.1. BENCHMARK-URI UTILIZATE ÎN SIMULARE. CARACTERISTICI.

3.1.1. BENCHMARK-URILE SPEC'95.

SPEC (Standard Performance and Evaluation Corporation) au fost dezvoltate de “*SPEC's Open Systems Group*” (OSG), care înglobează peste 30 de producători de calculatoare, integratori de sisteme, autori și consultanți din întreaga lume. Benchmark-urile reprezintă seturi de aplicații dezvoltate pentru evaluarea performanțelor calculatorului și pot fi folosite pe diverse versiuni de UNIX, Linux și Microsoft (dezvoltate de Microsoft).

Caracteristicile benchmark-urilor SPEC'95 precum și intrările lor aferente sunt descrise în tabelul următor:

Benchmark-urile SPEC '95		
<i>Benchmark</i>	<i>Intrare folosită</i>	<i>Caracteristici</i>
Applu	Applu.in	Rezolvă sisteme de matrici prin metode de pivotare
Apsi	Apsi.in	Calculează statistici asupra temperaturilor și gradelor de poluare
Cc1	1stmt.i	Compilează surse pre-procesate în cod optimizat pt. procesorul SPARC
Compress95	Bigtest.in	Comprimă un fișier text utilizând algoritmul adaptiv <i>Lempel-Ziv</i>
Fpppp	Natoms.in	Determină derivate multi-electron.
Go	9stone21.in	Joc de <i>Go</i> având înglobate metode strategice rafinate de IA
Hydro2d	Hydro2d.in	Calculul jeturilor galactice utilizând ecuațiile hidrodinamice ale lui <i>Navier - Stokes</i>
Ijpeg	Vigo.ppm	Comprimare prin algoritmi JPEG a unor fișiere tip imagine
Perl	Scrabbl.pl	Manipulări de texte și numere (anagrame, factorizări de numere prime)
Wave5	Wave5.in	Rezolvă ecuațiile lui Maxwell..

Su2cor	Su2cor.in	Calculul masei unor particule elementare utilizând teoria <i>Quark-Gluon</i>
Swim	Swim.in	Rezolvă ecuațiile lichidelor subțiri utilizând ecuații cu diferențe finite (singurul bench în simplă precizie)
Tomcatv	Tomcatv.in	Rezolvă probleme de geometrie computațională
Li	*.lsp	Interpreter de <i>Lisp</i>
Turb3d	Turb3d.in	Simulează turbulența într-un zonă cubică.
Vortex	Vortex.lit	Construiește și manipulează trei baze de date relaționale.
Mgrid	Mgrid.in	Determină potențialul unui câmp electromagnetic.

Tabelul 3.1. Caracteristicile benchmark-urilor SPEC'95

3.1.2. BENCHMARK-URILE SPEC2000.

SPEC CPU2000 este a 4-a versiune majoră a seturilor de benchmark-uri SPEC CPU, care, în 1989 a devenit primul standard acceptat la scară largă pentru compararea performanțelor la calcul intensiv pe o varietate de arhitecturi.

“Tehnologia sistemelor de calcul se dezvoltă așa de repede, încât trebuie să oferim noi pachete de benchmark-uri, pentru a asigura un mediu de testare adecvat. SPEC CPU95 a fost un mare succes, dar este timpul să facem trecerea la benchmark-uri standardizate, care reflectă îmbunătățirile tehnologice aferente microprocesoarelor, noi compilatoare, aplicații multimedia și transmisii de semnal audio/video/GSM, care s-a făcut în ultimii 5 ani; aceste benchmark-uri formează SPEC 2000.”(Kaivalya M. Dixit, președinte SPEC)

SPEC CPU2000 cuprinde 2 seturi de benchmark-uri: CINT2000 pentru măsurarea performanțelor în cazul calculului intensiv cu numere întregi și CFP2000 pentru performanțele în cazul calculului intensiv în virgulă flotantă. Cele 2 seturi măsoară performanțele procesorului, arhitecturii memoriei și compilatorului unui calculator. Îmbunătățirile aduse seturilor noi includ timp mai mare de execuție și probleme mai ample pentru benchmark-uri, o varietate mai mare a aplicațiilor, o ușurință mai mare de utilizare și platforme standard de dezvoltare care vor permite SPEC să producă versiuni adiționale pentru alte sisteme.

Setul CINT2000 cuprinde 12 benchmark-uri bazate pe aplicații, scrise în limbajele C și C++, iar CFP2000 cuprinde 14 benchmark-uri scrise în FORTRAN (77 și 90) sau C care realizează operații în virgulă mobilă.

În ultima decadă, substanțiale îmbunătățiri au avut loc în tehnologia compilatoarelor pentru extragerea și valorificarea paralelismului la nivelul instrucțiunilor (ILP). Majoritatea cercetărilor în acest domeniu s-au bazat pe

calculul de uz general, mai exact pe suta de benchmarkuri SPEC dezvoltate pentru asistarea în evaluarea comercială și marketing-ul variantelor desktop a sistemelor de calcul. În timp ce aceste aplicații au constituit un bun mijloc pentru direcționarea cercetării existente, ele nu cuprind toate elementele esențiale ale aplicațiilor multimedia și de comunicație.

Simulări efectuate folosind simulatorul *sim-outorder* [Bur97] cu parametrii implicați (vezi subcapitolul 3.2.6), din cadrul setului de instrumente SimpleScalar 3.0, au evidențiat câteva caracteristici prin care benchmark-urile SPEC'95 diferă de cele SPEC2000 [Pos00]. Rata globală de procesare, măsurată în instrucțiuni per ciclu de tact (IPC), este mai mică în cazul benchmark-urilor SPEC2000. Numărul mediu de instrucțiuni dinamice per salt variază între 4 și 6 în cazul benchmark-urilor SPEC'95 și respectiv între 4 și 8 în cazul testelor SPEC2000. Rata de miss în cache-ul de instrucțiuni (ICache) – de capacitate 16Ko, este foarte mică în cazul benchmark-urilor mai recente, exceptând testul *vortex*. Una din explicații o poate reprezenta dimensiunea foarte mică (sub 2500 linii) a codurilor sursă aferente unor programe de test (vezi *art*, *equaqe*, *mcf*). Rata de miss în ICache pe trei dintre benchmark-urile SPEC'95 (*compress*, *ijpeg* și *li*) este, de asemenea, foarte redusă, însă pe celelalte programe, aceasta este mult mai mare decât în cazul benchmark-urilor SPEC2000. La cache-urile de date rata de miss este mult mai mică pe testele SPEC'95 (valori între 2÷5%) comparativ cu cea rezultată pe SPEC2000 (valori între 4÷40%). O rată de miss ridicată se poate datora și optimizărilor realizate la nivelul compilatorului, care permit folosirea mai eficientă a regiștrilor procesorului prin eliminarea anumitor operații *load / store* mai simple și păstrarea doar a celor esențiale pentru algoritm. Spre exemplu, compilarea benchmark-ului *art* cu opțiunea de optimizare **-O2** provoacă o creștere a ratei de miss în cache-ul de date de la 15% (*art* compilat fără opțiuni de optimizare **-O0**) la 40% [Pos00]. O altă deosebire între cele două seturi de benchmark-uri se referă la fereastra dinamică de instrucțiuni (*register update unit*). Aceasta este mult prea ocupată în cazul benchmark-urilor SPEC2000 și necesită un număr mai mare de intrări (>16) pentru a evita diminuarea performanței globale de procesare.

Pornind de la aceste diferențe se poate observa că cele două suite de benchmark-uri SPEC'95 și SPEC2000 se completează reciproc în ceea ce privește o serie de caracteristici, motiv pentru care în studiile arhitecturale efectuate este bine să fie testate ambele suite. Un pas în această direcție a fost făcut și de către autorul acestei lucrări în [Vin05].

Din păcate, rezultatele generate pe SPEC2000, nu pot fi decât în extrem de mică măsură comparabile cu cele obținute pe benchmark-urile

SPEC'95. Unele teste au rămas din cele vechi, dar dimensiunea acestora este mult mai mare. Dintre aplicațiile pentru numere întregi, noi introduse, fac parte utilitarele de compresie fără pierderi: **gzip** și **bzip2** care vin să înlocuiască vechiul benchmark SPEC'95 **compress**. Alte aplicații vizează proiectarea optimă a circuitelor integrate (plasarea componentelor și autorutarea acestora) – **vpr**, un interpretor modificat de Perl – **perl2mbk**, un joc de șah – **crafty**, și o primă aplicație obiectuală (C++) ce tratează o problemă de grafică – **eon**. O parte din noile programe de test nu numai că solicită resurse hardware substanțiale față de predecesoarele SPEC'95, dar implică inclusiv un număr de trei compilatoare pentru execuția cu succes a benchmark-urilor SPEC2000 (gcc, g++ și f2c – necesar surselor scrise în Fortran – 77 sau 90). Condițiile de simulare presupun utilizarea unui harddisk de capacitate minimă de 1GByte și o memorie RAM de minim 256MByte. Din punct de vedere al frecvenței minime de procesare, trebuie specificat că unui procesor Sun Ultra10 la 300MHz îi sunt necesare două zile întregi pentru execuția completă a benchmark-urilor SPEC2000, iar procesoarele moderne (PentiumIV – 2.4GHz) necesită între câteva ore și o noapte întreagă, în funcție de parametrii de simulare și benchmark-ul utilizat.

Ca și concluzie se poate spune că noua suită de benchmark-uri încearcă să reflecte cât mai fidel tendința aplicațiilor și sistemelor viitoare de calcul, care fac uz de memorii de capacitate cât mai mare și rulează un timp îndelungat fără întrerupere. În continuare s-a trecut la descrierea individuală a programelor de test SPEC2000.

Cele 12 bechmark-uri din setul CINT2000, prezentate în tabelul 3.2, sunt scrise în C, cu excepția lui 252.eon care este scris în C++.

Benchmark	Limbajul în care a fost scris	Numar maxim de instrucțiuni dinamice executate
164.gzip	C	84.367.396.419
175.vpr	C	84.068.782.517
176.gcc	C	46.917.715.262
181.mcf	C	61.867.464.461
186.crafty	C	191.882.992.138
197.parser	C	546.749.947.290
252.eon	C++	80.614.082.986
254.gap	C	269.035.813.916
255.vortex	C	118.972.498.011
256.bzip2	C	143.565.170.182
300.twolf	C	346.485.090.453

Tabelul 3.2. Numărul maxim de instrucțiuni aferent benchmak-urilor SPEC2000 (de ordinul sutelor de miliarde)

161.zip

Comprimă / decomprimă un set de fișiere având capacitatea cumulată de 28 MBytes utilizând algoritmul de compresie fără pierderi, adaptiv Lempel-Ziv (LZ77). Toate operațiile se realizează la nivelul memoriei nemaifiind necesar accesul la discul magnetic. Fișierele folosite ca intrare sunt: un fișier imagine (TIFF) de dimensiune mare, un fișier log al unui webserver, un program binar, date aleatoare și un fișier sursă de tip *tar*.

175.vpr

Exemplu de program folosit în proiectarea optimă de circuite integrate. Implementează automat un circuit în FPGA. Benchmark-ul rezolvă probleme de alegere, poziționare și conectare a unităților de circuit pentru un anumit algoritm. Intrările benchmark-ului cuprind un fișier *netlist* al unui circuit, o descriere a arhitecturii FPGA în care circuitul este implementat, precum și informații de poziționare ale fiecărui element din *netlist*.

176.gcc

Compiler și optimizator de C bazat pe compilerul *gcc* versiunea 2.7.2.2., dedicat procesorului Motorola 88100. Simularea acestui benchmark presupune compilarea unor fișiere de intrare folosind multe din opțiunile de optimizare activate. Pentru simulare linia de comanda este linia tipică folosită la compilare sub Linux:

176.gcc *fișier_intrare.i* -o *detinatie*

Sunt disponibile 5 intrări, toate fiind cod preprocesat C (*.i*), cu o capacitate totală de 3.7MBytes:

integrate.i, *expr.i* – provin din sursele compilerului **gcc**.

166.i, *200.i*, *scilab.i* – au fost create prin concatenarea surselor FORTRAN ale unor benchmark-uri, apoi folosind translatorul **f2c** s-a produs codul C, care în final a fost preprocesat.

197.parser

Reprezintă un program de verificare sintactică a unui text scris în limba engleză (un set de fraze de capacitate ≤ 770 KBytes). Dicționarul acestuia este format din aproximativ 60.000 de cuvinte. Programul este robust și este capabil să treacă peste porțiuni de propoziții pe care nu le înțelege. La intrare este furnizat un singur fișier care conține o serie de propoziții. Pe lângă acesta pentru rulare programul mai are nevoie de dicționar. Acesta este compus din fișierul **2.1.dict** și directorul **words**.

Pentru simulare linia de comanda este:

197.parser *dictionar* –batch *fisier_intrare*

181.mcf

Aplicație care urmărește optimizarea problemelor de transport întâlnite la marile companii de transport în comun, și anume: minimizarea costurilor, crearea unor calendare orare incluzând durata sosirii vehiculelor, devieri de trasee.

186.crafty

Joc de șah. Înlocuiește practic aplicația **go**, de Inteligență Artificială, din setul SPEC'95. Datorită structurii aplicației, departe de a fi una liniară, poate fi utilizată cu succes în testarea structurilor de predicție implementate în procesoarele moderne. Rezolvă 5 partide pornind de la configurații diferite ale tablei de șah, cu căutări de „*adâncime*” variabilă în arborele de posibile mutări necesare găsirii celei optime.

252.eon

Reprezintă un program orientat – obiect care determină conturul / imaginea unor obiecte în format 3D. Pentru rezolvarea problemei sunt folosiți 3 algoritmi diferiți.

253.perlbnk

Reprezintă o versiune prescurtată a limbajului Perl versiunea 5.005_03, care rezolvă printre altele probleme de genul: conversia din format email – în format HTML, găsirea numerelor perfecte, generarea secvențelor de numere aleatoare.

245.gap

Constituie un program de calcul analitic în sfera algebrei discrete. Testul include câteva probleme de combinatorică, permutări, grupuri, etc.

255.vortex

Evidențiază modul de operare al unei baze de date obiectuale. Se lucrează cu trei tabele de date relaționale. Aplicația a fost modificată față de predecesora din suita SPEC'95 pentru a reduce influența operațiilor cu memoria externă, majoritatea acestora desfășurându-se la nivelul memoriei principale. Benchmark-ul este rulat de trei ori diferit, de fiecare dată sunt realizate joncțiuni cu tabele diferite și sunt adăugate, șterse sau modificate înregistrări ale acestei joncțiuni.

256.bzip2

Utilitar de compresie similar cu **gzip** doar că folosește la intrare o imagine, un fișier sursă și unul executabil, iar volumul total de date este aproape 20MBytes.

300.twolf

Program de proiectare a circuitelor integrate. Determină plasamentul și conexiunile globale dintre grupurile de tranzistoare care constituie microcipul.

După cum poate fi observat suita de teste nu este monotonă, acoperind o diversitate de aplicații, selectate cu atenție de corporația SPEC și agreate de marii producători de componente și sisteme de calcul: AMD, HP, Compaq (actualmente HP) și Intel.

În continuare sunt descrise benchmark-urile SPEC CFP2000, implementate în 3 limbaje de programare – C, Fortran 77 și Fortran 90; trebuie specificat că în principal se operează cu numere reale în dublă precizie. Numărul total de instrucțiuni dinamice executate în fiecare benchmark este tot de ordinul sutelor de miliarde.

<i>Benchmark</i>	<i>Limbajul în care a fost scris</i>	<i>Caracteristici</i>
168.wupwise	Fortran 77	Rezolvă una din cele mai importante ecuații din teoria fizicii cuantice referitoare la puternica interacțiune între componentele particulelor elementare (electroni).
171.swim	Fortran 77	Problemă de meteorologie folosită ca test standard pentru supercomputere. Rezolvă ecuațiile lichidelor subțiri utilizând ecuații cu diferențe finite.
172.mgrid	Fortran 77	Determină potențialul unui câmp electromagnetic. Este utilizat ca test standard pentru supercomputere.
173.applu	Fortran 77	Rezolvă probleme de fizica și dinamica fluidelor.
177.mesa	C	Biblioteca grafică de tipul 3D.
178.galgel	Fortran 90	Calcul numeric al parametrilor debitului lichidelor într-un spațiu închis.
179.art	C	Un model de rețea neurală folosit pentru recunoașterea obiectelor.
183.quake	C	Simulează propagarea undelor seismice elastice în văi largi și eterogene folosind metoda elementului finit.
187.facerec	Fortran 90	Procesare de imagini. Realizează implementarea unui sistem de recunoaștere a feței.
188.amp	C	Calcul chimice. Modelează un sistem bogat de molecule asociate cu cele biologice.

189.lucas	Fortran 90	Rezolvă probleme de teoria numerelor.
191.fma3d	Fortran 90	Utilizează metoda elementului finit în simularea răspunsului dinamic tranzient și inelastic al unor corpuri solide tridimensionale la sarcini bruște sau impulsiv aplicate.
200.sixtrack	Fortran 90	Soluționează probleme de fizică nucleară. Modelează un accelerator de particule și verifică stabilitatea pe termen lung a razelor într-o „ <i>gaură dinamică</i> ”.
301.apsi	Fortran 77	Calculează statistici asupra temperaturilor și gradelor de poluare. Este un benchmark utilizat în predicția vremii.

Tabelul 3.3. Caracteristicile benchmark-urilor SPEC CFP2000

După cum se poate observa și suita SPEC CFP2000 este extrem de variată, cu toate că unele benchmark-uri sunt puternic specializate. Sunt rezolvate o serie de probleme de fizică, chimie, matematică, meteorologie, procesare de imagini, inteligență artificială.

3.1.3. ALTE BENCHMARK-URI: SPEC JVM98, MEDIABENCH.

Odată cu dezvoltarea și creșterea în popularitate a Internetului (posibilitatea rezolvării a unei diverse game de probleme prin intermediul său), necesitatea unui limbaj de programare portabil devine tot mai acută. Caracteristici cum ar fi independența de platformă pentru portabilitate, un model orientat obiect, suport pentru multithreading, suport pentru programe distribuite și *automatic garbage collection* fac din Java un limbaj de programare foarte folosit în rândul dezvoltatorilor de programe. Prețul plătit de flexibilitatea Java se reflectă în performanța scăzută (viteză redusă de procesare) datorată gradului mare de abstractizare hardware pe care îl oferă.

Pentru portabilitate, codul sursă Java este tradus în arhitectura neutră *bytecode*, care poate fi executată pe orice platformă care suportă o implementare a mașinii virtuale Java (*JVM*). Majoritatea implementărilor JVM execută *bytecodul* Java fie prin interpretare, fie prin compilare *Just-in-Time (JIT)*. Deoarece atât interpretarea, cât și compilarea JIT necesită traducerea *runtime* a *bytecodului*, ambele conduc la o execuție înceată a aplicațiilor. Este evident că performanțele cresc atunci când sursele Java sunt compilate direct în cod mașină, dar cu prețul diminuării portabilității. O altă alternativă pentru executarea programelor Java este un procesor Java care implementează mașina virtuală Java hardware.

Benchmark-urile SPEC JVM98 – o suită de aplicații pentru numere întregi, au fost propuse pentru o mai bună înțelegere a „gâtuirilor” de performanță existente în mașinile virtuale Java și determinarea posibilelor optimizări. În continuare este realizată o scurtă descriere a fiecărui program de test din suită:

202_check

Este un benchmark utilizat pentru verificarea validității mașinii virtuale Java. Nu furnizează însă la ieșire informații privind metricile de performanță ale sistemului. Realizează o serie de teste simple privitoare la operațiile aritmetico-logice, de deplasare, rotire:

- verifică condițiile de încadrare în domeniul de definiție (erori de tipul „*out of range*”, în cazul tablourilor de date).
- creează superclase și verifică violările de acces la unele câmpuri.

Este un test foarte rapid comparativ cu celelate benchmark-uri.

222_mtrt

Reprezintă o modificare a benchmark-ului 205_raytrace, un desenator de contur care realizează portretul unui dinozaur. Se bazează pe un driver multithread, în care fiecare fir de execuție preia o scenă dintr-un fișier de intrare.

202_jess

Este un sistem expert scris în întregime în JAVA. Intenția testului Jess (Java Expert Shell System) este de a oferi apleturilor Java capabilitatea de a „*raționa*” de sine stătător prin aplicarea continuă a unui set de reguli. Benchmark-ul rezolvă un set de puzzle-uri.

201_compress

Reprezintă corespondentul benchmark-ului **129.compress** din suita SPEC'95 și folosește o metodă modificată a algoritmului Lempel-ZIV.

209_db

Realizează mai multe operații cu baze de date. Citește un fișier de 1 MBytes cu înregistrări și un alt fișier de 19 KBytes care conține un șir de operații (adăugare, ștergere, căutare, sortare) ce urmează a fi aplicate înregistrărilor respective.

222_mpegaudio

Benchmark-ul decompromă fișiere definite prin standardul ISO MPEG Layer-3. Prin tehnica de codare MP3 se realizează compresia semnalelor digitale audio până la un factor de 12, fără a pierde din calitatea sunetului perceput de urechea umană. Fișierul de intrare pentru decompresie are o dimensiune de 4MBytes.

228_jack

Reprezintă un parser Java bazat pe compilatorul realizat la Purdue University (PCCTS – Purdue Compiler Construction ToolSet), practic o versiune anterioară a copilatorului actual JavaCC. Testul de intrare al benchmark-ului îl reprezintă fișierul *jack.jack* care conține instrucțiuni pentru generarea benchmark-ului *jack*. Practic parser-ul se autogenează de mai multe ori (puternic recursiv).

213_javac

Este compilatorul Java pentru kit-ul de dezvoltare JDK 1.0.2.

În [Bow98] sunt introduse trei metrici pentru evaluarea benchmark-urilor SPEC JVM98. Prima o reprezintă frecvența fiecărui tip de instrucțiune, a doua o constituie adâncimea stivei necesară în execuția fiecărei instrucțiuni, iar cea de-a treia metrică calculează procentajul salturilor taken pe tipuri de instrucțiuni de salt condiționat. O concluzie certă după analiza benchmark-urilor SPEC JVM98 este că accesul de citire a memoriei (*load*) și operațiile aritmetico-logice sunt executate cu o frecvență mult mai mare decât scrierile în memorie (*store*). Practic mașina virtuală Java petrece majoritatea timpului citind operanzii din stivă (mașină bazată pe stivă). Lucrurile sunt oarecum normale dacă se ține cont de faptul că toate metodele în limbajul Java sunt virtuale, și după cum se poate vedea în capitolul 4, legarea dinamică realizată prin polimorfism se traduce prin accesarea tabelii de metode virtuale folosind o secvență de trei instrucțiuni load dependente urmate de un apel indirect de funcție. În ce privește adâncimea stivei rezultatele exprimă o concluzie interesantă: conținutul stivei poate fi stocat în 98% din timp folosind patru regiștri ai procesorului, evitându-se astfel accesul la memorie – mari consumatoare de timp. Benchmark-urile care nu se supun acestei concluzii sunt cele puternic recursive – 228_jack și 201_copress. Din punct de vedere al instrucțiunilor de salt condiționat, simulările cercetărilor [Bow98] au arătat că, cel puțin jumătate din ele au o puternică tendință de polarizare (taken sau notaken).

În ultima decadă, substanțiale îmbunătățiri au avut loc în tehnologia compilatoarelor pentru extragerea și valorificarea paralelismului la nivelul instrucțiunilor (ILP). Majoritatea cercetărilor în acest domeniu s-au bazat pe calculul de uz general, mai exact pe suita de benchmarkuri SPEC dezvoltate pentru asistarea în evaluarea comercială și marketing-ul variantelor desktop a sistemelor de calcul. În timp ce aceste aplicații au constituit un bun mijloc pentru direcționarea cercetării existente, ele nu cuprind toate elementele esențiale ale aplicațiilor multimedia și de comunicație.

În același timp, o mulțime de arhitecturi de microprocesor au apărut având structuri VLIW și SIMD care se potrivesc perfect necesităților compilatoarelor moderne. Majoritatea acestor procesoare sunt destinate suportului aplicațiilor dedicate (“*embedded applications*”) cum sunt cele multimedia sau comunicații, mai degrabă decât pentru sisteme de uz general. Din nefericire, există un decalaj între “comunitatea compilatoriștilor” și dezvoltatorii de aplicații dedicate. **MediaBench** constituie o suită de benchmarkuri, instrumente software pentru evaluarea și sintetizarea sistemelor multimedia și de comunicații, dedicate umplerii acestui “*gap*” [Lee97]. Majoritatea acestor aplicații implică optimizări manuale a rutinelor scrise în limbaje de asamblare, în special pentru cele care cuprind bucle imbricate. Ideea de bază în implementarea acestor optimizări o constituie identificarea codului din afara buclelor, a dependențelor de date, a buclelor vectorizabile și aplicarea tehnicilor de trace scheduling sau software pipelining.

Principalele scopuri urmărite de MediaBench sunt:

- ☐ reprezintă cu acuratețe o (bază de lucru) “*workload*” - o bază de simulare a noilor sisteme multimedia și de comunicație (un instrument soft de evaluare a sistemelor).
- ☐ se focalizează pe aplicații portabile scrise în limbaje de nivel înalt întrucât arhitecturile de procesoare și dezvoltatorii de software migrează înspre respectiva direcție.
- ☐ stabilirea cu precizie a avantajelor MediaBench comparativ cu alternativele existente (SPEC-int, SoftFloat).

Noile clase de arhitecturi de procesoare destinate suportului aplicațiilor multimedia combină o parte din caracteristicile procesoarelor de semnal (“*DSP devices*”) - posibilitatea de a executa concurrent mai multe operații, cu caracteristicile procesoarelor de uz general care constituie “*good targets*” pentru compilatoare (număr mare de seturi de regiștri generali, suport pentru execuția condiționată și speculativă).

MediaBench 1.0 reprezintă o suită de 19 aplicații complete, disponibile pe Internet oricărui utilizator, scrise în limbaje de nivel înalt și compilate de către multiple compilatoare independente pentru arhitecturi de

procesare diferite. Plaja de aplicații cuprinde procesare de imagini, compresii de sunet și imagine, comunicații, procesare de semnal. Componentele MediaBench sunt:

- ⌚ **JPEG:** JPEG este o metodă de compresie standardizată pentru imagini color sau în scală de gri. Compresia se realizează “cu pierderi” întrucât imaginea rezultată (după comprimare) nu este chiar identică cu cea inițială. Două aplicații diferite sunt derivate din codul sursă al JPEG: **cjpeg** – care realizează compresia și **djpeg** care realizează decompresia. De reținut că benchmark-ul JPEG este comun (identic ca și cod sursă) atât suitei MediaBench cât și SPECInt’95.
- ⌚ **MPEG:** MPEG2 reprezintă standardul actual predominant pentru transmisii video digitale de înaltă calitate. Nucleul de calcul îl constituie o transformată cosinus pentru **codare** (*mpeg2enc*) respectiv transformata inversă pentru **decodare** (*mpeg2dec*).
- ⌚ **GSM:** Se bazează pe standardul european GSM 06.10 pentru transcodare la rată maximă a vorbirii (sunetelor - *speech*), prI-ETS 300 036, care folosește semnalul rezidual de excitație pentru codificarea predicției pe termen lung la 13kbit/s.
- ⌚ **PGP:** PGP este folosit la realizarea semnăturilor electronice. Se bazează pe criptarea mesajelor cu ajutorul funcțiilor de dispersie.
- ⌚ **Ghostscript:** Reprezintă un interpretor de limbaj PostScript.
- ⌚ **Mesa:** Constituie o librărie grafică 3-D, similară cu OpenGL.
- ⌚ **RASTA:** Reprezintă un program pentru recunoașterea vorbirii care suportă următoarele tehnici: PLP, RASTA și Jah-RASTA. Tehnicile tratează simultan “zgomotul suplimentar” și “distorsiunea spectrală” prin filtrarea traiectoriilor temporale a unor spectre de bandă critice transformate neliniar.
- ⌚ **ADPCM:** Reprezintă una din cele mai simple și mai vechi forme de codificare audio și se bazează pe modulația adaptivă a semnalului audio.

Una din problemele care se pun este dacă programele de test MediaBench sunt cantitativ diferite de SPECInt pentru un număr de caracteristici de performanță pe care arhitecții le consideră importante. În urma unei analize statistice comparative a rezultatelor simulărilor efectuate pe cele două suite de benchmark-uri: MediaBench și SPECInt, există o diferență semnificativă în ce privește patru parametri de performanță, în favoarea primei suite. **SPECInt necesită aproape cu 300% mai multă lărgime de bandă decât MediaBench.** Doar programul de test *pegwitdec* surclasează media benchmark-urilor SPECInt. Acest trafic poate fi o consecință directă a unor rate de hit relativ scăzute la cache-ul de instrucțiuni și apare în cazul procesoarelor dedicate caracterizate de busuri limitate și capacitate redusă a memoriei. Ceilalți parametri care exprimă o

performanță superioară în cazul benchmark-urilor Media sunt: **rata de procesare** (instrucțiuni/ciclu), **rata de hit în cache-ul de instrucțiuni** și **rata de hit pentru citirile din cache-ul de date** (%).

Se știe că unul din cele mai stringente scopuri urmărite de proiectanții de sisteme dedicate este de a reduce costul, în mare parte realizat prin reducerea dimensiunii modulelor componente. Dacă arhitecții de procesoare ar dori să realizeze cipuri (unități centrale) pentru sisteme de comunicații și multimedia pe baza rezultatelor simulărilor efectuate pe benchmark-urile SPECInt, atunci cache-ul de instrucțiuni ar fi cu 18% mai mare decât dacă s-ar urma indicațiile de proiectare generate de simulările pe MediaBench. În ceea ce privește **acuratețea predicției și influența numărului de unități aritmetico-logice asupra ratei de procesare**, rezultatele simulărilor obținute sunt aproximativ identice, indiferent de benchmark-urile utilizate.

3.2. DEZVOLTAREA DE SIMULATOARE SUB MEDIUL SIMPLESCALAR UTILIZÂND BENCHMARK-URILE SPEC.

Istoria procesoarelor contrapune două paradigme pentru creșterea performanței, bazate pe software și respectiv pe hardware. În procesul de proiectare al procesoarelor, aferent generațiilor viitoare, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsă legătură cu aplicațiile potențiale. Se pornește de la o arhitectură de bază (generică), puternic parametrizată, care este modificată și îmbunătățită dinamic, prin simulări laborioase pe benchmark-uri reprezentative. Procesoarele se proiectează odată cu compilatoarele care le folosesc iar relația dintre ele este foarte strânsă: compilatorul trebuie să genereze cod care să exploateze caracteristicile arhitecturale, altfel codul generat va fi ineficient. **Simulatorul dedicat unei arhitecturi de calcul** constituie un instrument software (aplicație/program) utilizat în exploatarea / cercetarea / și îmbunătățirea performanțelor unei microarhitecturi. De obicei funcționarea microarhitecturii se simulează la nivel de ciclu mașină permițând vizualizarea tuturor resurselor hardware la finele fiecărui ciclu. Metodologia de simulare poate fi de două tipuri:

➤ **Execution driven simulation**

- ⇒ caracterizată de cunoașterea în fiecare moment (ciclu "pipe") a conținutului resurselor arhitecturale (regiștri, locații de memorie, unități funcționale).
- ⇒ Simularea se face foarte detaliat, la nivel de ciclu de execuție al procesorului.
- ⇒ **Outputs:** - conținutul resurselor, gradul de încărcare al acestora, rate de procesare, de hit etc.
 - fișiere **trace** care conțin toate instrucțiunile mașină ale programelor de test în ordinea în care se execută.

➤ **Trace driven simulation**

- ⇒ analizează secvențial toate instrucțiunile din trace-urile generate de simulatorul bazat pe execution driven, cu scopul de a determina instanța optimală a arhitecturii - *procesorul ce urmează a fi implementat în hardware*. TDS se pretează la *simularea cache-urilor* de date și instrucțiuni, mecanismelor de memorie virtuală etc., datorită faptului că oferă pattern-uri reale de adrese, în urma execuției unor programe reprezentative.

3.2.1. CE ESTE "SIMPLESCALAR TOOL SET" ?

Reprezintă o colecție de instrumente software, pusă la dispoziția cercetătorilor în arhitecturi moderne de calcul, și cuprinde: compilatoare, asamblatoare, link-editoare, simulatoare și instrumente de vizualizare a unei arhitecturi (super)scalare simple, generice (Exemple: arhitecturi **PISA**, **Alpha AXP**, **ARM**). Arhitectura Alpha AXP este un procesor RISC dezvoltat de firma DEC (fostă COMPAQ, actualmente HP), iar arhitectura SimpleScalar PISA (Portable ISA) se va detalia mai târziu (vezi subcapitolul 3.4). Setul mai cuprinde un depanator la nivel de cod sursă al benchmark-ului simulat (**DLite!**) și un generator de trace-uri la nivel de pipe a instrucțiunilor (afereț doar celui mai detaliat simulator). Setul de instrumente "*SimpleScalar*" este distribuit gratuit (poate fi găsit și descărcat pe site-ul web "<http://www.cs.wisc.edu/~mscalar/simplecalar.html>") și oferă posibilitatea simulării de tip *execution driven*, cât mai detaliată și de înaltă performanță a celor mai moderne microprocesoare. Programele simulate reprezintă parte integrantă a setului de instrumente și sunt programe de test precompilate (format cod obiect) pentru arhitecturile PISA și Alpha, și o parte din benchmark-urile SPEC'95. Cei interesați pot dispune

de compilatorul GNU GCC (precum și de utilitarele aferente), care permite fiecărui cercetător să compileze/asambleze/linkediteze propriile programe de test scrise pentru arhitectura SimpleScalar. Setul de instrumente "SimpleScalar" și modulele de portare a compilatorului GNU pe diverse platforme a fost scris de Todd Austin - începând cu anul 1994 pe când era cercetător la universitatea din Wisconsin-Madison, actualmente fiind membru al echipei de cercetători în paralelism la nivelul instrucțiunilor al firmei Intel Corporation. "SimpleScalar" este susținut în continuare de Doug Burger (autorul, în cea mai mare parte, a documentației) și de Todd Austin. Compilatorul GNU și instrumentele software adiționale (biblioteci, translatoare din Fortran în C) a fost scris de "Free Software Foundation".

3.2.2. AVANTAJELE UTILIZĂRII "SIMPLESCALAR TOOL SET".

⇒ **Distribuirea gratuită**, inclusiv a surselor C a tuturor modulelor componente. Printre adresele de web utile se numără:

- <ftp://ftp.cs.wisc.edu/sohi/Code/SimpleScalar/simplelim.tar>
- <http://www.cs.wisc.edu/~mscalar/simplecalar.html>

La prima adresă menționată pot fi găsite pe lângă sursele simulatoarelor și următoarele fișiere în format arhivă, fiecare cuprinzând anumite elemente (necesare sau opționale) ale setului:

- ☐ **Simplelim.tar.gz** - cuprinde sursele simulatoarelor procesorului virtual SimpleScalar, scripturi și macrodefiniții ale setului de instrucțiuni, sursele C și cod obiect (obținute după compilare/asamblare/linkeditare) a unor mici programe de test (nu SPEC). Este necesar pentru instalarea setului de instrumente și include o documentație exhaustivă asupra setului SimpleScalar (**hack_guide.pdf** sau **hack_guide.ps**).
- ☐ **Simpleutils.tar.gz** - conține sursele unor utilitare GNU (versiunea 2.5.2) portate pe arhitectura SimpleScalar. *Nu sunt necesare pentru execuția simulatoarelor dar sunt necesare la asamblarea și linkeditarea propriilor benchmark-uri scrise pentru arhitectura SimpleScalar.*
- ☐ **Simpletools.tar.gz** - conține sursele și bibliotecile compilatorului GNU (*gcc* 2.6.3, *glibc* 1.0.9 și translatorul din Fortran în C *f2c*) necesare compilării benchmark-urilor proprii pentru arhitectura

SimpleScalar (codul rezultat este format mnemonică de asamblare).
Nu sunt necesare pentru execuția simulatoarelor.

☐ **Simplebench.big.tar.gz** - cuprinde o serie de benchmark-uri SPEC'95, în cod obiect, compilate pentru arhitectura SimpleScalar rulând pe o stație cu ordinea octeților *big endian*.

☐ **Simplebench.little.tar.gz** - cuprinde aceleași benchmark-uri SPEC'95, în cod obiect, compilate pentru arhitectura SimpleScalar rulând pe o stație cu ordinea octeților *little endian*.

Ordinea octeților reprezintă o convenție de numerotare de către procesor a octeților din interiorul unui cuvânt de 32 de biți astfel încât octetul cu numărul cel mai mic este fie cel mai din stânga fie cel mai din dreapta. Procesoarele MIPS pot opera fie cu ordinea octeților:

big-endian

Byte #			
0	1	2	3

little-endian

Byte #			
3	2	1	0

sau

- ⇒ **Flexibilitatea sporită** (plaja largă de valori a parametrilor simulatoarelor precum și mulțimea de simulatoare de arhitecturi dezvoltate, începând cu unul funcțional foarte rapid dar extrem de simplu, cu execuție *in order - sim-fast* - și până la unul extrem de detaliat, cu execuție *out of order* și *speculativă*, dotat cu un sistem ierarhizat de memorii multinivel și un predictor avansat "*state of the art*" aferent instrucțiunilor de salt- **sim-outorder**).
- ⇒ **Portabilitatea**: simulatoarele rulează pe majoritatea platformelor UNIX pe 32 și 64 de biți și chiar pe platformă WinNT (sursele sunt compilate sub MS Visual C++), cu condiția ca uneltele software GNU să fie instalate pe calculatorul gazdă. Pentru modificarea/compilarea/link-editarea simulatoarelor/benchmark-urilor proprii poate fi folosit emulatorul Cygwin. Majoritatea utilizatorilor și dezvoltatorilor setului SimpleScalar lucrează pe sisteme de operare Linux pe mașini x86.
- ⇒ **Extensibilitatea**: pachetul SimpleScalar Tool Set cuprinde toate sursele permițând dezvoltarea ulterioară a setului - îmbunătățirea, atașarea de noi module (Exemple: simulator pentru predicția valorilor centrat pe diverse resurse, reutilizarea dinamică a instrucțiunilor, trace cache); este bine documentat. Setul de instrucțiuni proiectat suportă eventuale *adnotări* (câmp suplimentar) - modificări post-compilare - în fișierele asamblare, fără a fi necesară o recompilare a codului mașină, foarte util în introducerea informațiilor de profil aferente fiecărei instrucțiuni și necesar în implementarea diverselor arhitecturi sau tehnici novatoare de

procesare (vezi predictorul hibrid cu selecție bazată pe aritate din subcapitolul 5.3.3.1).

3.2.3. DEZVANTAJE ÎN UTILIZAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR".

- ❑ **Suportă doar seturile de instrucțiuni ale anumitor arhitecturi:** SimpleScalar PISA și Alpha AXP, ARM7ISA (nu și pentru Intel IA-64, etc).
- ❑ **Deocamdată SimpleScalar simulează doar medii uniprosesor;** în funcție de benchmark-ul simulat (și numărul de instrucțiuni) simularea poate dura foarte mult (vezi 3.1.2 privitor la simularea benchmark-urilor SPEC2000).
- ❑ **Simularea instrucțiunilor se realizează la nivel de utilizator** (nu este simulată execuția instrucțiunilor în interiorul sistemului de operare). Protocolul de tratare a **apelurilor sistem** (întreruperi) este următorul:
 - ❑ Decodificarea apelului sistem.
 - ❑ Copierea argumentelor (dacă există) în memoria simulatorului (**sim-***).
 - ❑ Executarea apelului sistem de către sistemul de operare (rutina de tratare a întreruperii).
 - ❑ Copierea rezultatelor dacă există în memoria programului simulat (**benchmark-ul SPEC**).

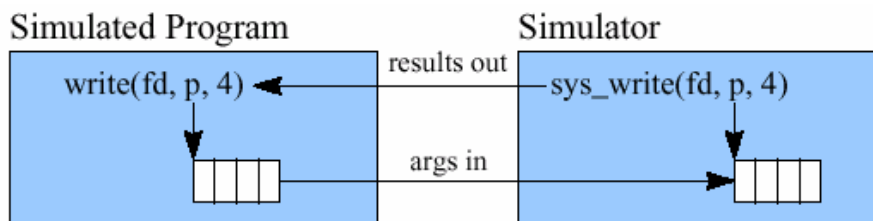


Figura 3.1. Protocolul de tratare a **apelurilor sistem**

- ❑ Modulul **syscall.c** implementează un subset de apeluri sistem specifice Unix. Pentru adăugarea oricărui nou apel sistem sau pentru portarea SimpleScalar pe un sistem de operare nou, este necesară implementarea software a rutinei de tratare în modulul **syscall.[hc]**.

3.2.4. UTILITARE GNU.

GNU reprezintă abrevierea expresiei "**GNU's Not Unix**". Proiectul GNU a startat în anul 1984 cu scopul de a dezvolta un sistem de operare asemănător UNIX-ului dar care să fie distribuit gratuit oricărui individ care dorește să-l utilizeze sau să-l dezvolte în continuare. Sistemul de operare GNU nu reprezintă o colecție de instrumente software proprii (specifice doar) GNU, ci cuprinde și alte programe existente ca "*free software*". Nucleul sistemului de operare este LINUX. Editorul de text folosit este TeX, realizat de Donald Knuth. Interfața bazată pe ferestre este identică cu cea a sistemului XWindows, realizată de Bob Scheifler. Componentele esențiale ale unui sistem de operare: compilatorul (*gcc*), asamblorul (*gas*) și link-editorul (*gld*) sunt proprii GNU. Pentru ca sistemul de operare să fie complet sunt necesare pe lângă instrumente de programare și un interpretor PostScript, biblioteci de C etc.

Chiar dacă nu există nici un avantaj tehnic al GNU asupra Unix, există totuși un avantaj social, permițând utilizatorilor să coopereze la utilizarea și dezvoltarea continuă a sistemului de operare și unul etic prin respectarea libertății individului. Creșterea în fiabilitate și viteză a unor componente (programe) GNU față de cele similare Unix s-a obținut prin tratarea diferită a fișierelor. Astfel, fișierele de dimensiuni foarte mari sunt încărcate în întregime în memoria principală, fără a mai avea probleme din punct de vedere al interfețelor de intrare/ieșire la citirea (parcurgerea) secvențială a conținutului.

GNOME (GNU Network Object Model Environment) reprezintă varianta desktop a proiectului GNU. Început de Miguel de Icaza în 1997 și continuat cu sprijinul companiei Red Hat Software, GNOME stabilește o serie de facilități la nivel desktop, dar folosind programe software exclusiv disponibile gratuit. Prezintă avantaje tehnice precum suportul unei varietăți de limbaje (nu doar C++).

Dintre utilitarele GNU amintim câteva și rolul îndeplinit de fiecare din ele:

- **ld** - link-editor GNU.
- **as** - asamblorul portabil GNU.
- **ar** - utilitar folosit la crearea, modificarea și extragerea din arhive.
- **objcopy** - copiază și translatează fișiere obiect.
- **objdump** - afișează informații din fișiere obiect.
- **size** - listează dimensiunea unei secțiuni a unui fișier obiect sau fișier arhivă.
- **windres** - compilator pentru fișiere de resurse Windows.

- **gprof** - afișează informații de profil.

Majoritatea acestor programe utilizează **BFD** ("*Binary File Descriptor*") - biblioteca binară a descriptorilor de fișiere), pentru a realiza o manipulare a fișierelor "*low-level*". Utilitățile GNU pot fi portate pe majoritatea variantelor de UNIX precum și pe sistemele Windows având procesoare Intel.

3.2.5. INSTALAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR".

Pentru început se prezintă pașii efectuați în instalarea setului de instrumente *SimpleScalar 3.0* [Bur97] pornind de la arhiva **simplesim-3.0b.tar.gz** disponibilă la adresa <http://www.cs.wisc.edu/~mscalar/simplecalar.html> și de la kit-ul de instalare **Cygwin** (aferent emulatorului de Linux pe sistemul de operare WinNT). Pașii descriși în continuare sunt valabili pe sistemele de operare Windows NT 4.0, Windows 2000 Profesional și Windows XP.

- ☐ Se descarcă de la adresa amintită anterior fișierul arhivă în format **tar.gz** "*simplesim-3.0b.tar.gz*". Se extrage din acesta conținutul său: directorul **SIMPLESIM-3.0/*.*** cu fișierele componente, dintre care și directorul **target-pisa**. Fișierele acestui director (care descriu funcționarea unei arhitecturi **pisa**) sunt copiate în directorul părinte(SIMPLESIM-3.0).
- ☐ Se instalează emulatorul de linux CYGWIN. După instalarea corectă se execută dublu-click pe iconița aplicației CYGWIN (sau *Start -> Programs -> Cygnus Solutions -> Cygwin Bash Shell*) și se intră în prompter-ul de linux. În acest moment (la primul acces după instalare) se generează automat un director **home** și un subdirector cu numele **user**-ului de pe stația respectivă (fie **Administrator**) în directorul CYGWIN. În directorul **home/Administrator** se copiază SIMPLESIM-3.0 cu tot conținutul său.
- ☐ Revenind în CYGWIN se tastează comanda **\$cd SIMPLESIM-3.0** și se ajunge în directorul tocmai copiat. Se tastează în continuare succesiunea de comenzi:

```
$make config-pisa
și
```

```
$make
```

Prima comandă determină instalarea componentelor (în vederea compilării instrumentelor setului pentru arhitectura *pisa*). Execuția

comenzii **make** poate genera o eroare la compilare deoarece tipul returnat de funcția *myrand()* din fișierul **misc.c** care apelează la rândul ei funcția *random()* diferă de cel returnat de funcția standard definită în **stdlib.h** (din **Cygwin\usr\include**). Eroarea se corectează modificând tipul funcției *random()* în **misc.c** (din **long** în **int**). Execuția din nou a comenzii **make** se va încheia cu mesajul "**My work is done here...**" ceea ce demonstrează compilarea / asamblarea / link-editarea cu succes a simulatoarelor setului SimpleScalar 3.0.

- ☐ Înainte de simularea propriu-zisă (lansarea în execuție a unuia din fișierele **sim-*.exe**) trebuie copiat fișierul **cygwin1.dll** din **Cygwin\bin** în directorul **Cygwin\home\Administrator\Simplesim-3.0**. În același director se vor copia și benchmark-urile SPEC (executabilele și intrările corespunzătoare). În caz contrar în momentul simulării trebuie specificată calea spre programele de test folosite în simulare.

Pentru rularea unui program de test (se consideră directorul curent ca fiind cel ce conține simulatorul - executabilul) se tastează secvența:

- ☐ din **prompter de DOS** sau **fereastra RUN din Windows**:

<Nume_simulator> <Cale_benchmark> [opțiuni]

- ☐ din **linux** (sau **Cygwin**):

(sim_path/sim-bin bmark_path/benchmark_bin input_set_path/input_set > output_file) > err_out_file

Exemplu:

```
./sim-bpred -redir:sim apsi_simout.res -redir:prog apsi_progout.res -max:inst
5000000 apsi.ss < apsi.in
```

Observații:

1. În loc de **sim-bpred** poate fi pus orice simulator.
2. Benchmark-ul **apsi.ss** poate fi înlocuit cu oricare altul din suita SPEC dar cu intrarea aferentă (**input_set_path/input_set**). Benchmark-urile SPEC2000 presupun o linie de comandă mai complexă, o exemplificare în acest sens fiind făcută ulterior pe parcursul acestui capitol.

Opțiunile sunt specifice fiecărui simulator și opționale. În subcapitolul 3.2.6. se vor detalia separat la fiecare din simulatoare. Există totuși un număr de 6 opțiuni disponibile tuturor simulatoarelor care permit afișarea mesajelor ajutătoare în funcționarea simulatorului, determină startarea afișorului de mesaje specifice depanatorului, startează execuția interactivă în mediul depanare, forțează încheierea execuției programului, citește și

încarcă / scrie parametrii simulatorului dintr-un / într-un fișier de configurare.

3.2.6. SIMULATOARELE SETULUI DE INSTRUMENTE "SIMPLESCALAR 3.0".

Sim-Fast	Sim-Safe	Sim-Profile	Sim-Cache/ Sim-Cheetah	Sim-Outorder
- 420 linii - functional	- 350 linii - functional - verificări - aliniere date - pe cuvânt	- 900 linii - functional - informații - de stare	- < 1000 linii - functional - informații - de stare - privind - cache-urile	- 3900 linii - performant - OoO issue - branch pred. - mis-spec. - ALU - cache - TLB

Figura 3.2. Studiu comparativ asupra simulatoarelor setului SimpleScalar

1. **Simulatorul funcțional** (*sim-fast*) - este cel mai rapid, elementar și puțin detaliat simulator. Simulează execuția **secvențială** a instrucțiunilor, nu presupune un sistem ierarhic de memorie, nu verifică dependențele dintre instrucțiuni. O versiune similară a acestui simulator este *sim-safe* care îndeplinește sarcinile simulatorului *sim-fast*, iar suplimentar verifică alinierea corectă a cuvintelor (instrucțiuni/date) și drepturile de acces pentru fiecare referință la memorie. Nici unul din cele două simulatoare nu acceptă parametrii suplimentari în linia de comandă. Ambele versiuni sunt foarte simple (codul sursă având mai puțin de 300 de linii) constituindu-se într-un excelent punct de start pentru înțelegerea funcționării interne a simulatorului.
2. **Simulatorul de cache-uri** (*sim-cache*) - reprezintă un simulator funcțional ideal pentru simularea rapidă a arhitecturilor care utilizează un sistem ierarhic de memorie, considerându-se suplimentar că timpul de acces la cache nu este relevant în ce privește performanța obținută. Pe lângă parametrii generali amintiți mai sus, *sim-cache* acceptă argumente în linia de comandă referitoare la: dimensiunea primului, respectiv celui

de-al doilea nivel de cache de instrucțiuni/date, dimensiunea buffer-ului de translatare a adreselor datelor/instrucțiunilor. Există opțiuni care permit golirea cache-urilor în cazul apelurilor sistem sau care determină remaparea instrucțiunilor pe 64 de biți la instrucțiuni echivalente pe 32 de biți. Parametrii de configurare a cache-ului trebuie să cuprindă: numele cache-ului, care trebuie să fie unic, numărul de seturi din cache, dimensiunea blocului (pentru bufferele TLB - dimensiunea paginii), asociativitatea cache-ului (putere a lui 2) și politica de înlocuire a blocurilor din cache (l | f | r), unde l = *LRU*, f = *FIFO*, r = *random*. Dimensiunea cache-ului se obține ca produs între numărul de seturi, gradul de asociativitate și dimensiunea în octeți a fiecărui bloc. Pentru a avea un nivel de cache unificat în ierarhie (arhitectură Princeton), "se pointează" la cache-ul de instrucțiuni cu numele cache-ului de date, pe nivelul corespunzător.

Exemplu:

- 1) **sim-cache** -redir:sim applu_simout.res -max:inst 5000000 -cache:il1 il1:128:64:1:l applu.ss<applu.in
- 2) **sim-cache** -redir:sim applu_simout.res -max:inst 5000000 -cache:il2 dl2 applu.ss<applu.in (cache unificat pe nivelul 2)

3. **Simulatorul de cache-uri cheetah** (*sim-cheetah*) - permite generarea rezultatelor simulării pentru configurații de cache multiple, printr-o singură simulare. Nucleul *Cheetah* simulează eficient cache-uri complet asociative, precum și o politică de înlocuire (uneori) optimă (algoritmul MIN al lui Belady - blocul cel mai târziu referit în viitor va fi selectat spre înlocuire). Politică se dovedește optimă pentru fluxuri de instrucțiuni (fișiere) *read-only*. Pentru cache-urile cu modalitate de scriere *write-back* algoritmul de înlocuire nu este întotdeauna optim (de exemplu poate fi mult mai costisitor să se înlocuiască blocul cel mai târziu referit în viitor dacă blocul trebuie scris în memoria principală ("*murdar*"), față de un bloc "*curat*" referit în viitor puțin mai devreme decât blocul "*murdar*" anterior). Nucleul *Cheetah* a fost conceput ca o bibliotecă de sine stătătoare, rezidentă în directorul **libcheetah/**. *Sim-cheetah* acceptă următorii parametri în linia de comandă, suplimentari celor generali, disponibili tuturor simulatoarelor din setul *SimpleScalar*:

Opțiune	Comanda realizată
-refs [inst data unified]	Specifică tipul fluxului (date/instrucțiuni) analizat.
-C [fa sa dm]	Tipul cacheului: complet asociat, set asociativ, mapat direct.
-R [lru opt]	Politica de înlocuire a blocurilor conflictuale din cache.

-a <sets>	\log_2 din limita inferioară a numărului de seturi simulate simultan.
-b <sets>	\log_2 din limita superioară a numărului de seturi.
-n <assoc>	Asociativitatea maximă de analizat.
-in <interval>	Intervalul, exprimat în dimensiunea cache-ului, la care se afișează rezultate, în cazul în care cache-ul este complet asociativ.
-M <size>	Dimensiunea maximă a cache-ului care este supusă atenției.
-C <size>	Dimensiunea maximă a cache-ului mapat direct care este analizată.

Tabelul 3.4. Opțiuni specifice *sim-cheetah*

Ambele simulatoare sunt ideale pentru studiul "high level" al cache-urilor, fără a ține cont de timpul de acces la cache (doar rata de miss fiind analizată). Pentru a măsura însă efectul organizării cache-ului asupra timpului de execuție al programelor de calcul trebuie utilizat simulatorul sim-outorder, mult mai complex, la nivel de execuție.

4. Predictorul de salturi (*sim-bpred*) – generează informații statistice privitoare la acuratețea de predicție, numărul și caracteristici ale instrucțiunilor de salt din programele de aplicație. Predicția salturilor este specificată prin alegerea flag-ului **-bpred** urmat de unul din următoarele 6 argumente:

- *nottaken* – saltul va fi prezis întotdeauna **not taken**;
- *taken* – saltul va fi prezis întotdeauna **taken**;
- *perfect* – predictorul va fi perfect din punct de vedere a acurateții de predicție;
- *bimod* – predictor bimodal, folosind un BTB ('branch target buffer') având automate de predicție pe 2 biți;
- *2lev* – predictor adaptiv pe 2 niveluri;
- *comb* – predictor combinat (bimodal și adaptiv pe 2 niveluri);

În funcție de tipul predictorului argumentele specifice sunt prezentate mai jos:

- ☐ **-bpred:bimod** <size> ⇔ stabilește dimensiunea tabeli predictorului bimodal la <size> intrări.
- ☐ **-bpred:2lev** <1size><2size><hist_size><xor> ⇔ precizează un predictor adaptiv pe două niveluri. Modelul de organizare este descris în figura 3.3.

- **<l1size>** - specifică numărul de intrări în tabela aflată pe primul nivel al predictorului (Exemplu: $N=1$ implică un registru de istorie globală, specific predictorilor GAg sau GAp).
- **<l2size>** - reprezintă numărul de intrări în tabela aflată pe cel de-al doilea nivel al predictorului.
- **<hist_size>** - identifică numărul de biți de istorie (globală) utilizați în procesul de predicție.
- **<xor>** - permite folosirea funcției de dispersie *xor* (dintre *adresa de salt* și *istoria salturilor*) pentru determinarea indexului de adresare în tabela de predicție aflată pe nivelul al doilea (selectarea *automatului de predicție* corespunzător).

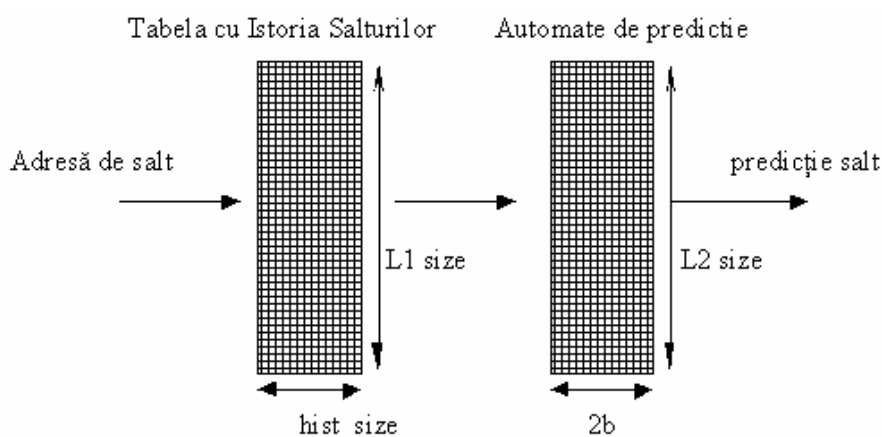


Figura 3.3. Structură de predictor adaptiv pe 2 niveluri

Tabelul 3.5 prezintă parametrii corespunzători unor scheme de predicție moderne. Valorile implicite pentru cei patru parametri anterior amintiți sunt **1** (**<l1size>**), **1024** (**<l2size>**), **8** (**<hist_size>**) și **0** (**<xor>**), respectiv.

PREDICTOR	L1_size	Hist_size	L2_size	Xor
GAg	1	W	2^W	0
GAp	1	W	$>2^W$	0
PAg	N	W	2^W	0
PAp	N	W	2^{N+W}	0
Gshare	1	W	2^W	1

Tabelul 3.5 Scheme moderne de predicție corelate pe două niveluri

- ☒ **-bpred:btb** <sets> <assoc> ⇔ configurează un predictor de tip BTB având <sets> seturi și gradul de asociativitate <assoc>. Valorile implicite sunt **512** seturi și asociativitate **4-way**.
- ☒ **-bpred:spec_update** <stage> ⇔ permite actualizarea speculativă a schemelor de predicție în fazele de decodificare sau de scriere rezultat în setul de regiștrii generali (<stage>=[ID/WB]). Prin nesetarea acestui parametru actualizarea predictorului se face nespeculativ în faza **Commit** (retragerea regiștrilor din bufferul de reordonare).

Exemplu:

- 1) **sim-bpred** -redir:sim applu_simout.res -max:inst 5000000 -bpred 2lev -bpred:2lev 1 1024 10 0 applu.ss < applu.in
- 2) **sim-bpred** -redir:sim applu_simout.res -max:inst 5000000 -bpred bimod -bpred:btb 512 4 applu.ss < applu.in

5. **Simulatorul de informații de profil** (*sim-profile*) - generează detaliat informații despre adrese și clase de instrucțiuni, simboluri (adrese de date/instrucțiuni), accese la memorie, instrucțiuni de salt.

Opțiune	Comanda realizată
-iclass	Tipul claselor de instrucțiuni (ALU, Branch, etc).
-iprof	Tipul instrucțiunilor (bnez, addi etc).
-brprof	Tipul clasei instrucțiunilor de salt (salturi directe/indirecte, apeluri de subrutină, salturi condiționate).
-amprof	Tipul modului de adresare (directă/indirectă/indexată).
-segprof	Zona de date accesată (statică, dinamică).
-tsymprof	Informații privind execuția (funcții utilizate, depanarea în interiorul acestora).
-dsymprof	Informații privind zona de date.
-taddrprof	Informații privind execuția la o anumită adresă.
-all	Setează pe <i>True</i> toate opțiunile.

Tabelul 3.6. Opțiuni specifice *sim-profile*

Generarea unor statistici a informațiilor de profil se poate realiza prin comanda:

***sim-profile* -pcstat sim_num_insn test-math >&! test-math.out**

Trei dintre simulatoarele setului simplecalar (*sim-profile*, *sim-cache* și *sim-outorder*) suportă exprimarea statistică a informațiilor de profil (descrierea detaliată în cadrul unui segment de text) aferentă diferitelor variabile contor de numere întregi (numărul de instrucțiuni din program, numărul de referințe la memorie, numărul miss-urilor pe primul nivel al cache-ului de instrucțiuni, acuratețea predicției instrucțiunilor de salt).

Exemplu:

- pcstat sim_num_insn** - caracteristici ale execuției: numărul instrucțiunilor executate.
- pcstat sim_num_refs** - profilul referințelor la memorie (numărul instrucțiunilor load/store).
- pcstat ill.misses** - profilul acceselor cu miss în primul nivel al cache-ului de instrucțiuni.
- pcstat bpred_bimod.misses** - profilul predicției instrucțiunilor de salt.

6. Simulatorul superspeculativ de complexitate ridicată out of order (*sim-outorder*)

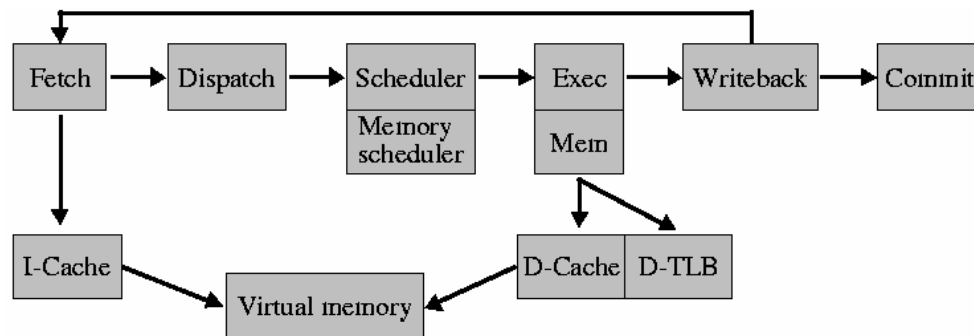


Figura 3.4. Structura pipeline a simulatorului Out of Order

Sim-outorder suportă expediere și execuție "out of order" bazat pe unitatea RUU (*register update unit*). Schema RUU utilizează un buffer de reordonare necesar în redenumirea automată a regiștrilor și reținerea rezultatelor pentru instrucțiunile aflate în așteptare. În fiecare ciclu, bufferul de reordonare retrage instrucțiunile încheiate în ordinea inițială a programului și le depune și în setul de regiștri generali.

Sistemul de memorie cuprinde și un buffer aferent instrucțiunilor Load/Store (coadă FIFO). Valorile de memorat sunt plasate în coadă dacă instrucțiunea Store este speculativă. Instrucțiunile Load sunt expediate spre unitățile de execuție cu referire la memorie doar când adresele tuturor instrucțiunilor Store anterioare sunt cunoscute (evitarea aliasurilor). Instrucțiunile Load pot fi satisfăcute fie prin execuție propriu-zisă fie printr-un mecanism de bypassing realizat hardware, preluând valoarea de înregistrare a unei instrucțiuni Store direct din coada de așteptare, în cazul unei potriviri de adresă. Instrucțiunile Load executate speculativ pot genera accese cu miss în cache, dar "miss-urile" speculative în TLB blochează structura pipeline până la cunoașterea condiției de salt.

Nucleul de execuție al simulatorului este structurat astfel:

```

ruu_init();
for ( ; ; ) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}

```

unde fiecare rutină descrie o fază de procesare a instrucțiunii.

Bucula de program este executată odată pentru fiecare ciclu mașină (A nu se înțelege perioadă de tact). La încheierea fiecărui program - cu apelul sistem `exit()` - simulatorul execută un apel `longjmp()` la funcția principală `main()` pentru generarea informațiilor statistice. Latențele tuturor unităților funcționale se regăsesc în structura `fu_config[]` din modulul `sim-outorder.c`.

Faza **fetch instrucțiune** a structurii pipeline este implementată de către funcția `ruu_fetch()`. Unitatea de fetch modelează lărgimea de bandă a fluxului de instrucțiuni și primește ca parametri de intrare următoarele: *PC*-ul instrucțiunii curente (adresa de la care se va face procesul de fetch), *starea predictorului* și *eventuale predicții greșite* rezultate din unitatea de execuție a instrucțiunilor de salt. În fiecare ciclu procesor, sunt citite și aduse instrucțiuni dintr-o singură linie a cache-ului de instrucțiuni (un acces cu miss blochează eventualele următoare citiri, până la rezolvarea respectivului *miss*). Instrucțiunile citite sunt depuse într-un buffer de prefetch, în așteptare, de unde vor fi expediate spre decodificare și execuție. De asemenea, se calculează cu ajutorul predictorului noua linie din cache-ul de instrucțiuni de unde va avea loc următorul proces de fetch.

Implementarea software a nivelului pipeline de **decodificare** și **clasificare/expediere** a instrucțiunilor spre unitățile funcționale de execuție se face în rutina `ruu_dispatch()`. Pe lângă decodificare are loc și *redenumirea regiștrilor* în vederea eliminării dependențelor WAR și WAW. Funcția folosește instrucțiunile din bufferul de prefetch ca pointeri spre unitatea RUU activă și spre unitatea (tabloul) de redenumire. Odată per ciclu procesor, sunt preluate din bufferul de prefetch un număr de instrucțiuni (cel mult numărul maxim de instrucțiuni ce pot fi executate simultan) și depuse într-o coadă cu instrucțiuni ce vor fi reorganizate de către scheduler. În această fază de procesare sunt efectuate predicțiile instrucțiunilor de salt. La apariția unei predicții greșite, simulatorul folosește speculativ buffere de stare, care sunt tratate printr-o politică **copy on write**. Rutina `ruu_dispatch()` introduce și conectează instrucțiuni la unitatea RUU și la coada de

instrucțiuni Load/Store; deoarece separă operațiile aferente instrucțiunilor cu referire la memorie în două (adunare/scădere pentru calcul efectiv de adresă și accesul efectiv la memorie).

Nivelul **issue** al structurii pipeline, de rutare efectivă (atașarea fiecărei instrucțiuni la câte o unitate funcțională de execuție) este implementat software prin procedurile *ruu_issue()* și *lsq_refresh()*. Rutinele păstrează de asemenea dependențele de date la resurse (regiștrii sau locații de memorie). În fiecare ciclu procesor, rutinele de scheduling localizează instrucțiunile pentru care regiștrii sursă sunt disponibili. Asignarea instrucțiunilor Load disponibile din punct de vedere al regiștrilor de intrare este stagnată dacă există o instrucțiune Store anterioară în coada Load/Store cu adresa efectivă necunoscută (nerezolvată). Dacă adresa instrucțiunii Store este chiar adresa de la care instrucțiunea Load, aflată în așteptare, urmează să citească atunci valoarea de memorat este înaintată spre utilizare acesteia. Altfel, instrucțiunea Load trebuie executată normal.

Faza de **execuție** a structurii pipeline este tratată, deasemenea, în rutina *ruu_issue()*. În fiecare ciclu mașină, rutina extrage cât mai multe (maxim **-issue:width**) instrucțiuni disponibile (și independente) pentru execuție din coada cu instrucțiuni reorganizate. Dacă unitățile funcționale de execuție sunt disponibile se startează execuția instrucțiunilor (fiecărei unități funcționale libere i se asociază (dacă există) o instrucțiune de același tip cu unitatea). În final, rutina organizează evenimentele de scriere a rezultatelor (resursele folosite - regiștrii, memorie) în funcție de latența unităților funcționale (se va ține cont de operațiile de acces la cache-ul de date/memorie) într-o coadă de priorități. Accesele cu miss la buferul de date TLB întârzie execuția operațiilor cu memoria până în faza *Commit* a structurii pipeline și li se alocă în mod curent o latență fixă. Latențele tuturor unităților funcționale se regăsesc în structura *fu_config[]* din modulul *sim-outorder.c*.

Scrierea efectivă a rezultatelor se realizează în faza **writeback**, implementată software în rutina *ruu_writeback()*. În fiecare ciclu mașină, rutina scanează coada de evenimente aferente instrucțiunilor care au încheiat faza de execuție. La găsirea unei instrucțiuni în această coadă, se parcurge lanțul dependențelor de date aferent instrucțiunii respective pentru a marca instrucțiunile dependente aflate în așteptare. Dacă o instrucțiune dependentă se află în așteptare pentru execuție, rutina o marchează ca fiind gata de execuție. În această fază se cunoaște cu exactitate dacă saltul a fost corect predicționat sau nu. Dacă a avut loc o predicție greșită, starea procesorului este reluată de la ultimul punct de verificare, eliminând (efectul) ultimele instrucțiuni executate eronat.

Procedura *ruu_commit()* tratează instrucțiunile din faza writeback care sunt gata să actualizeze corect (*in-order*) structurile hardware folosite (buffer reordonare, set de regiștri, coada de instrucțiuni load/store, unitatea RUU). De asemenea, se actualizează și cache-ul de date (sau memoria) și sunt tratate accesesele cu miss în bufferul de date TLB.

Datorită complexității sale *sim-outorder* rulează cu un ordin de mărime mai lent decât *sim-fast*. Parametrii din linia de comandă se pot clasifica în funcție de modulul component (nucleul de execuție, interfața procesor-cache, ierarhia de memorie, predictorul de salturi). Parametrii specifici nucleului de execuție al procesorului stabilesc numărul de instrucțiuni care sunt extrase simultan din cache/decodificate/transmise spre unitățile funcționale de execuție, modul de procesare (in/out order), capacitățile diferitelor buffere (coada Load/Store, unitatea RUU, unitățile funcționale ALU pentru întregi/flotante).

➤ Parametrii specifici *nucleului de execuție al procesorului*:

- ❑ **-fetch:ifqsize** <size> - setează numărul de instrucțiuni ce vor fi extrase simultan din cache / memorie principală. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-fetch:mplat** <cycles> - setează latența unei predicții eronate a unei instrucțiuni de salt (procesul de "recovery"). Implicit are valoarea 3.
- ❑ **-decode:width** <insts> - setează numărul de instrucțiuni ce vor fi simultan decodificate. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-issue:width** <insts> - setează numărul maxim de instrucțiuni ce vor fi simultan decodificate într-un ciclu. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-issue:inorder** – forțează simulatorul să lucreze *in-order*. Implicit ia valoarea "false".
- ❑ **-issue:wrongpath** – permite expedierea spre execuție a instrucțiunilor după o speculație greșită. Implicit are valoarea true.
- ❑ **-ruu:size** <insts> - capacitatea unității RUU (în instrucțiuni). Implicit ia valoarea 16.
- ❑ **-lsq:size** <insts> - capacitatea bufferului Load/Store (în instrucțiuni). Implicit ia valoarea 8.
- ❑ **-res:ialu** <num> - specifică numărul unităților ALU de numere întregi. Implicit ia valoarea 4.
- ❑ **-res:imult** <num> - specifică numărul unităților de înmulțire/împărțire întregi. Implicit ia valoarea 1.
- ❑ **-res:mempports** <num> - specifică numărul porturilor cache-ului de pe nivelul L1. Implicit ia valoarea 2.

- ❑ **-res:fpalu** <num> - specifică numărul unităților ALU în virgulă mobilă. Implicit ia valoarea 4.
- ❑ **-res:fpmult** <num> - specifică numărul unităților de înmulțire/împărțire în virgulă mobilă. Implicit ia valoarea 1.

Parametrii ce caracterizează *interfața procesor - sistemul ierarhizat de memorie* sunt constituiți din toți parametrii cache-ului inclusiv formatul acestora, utilizați în **sim-cache** și folosiți de către **sim-outorder** cu următoarele completări: specifică latența de hit în primul nivel al cache-ului de date, specifică dimensiunea în octeți a magistralei de memorie, reprezintă latența pentru deservirea unui miss în tabela TLB.

➤ Parametrii suplimentari (găsiți doar la **sim-outorder** nu și la **sim-cache**) ce caracterizează *interfața procesor - sistemul ierarhizat de memorie*:

- ❑ **-cache:dllat** <cycles> - specifică latența de hit în primul nivel al cache-ului de date. Implicit ia valoarea 1. Există opțiunea corespondentă și pentru cache-ul de instrucțiuni (pentru ambele nivele **il1**, **il2**).
- ❑ **-mem:width** <bytes> - specifică dimensiunea în octeți a magistralei de memorie.
- ❑ **-tlb:lat** <cycles> - reprezintă latența pentru deservirea unui miss în tabela TLB.

➤ Parametrii specifici *predictorului de salt*:

La fel ca simulatorul de cache-uri care constituie atât parte integrantă a simulatorului *out of order* cât și simulatorul de sine stătător, predictorul de salturi poate rula și individual, stabilind aceleași informații statistice: acuratețea de predicție, numărul de instrucțiuni de salt, numărul de interfețe în tabelele de predicție, numărul de accese cu hit în tabele, etc. Pot fi simulate o serie de predictoare, pornind de la tabela BTB la scheme de predicție adaptivă corelate pe 2 niveluri: GAg, GAp, PAg, PAp.

Dacă simulatoarele *sim-fast*, *sim-safe*, *sim-cache/cheetah* și *sim-bpred* sunt simulatoare funcționale care urmăresc influența diversilor parametri arhitecturali asupra acurateții predicției, ratei de hit în cache, etc, *sim-outorder* este singurul simulator component al setului SimpleScalar 3.0 care implementează *timing* (asociază câte o latență de execuție pentru fiecare tip de instrucțiune), modelează un procesor *superscalar* bazat pe o execuție *speculativă* și permite determinarea ratei globale de procesare. **Sim-outorder** este cel mai performant dintre simulatoarele setului SimpleScalar 3.0, dar și cel mai lent din punct de vedere al simulării. Pentru exploatarea câștigului de performanță introdus de orice tehnică (predicția valorilor pe

diverse resurse, reutilizare dinamică a instrucțiunilor), simulatorul superspeculativ și cu execuție *out-of-order* este singurul care poate fi considerat ca bază de plecare pentru acest experiment.

3.2.7. COMPILAREA ȘI EXECUȚIA PROGRAMELOR C ȘI C++ SUB LINUX. INSTALAREA ȘI COMPILAREA BENCHMARK-URILOR SPEC2000 PENTRU ARHITECTURA SIMPLESCALAR – PISA.

Creșterea în performanță și complexitate a microprocesoarelor moderne datorate tehnicilor avansate gen *pipelining*, execuție *out-of-order*, predicție și execuție speculativă, presupune un efort suplimentar de proiectare și verificare pentru dezvoltarea și implementarea de produse viabile. Pentru depășirea acestor probleme, proiectanții de microarhitecturi au explorat diverse modalități de *transfer de funcționalitate* la nivelul compilatorului. Începând cu procesoarele RISC VLIW și continuând cu cele EPIC – versiunile 1 și 2 de procesoare Intel Itanium, compilatorul a jucat un rol important în simplificarea arhitecturii la nivel hardware menținând totodată tendințele curente de creștere a performanței [Sias04]. În acest paragraf sunt prezentate etapele principale care trebuie parcurse pentru compilarea și execuția propriilor programe de test scrise în C (sau C++) sub sistemul de operare Linux folosind utilitarele GNU.

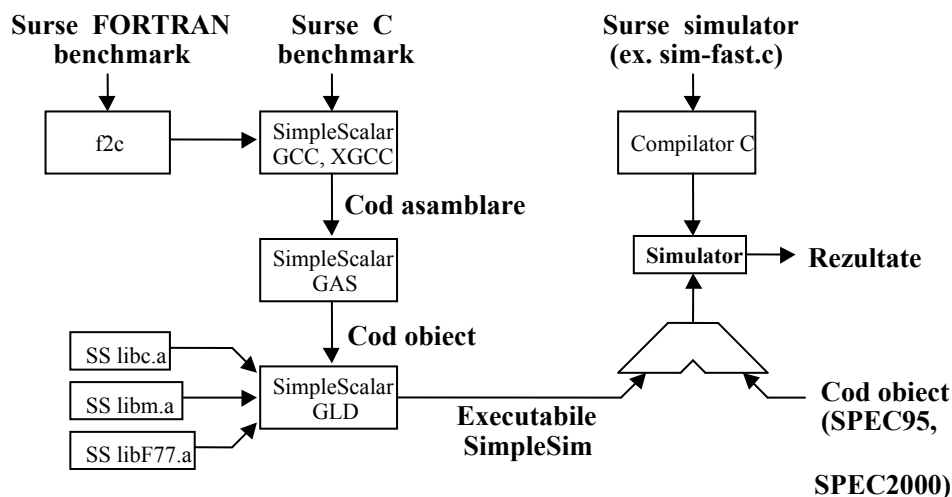


Figura 3.5. Interacțiunea instrumentelor în cadrul setului SimpleScalar

Etapele parcurse pornind de la sursa HLL (High Level Languages) a programelor de test și până la simularea de tip *execution driven* sunt evidențiate în figura anterioară (figura 3.5).

Sursele FORTRAN ale programelor de test proprii sunt convertite în C folosind translatorul *f2c*. Atât benchmark-urile C, C++ cât și cele convertite din FORTRAN sunt compilate cu ajutorul compilatorului GNU *gcc* (dedicat arhitecturii SimpleScalar) rezultând cod în format de asamblare pentru respectiva arhitectură. Codul rezultat este trecut prin asamblorul SimpleScalar *as* care generează un format foarte asemănător cu codul obiect dar nu identic (cuprinde simboluri + cod obiect; nu este generat codul obiect pentru funcțiile de bibliotecă și directivele de asamblare). Formatul final de cod obiect se obține cu ajutorul utilitarului SimpleScalar *ld* care link-editează codul rezultat la pasul anterior (*pseudo-obiect*) cu bibliotecile proprii arhitecturii SimpleScalar (*SS libc.a*, *SS libm.a*, *SS libF77.a*). Codul obiect se constituie ca parametru de intrare pentru simulatoarele arhitecturii SimpleScalar. Setul de instrumente SimpleScalar pune la dispoziție și câteva din benchmark-urile SPEC'95 în același format cod obiect (precompilate pentru arhitectura SimpleScalar). Dacă nu se dispune de simulatorul dorit în format executabil, ci doar de sursa C a acestuia, atunci acesta trebuie compilat cu compilatorul disponibil pe stația gazdă (orice compilator de ANSI C). Simularea se face foarte detaliat, la nivel de ciclu de execuție al procesorului, rezultatele generate fiind conținutul resurselor, gradul de încărcare al acestora, rate de procesare, de hit, acurateți de predicție, informații de profil, etc.

În continuare sunt prezentați pe scurt pașii care trebuie urmați pentru recompilarea instrumentelor SimpleScalar 3.0. în vederea obținerii codului obiect pentru arhitectura SimpleScalar, pornind de la benchmark-uri proprii scrise în limbajul C și având ca platformă sistemul de operare Linux Red Hat 7.3 cu procesor Intel Celeron (i686). Trebuie specificat că s-a încercat recompilarea instrumentelor și pe alte versiuni de Linux (Red Hat 7.0, Mandrake) însă nu întotdeauna s-a reușit. Mai mult, cu ajutorul noilor instrumente se poate genera cod obiect doar pentru fișiere sursă *.C* nu și *.C++*. Din programele obiectuale se pot obține totuși fișiere în limbaj de asamblare MIPS, utile în analiza numărului de salturi **statice** directe / indirecte cauzate de moștenire, polimorfism. Deși poate părea simplu și intuitiv de urmat, geneza pașilor ce vor fi prezentați este rodul a numeroase ore de muncă și implementări (corectări) repetate.

1. Pentru instalarea instrumentelor SimpleSim3.0 sunt necesare fișierele:

- **simplesim-3.0b.tar.gz**;
- **simpletools-2.0.tar.gz**;

- **simpleutils-2.0d.tar.gz;**

2. Fișierele de mai sus se dezarchivează într-un singur director, care nu trebuie să conțină spații în nume (presupunem că acest director se numește KIT și se află în rădăcină). Dezarhivarea se poate face folosind utilitarele GNU: *gunzip* urmat de *tar*. Succesiunea comenzilor este următoarea:

- **gunzip** simplesim-3.0b.tar
- **gunzip** simpletools-2.0.tar
- **gunzip** simpleutils-2.0d.tar

În urma acestor trei pași se obțin trei arhive **.tar** din care se extrage conținutul cu ajutorul comenzii:

- **tar -xvf** filename.tar, unde prin *filename* se înțelege oricare din numele de fișier anterioare: simplesim-3.0b, simpletools-2.0 sau simpleutils-2.0d.

În cadrul directorului /KIT se vor obține astfel următoarele subdirectoare:

- **binutils-2.5.2** (conține utilitare GNU – la nivel de cod C care urmează a fi compilat pentru diverse arhitecturi – în cazul nostru SimpleScalar 3.0)
- **simplesim-3.0** (conține codul sursă C al simulatoarelor procesorului virtual SimpleScalar, scripturi și macrodefiniții ale setului de instrucțiuni, sursele C și cod obiect (obținute după compilare/asamblare/link-editare) a unor mici programe de test (nu SPEC). Este necesar pentru instalarea setului de instrumente și include o documentație exhaustivă asupra setului SimpleScalar.
- **glibc-1.09.** – conține sursele bibliotecilor GNU, portate pentru arhitectura SimpleScalar.
- **gcc-2.6.3.** – reține sursele C ale compilatorului GNU, care pot fi compilate spre orice tip de arhitectură i386, SunSpark, SimpleScalar (în cazul nostru se dorește compilarea/asamblarea/link-editarea propriilor benchmark-uri spre arhitectura SimpleScalar).
- **f2c-1994.09.27** – conține sursele translatorului din Fortran în C, realizat în 1994 de către cercetătorii din laboratoarele AT&T Bell.
- **sslittle-na-sstrix**, **ssbig-na-sstrix** – directoarele țintă spre care vor porta cross-compilatorul, utilitarele GNU compilate (codul obiect rezultat), și bibliotecile de funcții C dedicate arhitecturii SimpleScalar. (Directorul ales va fi în funcție de ordinea octeților

de pe stația gazdă – în cazul nostru ordinea este *little-endian* [Flor03]).

3. Instalare **binutils-2.5.2**

- ❑ Se trece în subdirectorul **binutils-2.5.2** cu ajutorul comenzi **cd binutils-2.5.2**
- ❑ **./configure --host=i386-*-linux --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld** (eventual și **--prefix=/KIT**)
- ❑ **make**
- ❑ **make install**

4. Instalare **simplesim-3.0**

- ❑ Se trece în subdirectorul **simplesim-3.0** și se tastează comenzile:
- ❑ **make config-pisa**
- ❑ **make**

5. Instalare **gcc-2.6.3**.

- ❑ Se trece în subdirectorul **gcc-2.6.3** cu ajutorul comenzi **cd gcc-2.6.3**.
- ❑ **./configure --host=i386-*-linux --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld** (eventual și **--prefix=/KIT**)
- ❑ Fie (R) următoarea comandă: **make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc"**. Se tastează R și se repetă această comandă după fiecare corectare a erorilor eventual apărute.
 - O primă eroare care apare este la *linia 194* din fișierul **cccp.c** (conflict de tip). Linia respectivă poate fi înlocuită cu **extern const char *const sys_errlist[]**; sau poate fi pur și simplu comentată. Se tastează comanda R.
 - A doua eroare care apare este la *linia 57* din **sdbout.c**. Aceasta va fi înlocuită cu **#include "gsyms.h"** sau **#include <gsyms.h>**. Se tastează comanda R.
 - Următoarea eroare care apare este la *linia 17* din fișierul **bc-typedef.def**. Eroarea poate fi eliminată fie înlocuind linia respectivă cu **DEFTYPECODE(SFcode, "SF", SFmode, double)**; fie se corectează în *linia 13* din **bytetypes.h** (**float** devine **double**). Se repetă comanda **make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc"**.
 - O nouă eroare apare la *linia 172* din fișierul **gcc.c** (un nou conflict de tip). Pentru buna desfășurare a procesului de instalare această

linie va fi înlocuită cu **extern const char *const sys_errlist[]**; sau va fi comentată. Se reia comanda **R**.

- Următoarea eroare care apare se referă la *linia 90* din fișierul **/cp/g++.c**. Pentru înlăturarea erorii se comentează linia respectivă. Se tastează comanda **R**.
- În cazul în care apar erori în fișierul **collect2.c** (conflict de tip) se comentează cele două linii care au provocat erorile (*liniile 47 și 217*). Se tastează comanda **make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc"**.

În acest moment, singurele erori care au rămas sunt de bibliotecă (de ex: *stdio.h – no such file or directory*). Eliminarea acestora se realizează prin suprascrierea directorului **include** din directorul **gcc-2.6.3** cu subdirectorul **include** aflat în directorul **sslittle-na-sstrix**. Conținutul directorului **sslittle-na-sstrix/include**, mai puțin fișierul **assert.h** este copiat și în directorul **/usr/local/sslittle-na-sstrix/include**. De asemenea, fișierele **libc.a** și **crt0.o** din **/sslittle-na-sstrix/lib** se vor copia atât în directorul **gcc-2.6.3** cât și în **/usr/local/sslittle-na-sstrix/lib**. Se tastează pentru ultima dată comanda **make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc"**. În acest moment nu mai apar erori de compilare.

■ **make install**

După încheierea cu succes a celor 5 etape (complexe) în directorul **gcc-2.6.3** va fi creat fișierul **xgcc** – noul compilator cu ajutorul căruia se va determina codul asamblare / obiect pentru arhitectura virtuală SimpleScalar.

Comenzile cu ajutorul cărora obținem codul asamblare aferent programelor de test sunt:

■ **pentru sursa de program C:**

./xgcc nume_fisier.c -S

■ **pentru programele obiectuale CPP:**

./cc1plus nume_fisier.cpp -s

Codul obiect pentru arhitectura SimpleScalar se obține cu ajutorul comenzilor:

■ **pentru ambele tipuri de programe (atât C cât și C++):**

./xgcc nume_fisier_sursa.c -o nume_fisier_destinatie.ss

Se reamintește că opțiunea de compilare **-S** realizează preprocesarea și compilarea codului sursă, iar opțiunea **-s** generează doar codul asamblare fără faza de preprocesare. De asemenea, în cazul în care fișierele sursă **c** și

cpp nu există în directorul **gcc-2.6.3** atunci se va tasta odată cu numele și calea spre acest fișier.

În acest moment se vor copia fișierele executabile SimpleScalar (**.ss**) în directorul **simplesim-3.0** și pot fi folosite ca programe de test pentru oricare dintre simulatoarele setului SimpleScalar 3.0. De exemplu, pentru măsurarea localității și a numărului de salturi indirecte din benchmark-ul **back_ss** folosind simulatorul **sim-jindir.exe** se va tasta comanda:

```
./sim-jindir -redir:sim simout.res back_ss
```

În continuare sunt prezentate încă câteva opțiuni de compilare utile. Pentru compilarea programelor se recomandă specificarea opțiunilor **-Wall** pentru afișarea tuturor avertismentelor generate de compilator. Prin opțiunea **-c** este evitată link-editarea iar **-g** invocă opțiunea de depanare, impunând compilatorului generarea unei tabele de simboluri. Sunt generate doar codurile pseudo-obiect ale tuturor fișierelor sursă. Link-editarea se poate realiza ulterior cu comanda:

```
./xgcc file1.o file2.o ..... -o nume_fis_executabil
```

Folosind opțiunea **-llibrary** se realizează link-editarea bibliotecilor standard sau utilizator specificate prin **library**. Pentru utilizarea funcțiilor matematice trebuie ca fișierul sursă **.c** să cuprindă directiva de preprocesare **#include <math.h>** și legarea să fie făcută explicit cu comanda:

```
./xgcc calc.c -o calc -lm
```

Opțiunea **-Ldirectory** adaugă directorul specificat la lista de directoare care conțin librării de funcții în format obiect. Link-editorul caută întotdeauna după librăriile standard sau sistem în subdirectorul **/lib** sau **/usr/lib**. Exemplul următor poate fi înlocuit cu ușurință cu librăriile specifice arhitecturii SimpleScalar:

```
./xgcc prog.c -L/home/myname/mylibs mylib.a
```

Opțiunea **-Ipathname** adaugă calea de directoare în care se vor căuta bibliotecile specificate cu directiva **#include**. Este obligatoriu să nu înceapă cu **/**. Implicit preprocesorul caută după fișierele definite cu **#include** în directorul curent (cel care conține codul sursă al aplicației), apoi în directorul din calea specificată cu directiva **-Ipathname** și în final în directorul **/usr/include**. Astfel, includerea unei biblioteci utilizator aflată în calea de directoare **/home/myname/myheaders** se face prin comanda:

```
./xgcc prog.c -I/home/myname/myheaders
```

Obs: Fișierele de bibliotecă (header) aferente librăriilor sistem se găsesc în directorul /usr/include și nu sunt afectate de către opțiunea **-I**. Includerea acestora se face într-o manieră ușor diferită de cazul bibliotecilor utilizator.

În continuare sunt prezentați pașii pentru **instalarea benchmark-urilor SPEC2000 sub sistemul de operare LINUX**.

I1. Se creează un director pe discul destinație **/pisa/kit-spec2000**. Trebuie să existe cel puțin **1GByte** spațiu liber pe discul destinație.

I2. Se montează CD-ul cu sursele SPEC2000

\$mount /dev/cdrom /mnt/cdrom

I3. De notat că trebuie să existe privilegiile de *root* pentru a monta CD-ul.

I4. Se trece în directorul unde a fost montat CD-ul

\$cd /mnt/cdrom

I5. Se execută scriptul de instalare

./install.sh

Când sunteți întrebat, se va introduce directorul destinație (/pisa/kit-spec2000).

I6. După terminarea instalării se va afișa mesajul următor:

*Everything looks ok, source the shrc file
and have at it!*

I7. Se schimbă directorul curent în /pisa/kit-spec2000 și se execută scriptul *shrc*:

\$cd /pisa/kit-spec2000

\$. ./shrc <- adică punct spațiu punct slash....

I8. Efectul comenzilor anterioare este de a seta variabilele și calea spre benchmark-urile SPEC2000.

În acest moment benchmark-urile SPEC2000 sunt **instalate**. **Compilarea** benchmark-urilor pentru arhitectura PISA presupune urmarea pașilor:

C1. Se schimbă directorul curent în **/pisa/kit-spec2000/config** și se face o copie a fișierului **Tru64_Unix**:

\$cd /pisa/kit-spec2000/config

\$cp Tru64_Unix.cfg pisa.cfg

C2. Se trece la editarea fișierului de configurări *pisa* cu ajutorul unui editor.

În acest caz s-a folosit “*pico*”:

```
$pico pisa.cfg
```

C3. În fișierul **pisa.cfg** se vor modifica următoarele:

```
tune = base
```

```
ext = ss
```

```
CC = /pisa/gcc-2.6.3/xgcc -v
```

```
CXX = /pisa/gcc-2.6.3/xgcc -v
```

```
FC = kf90 -v
```

```
OPTIMIZE =
```

```
COPTIMIZE =
```

```
CXXOPTIMIZE = -O2
```

```
FOPTIMIZE = -O5
```

```
ONESTEP = yes
```

```
PASS1_CFLAGS = -funroll-loops
```

```
#PASS2_CFLAGS = -prof_use_feedback -prof_dir /tmp/prof
```

C4. Se editează fișierul **/pisa/kit-spec2000/benchspec/Makefile.defaults**.

La linia 69 se șterge **-lm**.

C5. În acest moment se poate trece la compilarea benchmark-urilor SPEC2000 pentru arhitectura PISA. Comanda universală pentru realizarea acestui lucru este:

```
$runspec --config=pisa --action=build benchmark
```

unde *benchmark* reprezintă programul de test care se dorește a fi compilat (ex: *parser*, *gzip*, *bzip2*, etc).

După compilare benchmark-ul poate fi găsit în directorul:

```
/pisa/kit-spec2000/benchspec/cint2000/benchmark/exe/
```

Prin metoda de mai sus vor putea fi compilate pentru arhitectura SimpleScalar următoarele benchmark-uri:

- 1) 164.gzip
- 2) 175.vpr
- 3) 176.gcc
- 4) 181.mcf
- 5) 186.crafty
- 6) 197.parser
- 7) 255.vortex
- 8) 256.bzip2
- 9) 300.twolf
- 10) 254.gap

Benchmark-ul 253.perlbnk necesită următorii pași suplimentari pentru obținerea codului executabil. Pentru început trebuie corectate următoarele erori:

```
pp_sys.c:58: sys/select.h: No such file or directory
```

Se comentează linia 58 din pp_sys.c aflat în directorul:

```
/pisa/kit-spec2000/benchspec/CINT2000/253.perlbnk/src
```

Se execută comanda:

```
$runspec --config=pisa --action=build perlbnk
```

3.2.8. FOLOSIREA DEBUGGER-ULUI GNU.

În acest subcapitol sunt prezentate etapele principale care trebuie parcurse pentru depanarea programelor scrise în C (sau C++) sub sistemul de operare Linux folosind utilitarele GNU.

Puține programe funcționează perfect la prima rulare. Mai devreme sau mai târziu va trebui să depanăm un program care funcționează conform așteptărilor. Deși există unele medii de programare care au unele facilități de depanare, există soluții foarte bune și fără a recurge la acestea. Cea mai simplă și mai folosită metodă de depanare este modificarea programului pentru a afișa unele informații esențiale în anumite puncte critice. Astfel, citind ieșirea programului se va putea analiza evoluția sa. Dublată de puțină experiență, această abordare se dovedește foarte eficientă. Standardul *ANSI C* prevede două facilități care ajută la depanare. Funcția *abort()* încheie execuția programului și generează un fișier *core*. Se poate apela la această funcție de câte ori se depistează un caz “imposibil”. Astfel, când se întâmplă un eveniment neprevăzut, programul se va opri și se va putea examina starea memoriei în acel moment, putând constata unde s-a greșit în raționament. Celălalt element de ajutor în depanare este funcția *assert()*. Aceasta primește un parametru, de obicei o condiție care este cu siguranță adevărată. Dacă condiția nu este îndeplinită se comportă asemănător cu *abort()*.

O abordare comodă o reprezintă folosirea depanatorului GNU – *gdb*, vizual sau în mod text, depinde de versiunea de Linux sau Cygwin (emulator de Linux în Windows) utilizată, pentru examinarea cursului

execuției programului și a evoluției variabilelor pentru cazuri de grade diferite de complexitate. Astfel,

- Fișierele obiect trebuie șterse dacă acestea există (în cazul în care au fost compilate anterior pentru sursele respective).
- Sursele trebuie compilate cu opțiunile de depanare: **-g -O**

gcc -c -g -O nume_fis1.c nume_fis2.c

- Asigurați-vă că a avut loc o compilare cu succes (dacă s-au creat fișierele obiect).

dir nume_fis*.o

- Se va reface executabilul prin comanda *make* (trebuie verificat ca fișierul *makefile* să existe și este cel dorit de dumneavoastră). Pentru surse obișnuite (nu module care extind setul de instrumente SimpleScalar) se poate încerca și comanda:

gcc nume_fis1.o nume_fis2.o -o nume_fis_executabil

Altfel, pentru surse SimpleScalar (fiind necesare și alte module componente setului, precompilate sau biblioteci arhivate) se va tasta din prompter cygwin (directorul curent fiind **simplesim-3.0**) sau mediu de comandă Linux:

```
$ gcc -o sim-vpred `./sysprobe -flags` -DEBUG -O0 -g -Wall sim-
vpred.o vpred.o main.o syscall.o memory.o regs.o loader.o endian.o
dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o
machine.o libexo/libexo.a `./sysprobe -libs` -lm
```

În ambele cazuri este generat cod mașină specific arhitecturii pentru care a fost scris și compilat codul sursă al gcc (în acest caz Intel x86).

- Se rulează programul *gdb*.

gdb sau

gdb nume_program_executabil sau

- În orice moment se poate tasta **gdb -help** pentru afișarea comenzilor.
- Alegerea programului (executabil) se face prin comanda *file*, dacă nu s-a specificat la inițializarea gdb-ului.

file nume_program_executabil

- Setarea argumentelor programului (dacă este nevoie) se face prin comanda *set args*.

```
set args -redir:sim applu.res -max:inst 5000000 -contextual 1 applu.ss <
applu.in
```

Notă: Pot exista unele probleme dacă nu se lasă spațiu între “aplu.ss” și “<” respectiv între “<” și “aplu.in”.

- Pentru a verifica dacă s-au introdus corect argumentele se poate apela comanda *show args*.
- Setarea punctelor de întrerupere (*breakpoints*) – se face prin comanda *break*.
 - ▣ *break* (fără parametrii) – setează întreruperea pe următoarea instrucțiune ce va fi executată și are efect doar dacă în prealabil s-a dat comanda *run*.
 - ▣ *break function* – stabilește un punct de întrerupere pe prima instrucțiune din funcție.
 - ▣ *break +/- offset* - setează un *breakpoint* la un număr de instrucțiuni (offset) înainte sau după linia curentă
 - ▣ *break nr_linie* – stabilește o întrerupere pe linia *nr_linie*
 - ▣ *break nume_fișier:nr_linie* - setează un *breakpoint* la linia *nr_linie* din fișierul *nume_fișier*.
 - ▣ *break ... if cond* – determină oprirea la punctul de întrerupere stabilit doar dacă evaluarea condiției este diferită de 0. În locul punctelor de suspensie (...) se poate pune orice argument mai sus amintit.
 - ▣ Comanda *tbreak args* are aceleași argumente ca și *break*, cu deosebirea că *breakpoint*-ul este șters după prima oprire în acel loc.
- Pentru a vizualiza lista punctelor de întrerupere se tastează *info breakpoints*.
- Stabilirea punctelor de vizualizare (*watchpoint*-uri) se face prin comanda *watch* (prin *watchpoint*-uri se poate opri execuția unui program de fiecare dată când evaluarea expresiei se modifică. Se pot seta cel mult două astfel de expresii:

watch expr

- Pentru a cunoaște lista punctelor de vizualizare se tastează *info watchpoints*.
- Se rulează apoi programul în mod interactiv (*debug*) prin comanda *run* (sau *run > outfile* – pentru indirectarea mesajelor către fișierul *outfile*).

- Puncte de întrerupere se pot seta și după lansarea comenzii *run* prin aceleași comenzi enunțate mai sus (în cazul în care consola deține controlul).
- Parcurgerea sursei se poate face fie pas cu pas, fie până la următorul punct de întrerupere, fie până la sfârșitul funcției, prin comenzi asemănătoare utilităților de depanare din limbajele consacrate (de nivel înalt).
- Comanda ***continue [ignore_count]*** reia execuția programului de la adresa la care s-a oprit ultima dată; oricare punct de întrerupere setat la acea adresă este sărit (*bypassing*); argumentul opțional *ignore_count* permițând ignorarea punctului de întrerupere de la adresa la care s-a oprit de un număr de ori egal cu *ignore_count*; comanda poate fi abreviată cu *c*.
- Comanda ***next [count]*** – (*step over*) reia execuția programului și se oprește la instrucțiunea următoare din contextul curent al stivei (din fișierul curent). Dacă se specifică argumentul *count*, rămâne în *next* – nu mai predă controlul consolei decât după execuția următoarelor *count* instrucțiuni.
- Comanda ***step [count]*** – este asemănătoare comenzii *next*, cu deosebirea că, dacă instrucțiunea curentă este o funcție compilată cu opțiunea pentru depanare (-g), nu se trece la următoarea instrucțiune, ci se intră în funcția respectivă (*step into*). Reciproc, dacă funcția nu a fost compilată cu opțiune pentru depanare, efectul va fi același cu cel al comenzii *next*. Pentru a intra în funcțiile fără informații de depanare, se poate folosi comanda ***stepi*** (abreviată ***si***) – care are ca efect intrarea în funcția respectivă, cu afișarea instrucțiunii mașină următoare; împreună cu *stepi* se poate folosi comanda ***nexti*** (abreviată ***ni***) care reia execuția programului și se oprește la următoarea instrucțiune mașină.
- Vizualizarea datelor se face cu ajutorul comenzii ***print***, abreviată ***p***. Sintaxa acestei comenzi este:
print [/f] [exp]
- Dacă argumentul *exp* lipsește, *print* va afișa ultima valoare afișată. */f* poate specifica formatul în care se face afișarea, astfel:
 - ***x*** afișează întregul din valoare ca hexazecimal.
 - ***d*** afișează întregul din valoare ca zecimal cu semn.
 - ***u*** afișează întregul din valoare ca zecimal fără semn.
 - ***o*** afișează întregul din valoare ca octal.
 - ***t*** afișează întregul din valoare în binar.
 - ***c*** afișează întregul din valoare ca și caracter constant.

- **f** ia biții din valoare ca număr în virgula mobilă și afișează valoarea.
- Se pot vizualiza variabile sau expresii din alte funcții sau fișiere:

fișier::variabilă
funcție::variabilă

Notă: Ocazional, o variabilă locală poate avea valori greșite chiar la începutul intrării în funcție, sau chiar înainte de ieșire din funcție. Acest lucru se datorează în parte faptului că este nevoie pe majoritatea mașinilor de mai multe instrucțiuni pentru a comuta contextul (și variabilele locale implicit). Acest lucru se întâmplă doar la parcurgerea sursei la nivel de instrucțiuni mașină.

- Părăsirea utilitarului gdb se face prin comanda *quit*.

Exemplul următor este dat pentru predictorul de tip *LastValue* aferent instrucțiunilor de salt indirect (*contextual = 3* – parametrul de intrare pentru simulatorul *sim-vpred.exe*) ce conține ca și module principale fișierele: *sim-vpred.c*, *vpred.h* și *vpred.c*.

1. `gcc -c -g -O *vpred.c`
2. `dir *vpred.o`
3. `make //` sau comanda echivalentă (vezi expunerea anterioară)
4. `gdb`
5. `file sim-vpred //` s-a încărcat `sim-vpred.exe` în memoria aplicației gdb
6. `set args -redir:sim c.res -max:inst 500000000 -contextual 3 -jvpt 256 -history 1 applu.ss < applu.in`
7. `show args`
8. `break foundAssociativeJVPTAddress`
// mesajul care va apare este: *Breakpoint 1 at 0x40a8cd: file vpred.c, line 304.*
9. `run`
/* mesajul care va apare este: *Breakpoint 1, foundAssociativeJVPTAddress (addr=4346064, value=4349632, history=1, jvpt=0x430d4c, JVPTdim=256, contextual=3, pattern=3) at vpred.c:304*
*Starting program: /home/Administrator/simplesim-3.0/sim-vpred.exe -redir:sim c.res -max:inst 500000000 -contextual 3 applu.ss < applu.in */*

3.3. EXTINDEREA MEDIULUI SIMPLESCALAR 3.0 CU UN MODUL PROPRIU.

3.3.1. DESCRIEREA SUMARĂ A CODULUI SURSĂ AFERENT SIMULATOARELOR SIMPLESCALAR EXISTENTE.

Setul de instrucțiuni aferent arhitecturii SimpleScalar este definit prin intermediul unor macrouri în modulul **ss.def** [Bur97]. Fiecare macro definește codul operației, numele instrucțiunii, diverse flag-uri (indicatori de condiție), operanzii sursă și destinație, precum și acțiunea ce trebuie îndeplinită pentru fiecare instrucțiune în particular. Acțiunile instrucțiunilor comune tuturor simulatoarelor sunt definite în fișierul **ss.h**. Acele acțiuni care necesită implementări distincte pentru simulatoare diferite, sunt definite în mod specific în codul sursă al fiecărui simulator.

La rularea unui simulator, rutina *main()* (definită în modulul **main.c**) realizează toate inițializările (ceasul simulatorului, biblioteca BFD, numărul de instrucțiuni procesate, unitatea de decodificare a instrucțiunilor etc.), procesarea (identificarea) opțiunilor generale de simulare, iar apoi încarcă în memorie codul obiect (executabil) al benchmark-ului care va fi simulat, verifică valorile parametrilor din linia de comandă. În final, *main()* apelează subrutina *sim_main()* specifică și descrisă distinct în fiecare simulator în parte. *Sim_main()* predecodifică întregul segment de text (codul și datele benchmark-ului) pentru o mai rapidă simulare, și apoi startează simularea din punctul specificat (instrucțiunea 0, sau după simularea unui anumit număr de instrucțiuni etc.). Pentru o mai bună înțelegere a funcționării interne a simulatoarelor setului de instrumente SimpleScalar 3.0, se prezintă în continuare codul sursă al fișierelor componente. Toate aceste fișiere se regăsesc și la adresa de web: <ftp://ftp.cs.wisc.edu/sohi/Code/Simplescalar/simplesim.tar>.

- ☐ **bpred.[c,h]:** Verifică instanțierea, funcționarea și actualizarea predictoarelor de salturi. Cele mai importante funcții de interfață sunt *bpred_create()*, *bpred_loockup()*, *bpred_update()*.
- ☐ **cache.[c,h]:** Conține funcțiile necesare configurării unor multiple tipuri de cache (TLB-uri, cache-uri de date și instrucțiuni). Structurile de date folosite sunt dinamice (liste simplu înlănțuite) care ajută la compararea tag-urilor în cache-urile cu asociativitate scăzută (cel mult patru), și tabele de dispersie pentru comparațiile de tag-uri în cache-urile de cu grad ridicat de asociativitate. Funcțiile cheie folosite sunt:

cache_create(), *cache_acces()*, *cache_probe()*, *cache_flush()*, și *cache_flush_adr()*.

- ☐ **endian.[c,h]:** Definește câteva funcții simple pentru a determina ordinea la nivel de octet și cuvânt de date (byte și word) pe platformele sursă (unde au fost compilate) dar și pe cele țintă (unde sunt lansate în execuție).
- ☐ **eventq[c,h]:** Definește funcțiile și macro-urile pentru administrarea cozilor ordonate de evenimente (folosite pentru ordonarea și arbitrarea scrierilor în faza “write-back”). Metodele mai importante sunt: *eventq_queue()* și *eventq_serve_events()*.
- ☐ **loader.[c,h]:** Încarcă programul țintă (ex. *applu.ss*) în memoria simulatorului, setează mărimea segmentelor de date, cod și stivă, inițializează stiva de program, și stabilește “punctul de intrare” în programul de simulat. Funcția cheie care realizează acest lucru este *ld_load_prog()*.
- ☐ **main.c:** Execută toate inițializările și funcțiile principale ale simulatorului. Metodele cheie sunt: *sim_options()*, *sim_config()*, *sim_main()*, și *sim_stats()*.
- ☐ **memory.[c,h]:** Cuprinde funcțiile pentru citirea din sursă, scrierea în destinație, inițializare, și afișare a conținutului memoriei principale. Memoria este implementată ca și un spațiu vast compartimentat, fiecare fracțiune a ei fiind alocată la cerere. Funcția de bază este *mem_acces()*.
- ☐ **misc.[c,h]:** Conține numeroase funcții foarte utile, ca de exemplu: *fatal()*, *panic()*, *warn()*, *info()*, *debug()*, *getcore()*, și *elapsed_time()*.
- ☐ **options.[c,h]:** Conține codul responsabil cu opțiunile simulatorului, folosit pentru procesarea argumentelor din linia de comandă și/sau opțiunile specificate în fișierele de configurare. Opțiunile sunt stocate într-o bază de date (vezi funcțiile *opt_reg_**()). *opt_print_help()* generează o listă de mesaje de ajutor, iar *opt_print_options()* afișează starea curentă a fiecărei opțiuni.
- ☐ **ptrace.[c,h]:** Furnizează codul necesar generării de *trace*-uri la nivel de faze *pipeline* de procesare din simulatorul *sim-outorder*.
- ☐ **regs.[c,h]:** Evidențiază funcțiile privind inițializarea și afișarea conținutului seturilor de regiștri generali: întregi, flotanți.
- ☐ **sim.h:** Conține câteva declarații de variabile externe și prototipuri de funcții.
- ☐ **stats.[c,h]:** Cuprinde rutinele necesare manipulării rezultatelor statistice ale simulării. Funcții cheie folosite sunt: *stat_reg_**(), *stat_reg_formula()*, *stat_print_stats()*, *stat_reg_dist()* și *stat_add_sample()*.

- ☐ **syscall.[c,h]**: Conține codul care acționează ca o interfață între *apelurile sistem* aferente arhitecturii SimpleScalar și *apelurile sistem* caracteristice sistemului de operare care rulează pe calculatorul gazdă.
- ☐ **sysprobe.c**: Determină ordinea la nivel de octet și cuvânt (*byte and word order*) existentă pe calculatorul gazdă, și generează flag-urile de compilare corespunzătoare.

3.3.2. IMPLEMENTAREA SIMULATORULUI PROPRIU.

Pentru studierea unei anumite idei/concept/metode și raportare în concordanță cu majoritatea cercetătorilor în domeniul arhitecturilor de calcul trebuie ca simularea să fie realizată pe benchmark-urile SPEC, SoftFloat sau *MediaBench*. Spre exemplu, în anul 2000, mai mult de o treime din lucrările publicate în conferințele de top dedicate arhitecturii calculatoarelor au folosit setul *SimpleScalar* pentru simularea / evaluarea propriilor idei de proiectare. Setul de instrumente "*SimpleScalar*" realizează o standardizare a procesului de simulare a unei microarhitecturi. Drept consecință, este dificil de realizat un **simulator independent** care să proceseze benchmark-urile SPEC. **Soluția** constă în: se alege unul din simulatoarele sursă existente (cel care s-ar potrivi cel mai mult cu "*background-ul*" necesar cercetării proprii - de exemplu *sim-fast*, *sim-bpred*) și se urmează pașii descriși în continuare. Se exemplifică printr-un modul care determină localitatea valorilor pe diverse tipuri de resurse și predicționează valorile instrucțiunilor. O descriere integrală a acestui simulator este realizată în subcapitolul 6.1.1.

- A) Pe scheletul simulatorului existent (fie acesta **sim-bpred.c**), redenumindu-l, se construiește un fișier sursă nou, numit **sim-VPred.c**. În cadrul acestui fișier se vor adăuga (sau elimina) noi informații.
- ⇒ **Eliminarea** se referă la directive de compilare, etc, având ca "*target*" o altă platformă (Exemplu: *Alpha*, *ARM*). **Atenție**, acest lucru trebuie făcut cu multă grijă! Rămânerea informațiilor ne-necesare în codul sursă al simulatorului nu va afecta funcționarea corectă a acestuia pe platforma pe care se lucrează (*PISA*).
 - ⇒ **Adăugarea** constă în **opțiuni noi** (parametrii de intrare ai simulatorului), **statistici generale** precum și codul aferent **simulării efective** (rutina **sim_main()** - specifică fiecărui simulator, și apelată din modulul extern **main.c**, după inițializări prealabile) a ideii/tehnicii propuse spre investigare.

- B)** Se construiește un fișier sursă nou **VPred.c** care să conțină definirea simbolurilor/parametrilor, structurilor (spre exemplu: **LVPT**), implementarea funcțiilor apelate în modulul **sim-VPred.c** (spre exemplu: *clasificarea (predictibilă / nepredictibilă) a intrării în structura de predicție* etc). Se poate păstra scheletul existent (fie **bpred.c**), redenumindu-l.
- C)** Completarea modulului **Makefile** (în cadrul căruia are loc compilarea / asamblarea / link-editarea) cu noul modul oriunde este nevoie (fișiere sursă **.c**, fișiere header **.h**, fișiere obiect **.o**, biblioteci suplimentare, fișiere executabile - ce vor fi generate). Apariția mesajului "*My work is done here !*" exprimă încheierea cu succes a operațiunii de compilare / asamblare / link-editare a simulatorului implementat.

Obs:

- i) Pasul **B)** poate lipsi dar în acest caz modulul **sim-VPred.c** ar deveni prea complex cuprinzând și definițiile, implementările funcțiilor apelate în interiorul său. Este deci nerecomandabil.
- ii) Programarea se realizează în limbajul **C standard** sub sistemul de operare **Linux**, neexistând situații de gen "*Not Enough Memory*" la compilare în ciuda complexității aplicației.

Înainte de a începe simularea în sine simulatorul face uz de câteva funcții predefinite, astfel:

- ❖ înregistrează opțiunile de intrare în funcția *sim_reg_options* (struct *opt_odb_t *odb*) folosind funcțiile *opt_reg_uint*, *opt_reg_string*.
- ❖ verifică valorile argumentelor de intrare în funcția *sim_check_options* (struct *opt_odb *odb*, int *argc*, char **argv*).
- ❖ înregistrează informațiile statistice de ieșire în *sim_reg_stats* (struct *stat_sdb_t *odb*) folosind *stat_reg_counter*, *stat_reg_int*, *stat_reg_formula*.
- ❖ inițializează simulatorul cu *sim_init*(void).
- ❖ încarcă programul (benchmark-ul selectat) în starea simulată cu *sim_load_prog*(char **fname*, int *argc*, char ***argv*, char ***envp*).

Funcțiile descrise sunt apelate în multe etape ale simulării și pot fi folosite în structura simulatorului (dar nu este necesar). Folosind funcțiile de ajutor furnizate, simulatorul nou creat este integrat în structura SimpleScalar și va obține interfața standard pentru opțiunile de intrare, și una pentru furnizarea rezultatelor. Cu aceste noi funcții de ajutor avem să definim câteva macrouri pentru a accesa regiștri pe care îi folosește simulatorul (**SET_NPC**, **SET_TPC**, etc), pentru identificarea dependențelor dintre instrucțiuni provocate de regiștri generali (**DGPR**, **DFPR_L**, **DFPR_F**, etc.),

funcții de ajutor pentru determinarea stării precise a structurii ierarhice de memorie (READ_BYTE, WRITE_BYTE etc).

Funcția *sim_main()* conține nucleul simulatorului. Ea buclează într-un ciclu infinit unde se aduc, se decodifică și execută instrucțiunile din program. Când programul simulat se încheie sau ajunge la un număr maxim de instrucțiuni ciclul se încheie ca de altfel și simularea.

Operația de aducere și decodificare se implementează prin două macrouri:

- ▣ MD_FETCH_INST(inst, mem, regs, regs_PC).
- ▣ MD_SET_OPCODE.

Execuția instrucțiunii se încheie folosind o întrerupere unde este inclusă definiția mașinii (**machine.def** care pointează la **pisa.def** ori **alpha.def**). Execuția instrucțiunii actuale este încheiată în **machine.def** folosind macrourele definite anterior (SET_GPR, SET_NPC, SET_FPR etc). Mai mult, în timpul fazei de execuție a instrucțiunii se realizează simularea proprie (determinarea localității pe tipuri de instrucțiuni, pe regiștrii procesorului MIPS și eventual predicția cu diverse scheme: "*Last Value, Incrementală, Contextuală, Hibridă*").

Fișierele **VPred.h** și **VPred.c** conțin declarațiile de structuri și de date, definițiile funcțiilor folosite în implementare în vederea cuantizării localității valorilor: **addrList**, **valueList**, **pushAddress**, **pushValue**, **foundAddress**, **foundValue**, etc.

Blocul de registre generale folosit în simulatorul SimpleScalar este implementat ca o structură definită în **regs.h**. Flag-urile folosite sunt:

- F_ICOMP – identifică operații aritmetico - logice
- F_FCOMP – identifică operații în virgulă mobilă (Floating Point)
- F_CTRL – instrucțiune de control (jump)
- F_UNCOND – salt necondiționat
- F_COND – salt condiționat
- F_MEM – instrucțiune de acces la memorie (cuprind atât instrucțiuni Load cât și instrucțiuni Store)
- F_LOAD – încarcă instrucțiunea Load (citire din memorie)

Alte flag-uri sunt definite în fișierul **pisa.h**.

3.4. ARHITECTURA SIMPLESCALAR

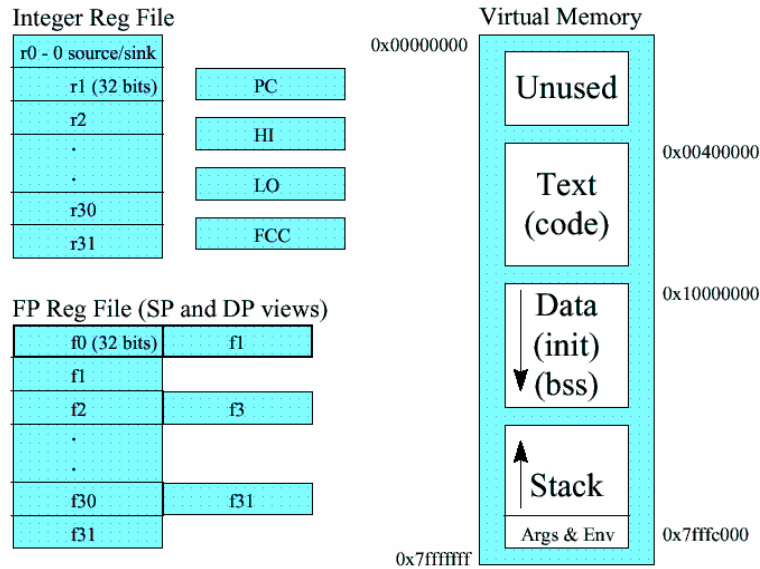


Figura 3.6. Arhitectura virtuală "SimpleScalar" – regiștrii/memorie. Orientare inversă.

Arhitectura SimpleScalar este derivată din arhitectura procesorului MIPS - IV ISA. Organizarea memoriei în sistemele bazate pe arhitectura SimpleScalar este convențională. Spațiul de adrese utilizator (pe 31 de biți) este compus din trei părți: cod, date și stivă program. Prima zonă din spațiul de adrese (începe la 0x400000) este segmentul text, care memorează instrucțiunile programului. Deasupra segmentului de text este segmentul de date (începând de la adresa 0x1000000) și este împărțit în două părți. Zona de date statică conține obiecte a căror mărime și adresă sunt cunoscute de către compilator și link-editor. Imediat, deasupra acestei zone se află datele dinamice. La alocarea spațiului dinamic de memorie pentru un program (prin **malloc** și apelul sistem **sbrk**) marginea superioară a segmentului de date se deplasează în sus. La marginea superioară a spațiului de adrese (0x7ffc000) este stiva de program, care crește în jos, spre segmentul de date.

Setul de instrucțiuni SimpleScalar PISA (Arhitectura cu set de instrucțiuni portabilă) este o extensie a setului de instrucțiuni al procesorului DLX conceput de *Hennessy* și *Patterson*, incluzând de asemenea un număr de instrucțiuni și moduri de adresare specifice procesoarelor MIPS IV și

RS/6000. Există totuși câteva diferențe notabile dar și caracteristici suplimentare din care amintim:

- ◆ Codificarea instrucțiunilor este pe 8 octeți.
- ◆ Cuprinde o instrucțiune nouă, inexistentă la procesoarele amintite mai sus, de extragere a rădăcinii pătrate atât în simplă cât și în dublă precizie.
- ◆ Instrucțiunile cu referire la memorie (load/store) suportă două moduri de adresare - pentru toate tipurile de date - suplimentare celor existente la procesorul MIPS IV: *indexat (registru + registru)* și *auto - incrementare/decrementare*.

Ca și în cazul arhitecturii MIPS există trei formate de instrucțiuni: *registru (R-tip)*, *imediat (I-tip)* și de *salt (J-tip)*.

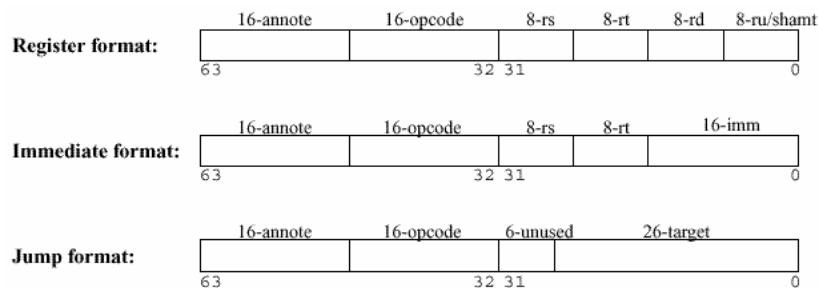


Figura 3.7. Formatul instrucțiunilor pentru arhitectura SimpleScalar

Formatul registru este specific instrucțiunilor de calcul (aritmetico - logice). Formatul imediat este folosit de către instrucțiunile de salt condiționat și de transfer. Cuprinde un câmp constantă imediată pe 16 biți. Al treilea format se numește J-tip, și cuprinde printre altele și un câmp de adresă de 24 de biți. Câmpurile alocate fiecărui registru sunt pe 8 biți în scopul suportării unei extensii ulterioare a arhitecturii de registre de 256 regiștri întregi și 256 flotanți. Câmpul *opcode* este pe 16 biți facilitând o decodificare rapidă a instrucțiunii. Deși, multiplele formate complică hardware-ul, prin păstrarea unor formate similare se poate reduce această complexitate. Astfel, primele două câmpuri ale tuturor formatelor prezentate mai sus sunt identice.

Setul de instrucțiuni proiectat suportă eventuale *adnotări* (câmp suplimentar pe 16 biți) - modificări, sintetizări a instrucțiunilor post-compilare - în fișierele asamblare, fără a fi necesară o recompilare a codului mașină. Există două tipuri de adnotări: pe bit sau la nivel de câmp. O adnotare pe bit este următoarea:

lw /a \$r6, 4(\$r7)

Adnotarea */a* urmărește setarea primului bit al câmpului adnotare. Adnotarea pe bit de la */a* la */p* stabilește setarea biților de la 0 la 15 din respectivul câmp. O adnotare pe bit are următoarea formă:

lw /6:4(7) \$r6, 4(\$r7)

Adnotarea de mai sus setează câmpul de 3 biți (de la 4 la 6) în interiorul câmpului de 16 biți la valoarea 7. Aceste adnotări ar putea conține informații utile hardului, furnizate de către compilator. Introducerea informațiilor de profil aferente fiecărei instrucțiuni poate fi necesară în implementarea diverselor arhitecturi sau tehnici novatoare de procesare (vezi predictorul hibrid cu selecție bazată pe aritate din subcapitolul 5.3.3.1, identificarea salturilor anterioare cu adevărat relevante pentru predicția salturilor curente). Prin intermediul lor, s-ar putea proiecta o interfață hardware – software extrem de utilă, care să faciliteze execuția *run-time* a instrucțiunilor.

Setul de instrucțiuni precum și apelurile sistem utilizate în cadrul setului de instrumente *SimpleScalar 3.0* se regăsesc în [Bur97, Flo03].

4. INFLUENȚA UNOR CONCEPTE DE PROGRAMARE PROCEDURALĂ / OBIECTUALĂ ASUPRA GENERĂRII SALTURILOR INDIRECTE

4.1. ANALIZA LIMBAJELOR C ȘI C++ DIN PUNCT DE VEDERE AL PROCESĂRII LOR PE ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCȚIUNILOR

Istoria procesoarelor contrapune două paradigme pentru creșterea performanței, bazate pe software și respectiv pe hardware. În ciuda scopului comun de exploatare și creștere a paralelismului la nivelul instrucțiunilor comunitatea cercetătorilor se împarte în două entități aproximativ „disjuncte” în încercările lor de a-l îndeplini. În timp ce arhitecții de calculatoare își canalizează eforturile pentru exploatarea / optimizarea tehnicilor de procesare existente prin simulări substanțiale pe programe de test reprezentative în format cod obiect, fără a ține cont de semantica codului sursă (H.L.L), autorii de compilatoare urmăresc optimizarea codului obiect, reducerea necesarului de memorie etc. Knuth afirmă în [Knu71], analizând comportamentul static și dinamic al unei bogate colecții de programe Fortran, că programatorii au o “săracă intuiție” în ce privește secțiunile de program mari consumatoare de timp. De aceea, cunoașterea profilului de execuție al programelor i-ar ajuta pe autorii acestora în efortul de îmbunătățire a performanțelor (creșterea vitezei de execuție a programelor). În sprijinul afirmațiilor lui Knuth se află celebra “regulă 90 / 10” care enunță că *cca. 90% din timpul de rulare al unui program se execută doar cca. 10% din codul acestuia*. Ideea că arhitectura procesoarelor interacționează "accidental" cu domeniul software este complet greșită, între hardware și software existând în realitate o simbioză și o interdependență puternică, încă neexplorate corespunzător. Procesoarele se proiectează odată cu compilatoarele care le folosesc iar relația dintre ele este foarte strânsă:

- benchmark-urile sunt compilate pentru arhitectura respectivă (Ex: *gcc* în sistemul de operare Linux poate transforma un cod sursă C în cod obiect fie pentru procesor Intel, fie pentru arhitectura virtuală SimpleScalar, în funcție de bibliotecile și instrumentele software folosite – asamblor, link-editor, interpretor) iar,
- compilatorul trebuie să genereze cod care să exploateze caracteristicile arhitecturale, altfel codul generat va fi inefficient.

Una dintre metodele moderne care se află la intersecția celor două tipuri de abordări se bazează pe clasificarea structurilor de program (*SingleLast*, *SingleStride*, *MixedPattern*) în funcție de arhitectura pe care rulează mai bine (vezi cazul predictoarelor de valori decuplate [Lee93]).

O altă soluție ar putea consta în modificarea compilatoarelor existente astfel încât codul obiect rezultat să fie executat foarte eficient pe arhitecturile implementate la ora actuală.

După cum este binecunoscut, tehnologia sistemelor de calcul se dezvoltă extrem de rapid (*Legea lui Moore*) făcând necesară apariția de noi programe de test standardizate care să reflecte îmbunătățirile tehnologice aferente microprocesoarelor, compilatoare noi, cele mai recente aplicații multimedia, compresii / transmisii de semnale audio / video / GSM. Cele mai reprezentative benchmark-uri care pot asigura un mediu adecvat de testare / simulare a ideilor arhitecturale novatoare sunt cele dezvoltate de consorțiul SPEC. Ultima versiune a acestor benchmark-uri (SPEC 2000) a fost lansată oficial la 30 iunie 2000 și cuprinde 26 de programe scrise în C, C++ și Fortran dintre care 19 în premieră [Henn00, SPEC]. Ținând cont că benchmark-urile SPEC'95 erau în întregime procedurale și datorită tendințelor, manifestate în ultimii ani, de trecere la limbajele obiectuale bazate pe concepte avansate: încapsulare, moștenire, polimorfism, era de așteptat ca noua suită (SPEC 2000) să conțină cât mai multe benchmark-uri obiectuale scrise în C++. Cu toate acestea, SPEC a primit ca propuneri din partea membrilor și a publicului larg, doar două aplicații scrise în C++ dintre care una nu s-a dovedit portabilă pe compilatorul *ANSI C++* (doar pe *gnu g++*) nefiind astfel votată de către membrii SPEC; cealaltă aplicație - *252.eon* - este însă extrem de portabilă fiind compilată "*fără probleme*" pe 17 compilatoare de C++ distincte.

Sunt doar câteva motive pentru care aplicațiile obiectuale pot fi considerate la momentul actual *o mare provocare* atât pentru comunitatea compilatoriștilor cât și pentru cea a arhitecților de microprocesoare. Provocarea este cu atât mai mare cu cât încă din 1994 cercetătorii americani [Cal94] au demonstrat, bazat pe simulare, că există diferențe semnificative între caracteristicile programelor procedurale (C) și cele ale programelor

obiectuale (C++), cu implicații asupra performanțelor acestor programe (viteză de execuție, consum de memorie) și asupra microarhitecturii.

Caracterizarea empirică a celor două tipuri de programe (C și C++) este utilă atât pentru realizatorii de compilatoare cât și pentru proiectanții de arhitecturi de calcul în încercarea acestora de a exploata eficient diferențele comportamentale dar și de a înțelege nivelul de complexitate al benchmark-urilor obiectuale. Astfel, se urmărește detectarea acelor optimizări benefice din aplicațiile scrise în C dacă sunt la fel de eficiente și în programele C++.

Programarea structurată (în limbajul C) este deficitară în ceea ce privește posibilitatea reutilizării programelor, scalabilității, mentenabilității, depanării și extinderii unor module de program. Pornind de la *„ecuația programării structurate”* enunțată de Niklaus Wirth:

Structuri de date + Algoritmi = Program

se poate afirma că, principala deficiență a programării procedurale constă în tratarea separată a algoritmilor și a structurilor de date ce se prelucrează [Aco02]. Prin analogie cu obiectele din natură, caracterizate atât prin structură, cât și prin comportament, poate fi enunțată o ecuație ce caracterizează programarea orientată pe obiecte (OOP):

Date + Metode = Clasă

Relația se bazează pe principiul fundamental al *„încapsulării datelor”*, conform căruia accesul la datele membre (atribute, proprietăți) se poate face numai prin intermediul metodelor asociate. Pe lângă încapsulare, concepte fundamentale ca polimorfism, moștenire (simplă, multiplă) urmăresc realizarea unei ierarhii de clase, care să modeleze sisteme complexe. Construirea ierarhiei de clase constituie activitatea esențială în realizarea unei aplicații orientate-obiect, practic faza de proiectare a respectivului sistem.

Stilul de programare orientată-obiect propune împărțirea aplicațiilor în mai multe module, astfel încât cel ce dezvoltă un modul nu trebuie să cunoască detaliile de implementare a altor module. Consecințele imediate sunt scăderea timpului de dezvoltare a aplicațiilor, simplificarea activității de întreținere a modulelor și creșterea calității programelor.

În continuare sunt ilustrate câteva din caracteristicile de program prin care aplicațiile procedurale diferă de cele obiectuale. Înainte însă, se va explica semnificația unora din termenii utilizați. Astfel, prin noțiunea de *“funcție C++”* se înțelege o funcție compilată cu un compilator de C++ (de exemplu *gnu g++*) în timp ce o *“funcție C”* este orice funcție compilată cu

un compilator de C (fie *gnu gcc*). *Funcțiile* (metodele) *membru C++* sunt acele funcții asociate obiectelor.

- Din punct de vedere static, multe programe obiectuale au un număr relativ redus de funcții C++ însă ele conțin un procent destul de ridicat de funcții C datorită utilizării funcțiilor de bibliotecă C predefinite.
- Programele obiectuale conțin în medie mai puține instrucțiuni per funcție comparativ cu cele procedurale putând beneficia astfel mai mult de pe urma tehnicii de “**in-lining**” aplicată funcțiilor. În plus metodele C++ conțin mai puține instrucțiuni decât funcțiile C++ (consecință a moștenirii care are printre principalele avantaje economia și reutilizarea de cod, polimorfism, extensibilitate [Brea02]).
- Din punct de vedere dinamic, funcțiile din limbajul C execută de aproximativ trei ori mai multe instrucțiuni decât funcțiile și metodele din limbajul C++. Dimensiunea funcțiilor reprezintă un factor important atât în optimizări precum macroexpandarea, fiind direct proporțională cu spațiul de “**overhead**” al fiecărei funcții - destinat salvării regiștrilor (scrieri / citiri repetate din stivă), transmiterii parametrilor, păstrării rezultatului, precum și în performanța cache-ului de instrucțiuni. Caracteristica programelor obiectuale de a realiza multe apeluri de funcții “*scurte*” (reduse ca număr de instrucțiuni) constituie unul din principalele motive pentru care acest tip de programe nu beneficiază de avantajele localității spațiale oferite de blocurile “*mari*” de cache. În cercetările sale, Calder afirmă că programele obiectuale necesită un cache de instrucțiuni de două mai mare pentru a obține rate de miss egale cu programele procedurale [Cal94].
- Dimensiunea *basic-block*-urilor este aproximativ identică atât în programele procedurale cât și în cele obiectuale și este situată sub valoarea de 6 instrucțiuni (5.4 instrucțiuni / *basic-block* în programele obiectuale **vs.** 5.9 instrucțiuni / *basic-block* în programele procedurale). Acest aspect este fundamental în ceea ce privește limitarea ratei de aducere a instrucțiunilor din memorie în cadrul procesorului (*fetch bottleneck*), sau în optimizarea codului executat pe o arhitectură prin scheduling static – metodă care poate duce chiar și la eliminarea miss-urilor de conflict în cache-ul de instrucțiuni.
- Cunoscută fiind tendința programelor obiectuale de a avea mai multe metode cu un număr redus de instrucțiuni, descrise la nivel de cod mașină prin intermediul apelurilor de funcții indirecte – mecanism furnizat de C++ și pentru implementarea moștenirii, se poate afirma că: „în timp ce programele C execută mai multe instrucțiuni de salt

condiționat (*bnez \$t0*), aplicațiile orientate pe obiecte execută mai multe apeluri de rutine (directe – *jal <nume_procedură>* și indirecte – *jr \$t1*) dar și reveniri – *jr \$ra* în rutina apelantă” [Cal94]. Bazat pe simulări laborioase [Cal94] s-a obținut că 23.1% din numărul total de apeluri dinamice de funcții sunt indirecte în cadrul programelor obiectuale, în timp ce acest procentaj pe testele procedurale este de doar 8.3%. Acest fapt subliniază necesitatea proiectării de scheme de predicție distincte pentru atingerea de acurateți ridicate de predicție la execuția celor două tipuri de programe (obiectuale respectiv procedurale). Trebuie amintită aici și afirmația lui Fisher [Fis92] care afirmă despre apelurile indirecte de funcții că reprezintă stagnări greu evitabile în procesarea fluxului de instrucțiuni și că există puține compilatoare sau artificii arhitecturale aplicabile la nivel hardware care să elimine aceste bariere în vederea creșterii paralelismului la nivelul instrucțiunilor. De exemplu, procesoarele superscalare DEC Alpha AXP 21064, respectiv Intel Pentium4, cu structuri pipeline extrem de complexe (pe lungime – faze de procesare, pe adâncime – număr de unități execuție) determină stagnarea și reexecuția unui număr de 10 [Cal94b], respectiv 20 [Intel02], instrucțiuni în cazul unei predicții eronate a fluxului de instrucțiuni. Penalitatea în timp poate crește dacă instrucțiunea țintă nu se află în cache și trebuie adusă din memoria principală.

- Pentru o mai bună înțelegere a comportamentului programelor, este de un real ajutor cunoașterea proporției (procentajul) de instrucțiuni statice (de salt indirect, condiționat) prezente în program și care au o semnificație deosebită în timpul rulării acestora. Pentru o cuantificare a termenului de “semnificație” s-a determinat câte instrucțiuni statice din program exprimă un procentaj **dat** din execuția tuturor salturilor efectuate dinamic. Rezultatele, extrem de interesante, au la bază observația că nu toate instrucțiunile de salt statice (mai ales cele condiționate) sunt executate în timpul procesării dinamice a codului sursă, deoarece ele rezidă în unele condiții de eroare, sau pot apare într-un anumit context la care nu se ajunge întotdeauna (salturi corelate – “ **if cond1**
if cond2 ”).

Atât pentru programele obiectuale cât și pentru cele procedurale, în medie 95% dintre salturile dinamice efectuate au la bază doar 10% dintre cele statice prezente în codul sursă, în timp ce, 50% dintre salturile dinamice sunt cauzate de mai puțin de 1% dintre salturile statice [Cal94]. De asemenea, se remarcă numărul relativ redus de

salturi indirecte statice prezente în programele de calcul, dar și faptul că sunt mai multe salturi indirecte “*semnificative*” în programele obiectuale (25%) decât în cele procedurale (10%) pe benchmark-urile măsurate de Calder [Cal94]). O consecință a acestor rezultate ar fi că, pentru multe din programele de calcul, prin tehnica de *branch prediction* nu ar fi necesară predicția tuturor instrucțiunilor de salt, ci mult mai eficientă ar fi predicția corectă a unui procentaj cât mai însemnat dintre salturile importante (“semnificative din punct de vedere dinamic”).

- În stabilirea **diferențelor** comportamentale dintre programele procedurale și cele obiectuale un rol important îl au compilatoarele folosite pentru generarea codului pseudo-obiect (necesar asamblării și link-editării cu bibliotecile aferente în vederea obținerii codului obiect). În urma simulărilor, cercetătorii au sesizat pe lângă asemănări și unele diferențe în urma compilării cu DEC C++ respectiv GNUC++ [Cal94, Aig96, Dri98, Flo04]. La aceste aspecte se poate adăuga și afirmația lui Stroustrup care afirmă că: “*utilizatorii au început să folosească C++ înainte ca specialiștii să aibă timpul necesar să-l instruiască pentru a-l folosi cu randament maxim*” [Aco02]. Într-adevăr, s-a constatat că o mare parte dintre compilatoarele de C++ existente nu sunt folosite decât pentru dezvoltarea de software structurat și nu orientat-obiect (altfel spus, se lucrează în C pe un compilator de C++). Personal, am dezvoltat unele programe de test simple care relevă deosebiri în generarea codului cu compilatorul de C al firmei Borland (BCC) față de cel obținut prin compilarea cu GNU C. Programele asamblare rezultate în urma compilării cu GNU C conțin anumite salturi indirecte „*pure*” care nu se regăsesc în urma compilării aceluiași cod sursă C cu BCC. (Ex: un program care folosește o instrucțiune de selecție *switch/case* pentru identificarea numărului de cifre, spații și a altor caractere dintr-un șir citit de la tastatură).

Codul generat cu GNUC++ este puțin mai inefficient decât cel compilat cu DECC++. Programele compilate cu G++ execută cu 2.5% mai multe instrucțiuni și apelează cu 8% mai multe funcții. Cu toate că numărul de instrucțiuni de salt executate este aproape identic, programele compilate cu G++ generează cu aproape 28% mai puține salturi condiționate. Majoritatea acestor diferențe apar datorită bibliotecilor dinamice de funcții și se manifestă și în cazul programelor procedurale.

- Așa cum compilatorul influențează comportamentul programelor obiectuale respectiv procedurale și optimizările din timpul link-editării

constituie un factor important în analiza performanțelor celor două tipuri de aplicații (C și C++).

Limbajele orientate obiect (C++) permit extinderea ierarhiei claselor fără afectarea funcționalității procedurilor anterior compilate. Spre exemplu, un programator poate folosi clasa X și să compileze unele module folosind doar interfața clasei respective. Dacă ulterior, se declară o clasă Y, derivată din X – aflată în programul original, atunci se poate opera atât cu instanțe ale clasei de bază cât și cu ale celei derivate dar optimizările permise prin folosirea clasei Y nu vor fi însă detectate. În general, aceste optimizări nu vor fi detectate până în momentul link-editării, când toate modulele (corpurile metodelor, funcțiilor) vor fi asamblate și legate dinamic. Doar în acest moment, ierarhia de clase este complet vizibilă pentru compilator și doar atunci pot fi considerate optimizările specifice clasei derivate.

Într-o altă lucrare Calder afirmă că bazat pe informații de profil (*type feedback*), prin optimizări realizate în momentul link-editării (*macroexpandarea* funcțiilor și execuția lor condiționată) are loc o reducere a numărului de apeluri indirecte de funcții cu 31% [Cal94b]. În ciuda unor avantaje evidente (predictibilitate mai mare a apelurilor directe de funcții, reducerea numărului de instrucțiuni executate pe o arhitectură RISC modernă) există și unele dezavantaje și anume:

- ❖ nu întotdeauna prin „*inlining*” aplicat funcțiilor se reduce timpul de execuție [Dav92]. O explicație ar putea consta în expansiunea codului după aplicarea acestei tehnici și creșterea ratei de miss în cache-ul de instrucțiuni. Măsurători efectuate de cercetători pe 8 programe de test C++ (porky, richards etc) înglobând peste 90.000 de linii de cod, compilate cu ajutorul GNU C++ 2.6.3. pe un procesor SuperSPARC cu sistem de operare Solaris 2.5 au indicat o creștere medie a codului cu 9% după aplicarea tehnicii de „*inlining*” funcțiilor [Aig96].
- ❖ eficiența metodei este ridicată dacă există o mare posibilitate de apel repetat a respectivei funcții.
- ❖ dependența de arhitectură [Cal94] și deci lipsa de compatibilitate respectiv portabilitate.

Pentru o mai bună înțelegere, se exemplifică considerând o „*clasă*” oarecare, având o metodă (fie „*foo*”) și două sau mai multe obiecte din această clasă (fie A, B, ...). Atunci apelul indirect de funcție, determinat de apelul metodei „*object->foo()*,” poate fi convertit la un apel direct de procedură completat cu verificarea în timpul execuției

programului a tipului obiectului, cu avantajele și dezavantajele amintite anterior:

```

if (typeof(object) == A)
    object->A::foo();
else
    object->foo();

```

- Deși programele procedurale au nevoie de mai puțin spațiu pe stivă decât cele obiectuale, “*adâncimea stivei*” este destul de mică pentru ambele tipuri de programe (9.9 - C vs 12.1 - C++) fiind puternic determinată de domeniul aplicațiilor și tehnicile de programare folosite (recursivitate, backtracking, programare dinamică) [Cal94].
- Operațiile cu memoria (citiri / scrieri) influențează semnificativ performanța arhitecturilor moderne de procesare – prin penalități “serioase” în timp în cazul în care procesorul nu dispune de datele necesare (este binecunoscut decalajul dintre *performanța procesoarelor* care se dublează la fiecare doi ani și timpul mediu de acces la memoria principală (DRAM) care scade cu doar aproximativ 7% pe an). Mecanismele hardware moderne (memorii cache, scheme diferite de predicție a valorilor, buffer-ele de reutilizare) contribuie la eliminarea acestor penalități, nemaifiind necesar accesul la memoria principală. Programele obiectuale trimit spre execuție mai multe instrucțiuni Load și Store decât programele procedurale (însă cu un procentaj nesemnificativ, între 2÷4%); comportamentul fiind însă diferit în funcție de arhitectură (MIPS pe 32 biți, Alpha pe 64 biți). Metodele par să necesite mai multe instrucțiuni Load decât celelalte funcții C++, iar funcțiile apelabile indirect tind să utilizeze puțin mai multe Load-uri decât apelurile directe de funcții, cel puțin în C++ (acces la tabelele de metode virtuale [Roth99]). Contribuția semnificativă la procentajul ridicat de operații cu memoria în C++ se datorează și salvărilor / restaurărilor de regiștri din timpul apelurilor și revenirilor din proceduri; se reamintește faptul că programele obiectuale execută mult mai multe apeluri de funcții și reveniri decât programele procedurale.
- Programele obiectuale alocă mult mai multă memorie în “heap” pentru obiectele folosite în aplicații decât pentru variabilele dinamice din structurile de date aferente programelor procedurale. Dimensiunea obiectelor din programele C++ este destul de redusă, ceea ce face ca traficul cu memoria să fie extrem de ridicat. Prin timpul de viață scurt al acestor obiecte se reduce însă complexitatea interfeței aplicației.

Probabil că unul din principalele motive pentru care gradul de utilizare a memoriei în programele C++ este semnificativ îl reprezintă tendința programelor obiectuale de a crea și folosi componente reutilizabile.

Un alt motiv poate fi de natură istorică. Capacitatea redusă a memoriei sistemelor din anii 70 – 80 (limbajul C a fost conceput de către Kernigan și Ritchie în anul 1972) a condus la constrângeri privind execuția programelor procedurale într-un spațiu mic de adrese (de memorie) și alocări reduse pe stivă sau în “heap”, ceea ce la limbajul C++ (mult mai nou –1983) nu mai este cazul (executabil portabil pe arhitecturi moderne de procesare, fără prea mari constrângeri din punct de vedere al capacității resurselor).

Un aspect important privind diferențierea dintre limbajele C și C++ îl constituie modul în care acestea asigură suportul pentru alocarea dinamică în memoria “heap”. În timp ce C asigură câteva funcții simple de bibliotecă (malloc, calloc), C++ furnizează acest suport atât cu ajutorul funcțiilor de bibliotecă (new, delete) cât și prin constructori (impliciți, expliciți) dar și prin destructori.

Cu toate că benchmark-urile SPEC reprezintă cel mai adecvat mediu pentru testarea / simularea ideilor novatoare din domeniul arhitecturii calculatoarelor, cele mai multe sisteme de calcul au ca platformă sistemul de operare Microsoft Windows (9x, NT, 2000, XP) cu procesor Intel din familia x86. Multe dintre acestea execută aplicații de uz personal (multimedia, editare, calcul tabelar, prelucrare de imagini, email, navigare pe Internet) și nu bazate pe cercetări științifice, de uz industrial, militar. În acest sens, vor fi descrise în continuare câteva caracteristici ce privesc aplicațiile la nivel desktop executate sub sistemul de operare WindowsNT pe sisteme cu procesor Intel x86, comparativ cu cele aferente benchmark-urilor SPEC’95.

Un studiu sumar dezvăluie câteva diferențe evidente existente între aplicațiile desktop și benchmark-urile tradiționale care intuitiv sugerează diferențe atât în utilizarea resurselor hardware cât și în comportamentul programelor la nivel microarhitectural (salturi indirecte, condiționate, utilizare funcții de bibliotecă).

- ☐ Deși *caracterul interactiv al aplicațiilor desktop poate presupune un comportament propriu mai puțin predictibil*, din punct de vedere al predicției instrucțiunilor de salt, performanțele acestora sunt similare cu cele ale programelor de test SPEC’95 [Lee98], subliniind faptul că gradul de nepredictibilitate introdus de utilizator este mai degrabă la un nivel masiv de granularitate (*coarse*) decât la unul care să afecteze

microarhitectura (*fine*). Spre exemplu, în spatele unui *click de mouse* din cadrul aplicației Microsoft Office Winword.exe se află chiar sute de instrucțiuni de salt condiționate. Intervenția utilizatorului în acest caz constă în a spune programului “*ce să facă*” (click de mouse la locația (x,y) – pe o anumită resursă) și nu “*cum să facă*” (cum să trateze respectiva întrerupere – implementată în rutina de tratare a “*click*”-ului de mouse).

- ▣ Pe lângă caracterul interactiv, *majoritatea aplicațiilor desktop sunt caracterizate și de o interfață grafică intuitivă, bazate pe meniuri, pagini de proprietăți, ferestre de dialog etc.* În timp ce rezultatele benchmark-urilor tradiționale constau în câteva pagini de fișiere text, *ieșirile aplicațiilor desktop sunt atât sub format text dar mai ales grafic* (controale de tip ActiveX, OLE, tabele Excel etc) *ceea ce se traduce la nivel microarhitectural în cod suplimentar de executat, necesitând resurse adiționale* (structuri de date). Cu toate acestea, din punct de vedere al cache-ului de date și al buffer-ului TLB de date comportamentul aplicațiilor desktop este similar cu cel al benchmark-urilor SPEC’95.
- ▣ În timp ce benchmark-urile tradiționale de cele mai multe ori execută un singur “task” (program) la un moment dat, *majoritatea aplicațiilor desktop efectuează și alte funcții pe lângă “task”-ul principal* (de exemplu, desenarea unor grafice MicrosoftExcel Chart în cadrul unei prezentări Powerpoint). *Aceste caracteristici conduc la coduri executabile mai mari și apeluri de proceduri dincolo de spațiul de adresă alocat inițial.* Practic, diferența semnificativă dintre aplicațiile desktop și programele de test SPEC’95 o reprezintă dimensiunea foarte mare a codului obiect aferent primelor (de 3 până la 10 ori mai mare decât a celor din urmă) [Lee98]. Numărul funcțiilor statice existente într-o aplicație și executate cel puțin o dată contribuie substanțial la dimensiunea codului obiect (fișierul executabil) al aplicației respective, cu implicații defavorabile asupra performanței (consum ridicat de resurse). Din acest punct de vedere, aplicațiile desktop apelează cu cel puțin un ordin de mărime mai multe funcții statice decât programele de test SPEC’95, fapt deloc surprinzător întrucât dimensiunile funcțiilor sunt sensibil egale în ambele cazuri iar codul obiect al aplicațiilor desktop este de până la 10 ori mai mare. Tendința execuției programelor desktop de dispersare în mai multe funcții crește probabilitatea de apariție a miss-urilor de capacitate și conflict la cache-urile de instrucțiuni. Același comportament defavorabil îl prezintă și TLB-ul de instrucțiuni [Lee98].

- ☐ *Multithread*-ingul caracterizează aproape toate aplicațiile desktop constituind și o modalitate de a masca operațiile cu latențe mari (o parte dintre ele introduse prin intervenția utilizatorului). Simulări efectuate pe cinci aplicații desktop reprezentative (*acord32* - cititor de documente în format portabil, *netscape* - "browser" de Internet, *photoshp* - pachet de editare de imagini, *powerpnt* - pachet util în realizarea de prezentări, *winword* - procesor de texte) au arătat că patru dintre ele execută în mai mult de 97% din timp instrucțiuni dintr-un singur thread, rezultând un impact minim asupra performanței datorat disputei între fire asupra resurselor [Lee98].
- ☐ O altă diferență majoră constă în faptul că aplicațiile desktop folosesc foarte mult bibliotecile legate dinamic (DLL), multe dintre acestea partajate între diferite alte aplicații. DLL-urile cuprind servicii ale sistemelor de operare, funcții de interfață, biblioteci grafice, funcții C de bibliotecă, etc. Utilitatea DLL-urilor constă atât în economisirea spațiului de memorie precum și în crearea unei interfețe standard cu utilizatorul. Simulări realizate pe aceleași cinci aplicații desktop [Lee98] au demonstrat utilizarea în medie a 30 de DLL-uri dintre care între 21 și 25 au fost partajate cu alte aplicații. De asemenea, s-a remarcat că aplicațiile desktop execută în medie 11% din instrucțiunile lor în trei biblioteci sistem (*user32.dll*, *gdi32.dll* și *kernel32.dll*) în timp ce benchmark-urile SPEC'95 folosesc doar *ntdll.dll* și *kernel32.dll*, fiecare dintre acestea contribuind cu mai puțin de 1% din totalul instrucțiunilor procesate de respectivele benchmark-uri.

Pentru a determina și a înțelege cum bibliotecile legate dinamic sunt folosite de către aplicațiile desktop, trebuie analizați doi parametri: lungimea unui DLL - numărul de instrucțiuni executate între apelul unui DLL și revenirea din acesta, respectiv rezidența într-un DLL - numărul de instrucțiuni executate din cadrul unui singur DLL până la trecerea într-altul. Bazat pe simulare a rezultat că majoritatea DLL-urilor sunt scurte și implică apelul altor DLL-uri [Lee98]. Practic, în jumătate din apeluri lungimea acestora este mai mică sau egală cu 45 de instrucțiuni iar în 95% din apeluri dimensiunea maximă a acestora este de 571 de instrucțiuni. În ceea ce privește rezidența într-un DLL s-a observat că majoritatea secvențelor execută mai puțin de 60 de instrucțiuni în cadrul unui anumit DLL înainte de a preda controlul altui DLL.

Există câteva motive pentru care apelurile de biblioteci legate dinamic (DLL) sunt considerate mai costisitoare decât apelurile funcțiilor legate static, și anume:

- **Apelurile DLL sunt implementate ca și apeluri indirecte de funcții.** Simulări efectuate pe cele cinci aplicații desktop amintite anterior [Lee98] au relevat un procentaj ridicat (în medie de 25%) de apeluri indirecte (pure + DLL) comparativ cu unul extrem de redus (sub 4%) obținut pe benchmark-urile SPEC'95 (vezi și subcapitolul 7.1.1 al prezentei lucrări). Ținând cont de concluzia lui Calder [Cal94] care susține că programele obiectuale generează mai multe apeluri indirecte datorită legării dinamice prin polimorfism, față de programele procedurale, am fi tentați să credem că cele cinci aplicații desktop sunt în întregime obiectuale. În realitate doar *netscape*, *photoshp* și *powerpnt* sunt obiectuale (scrise în C++) în timp ce *winword* și *acrord32* sunt implementate în C [Lee98].
- DLL-urile sunt partajate între aplicații astfel încât îmbunătățirea localității instrucțiunilor prin algoritmi tradiționali de reordonare a codului, statici sau dinamici, este mai dificilă [Pett90]. Afirmatia are la bază experiența și realizările a doi cercetători în optimizare de cod, angajați ai firmei Hewlett-Packard, Pettis și Hansen, care pornind de la limitările tehnicilor de optimizare tradiționale bazate pe "eliminarea / adăugarea" unor instrucțiuni, au propus o tehnică nouă de optimizare globală "*procedure splitting*", în vederea reducerii ratei de miss la cache-ul de instrucțiuni și îmbunătățirea performanței arhitecturilor de calcul. Având la bază informații de profil culese în urma execuției anterioare a programelor, *procedure splitting* urmărește partajarea în zone fizice de memorie distincte a basic-block-urilor executate mai frecvent (codul util) și separat basic-block-urile executate foarte rar sau deloc. Prin această separare se creează noi proceduri, mai scurte și mai dense (executate în întregime mult mai des) și care pot fi stocate împreună într-un număr mai mic de pagini de memorie decât inițial, când codul util era dispersat. Basic-block-urile foarte rar executate sunt mutate la sfârșitul procedurii, pagina care le cuprinde nefiind necesar a fi încărcată în memorie decât foarte rar. De asemenea, în urma aplicării acestei tehnici, la implementarea mecanismului de memorie virtuală, dimensiunea paginii de memorie poate fi aleasă mai mică, de capacitate optimă însă, astfel încât să compenseze timpul mare de acces la disc. *Procedure splitting* a fost implementată cu succes în cadrul optimizatorului de cod realizat pentru procesorul PA-RISC al companiei Hewlett-Packard [Pett90].
- Apelurile frecvente de DLL-uri din cadrul altor DLL-uri necesită ca paginile de memorie cu ambele biblioteci să fie rezidente în spațiul de adrese aferent aplicației chiar dacă doar o fracțiune redusă din aceste

pagini sunt de fapt folosite, în caz contrar creându-se o fragmentare internă.

Foarte pe scurt, aplicațiile desktop executate pe platformă WindowsNT cu procesor Intelx86 prezintă comportamente similare cu benchmark-urile tradiționale (SPEC'95) pentru o serie de metrici (influența cache-ului și a buffer-ului de translatat de adrese, dimensiunea funcțiilor, dimensiunea basic-block-urilor, procentajul tipurilor de instrucțiuni în cadrul aplicațiilor) dar și diferă substanțial în cel puțin două privințe: dimensiunea codului obiect al aplicației respectiv procentajul semnificativ de salturi indirecte introdus de apelurile DLL-urilor.

În continuare, subcapitolul 4.2 descrie câteva modalități de generare a apelurilor indirecte prin intermediul unor aplicații originale atât procedurale cât și obiectuale, iar în subcapitolul 4.3 sunt prezentate un număr de salturi dinamice indirecte cu caracter polimorf observate în urma simulării benchmark-urilor SPEC.

Capitolul de față încearcă să investigheze cele două "*emisfere*", hardware și software, doar aparent disjuncte, în care își desfășoară activitatea cercetătorii din știința calculatoarelor. Preocupările programatorilor nu trebuie să vizeze doar interfața care atrage sau diversele artificii care fac din utilizator un simplu robot ci și implicațiile pe care aplicația creată o are asupra microarhitecturii. Scopul aplicației trebuie să fie utilizarea cu justete atât a resurselor software (biblioteci, elemente de interfață) avute la dispoziție cât și a algoritmilor / conceptelor de programare cunoscute (declarații de funcții virtuale, apeluri de funcții prin pointer chiar și acolo unde nu este cazul). În caz contrar, "*răul*" (a se citi în primul rând salturi indirecte, cod obiect masiv, resurse hardware suplimentare) se răsfrânge asupra performanțelor arhitecturii. În ce-i privește pe proiectanții de arhitecturi, schemele propuse de aceștia ar putea fi mai eficiente dacă nu ar analiza numai codul obiect al benchmark-urilor avute la dispoziție (dezbrăcat de orice semantică) ci ar privi "*mai sus*" spre sursa de nivel înalt a programelor simulate.

4.2. INVESTIGAȚII PRIVIND GENERAREA SALTURILOR / APELURILOR INDIRECTE PRIN APLICAȚII PROCEDURALE RESPECTIV OBIECTUALE

Pentru o mai bună înțelegere a ceea ce va urma se reamintesc câteva noțiuni legate de metodele virtuale respectiv legarea statică versus dinamică. Preluat după alți autori [Brea02], este numită **legare** (“binding”) mecanismul prin care compilatorul generează cod pentru apelul unei anumite metode (cum face legătura între numele metodei și codul acesteia). Există două posibilități:

- **legarea statică** (timpurie – “**early binding**”) – caz în care compilatorul și editorul de legături fac corespondența între numele metodei și adresa acesteia din segmentul de cod. Odată fixată această legătura **în momentul compilării** ea este definitivă în tot timpul rulării programului. La nivelul cel mai de jos, legarea statică corespunde apelului direct al unei proceduri. Acesta este modul de legare clasic folosit de toate limbajele structurate procedurale și de limbajele obiectuale pentru metodele ne-virtuale (clasice).
- **legarea dinamică** (târzie – “**late binding**”) – situație în care compilatorul și editorul de legături fac corespondența între numele metodei și adresa acesteia din segmentul de cod prin intermediul unui *tabel de adrese posibile de salt*. Alegerea adresei efective de apel se face **în momentul rulării**, în funcție de obiectul pentru care se apelează (deci, decizia legării este întârziată). La nivelul cel mai de jos, legarea dinamică corespunde apelului indirect al unei proceduri (folosind de exemplu o instrucțiune de tip **jal \$reg** – pentru procesorul MIPS). Acesta este modul de legare folosit de limbajele obiectuale pentru metodele virtuale.

Polimorfismul, fiind un exemplu de soluție având la bază indirectarea, este caracterizat de următoarele avantaje și dezavantaje ale acestor metode:

- **avantaje**: se asigură un grad de **flexibilitate**.
- **dezavantaje**: ca în orice apel indirect, se înrăutățesc performanțele de **viteză** (datorită dificultății predicției acestor apeluri).

Pentru fiecare clasă compilatorul construiește o tabelă cu metodele virtuale (la offset-uri predeterminate) ale clasei respective – VMT și include în fiecare instanță (obiect) a clasei un pointer *VMT_ptr* spre tabela VMT corespunzătoare clasei (vezi figura 4.1). Pentru o metodă ne-virtuală legarea este necondiționat statică iar pentru o metodă virtuală legarea este

necondiționat dinamică (atât la apel direct prin obiect sau prin pointer la obiect cât și la apel din cadrul altor metode) [Brea02].

Întrucât target-urile apelurilor de funcții virtuale sunt determinate de tipul obiectului este de înțeles faptul că metodele virtuale care folosesc ca referință același obiect sunt puternic corelate (se găsesc în aceeași tabela VMT). În consecință, de îndată ce prima adresă destinație aferentă unui apel de metodă virtuală este cunoscută, următoarele apeluri de metode virtuale generate de același obiect pot fi cu ușurință prezise prin tehnici bazate pe istorie (vezi structura TargetCache din subcapitolul 5.2.1). Însă predicția adresei destinație aferentă primului apel indirect este foarte dificilă, șansele de succes ale schemelor bazate pe istorie fiind doar dacă obiectele sunt ordonate după tip, ceea ce nu se întâmplă întotdeauna; ordinea obiectelor poate fi aleatoare – cazul benchmark-ului *richards* (simulator de sistem de operare) sau dependentă de intrarea aplicației – cazul benchmark-urilor *porky* respectiv *ixx* (parser IDL) [Roth99].

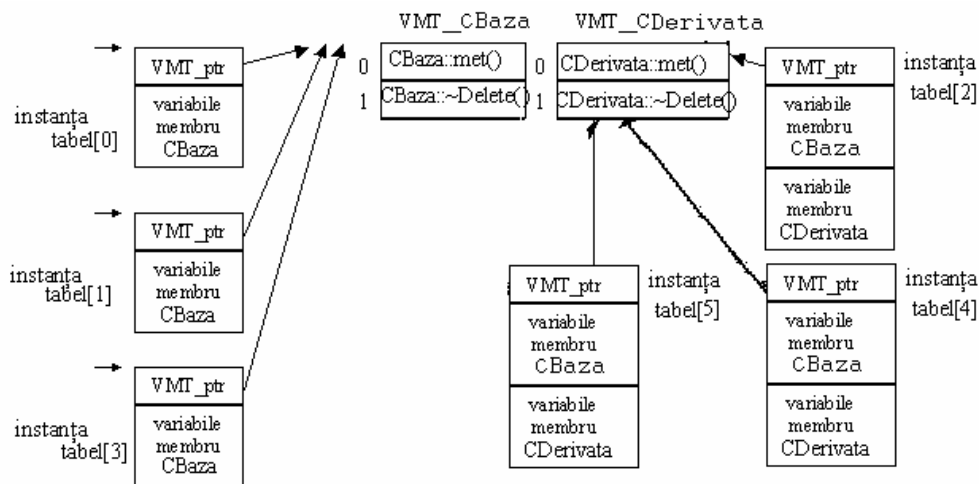


Figura 4.1 Tabelele VMT aferente aplicației propuse (**moștenire_simplă_3.cpp**)

O abordare complementară schemei de predicție a salturilor indirecte bazate pe istorie constă în **pre-calcularea target-urilor** apelurilor de funcții virtuale (**v-call**) folosindu-se dependențele dintre instrucțiunile de tip *Load* și cele de *salt indirect*. Tehnica identifică dinamic secvențe de operații ce concură la determinarea respectivelor target-uri. La întâlnirea primei instrucțiuni din secvență un “*mic motor de execuție*” speculativ și agresiv efectuează în avans restul operațiilor. Adresa destinație precalculată astfel este memorată, fiind folosită abia când în secvența de instrucțiuni ce urmează se ajunge la saltul indirect care trebuie predicționat [Roth99].

Acuratețea scăzută generată de schemele de predicție bazate pe istorie în cazul apelurilor indirecte de funcții este datorată dificultății de a identifica cu precizie aceste apeluri în momentul predicției; practic, salturile condiționate nu reflectă *contextul exact* în care apar salturile indirecte. **Folosind informații extrase din momentul execuției programului mecanismul de pre-calculare a target-urilor instrucțiunilor de salt indirect se bazează mai degrabă pe instrucțiunile care concură la calcularea acestui target decât pe adresele destinație anterioare ale respectivei instrucțiuni de salt** (vezi cazul Target Cache-ului).

Pre-calcularea target-urilor necesită o perioadă scurtă de “învățare” și poate fi aplicată cu succes chiar și în lipsa informațiilor statistice de corelare. Deși este o tehnică de uz general care poate fi aplicată cu succes oricărui tip de instrucțiune, pre-calcularea target-urilor apelurilor de metode virtuale se bazează pe o implementare hardware simplă, care extinde *mecanismul de prefetch aplicat structurilor de date cu legături* [Roth98]. Prin această metodă crește acuratețea de predicție față de un predictor BTB cu 46%, iar față de unul corelat pe două niveluri cu 24% [Roth99].

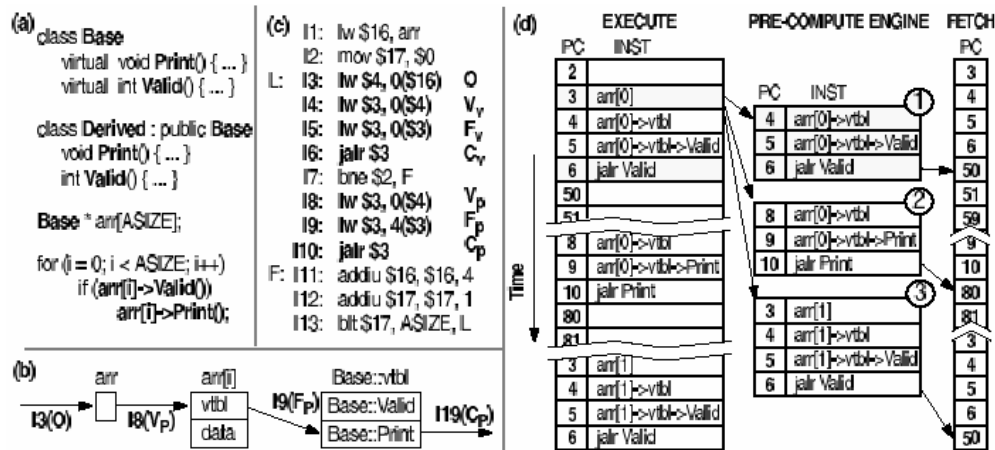


Figura 4.2 Mecanismul de pre-calculare a adreselor destinație aferente apelurilor de metode virtuale

Polimorfismul permite programatorului tratarea uniformă a unui tablou de obiecte de clase diferite (**Base** și **Derived** în exemplul din figura 4.2., similare cu **CBaza** și **CDerivata** din aplicația *moștenire_simplă3.cpp* (vezi figura 4.1) sau **Back** respectiv **Dame** din aplicația *moștenire_simplă1.cpp*) și folosirea apelurilor de metode virtuale pentru alegerea corectă a funcțiilor în fiecare caz în parte. Practic legarea dinamică realizată prin polimorfism [Brea02] se traduce prin accesarea tabelii de metode virtuale folosind o secvență de trei instrucțiuni *load* dependente

urmate de un *apel indirect de funcție* (mecanism numit OVFC [Roth99], vezi figura 4.2).

Mecanismul de pre-calculare cuprinde trei componente principale: primul este responsabil cu detecția dependențelor reale de date dintre instrucțiunile *load* precum și dintre ultima instrucțiune *load* și cea de *salt indirect*, urmată de reprezentarea internă a acestor dependențe. A doua componentă o reprezintă un motor simplu de execuție care folosește reprezentarea internă a dependențelor de date pentru procesarea anticipată a instrucțiunilor din lanțul care se încheie cu instrucțiunea de salt indirect. Ultima componentă colectează rezultatele anterior calculate, în principal adresa destinație a instrucțiunii de salt indirect și o predă structurii de predicție principale din cadrul procesorului fiind necesară abia în momentul execuției efective a acesteia.

Pe exemplul din figura 4.2, primul Load (**I3**) accesează adresa de bază a obiectului (**O**), al doilea (**I4**) folosește adresa de bază a acestuia pentru a accesa tabela metodelor virtuale VMT (**V**). De la deplasamentul corespunzător din VMT, cu cea de-a treia instrucțiune de tip Load (**I5**) este preluată adresa metodei virtuale (**F**) care este utilizată în apelul indirect (**C**). Secvențele OVFC corespondente metodelor virtuale Valid() și Print() sunt ilustrate cu litere aldine în figura 4.2. De asemenea, puțin mai târziu pe parcursul acestui capitol, s-a prezentat mecanismul OVFC corespondent metodei *met()* virtuală și a destructorului virtual din cadrul aplicației propuse de autor *moștenire_simplă3.cpp*.

De îndată ce instrucțiunea I3 (**O**) se încheie, se consultă structura internă de reprezentare a dependențelor și se execută anticipat lanțul de instrucțiuni I4-I6 (VFC). Rezultatul precalculat constituie adresa destinație a instrucțiunii care urmează lui I6 (I50 în acest exemplu).

Se observă că ambele secvențe VFC folosesc referința la același obiect O. Astfel, o singură secvență de acest gen poate fi atașată dinamic mai multor obiecte, folosind instrucțiuni condiționate. Întrucât apelurile metodelor virtuale (în exemplul dat) se referă la un singur obiect mecanismul se numește *pre-calculare simplă a target-urilor*. Dacă în loc de obiecte izolate se folosesc tablouri sau liste de obiecte (structuri de date cu legături) mecanismul se numește *pre-calculare "n-lookahead"*, unde *n* reprezintă indexul obiectului în respectiva structură. În acest caz, adresa obiectului aflat la un anumit deplasament poate fi și ea prezisă pornind de la adresa de bază a tabloului de obiecte și spațiul de memorie consumat de fiecare obiect.

Ineficiența uneori a mecanismului de pre-calculare a target-urilor apelurilor de metode virtuale se datorează fie timpului insuficient de

calculare a acestuia (nu există suficiente instrucțiuni între apel și referința la obiectul respectiv), fie incapacității de a predicționa corect adresa următorului obiect.

În continuare sunt prezentate integral sau sumar câteva programe de test simple propuse de autor [Flo03a], care relevă **caracteristicile de program** prezente în programele de nivel înalt **procedurale**, sau **concepte fundamentale ale programării orientate pe obiecte** care conduc la **generarea salturilor indirecte la nivel de cod asamblare / obiect**.

Un prim exemplu ilustrează cum se reflectă o “**moștenire simplă**” la nivelul codului obiect. Aplicația **moștenire_simplă1.cpp** (**moștenire_simplă1.s** după compilare) rezolvă prin *backtracking recursiv* două probleme. Prima se referă la generarea permutărilor de n elemente iar a doua problemă urmărește așezarea a n regine pe tabla de șah fără ca ele să se atace. Este implementată o clasă de bază **back** având ca atribute (membre) stiva cu soluții, dimensiunea stivei (n -ul citit de la tastatură), iar ca metode rutina principală de execuție a backtracking-ului (**compl**), metoda de afișare a soluțiilor (**show**), metoda care testează dacă soluția parțială este validă (**valid**), metoda care indică dacă s-a ajuns la ultimul element de pe stivă (**final**), ultimele trei metode fiind declarate virtuale. Din clasa de bază se derivează clasa **dame** care moștenește membre (atribute, metode) ale clasei **back** și supraîncarcă cele două metode virtuale cu codul propriei aplicații.

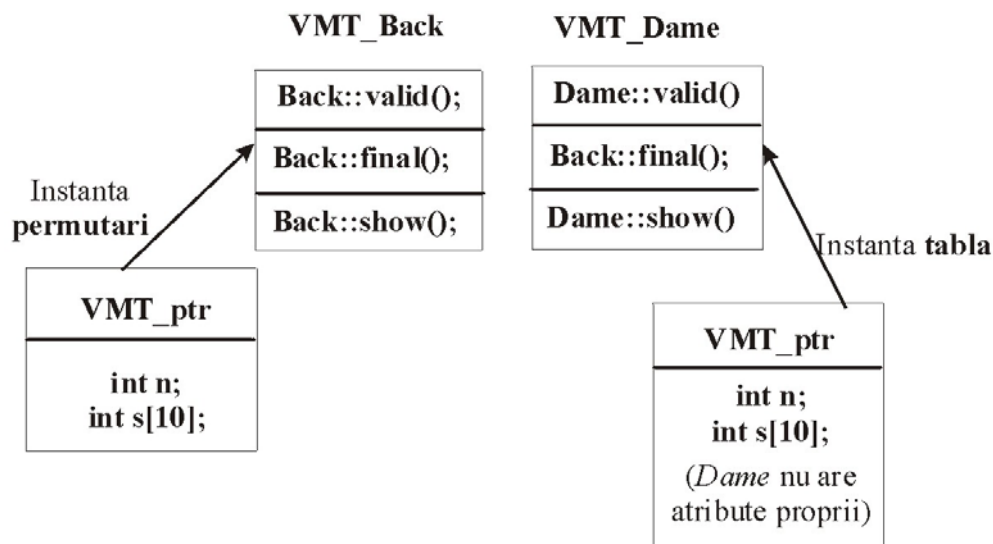


Figura 4.3. Tabelele VMT aferente aplicației **moștenire_simplă1.cpp**

Declarațiile de obiecte pot fi statice sau dinamice. Rezultatele obținute în urma compilării determină prezența a *3 funcții apelabile indirect*, datorate

în opinia autorului, celor trei apeluri de metode declarate virtuale din metoda **back::compl**. Din punct de vedere dinamic, cele 3 salturi indirecte statice determină **322.561** salturi indirecte dinamice (dacă se consideră o tablă de șah obișnuită $n=8$). De asemenea, numărul instrucțiunilor executate este 302.538.522, instrucțiuni de salt dinamice 55.529.324 (statice doar 585) și instrucțiuni de revenire din subrutină 3.899.790 (statice - 57).

Surprinzător la o primă vedere, rezultatele statistice (parametrul – **număr salturi indirecte statice**) obținute în urma compilării diferă puțin de cele obținute după simularea pe arhitectura SimpleScalar a codului obiect al respectivei aplicații. Cu toate acestea ele sunt justificate. Practic, metoda **final** declarată virtual în clasa de bază **back** nu va fi supraîncărcată în clasa derivată (**dame**). Astfel, datorită acestei declarații, la apelul metodei **final** în cadrul metodei **compl** (rutina principală din aplicație), la compilare va fi generat un salt indirect (**jal \$31,\$2 – trei în total**, două datorate celorlalte metode apelate – **show** și **valid**). În realitate, când va fi executată metoda **compl**, apelată printr-un obiect al clasei derivate nu se va apela altă metodă **final** decât tot cea din clasa de bază (identică ca într-o legare statică – *final nevirtuală*). Practic în statistica generată dinamic prin execuția acestui benchmark pe arhitectura SimpleScalar vor rezulta din **14 salturi indirecte statice** (generate după faza de link-editare și execuție cu $n=4$) **trei** datorate apelurilor virtuale **valid**, **final** și **show** (la momentul compilării), celelalte **11** fiind datorate funcțiilor de bibliotecă. O cauză a acestei situații poate consta în optimizările realizate de compilatorul **xgcc** în momentul link-editării (asemeni *in-lining*-ului coroborat cu verificarea tipului obiectelor din [Cal94b], vezi și subcapitolul 4.1). Faptul că metoda **final**, deși virtuală este doar moștenită și nesupraîncărcată în clasa derivată (**dame**), este reflectat în tabela metodelor virtuale aferentă clasei **dame** (vezi figura 4.3), constituindu-se într-un real suport pentru compilator ca în momentul execuției apelul indirect generat de respectiva metodă să se transforme într-unul direct. Tehnica de optimizare statică care realizează acest lucru se numește "*analiza ierarhică a claselor*" [Aig96]. Evident în acest caz este trivială declarația metodei **final** ca virtuală, însă ea a fost introdusă și pentru extinderea aplicației cu alte clase pentru rezolvarea *aranjamentelor, combinărilor, produsului cartezian* și a altor aplicații reprezentative, dar în care metoda **final** va trebui supraîncărcată.

*****Secvență de cod din **moștenire_simplă1.cpp*******

```

class back{
public:
    int n,s[10];
    back *x;
    back();
    void compl(int);
    virtual int valid(int );
    virtual int final(int );
    virtual void show(int );
};

class dame:public back{
public:
    dame():back(){};
    int valid(int );
    void show(int );
};

void back::compl(int p){
    int v;
}

for(v=1;v<=n;v++){
    s[p]=v;
    if(valid(p))
        if(final(p))show(p);
        else compl(p+1);
}

void main(){
    ...
    back* permutari;
    permutari = new back;
    ...
    permutari->compl(0);
    ...
    dame *tabla;
    tabla = new dame;
    tabla->compl(0);
    ...
}

```

*****Secvență de cod din **moștenire_simplă1.s*******

```

.ent    compl_4backi
compl_4backi:
    .subu    $sp,$sp,40          # alocare spațiu pe stivă și salvare adresă de revenire, pointer de
    cadru, eventual alți parametri
    sw     $31,$32($sp)
    sw     $fp,$28($sp)
    sw     $16,$24($sp)
    move   $fp,$sp
    move   $16,$4
    sw     $5,$4($fp)
    ...
$L58:
    lw     $2,$4($fp)
    move   $3,$2
    sll   $2,$3,2
    addu  $3,$2,$16
    addu  $2,$3,4
    move   $3,$2
    lw     $2,$16($fp)
    sw     $2,$0($3)
    lw     $3,$48($16)
    addu  $2,$3,8
    move   $3,$2
    move   $2,$3
    lh     $3,$0($2)
    addu  $4,$16,$3
    lw     $2,$4($2)
    lw     $5,$44($fp)
    jal   $31,$2                # apel virtual metoda valid(p) în sursa .cpp
    sltu  $3,$0,$2
    move  $2,$3

```

```

andi    $3,$2,0x00ff
beq     $3,$0,$L59
lw      $3,48($16)
addu   $2,$3,16
move    $3,$2
move    $2,$3
lh      $3,0($2)
addu   $4,$16,$3
lw      $2,4($2)
lw      $5,44($fp)
jal   $31,$2      # apel virtual metoda final(p) în programul sursă
sltu   $3,$0,$2
move    $2,$3
andi    $3,$2,0x00ff
beq     $3,$0,$L60
lw      $3,48($16)
addu   $2,$3,24
move    $3,$2
move    $2,$3
lh      $3,0($2)
addu   $4,$16,$3
lw      $2,4($2)
lw      $5,44($fp)
jal   $31,$2      # apel virtual metoda show(p) în sursa .cpp
j       $L61
$L60:
...
```

Cele trei apeluri indirecte apar doar în metoda **compl()** a clasei **back** (de bază) deoarece aceasta este moștenită și legată static în clasa derivată (**dame**). Dacă **compl()** ar fi fost virtual declarată în clasa de bază și supraîncărcată în clasa derivată atunci cele trei metode componente (*valid*, *final* și *show*) ar fi generat un număr dublu de instrucțiuni de salt indirect în funcție de numărul de obiecte sau pointeri la obiecte din cele două clase. Un exemplu în acest sens îl constituie programul de test **back_.cpp** care extinde problema rezolvată în **moștenire_simplă1.cpp** cu o nouă clasă (**cinci**).

Ulterior am analizat problema eliminând declarația ca virtuală a metodei **final**. În acest sens, după compilare a rezultat în locul saltului indirect anterior (**jal \$31, \$2**) un apel direct de subrutină (**jal final_4backi**), deci statistic au fost generate la nivel de cod asamblare doar **două** salturi indirecte dintr-un total de **13** statice (în aceleași condiții ca mai sus). Evident rezultatele dinamice ale simulării au fost identice cu cele obținute anterior (când metoda **final** era virtuală).

Metodele **clasei cinci** încearcă să rezolve următoarea problemă: *să se genereze toate numerele de 5 cifre care dau prin suma cifrelor valoarea 20*. Dintr-un total de **22** de salturi statice indirecte (rezultate după link-editare) doar **11** sunt generate după faza de compilare (datorate metodelor **compl**, **final**, **valid**, **show** declarate virtuale în clasa de bază - **back** și supraîncărcate în clasa derivată - **cinci**). Dinamic au fost generate **145236**

instrucțiuni de salt indirect. Declarația a doar 3 pointeri la obiecte din 3 clase diferite pe parcursul execuției programului implică un grad ridicat de localitate a valorii aferent instrucțiunilor de salt indirect dovedind astfel predictibilitatea sporită a acestora (vezi rezultatele grafice din capitolul 7.1.2).

O altă aplicație elocventă, simplă și tot orientată obiect, se bazează pe conceptul de **moștenire simplă, polimorfism, conversii de tip între clase derivate, alocare dinamică de obiecte și utilizare constructori / destructori în contextul derivării**. Întrucât o clasă derivată poate fi privită ca o extensie a clasei de bază C++ se permite conversia unui obiect de clasă derivată la un obiect al clasei de bază iar pointerii la clasa de bază pot ține adresa unei instanțe a clasei derivate [Brea02]. Prin legarea dinamică pe care o presupune, polimorfismul permite o tratare uniformă a masivelor eterogene tablouri, liste de obiecte. De asemenea, la distrugerea unui obiect dintr-o clasă derivată, întâi se execută corpul destructorului clasei derivate și abia apoi destructorul clasei de bază.

Aplicația **moștenire_simplă3.cpp[.s]** realizează **tratarea uniformă a unui tablou de obiecte**, alocate dinamic, prin polimorfism. Clasa **Baza** cuprinde un constructor explicit simplu (o atribuire aplicată membrului clasei), o metodă virtuală (un banal “*printf*”) și un destructor virtual (tot un “*printf*”).

Clasa **Derivată**, pe lângă un alt atribut cuprinde aceeași metodă și un destructor similar, dar nevirtuale. Programul principal presupune existența unui tablou de 6 obiecte dinamice care parțial aparțin clasei de bază și parțial clasei derivate. Dimensiunea tabloului poate fi parametrizabilă, cu cât acest parametru este mai mare cu atât numărul de apeluri indirecte de instrucțiuni dinamice este mai mare. Urmează apoi două bucle de program în care fiecare obiect creat apelează metoda “**met**” aferentă clasei din care face parte iar ulterior este distrus.

În urma compilării rezultă **două** apeluri indirecte de funcții statice în fișierul **.s** și **11** astfel de apeluri după simularea codului obiect pe arhitectura SimpleScalar. Primul apel **jal \$31,\$2** se datorează apelului metodei virtuale “**met**” aferentă obiectelor dinamice anterior definite – în bucla *for*. Apelul indirect de funcție se justifică prin necunoașterea exactă în momentul compilării a metodei care va fi executată: cea din clasa derivată sau cea de bază. Al doilea apel **jal \$31,\$2** apare în cazul apelării destructorilor virtuali (nu se cunoaște în momentul compilării din ce clasă face parte, iar în cazul în care obiectul aparține clasei derivate vor fi apelați destructorii ambelor clase – întâi cel al clasei derivate și apoi cel al clasei de bază). Din punct de vedere dinamic apelurile statice indirecte determină **72** de salturi indirecte

dinamice. Rezultatele statistice bazate pe simularea codului obiect (opțiune de compilare **-o**) prezintă cu **9** apeluri statice mai mult decât s-a determinat în urma compilării (opțiune de compilare **-S**), probabil datorate apelurilor funcțiilor de bibliotecă.

În continuare se descrie comparativ codul sursă - de nivel înalt (**.cpp**) și cel asamblare - apropiat de arhitectură (**.s**) rezultat în urma compilării, pentru a ilustra cum **legarea dinamică (binding) realizată prin polimorfism** generează apeluri indirecte de funcții. Asupra codului obiect obținut se poate aplica cu succes metoda de **predicție a target-urilor apelurilor de funcții virtuale prin pre-calcularea adresei destinație** bazată pe dependențe de date [Roth99]. Sunt prezentate doar secvențele de cod cele mai reprezentative.

*****Secvență de cod din **moștenire_simplă3.cpp*******

```
#include <stdio.h>
class Baza {
public:
    int x;
    Baza(int x0) {x=x0;}
    virtual ~Baza() { printf("\nDestructor clasa de baza
(%d)",x);}
    virtual met() { printf("\n Metoda clasa de baza
(%d)",x);}
};

class Derivata:public Baza
{
public:
    int y;
    Derivata(int x0, int y0):Baza(x0) {y=y0;}
    ~Derivata() { printf("Destructor clasa derivata (%d,
%d)", x, y);}
    met(){printf("\nMetoda clasa derivata (%d, %d)", x,
y);}
};

void main()
{
    Baza *tabel[6];
    tabel[0]=new Baza(1);
    tabel[1]=new Baza(2);
    tabel[2]=new Derivata(3, 4);
    tabel[3]=new Baza(5);
    tabel[4]=new Derivata(6, 7);
    tabel[5]=new Derivata(8, 9);

    for(int k=0; k<6; k++)
        tabel[k]->met();

    for(k=0; k<6; k++)
        delete tabel[k];
}
```

*****Secvență de cod din **moștenire_simplă3.s*******

```
.text
...
main: # programul principal în care are loc alocarea dinamică de memorie pentru fiecare obiect prin intermediul
constructorului explicit
...
subu    $sp,$sp,56          # alocare de spațiu pe stivă
sw     $31,52($sp)        # întrucât apelurile de funcții salvează în r31 adresa de revenire în programul
apelant, în cazul apelurilor
```

```

# recursive r31 trebuie salvat în stivă
sw    $fp,48($sp)
move  $fp,$sp
jal   __main
li    $4,0x00000008 # 8
jal   __builtin_new
move  $4,$2
li    $5,0x00000001 # 1
jal   __4Bazai      # obiect din clasa de bază
sw    $2,16($fp)
...
li    $4,0x0000000c # 12
jal   __builtin_new
move  $4,$2
li    $5,0x00000003 # 3
li    $6,0x00000004 # 4
jal   __8Derivataii # obiect din clasa derivată
sw    $2,24($fp)
...
$L48:
lw    $2,40($fp)
move  $3,$2
sll   $2,$3,2
addu  $3,$fp,16
addu  $2,$2,$3
move  $3,$2
lw    $2,0($3)      # registrul $2 va conține adresa de bază a obiectului - Omet
lw    $3,4($2)      # registrul $3 va conține adresa de bază a tabeli VMT - Vmet
                        # (afată la un deplasament față de adresa obiectului) – vezi figura
                        # 2.1
addu  $2,$3,16      # metoda virtuală este la un deplasament de 16o ⇔
                        # tabel[k]->met();
move  $3,$2
move  $2,$3
lw    $3,40($fp)
move  $4,$3
sll   $3,$4,2
addu  $4,$fp,16
addu  $3,$3,$4
move  $4,$3
lh    $3,0($2)
lw    $4,0($4)
addu  $3,$3,$4
lw    $2,4($2)      # registrul $2 va conține adresa metodei obiectului care va fi
                        # apelată - Fmet
move  $4,$3
jal   $31,$2      # apelul indirect al metodei virtuale - Cmet
...
$L52:
lw    $2,40($fp)
move  $3,$2
sll   $2,$3,2
addu  $3,$fp,16
addu  $2,$2,$3
move  $3,$2
lw    $2,0($3)
beq   $2,$0,$L53
lw    $2,40($fp)
move  $3,$2
sll   $2,$3,2

```



```

addu $3,$fp,16
addu $2,$2,$3
move $3,$2
lw $2,0($3) # registrul $2 va conține adresa de bază a obiectului - OD
lw $3,4($2) # registrul $3 va conține adresa de bază a tabeli VMT - VD
# (aflată la un # deplasament față de adresa obiectului) – vezi
# figura 2.1

addu $2,$3,8 # metoda virtuală este la un deplasament de 8o ⇔ delete tabel[k];
move $3,$2
move $2,$3
lw $3,40($fp)
move $4,$3
sll $3,$4,2
addu $4,$fp,16
addu $3,$3,$4
move $4,$3
lh $3,0($2)
lw $4,0($4)
addu $3,$3,$4
lw $2,4($2) # registrul $2 va conține adresa Destructorului de obiect care va
# fi apelată - FD

move $4,$3
li $5,0x00000003 # 3
jal $31,$2 # apelul indirect al destructorului virtuale - CD
...

```

A treia aplicație exemplifică un **apel indirect de funcție prin pointer** (cazul aplicațiilor de sortare selectate printr-o **construcție de tip switch/case**). Aplicația sortează aleator (pe baza unui contor generat de o funcție) printr-una din metodele clasice de sortare implementate Bubblesort, Quicksort, SelectionSort, InsertionSort. În urma compilării au rezultat **5 apeluri indirecte statice de funcție** la *apelul unor funcții prin pointeri*, construcții de tipul *switch/case* (în aplicația de nivel înalt C). Diferența de la 5 apeluri statice de funcții (obținute după faza de compilare) la 14 apeluri statice (generate după link-editare) se datorează apelului indirect a altor funcții de bibliotecă folosite în cadrul aplicației (*printf*, *scanf*, *rand*). Din punct de vedere dinamic numărul apelurilor indirecte de funcții a fost 1035.

În continuare se vor prezenta comparativ, exemplificându-se la nivel de instrucțiune, secvențe din codul sursă **.c** (de nivel înalt) versus **.s** (asamblare) al aplicației **sort.[c,s]**.

```

*****Secvență de cod din sort.c*****
int Nr_metoda;                               Nr_metoda=x;
... // alte declarații: variabile și        return 0;
prototipuri de funcții                       }
int AfisareMetoda(int (*p)());
int GenMetoda();                             int GenMetoda(){
...                                           return rand() % 5;
}
int AfisareMetoda(int (*p)())
{
int x=p();
switch (x){
case 0: printf("\nBubbleSort");
break;
case 1: printf("\nQuickSort");
break;
case 2: printf("\nSelectSort");
break;
case 3: printf("\nInsertSort");
break;
case 4:
printf("\nInsertSortModif");
break;
}
}

void main()
{
int i, x, k, nr_sort;
printf("\nNr sortari = ");
scanf("%d",&nr_sort);
for (k=0; k<nr_sort; k++)
{
...
AfisareMetoda(GenMetoda);
... // apelul diverselor metode de
sortare
}
}

```

```

*****Secvență de cod din sort.s*****
.align 2
$LC1: .ascii "\n" # stocarea în zona de date a adreselor de mesaje: numele funcțiilor
      .ascii "BubbleSort\000"
      .align 2
$LC2: .ascii "\n"
      .ascii "QuickSort\000"
      .align 2
$LC3: .ascii "\n"
      .ascii "SelectSort\000"
      .align 2
$LC4: .ascii "\n"
      .ascii "InsertSort\000"
      .align 2
$LC5: .ascii "\n"
      .ascii "InsertSortModif\000"
      .text # declararea numelui funcțiilor ca pointer global (în acest
            # caz AfisareMetoda)
      .align 2
      .globl AfisareMetoda
      .align 2
$LC6: .ascii "\n"
      .ascii "Nr sortari = \000" # mesaj folosit în cadrul funcției
...
.ent AfisareMetoda

```

AfisareMetoda:

```

subu $sp,$sp,40          # alocare spațiu pe stivă și salvare parametri:
sw $31,$32($sp)        # adresa de revenire în programul apelant - $31
sw $fp,$28($sp)        # frame pointer-ul
sw $16,$24($sp)
move $fp,$sp
sw $4,$40($fp)

lw $16,$40($fp)

    jal $31,$16
    # salvare pe stivă a parametrului funcției ($4): adresa
    # funcției GenMetoda
    # preluare din stivă în registrul $16 a adresei funcției
    # GenMetoda în vederea apelului indirect de funcție
    # apelul este indirect pentru că în momentul compilării din
    # punct de vedere al metodei AfisareMetoda nu este
    # cunoscut exact numele funcției, ci doar adresa acesteia.
    # La revenirea din funcția GenMetoda rezultatul acesteia,
    # aflat în registrul $2 [F1003] va fi salvat în stivă. Acesta
    # va reprezenta pentru AfisareMetoda o opțiune de
    # selecție în cazul mecanismului switch/case.

lw $2,$16($fp)
sltu $3,$2,$5          # Există 5 opțiuni de sortare: de la 0 la 4. Se compară dacă
                      # rezultatul funcției GenMetoda se află printre cele 5
                      # valori posibile

beq $3,$0,$L11        # în caz contrar efectuându-se salt la $L11 – ieșirea din
                      # funcția AfisareMetoda.

lw $2,$16($fp)
move $3,$2
sll $2,$3,$2          # Calcul deplasament față de adresa de bază a instrucțiunii
                      # switch (adresele instrucțiunilor fiind pe 4 octeți – vezi
                      # directivele .word începând cu adresa $L10).
                      # Deplasamentul va fi multiplu de 4.

la $3,$L10            # În $3 se încarcă adresa de bază a instrucțiunii switch –
                      # aferentă primului case:

addu $2,$2,$3         # Calcul adresă exactă instrucțiune de selecție.
move $3,$2
lw $2,$2,0($3)       # Preluarea de la adresa anterior determinată a
                      # instrucțiunii de selecție efective - fie case 3:

    j $2
    # Execuție salt necondiționat de la PC-ul instrucțiunii
    # determinată anterior.

.rdata
.align 3
.align 2
$L10:
.word $L5            # Adresele la care se găsesc parametrii de afișare pentru
                    # fiecare din cazuri

.word $L6
.word $L7
.word $L8
.word $L9
.text

$L5:
la $4,$LC1
jal printf          # Apelul funcției de bibliotecă printf după încărcarea în
                    # prealabil în $4 a parametrilor de afișare

j $L4              # Salt necondiționat la ieșirea din funcția AfisareMetoda.

$L6:
la $4,$LC2
jal printf
j $L4

$L7:
la $4,$LC3

```

```

        jal    printf
        j      $L4
$18:
        la    $4,$LC4
        jal    printf
        j      $L4
$19:
        la    $4,$LC5
        jal    printf
        j      $L4
$11:
$14:
        lw    $2,16($fp)           # rezultatul returnat de funcția GenMetoda aflat pe stivă
                                   # este adus în $2 și
        sw    $2,Nr_metoda        # stocat în memorie în variabila globală Nr_metoda
        move  $2,$0               # rezultatul returnat de AfisareMetoda (adică 0) se va
                                   # depune în $2.

$13:
        j      $L3
        move  $sp,$fp             # revenire din funcția AfisareMetoda.
        lw    $31,32($sp)        # preluare din stivă a adresei de revenire în programul
                                   # apelant
        lw    $fp,28($sp)
        lw    $16,24($sp)
        addu  $sp,$sp,40
        j      $31
        .end  AfisareMetoda

main:
        ...
        la    $4,GenMetoda       # pregătire parametrii de apel funcție AfisareMetoda.
        jal    AfisareMetoda
        lw    $2,Nr_metoda       # la revenirea din funcție se citește variabila Nr_metoda
                                   # necesară ulterior.
        ...

```

Construcțiile de tip `switch/case` sunt foarte des utilizate în fazele de analiză (lexicală, sintactică) componente ale compilatoarelor. Întrucât schemele de predicție bazate pe istorie, focalizate pe îmbunătățirea acurateții de predicție prin optimizarea algoritmilor sau varierea numărului de instrucțiuni de salt care determină contextul de apariție al saltului, sunt ineficiente în cazul acestor construcții, a fost propus un nou mecanism **Case Block Table (CBT)** menit atât să reducă numărul instrucțiunilor de salt greșit predicționate cât și numărul instrucțiunilor executate dinamic [Kae97]. CBT folosește valoarea variabilei de selecție (fie `x` în instrucțiunea **switch (x)**), pentru a găsi adresa exactă a următoarei instrucțiuni (blocul *case* care se va executa). Deoarece CBT înregistrează comportamentul anterior al fiecărui bloc, asociind adresei destinație a instrucțiunii de salt (`switch`) valoarea variabilei care determină selecția unui anumit bloc, structura se dovedește a fi un predictor optimal după ce fiecare bloc a fost accesat cel puțin o dată.

Pentru fiecare bloc *case* va exista o intrare unică în CBT aferentă fiecărei valori unice de selecție. O intrare în CBT este alcătuită din adresa de start a blocurilor (adresa instrucțiunii *switch*), valoarea operandului de selecție și adresa destinație a saltului (adresa exactă a blocului *case* care se va executa în continuare). CBT este accesat în momentul în care este recunoscută adresa de început a unei construcții *switch/case*. În tabela CBT are loc o căutare asociativă după adresa instrucțiunii de salt *switch* și valoarea operandului de selecție. În caz de hit se execută un salt necondiționat la adresa țintă (a noului bloc) din locația corespondentă din CBT.

Pentru a permite structurii CBT să identifice corect adresa de intrare și ieșire în / din construcția *switch/case*, setul de instrucțiuni al procesorului trebuie îmbogățit cu două instrucțiuni:

- **BEGINCASE(VARADDR)** – unde VARADDR reprezintă numele variabilei (operandului de selecție) folosit pentru căutarea în tabela CBT. Funcțional nu realizează nimic însă servește la identificarea adresei de start a construcției *switch/case*.
- **ENDCASE(TARGADDR)** – unde TARGADDR specifică adresa de ieșire din construcția *switch/case*. Funcțional nu realizează nimic.

Instrucțiunile de salt condiționat existente în interiorul construcției *switch/case* compilate, delimitate de **BEGINCASE** și **ENDCASE**, nu vor fi predicționate cu predictorul folosit în mod obișnuit, în cazul unei căutări cu hit în tabela CBT acestea fiind evitate prin *bypassing*, prevenind astfel predictorul să facă un număr suplimentar de predicții greșite. În caz de miss după execuția instrucțiunilor de salt condiționat și aflarea adresei destinație a noului bloc *case* aceasta va fi introdusă împreună cu adresa instrucțiunii **BEGINCASE** și valoarea operandului de selecție într-o locație din CBT.

Rezultate statistice bazate pe simulare obținute pe trei dintre benchmark-urile SPEC'95 (*gcc*, *li*, *espresso*), caracterizate de un procentaj ridicat de construcții *switch/case*, au determinat o reducere a numărului de predicții greșite cu 4.5% iar a numărului de instrucțiuni cu 9%, îmbunătățind astfel performanța [Kae97]. Dezavantajul utilizării tablei CBT îl constituie indisponibilitatea valorii operandului de selecție la timp pentru extragerea din cache a instrucțiunii din subcazul *target* corespunzător, în cazul procesoarelor *superscalare* cu execuție speculativă.

O soluție alternativă pentru predicția construcțiilor *switch/case* poate consta în utilizarea unei table de salturi care să stocheze *target*-urile fiecărui subcaz în parte, tabela fiind indexată cu valoarea operandului de

selecție. Avantajul CBT față de această structură constă doar în reducerea căii de execuție a programului.

Întrucât sistemele dedicate în timp real cunosc o largă răspândire la ora actuală iar propriile aplicații implementate software rulează pe procesoare cu puteri de calcul limitate, optimizarea codului devine o necesitate. Câteva din aceste optimizări vizează și construcțiile **switch/case**.

Exemplul anterior (aplicația sort.[c,s]) arată cum instrucțiunea de selecție multiplă **switch/case** generează salturi indirecte în programele procedurale. În continuare se va arăta tot bazat pe exemple concrete cum o aceeași instrucțiune se comportă diferit din punct de vedere al codului asamblare generat în urma compilării, în funcție de numărul de opțiuni (cazuri ce trebuie tratate).

switch 04.cpp	switch 05.cpp
<pre>#include <stdio.h> void main(){ char c; int v=random(4); switch (v){ case 0: c='a'; break; case 1: c='b'; break; case 2: c='c'; break; case 3: c='d'; break; } }</pre>	<pre>#include <stdio.h> void main(){ char c; int v=random(5); switch (v){ case 0: c='a'; break; case 1: c='b'; break; case 2: c='c'; break; case 3: c='d'; break; case 4: c='e'; break; } }</pre>
switch 04.s	switch 05.s
<pre>... li \$3,0x00000001 # 1 beq \$2,\$3,\$L29 slt \$3,\$2,2 beq \$3,\$0,\$L34 beq \$2,\$0,\$L28 j \$L33 \$L34: li \$3,0x00000002 # 2 beq \$2,\$3,\$L30 li \$3,0x00000003 # 3 beq \$2,\$3,\$L31 j \$L33</pre>	<pre>... sltu \$3,\$2,5 # There are 5 options for the selection parameter beq \$3,\$0,\$L34 # If the selection parameter value is bigger than 5 # (number of options) a jump is executed outside of # the switch/case construction (\$L34 label) lw \$2,20(\$fp) # It is computed the offset for the base address of # switch/case statement sll \$2,\$3,2 # Every subcase address is a word (4 bytes) la \$3,\$L33 # In \$3 it is loaded the base address of switch # instruction proper to the first case: addu \$2,\$2,\$3 # It is computed the right (correct) address of # selection instruction move \$3,\$2 # Taking from the just computed address of the lw \$2,0(\$3) # first effective instruction (let it be case 3: - \$L31) j \$2 # The sequence is composed by a load instruction # followed of an indirect jump (unconditional jump to # the first instruction of specifical case) rdata .align 3 .align 2 \$L33: # The base address of switch/case statement from .word \$L28 # the application data segment storing the five .word \$L29 # addresses of every possible case. .word \$L30 .word \$L31 .word \$L32</pre>
a)	b)

Figura 4.4. Comportament diferit al construcției **switch/case** din punct de vedere al generării statice a salturilor indirecte

Astfel, dacă instrucțiunea de selecție multiplă oferă mai mult de 5 ramuri posibile (cazurile testate nu cuprind o situație implicită *default*), în codul limbaj de asamblare rezultat apare o instrucțiune de salt indirect, în caz contrar aceasta fiind înlocuită de o secvență de salturi condiționate. Compilarea surselor s-a făcut cu GNU C++ 2.6.3. și opțiunea de optimizare `-O3` pe un procesor Intel80x86 și sistem de operare Linux Red Hat 7.3. Opțiunea `-O3` pe lângă reducerea dimensiunii codului și a timpului de execuție permite inlining-ul aplicat funcțiilor și aplicarea *delay slot*-ului pentru instrucțiunile de salt (*f_inline_functions*, *f_delayed_branch*).

Semantica instrucțiunii `switch/case` se “traduce” în limbaj de asamblare într-o secvență de cel puțin două instrucțiuni: una de încărcare (**li \$3, 0x3**) respectiv una de salt condiționat (**beq \$2, \$3, \$L31**) – vezi figura 4.4. Este de înțeles atunci că, la un număr mare de opțiuni numărul instrucțiunilor de executat (introduse prin aceste opțiuni) se dublează. Problema se poate rezolva prin scrierea mult mai **generică** a secvenței de cod, introducându-se însă în locul instrucțiunilor de salt condiționat una de salt indirect și a unui tablou de adrese de subcazuri în memoria de date aferentă aplicației. Pragul de *5 ramuri ale instrucțiunii de selecție* a fost ales întrucât, până la inclusiv 4, numărul instrucțiunilor folosite pentru emularea construcției `switch/case` a fost mai mic sau egal decât numărul instrucțiunilor din secvența generică (cu salt indirect vezi figura 4.4(a, b)). Practic prețul **genericității** aplicației la nivel înalt se plătește la nivelul codului obiect întrucât salturile indirecte sunt foarte dificil de predicționat și mai mult, o singură instanță statică a acestora poate determina și 6000 de instanțe dinamice (cazul benchmark-ului *go.ss* din SPEC’95 [Flo04, Dri98a]).

Privind prin prisma construcției `switch/case` cu cel mult 4 opțiuni, în care compilatorul generează o cascadă de instrucțiuni **if-else-if** pentru compararea variabilei de selecție cu adresa fiecărui caz în parte, este justificată încercarea de optimizare a codului prin amplasarea cazului cel mai frecvent executat pe prima poziție în lista de opțiuni. În această situație numărul de comparații efectuate este automat diminuat. Identificarea celui mai frecvent caz se poate face cu ajutorul informațiilor de profil. Un exemplu în care această tehnică de optimizare ar avea realmente succes este cazul benchmark-ului SPEC’95 **li.ss** în care funcția *livecar* conține o instrucțiune de selecție multiplă cu un caz (corespondent tipului *LIST*) care este accesat cu o frecvență de peste 80% [Watt01].

O metodă de **optimizare bazată pe reducerea numărului de apeluri indirecte prin transformarea acestora în salturi condiționate**, folosită în situația existenței unui număr suficient de mare de ramuri (>4) ale

instrucțiunii de selecție multiplă **constă în imbricarea construcțiilor switch/case**. Cazurile frecvente sunt plasate ca și ramuri (efective) ale instrucțiunii de selecție multiplă, în timp ce cazurile mai puțin frecvente sunt tratate pe varianta implicită printr-o nouă construcție switch/case (vezi figura 4.5).

```

pMsg = ReceiveMessage();
switch (pMsg->type) {
    case Mesaj1_FRECVENT:
        tratează_Mesaj1Frecvent();
        break;

    case Mesaj2_FRECVENT:
        tratează_Mesaj2Frecvent();
        break;
    ...
    case Mesajn_FRECVENT:
        tratează_MesajnFrecvent();
        break;

    default: // construcția switch pentru tratarea mesajelor infrecvente.
        switch (pMsg->type) {
            case Mesaj1_INFRECVENT:
                tratează_Mesaj1Infrecvent();
                break;

            case Mesaj2_INFRECVENT:
                tratează_Mesaj2Infrecvent();
                break;
            ...
            case Mesajn_INFRECVENT:
                tratează_MesajnInfrecvent();
                break;
        }
}

```

Figura 4.5. Separarea cazurilor frecvente de cele mai puțin frecvente prin imbricarea construcțiilor **switch/case**

În situația în care construcția de bază switch/case (neoptimizată) ar avea cel mult 8 ramuri atunci prin optimizarea propusă anterior s-ar putea elimina cu succes saltul indirect, fiind înlocuit cu salturi condiționate.

La o primă vedere pot părea surprinzătoare rezultatele obținute după compilarea respectiv simularea execuției codului obiect pe arhitectura SimpleScalar a unui program de test extrem de simplu care cuprinde doar funcția main și care nu face efectiv nimic **main**}. Codul asamblare salvează în stivă *frame pointer*-ul și *adresa de revenire* în sistemul de operare (contextul din care a apărut) – vezi și secvențe de cod din programul de test **qsort.s**, efectuează un apel direct de subrutină **jal __main** (rutină aferentă

sistemului de operare). Se restaurează din stivă *registrul de revenire* și *frame pointer*-ul după care printr-un **return** (j \$31) se revine în sistemul de operare – în contextul din care a fost lansat. Aparent nici un salt indirect. La simularea codului obiect însă rezultă **6** apeluri statice indirecte care generează **8** apeluri dinamice indirecte. De asemenea, un fapt interesant obținut în urma simulării pe toate programele de test folosite (inclusiv SPEC '95) sugerează că **există mai multe apeluri call dinamice decât cele return dinamice**.

Un exemplu suplimentar în sprijinul afirmației că “**funcțiile de bibliotecă determină salturi indirecte** (*new, qsort, scan, printf*)” se constituie programul de test **qsort[.c,.s]**.

Codul asamblare nu prezintă nici un apel indirect de funcții de bibliotecă (*jal qsort, jal strcmp*). La analiza codului obiect însă (după link-editarea sursei cu bibliotecile în cauză) au rezultat **11** apeluri statice indirecte care au generat **61** apeluri dinamice indirecte.

Programul sursă C – preluat din help-ul oferit de mediul BorlandC – sortează un tablou de șiruri de caractere prin intermediul funcției de bibliotecă *qsort* care primește ca parametri adresa tabloului, numărul de elemente, dimensiunea fiecărui element și o funcție de comparare de două șiruri de caractere. Practic o sursă a celor 11 apeluri indirecte o constituie și **apelul indirect prin pointer la funcția de comparare** – pas realizat în cadrul funcției *qsort* (*precompile*).

În continuare se exemplifică pe baza descrierii funcției din fișierul **...\BorlandC\Crtl\Clib\qsort.cas**.

Nume funcție **qsort** – sortează un tablou de elemente pe baza metodei de sortare rapidă "selecția *elementului median din trei*", prin apelul repetat al unui pointer la o funcție definită de utilizator (**(*fcmp)**).

Utilizare **void qsort(void *base, int nelem, int width, int (*fcmp)());**

Prototipul funcției se află în **stdlib.h**.

**fcmp* *funcție de comparare* care acceptă două argumente: *elem1* și *elem2*, adresele a două elemente ale tabloului de sortat. Rezultatul returnat este următorul:

Dacă elementele sunt în relația	<i>fcmp</i> returnează
*elem1 < *elem2	Un întreg negativ (<i>elem1</i> se va afla în tabloul sortat

	înaintea	lui
	elem2)	
*elem1 == *elem2	0	
*elem1 > *elem2	Un	întreg
	pozitiv (>0)	

Qsort nu returnează nimic.

În continuare se vor prezenta comparativ, exemplificându-se la nivel de instrucțiune, secvențe din codul sursă **.c** (de nivel înalt) versus **.s** (asamblare) al aplicației **qsort.[c,s]**.

*****Secvență de cod din **qsort.c*******

```
#include <stdio.h>          int main(void)
#include <stdlib.h>         {
#include <string.h>         int x;

char list[5][4] = { "cat", "car", "cab",
"cap", "can"};           qsort((void *)list, 5, sizeof(list[0]),
                          sort_function);
                          for (x = 0; x < 5; x++)
int sort_function( const void *a, const void
*b)                       printf("%s\n", list[x]);
                          return 0;
                          {
                          }
                          return( strcmp((char *)a,(char *)b) );
                          }
}
```

*****Secvență de cod din **qsort.s*******

```
.globl list
.data
.align 2
list:
.ascii "cat\000"           # stocarea în zona de date a adreselor de mesaje:
                          # elementele de tablou care vor fi sortate.
.ascii "car\000"
.ascii "cab\000"
.ascii "cap\000"
.ascii "can\000"
...
.text
.align 2
.globl main
.align 2
.globl sort_function__FPCvT0 # adresa funcției de sortare
.text
.ent main
main:
subu $sp,$sp,32           # alocare spațiu pe stivă
sw $31,28($sp)           # salvare # adresa de revenire în programul apelant - $31
                          # (sistemul de operare)
sw $fp,24($sp)
move $fp,$sp
jal __main
la $4,list               # pregătire parametrii de apel $4 – adresa tabloului de
                          # sortat
li $5,0x00000005         # $5 <- 5 = numărul de elemente al tabloului
```

```

li      $6,0x00000004      # $6 <-4 = dimensiunea în octeți a fiecărui element de
                                # tablou
la      $7,sort_function__FPCvT0 # $7 <- adresa funcției folosită de către qsort în apelul său
jal     qsort              # la apelul unei funcții primii 4 parametri de apel sunt
                                # salvați în regiștrii $4 ÷ $7 restul, dacă e cazul sunt depuși
                                # pe stivă [Flor03]. Funcția qsort este precompilată.

sw      $0,16($fp)
...
$L36:
move   $sp,$fp      # sp not trusted here
lw     $31,28($sp)
lw     $fp,24($sp)
addu   $sp,$sp,32
j      $31
.end   main
.ent   sort_function__FPCvT0
sort_function__FPCvT0:
subu   $sp,$sp,24
sw     $31,20($sp)
sw     $fp,16($sp)
move   $fp,$sp
sw     $4,24($fp)      # salvare pe stivă a parametrilor de apel în cadrul rutinei
                                # apelate indirect prin qsort – adresa și
                                # numărul de elemente al tabloului

sw     $5,28($fp)
lw     $4,24($fp)
lw     $5,28($fp)
jal    strcmp          # apel funcție de bibliotecă pentru compararea a două
                                # șiruri de caractere

move   $3,$2
move   $2,$3
...
$L41:
move   $sp,$fp
lw     $31,20($sp)
lw     $fp,16($sp)
addu   $sp,$sp,24
j      $31
.end   sort_function__FPCvT0

```

În urma analizei programelor de test (rezultate după compilare [.s]) se desprinde o concluzie generală în ceea ce privește salturile indirecte: *Întrucât instrucțiunile de apel de subrutină salvează automat în registrul \$31 (\$ra) adresa de revenire (PC_curent+4) [Flor03], la apelul recursiv întâlnit în cadrul diverselor apeluri indirecte de funcții (vezi back_.cpp) adresa de revenire trebuie salvată explicit în stivă de către programator, înaintea fiecărui astfel de apel (practic compilatorul realizează acest lucru).*

Deși au fost analizate doar câteva programe simple de test, cu număr redus (<5.000.000) de instrucțiuni dinamice, pe baza rezultatelor obținute se desprind câteva concluzii clare.

- Instrucțiunile de salt indirect apar mult mai frecvent în programele obiectuale decât în cele procedurale.

- Prezența salturilor indirecte în **programele procedurale** se datorează în *principal* următoarelor două aspecte:
 - **Apelurilor indirecte de funcții prin pointeri.**
 - **Construcțiilor de tip switch/case.**
- Unul dintre *motivele secundare* care favorizează prezența salturilor indirecte în programe îl reprezintă **prezența funcțiilor de bibliotecă** (vezi cazul *qsort* precum și alte librării legate dinamic din cadrul aplicațiilor desktop – *DLL* [Lee98]).
- **Legarea dinamică (*late binding*) realizată prin polimorfism** (care se bazează la rândul său pe conceptul de **moștenire**) generează apeluri indirecte de funcții în cazul **programelor obiectuale** (C++, Java, Smalltalk).

4.3. INSTRUCȚIUNI DE SALT INDIRECT CU CARACTER DINAMIC POLIMORF

Subcapitolul de față ilustrează pe 5 din benchmark-urile SPEC'95 comportamentul dinamic al salturilor indirecte, dispersia target-urilor acestora și câteva cazuri în care deși clasificarea bazată pe aritate (numărul target-urilor distincte generate de fiecare instanță de salt indirect) a indicat salturi duomorfe sau polimorfe, ar fi fost mai utilă o clasificare a branch-urilor după numărul de situații în care acestea suferă o modificare a target-ului datorită saltului care se face preponderent la doar una din adresele sale destinație. Pentru acest experiment simulările au fost efectuate doar pentru 5.000.000 instrucțiuni dinamice executate aferente fiecărui benchmark, concluziile fiind suficient de elocvente. În continuare sunt ilustrate adresele instrucțiunilor de salt indirect, target-urile posibile aferente respectivelor salturi, tipul saltului (salt – jr \$reg, sau apel – jalr \$reg) precum și frecvența cu care este accesat fiecare target.

Salturi cu caracter duo sau polimorf dar cu execuție preponderentă doar la una din adresele destinație

Benchmark-ul **hydro2d.ss**

PC: **0x425cb0 jr \$2**

Target1 = 0x426950 freq = 2

Target2 = 0x425cb8 freq = 782

PC: **0x41e858 jr \$2**

Target1 = 0x41ecf8 freq = 1565

Target2 = 0x41eb38 freq = 6

PC: **0x4108e0 jr \$2** Target1 = 0x4108e8 freq = 1565
Target2 = 0x410f50 freq = 6

Benchmark-ul **cc1.ss**

PC: **0x420598 jr \$2** Target1 = 0x420750 freq = 1
Target2 = 0x4205a0 freq = 306

PC: **0x5071d0 jr \$2** Target1 = 0x507220 freq = 2
Target2 = 0x507250 freq = 71

PC: **0x462750 jr \$2** Target1 = 0x464f18 freq = 1
Target2 = 0x463340 freq = 306
Target3 = 0x462770 freq = 4

Salturi cu dispersie ridicată a target-urilor

Benchmark-ul **cc1.ss**

PC: **0x400a68 jr \$2**

Target1 = 0x402b70 freq = 7	Target23 = 0x401fa8 freq = 14
Target2 = 0x401228 freq = 1	Target24 = 0x402a98 freq = 146
Target3 = 0x400fd0 freq = 15	Target25 = 0x402ae0 freq = 146
Target4 = 0x402ca8 freq = 15	Target26 = 0x402b88 freq = 122
Target5 = 0x402950 freq = 35	Target27 = 0x403188 freq = 153
Target6 = 0x402a28 freq = 1	Target28 = 0x403178 freq = 153
Target7 = 0x402c88 freq = 13	Target29 = 0x401f50 freq = 161
Target8 = 0x4027f0 freq = 17	Target30 = 0x402840 freq = 17
Target9 = 0x400de0 freq = 17	Target31 = 0x402818 freq = 13
Target10 = 0x400df8 freq = 17	Target32 = 0x402a08 freq = 10
Target11 = 0x401fd0 freq = 246	Target33 = 0x401cc0 freq = 24
Target12 = 0x402bb8 freq = 31	Target34 = 0x401cf8 freq = 24
Target13 = 0x4028f8 freq = 3	Target35 = 0x4029c0 freq = 8
Target14 = 0x4028d8 freq = 3	Target36 = 0x402c50 freq = 298
Target15 = 0x4027a0 freq = 282	Target37 = 0x402c68 freq = 293
Target16 = 0x4044d8 freq = 282	Target38 = 0x402970 freq = 8
Target17 = 0x404360 freq = 282	Target39 = 0x402100 freq = 365
Target18 = 0x4044a8 freq = 282	Target40 = 0x401bf8 freq = 497
Target19 = 0x4028b8 freq = 41	Target41 = 0x401d20 freq = 351
Target20 = 0x402860 freq = 17	Target42 = 0x4045f8 freq = 4165
Target21 = 0x404328 freq = 20	Target43 = 0x401df8 freq = 351
Target22 = 0x4027d0 freq = 86	Target44 = 0x4041c8 freq = 1915

Benchmark-ul **li.ss**

PC: **0x406bb0 jalr \$2, \$31**

Target1 = 0x40a0b8 freq = 21	Target5 = 0x40a128 freq = 64
Target2 = 0x40a868 freq = 21	Target6 = 0x40cd20 freq = 22
Target3 = 0x401938 freq = 21	Target7 = 0x40c8d0 freq = 44
Target4 = 0x400c98 freq = 22	Target8 = 0x402278 freq = 22

Target9 = 0x40a080 **freq = 86**
 Target10 = 0x40cb60 freq = 68
 Target11 = 0x401e90 freq = 46

Target12 = 0x400480 freq = 67
 Target13 = 0x401350 **freq = 13**
Target14 = 0x401318 freq = 30

Benchmark-ul **hydro2d.ss**

PC: **0x4125b0 jr \$2**

Target1 = 0x412918 freq = 1570
 Target2 = 0x412690 **freq = 3141**

Target3 = 0x4127b8 freq = 1571
Target4 = 0x4127f8 **freq = 1571**

Benchmark-ul **apsi.ss**

PC: **0x43a0b0 jr \$2**

Target1 = 0x43a190 **freq = 1499**
 Target2 = 0x43a2b8 freq = 568
 Target3 = 0x43a418 **freq = 252**

Target4 = 0x43a140 freq = 410
Target5 = 0x43a2f8 freq = 253

Benchmark-ul **go.ss**

PC: **0x47ec68 jr \$2**

Target1 = 0x47ed98 **freq = 760**
 Target2 = 0x47ed50 freq = 760
 Target3 = 0x47ecf8 freq = 760
 Target4 = 0x47ec90 freq = 760

Target5 = 0x47eca8 **freq = 808**
 Target6 = 0x47ece8 freq = 808
 Target7 = 0x47ec80 freq = 808
 Target8 = 0x47ec70 freq = 808

Exceptând benchmark-ul *go.ss* mai există și alte salturi polimorfe (cu 3, 4 și până la 9 target-uri distincte) în fiecare din celelalte programe de test (rezultatele se bazează pe execuția a 5.000.000 instrucțiuni dinamice) însă le-am ales pe cele cu dispersia cea mai mare în fiecare din cazuri. Practic **benchmark-urile cu un număr** (și procentaj) **redus** (chiar foarte mic) **de instrucțiuni dinamice de salt indirect** (*aplu*, *go*) **se caracterizează printr-o dispersie redusă a target-urilor**. Rezultatul este similar cu cel obținut de Chang în simulările sale [Cha97]. Dispersia target-urilor salturilor indirecte tinde să crească odată cu simularea unui mai mare număr de instrucțiuni dinamice. Dificultatea predicției se datorează **dispersiei extrem de ridicate a target-urilor unora dintre aceste instrucțiuni** (simulări pe benchmark-ul *gcc* – SPEC2000 au arătat că există salturi care generează pentru 100.000.000 de instrucțiuni dinamice, **106** target-uri distincte, cu frecvențe de repetiție a acestora între **1** și **12039**). O contribuție proprie care vizează predicția salturilor indirecte, prezentată în subcapitolul 5.3, utilizează informațiile de aritate a salturilor în cadrul unei structuri hibride de predicție.

5. CONTRIBUȚII LA PREDICȚIA SALTURILOR / APELURILOR INDIRECTE

5.1. PREDICȚIA DINAMICĂ A RAMIFICAȚIILOR DE PROGRAM. NECESITATE. SOLUȚII

Rezultate statistice bazate pe simulări laborioase pe benchmark-uri reprezentative (SPEC'95, 2000) arată că o instrucțiune de salt apare la fiecare 5÷8 instrucțiuni dinamice executate, ceea ce înseamnă că rata de aducere a instrucțiunilor (*fetch rate* – FR) este limitată la cel mult 8, aducerea simultană a mai multor instrucțiuni fiind inutilă (*fetch bottleneck*). Această limitare fundamentală ar avea consecințe defavorabile și asupra ratei medii de execuție (*issue rate* – IR) a instrucțiunilor întrucât $IR \leq FR$. Pentru creșterea gradului de paralelism la nivelul instrucțiunilor este necesară dezvoltarea și implementarea de noi tehnici care să reducă întârzierile în procesare (*hazarduri*) pe oricare din cele două fluxuri: de instrucțiuni (*control-flow*) și respectiv de date (*data-flow*). Instrucțiunile de ramificație (*branch*) acționează la nivelul control-flow generând pierderi de performanță prin necunoașterea la timp (în momentul fazei de aducere a instrucțiunilor) a direcției și adresei saltului. Reducerea efectelor defavorabile se poate face prin metode software bazate pe reorganizarea programului sursă (*scheduling*) sau prin metode hardware (predicția ramificațiilor de program care favorizează astfel execuția speculativă). Există scheme de predicție numite *statice* caracterizate prin faptul că predicția saltului pentru o anumită istorie a sa este fixă, bazată în general pe studii statistice ale comportamentului salturilor și uneori pe diverse euristici [Cal95]. Compilatoarele se bazează pe predicția statică și estimări ale execuției codului în implementarea de optimizări gen *trace-scheduling*.

Predicția statică a branch-urilor determină calea cea mai frecvent urmată prin graful de control al unui program, prezicând direcția care se va alege în cazul instrucțiunilor de salt. Îmbunătățirea predicției statice a branch-urilor constituie o problemă importantă cu numeroase implicații, variind de la optimizări ale compilatoarelor, până la creșterea

performanțelor sistemelor dedicate (*embedded systems*) care nu înglobează un predictor dinamic de salturi din motive de cost hardware. De asemenea, se urmărește îmbunătățirea predicției dinamice prin transmiterea de informații de la nivel *high* (software) la nivel *low* (hardware). Astfel de informații pot fi: se face / nu se face saltul (T/NT), direcție target (forward / backward), opcode salt, etc.

Predicția prin hardware numită dinamică, reprezintă una din cele mai performante strategii actuale de gestionare a ramificațiilor de program și presupune verificarea condiției de salt (*se face* sau *nu se face*) iar dacă saltul se face determinarea adresei instrucțiunii destinație. Procesul de predicție a ramurii de salt condiționat precum și determinarea în avans a noului PC se realizează „*run-time*” utilizând tabele hardware mai mult sau mai puțin complexe (vezi în continuare subcapitolele 5.1.1 și 5.1.2). Cercetările recente insistă pe această problemă întrucât s-ar elimina necesitatea reorganizărilor soft ale programului sursă obținându-se astfel o independență față de mașină. În principiu, o schemă dinamică de predicție este superioară uneia statice datorită adaptabilității sale.

În condițiile creșterii cantității de cod executat speculativ, predicția cu acuratețe a instrucțiunilor de ramificație (salturi) în general, și a celor indirecte în special, constituie un factor esențial în îmbunătățirea performanței globale a arhitecturilor de calcul. Complexitatea arhitecturală a procesoarelor actuale (structuri pipeline cu foarte multe niveluri – 20 la procesorul INTEL Pentium4 și „ferestre largi” de instrucțiuni care pot fi lansate simultan spre execuție) dar și complexitatea tehnologică (perioade de tact extrem de scăzute – 500ps la același procesor [Spra02]) agravează impactul negativ asupra performanței cauzat de o predicție greșită. Întrucât structurile pipeline moderne rețin până la sute de instrucțiuni în execuție nefinalizate (*in flight*), dar mai ales faptul că, ramura de program executată speculativ trebuie părăsită în cazul unei predicții eronate a destinației unei instrucțiuni de salt (cu consecințe defavorabile din punct de vedere al timpului de execuție), arhitecturile moderne de procesare trebuie să înglobeze scheme de predicție cât mai eficiente. Simulări efectuate pe o suită amplă de benchmark-uri (SPEC'95, '2k, aplicații desktop, multimedia și Internet, recunoașterea vorbirii, CAD etc.) având la bază o arhitectură superscalară echivalentă procesorului Intel Pentium 4 au demonstrat diminuarea ratei globale de procesare (IPC – instruction per cycle) cu **0.45%** pentru fiecare ciclu de tact suplimentar necesar în cazul unei predicții greșite a salturilor [Spra02], iar o înrăutățire a acurateții predicției salturilor de la 4% la 7% determină descreșterea ratei globale de procesare cu 11% [Jim02].

Deși cele mai mari acurateți de predicție a salturilor (pe direcție) raportate în literatură se situează între 95% [McFar93] și 97% [Yeh92], costul ridicat ca și timp necesar refacerii structurii pipeline și a contextului procesorului la starea dinaintea predicției greșite reprezintă unul dintre cele mai mari impedimente în calea creșterii performanței procesoarelor moderne [Ega03]. La procesorul Alpha 21264 acest timp este de 7 cicli de tact procesor, la Intel Pentium II este de 10 cicli iar la Pentium 4 procesul de *recovery* se realizează în 20 cicli de tact.

Un alt aspect de care trebuie ținut cont de către arhitecții de calculatoare se referă la costul hardware impus de o acuratețe ridicată de predicție. De exemplu, la campionatul mondial de branch prediction din 2004 bugetele hardware destinate implementării predictoarelor a fost limitat la 64Kbiți+256biți [JILP04]. Studiile cercetătorilor [Ega03] accentuează faptul că generațiile viitoare de predictoare ar putea consuma un buget hardware foarte ridicat (de ordinul megaocteților). Ideal ar fi ca predictoarele viitoare să prezică cu acuratețe cât mai apropiată de 100% dar cu costuri hardware nu foarte mari. Pentru predictoarele de capacitate redusă ($\leq 16\text{Ko}$ [Tho03]) a căror acuratețe este în principal limitată de interferențe destructive, o mică creștere în dimensiune determină o îmbunătățire substanțială a acurateții. Predictoarele globale de salturi îmbunătățesc acuratețea predicției prin corelarea comportamentului saltului curent (T / NT) cu istoria celor mai recente salturi dinamice precedente acestuia, o creștere liniară a lungimii istoriei cauzând o creștere exponențială a capacității predictorului. Din fericire, chiar și aceste structuri „*imense*” de predicție pot fi implementate în Siliciu, bugetul de tranzistoare (între 55 milioane – la *Intel Pentium 4 Northwood* [Intel02] și până la 178 milioane – la *Intel Pentium 4 Extreme Edition* [Intel03]) existent la ora actuală permițând acest lucru, singurul neajuns constituindu-l întârzierea predicției.

Dacă până nu demult, un număr însemnat de cercetări a fost canalizat asupra a două dimensiuni aferente predicției – acuratețe și respectiv consum hardware al schemelor implementate, odată cu creșterea în complexitate a procesoarelor actuale (structuri pipeline foarte adânci) și necesitatea predicției într-un singur ciclu de tact CPU, se pune problema luării în calcul a celui de-al treilea aspect al predicției – latența [Jim02]. Îmbunătățirile aferente procesului tehnologic de fabricare, realizate prin creșterea capacității de integrare a tranzistoarelor și reducerea timpilor de comutație, determină creșterea frecvenței procesoarelor actuale și viitoare. Îngustarea continuă a perioadei de tact a acestora conduce la imposibilitatea accesării într-o singură perioadă de tact a structurilor de predicție dinamice de foarte mare capacitate cu efect imediat și asupra vitezei globale de execuție a programelor măsurat în IPC (instrucțiuni per ciclu). Evident predictoarele de

capacitate redusă determină diminuarea acurateții de predicție și o penalitate suplimentară rezultând de asemenea, reducerea vitezei globale de execuție. Practic trebuie realizat un compromis între dimensiunea tabelelor de predicție și frecvența de procesare. O tabelă de predicție de capacitate mai mare poate extrage mai bine informația de corelație rezultând o acuratețe de predicție ridicată dar un timp de acces la structură mai mare ($> 1Tact_{CPU}$). O frecvență de procesare mai modestă permite un timp mai mare pentru efectuarea unei predicții care se poate realiza astfel într-o singură perioadă de $Tact_{CPU}$. În [Jim02] se arată că, din punct de vedere al costului de execuție al unei instrucțiuni de salt, acceptarea unor predicții realizabile într-un număr mai mare de cicli de tact folosind structuri de predicție de capacități foarte mari, în vederea creșterii acurateții, nu reprezintă niciodată un bun compromis. Ecuația următoare, preluată din [Jim02] descrie în mod grosier costul de execuție al unei instrucțiuni de salt:

$$C = d + r \times p, \text{ unde}$$

d – reprezintă timpul de acces la structura de predicție

$r = 1 - \text{Acuratețea predicției}$, și reprezintă procentajul de predicții greșite

p – penalitatea în cazul unei predicții greșite.

Întrucât r este în mod clar subunitar rezultă influența mult mai pronunțată a parametrului d decât cea a lui r asupra costului C .

În sprijinul celor afirmate mai sus se exemplifică cu două implementări comerciale, existente pe piața mondială, referitoare la procesoarele firmei AMD. Astfel, predictorul de salturi încorporat în procesorul AMD Athlon reprezintă un pas înapoi comparativ cu predecesorul său încorporat în AMD K6. În timp ce K6 deținea un predictor performant GAs (vezi terminologia din subcapitolul 5.1.1) de 8K intrări, Athlon folosește un predictor mai puțin eficient, de același tip dar cu doar 2K intrări [Die98]. Prin această modificare se reduce timpul mediu de efectuare a unei predicții favorizând atingerea unei frecvențe de ceas de 1.4 GHz, mai agresive decât cea aferentă procesorului K6. Fără îndoială că, viteza globală de execuție a programelor (măsurată în IPC) pe un Athlon a scăzut datorită predictorului de salturi mai puțin performant decât al predecesorului său.

Astfel, toate ideile relevate anterior conduc la o motivație clară: necesitatea găsirii și implementării unor predictoare cu acurateți extrem de ridicate și un timp de acces cât mai redus. Se pare că, *structurile viitoare de predicție vor fi compuse din predictoare multi-nivel (hibride sau cascade) cu latențe și acurateți de predicție progresiv crescătoare (rezultate în urma unui mecanism selectiv de corecție)* [Tho03, Dri98b, Dri99, Jim02].

În continuare în subcapitolul 5.1.1. sunt ilustrate pe scurt cele mai însemnate structuri clasice de predicție cu performanțele și limitările lor, urmând ca în subcapitolul 5.1.2. să fie prezentate câteva soluții de depășire a celor mai bune predictoare clasice. Predictoarele markoviene și cele bazate pe rețele neuronale par să reprezinte soluția viabilă cea mai avantajoasă care să fie implementată în Siliciu. În ciuda unor dezavantaje legate de timpul de realizare a predicției, rețelele neurale au fost sugerate ca o tehnologie promițătoare pentru procesoarele viitoare [Sez02]. Spre exemplu, unul din simulatoarele procesorului Intel IA'64, utilizate în exploatarea, cercetarea și dezvoltarea microarhitecturilor, are în componență un predictor neural [Bre02].

5.1.1. STRUCTURI CLASICE DE PREDICȚIE A SALTURILOR CONDIȚIONATE.

Structura de predicție *branch target cache (BTC)* [Hen03] sau *branch target buffer (BTB)* în alte documentații [Per93], reține următoarele informații privitoare la salturile executate anterior: adresa instrucțiunii (PC-ul saltului), adresa destinație (target-ul saltului) și informații privind istoria acestuia (automat finit de stare pe 1 sau 2 biți care în funcție de starea sa curentă prezice dacă saltul se va face sau nu). Automatele de predicție astfel încorporate, conferă predictorului *dinamism*, comportarea lor adaptându-se în funcție de istoria saltului respectiv. Branch-urile sunt predicționate, folosind PC-ul pentru indexarea structurii BTB, în paralel cu procesul de aducere a instrucțiunii din cache. Dacă din tabela BTB rezultă branch-ul de prezis ca fiind taken, și întrucât target-ul acestuia este și el disponibil la finele ciclului de fetch al saltului, rezultă că următoarea instrucțiune va fi extrasă din cache de la adresa specificată de respectivul target. Bazat pe simulări laborioase [Per93], se arată că acuratețea de predicție obținută folosind structura BTB este de 88% folosind un singur bit de predicție, respectiv de 93% folosind 16 biți de predicție. Totuși, chiar și aceste acurateți ale predicției sunt insuficiente pentru marea performanță a procesoarelor superscalare, unde fiecare procent de acuratețe suplimentar are un impact pozitiv deosebit asupra performanței globale (vezi 5.1). Dintre implementările comerciale microprocesoarele DEC Alpha 21064 și respectiv AMD K5 au avut încorporat un predictor de ramificații bazat pe un BTB cu un singur bit de predicție, iar microprocesoarele PowerPC 604 și MIPS R10000 predictoare BTB cu doi biți de predicție.

Schema de predicție anterior prezentată se baza pe comportarea recentă a unei instrucțiuni de salt, de aici predicționându-se comportarea viitoare a acelei instrucțiuni de salt. Este posibilă îmbunătățirea acurateții predicției dacă aceasta se va baza pe comportarea recentă a altor instrucțiuni de salt, întrucât frecvent aceste instrucțiuni pot avea o comportare corelată în cadrul programului. Începând cu 1991 au fost dezvoltate structuri de predicție adaptive, corelate pe două niveluri, cu acurateți de predicție superioare celor obținute folosind structuri de tip BTB. *Schemele de predicție corelate* au fost introduse pentru prima dată în 1992 în mod independent de către Yeh și Patt și respectiv de Pan și Rahmeh [Hen03, Yeh92, Pan92]. Există în implementare două niveluri. Primul îl reprezintă un *registru de predicție* (registru binar de deplasare pe m ranguri) care reține *istoria* (s-a făcut / nu s-a făcut) a celor mai recent executate m salturi din program. Această informație poate fi sau nu, concatenată cu cei mai puțin semnificativi biți ai PC-ului instrucțiunii de salt pentru a pointa la un cuvânt din *tabela de predicție* (practic al doilea nivel din structură). Acest cuvânt conține automatul de predicție, target-ul saltului etc). Succesul noilor structuri se datorează în primul rând capacităților ridicate ale acestora (tablouri de dimensiuni mari de automate de predicție sau de pattern-uri de istorie a fiecărui salt – *Prediction History Table* – PHT). Întrucât capacitatea PHT crește exponențial în funcție de lungimea istoriei, costul PHT devine excesiv, fiind dificil de exploatat eficient o cantitate mare de istorie a branch-urilor. Predictoarele adaptive pe două niveluri prezintă încă alte două dezavantaje. Primul se referă la faptul că, în majoritatea implementărilor fiecare automat de predicție este partajat între mai multe salturi. Interferențele astfel generate conduc la scăderea acurateții de predicție datorită miss-urilor de conflict. Pe de altă parte, structurile foarte mari de predicție reduc din aceste interferențe însă necesită un timp suplimentar de antrenare (*setup*) al automatelor înainte de a prezice cu succes.

În literatură au fost dezvoltate două tehnici de predicție: *globală* și respectiv *locală* (per adresă). Dacă registrul de predicție (istorie) reține comportamentul ultimelor k (HR) salturi condiționate anterioare saltului curent atunci această istorie precum și tehnica de predicție este una globală. Dacă registrul de istorie reține comportamentul saltului curent în ultimele sale k (HR) instanțe se spune atât despre istorie cât și despre tehnica de predicție că este locală. Yeh și Patt [Yeh92] au propus un sistem de clasificare al predictoarelor adaptive corelate pe două niveluri. Sunt cunoscute 6 tipuri de astfel de predictoare: GAg, GAp, GAs, PAg, PAp și PAs. Expresiile care definesc fiecare tip de predictor nu sunt întâmplător alese și au la bază următoarele considerații:

- Prima literă scrisă cu majusculă specifică tipul mecanismului de pe primul nivel: „G” – global sau „P” – per adresă (local).
- Cea de-a treia literă identifică tipul istoriei de pe cel de-al doilea nivel. Aceasta poate fi: „g” – globală, „p” – per adresă (locală) sau „s” – per set (mai multe branch-uri partajează aceeași istorie fiind mapate în aceeași locație a tabelului de predicție PHT).
- Litera „A” din mijlocul fiecărei expresii evidențiază caracterul adaptiv (dinamicitatea) predictorului.

Prima implementare comercială a unei scheme de predicție corelată pe două niveluri a fost predictorul **GAg** (vezi figura 5.1) în cadrul microprocesorului Intel Pentium Pro. Au urmat apoi implementări ale schemei GAg în cadrul microprocesoarelor Intel Pentium II și respectiv ale schemei GAs în AMD K6.

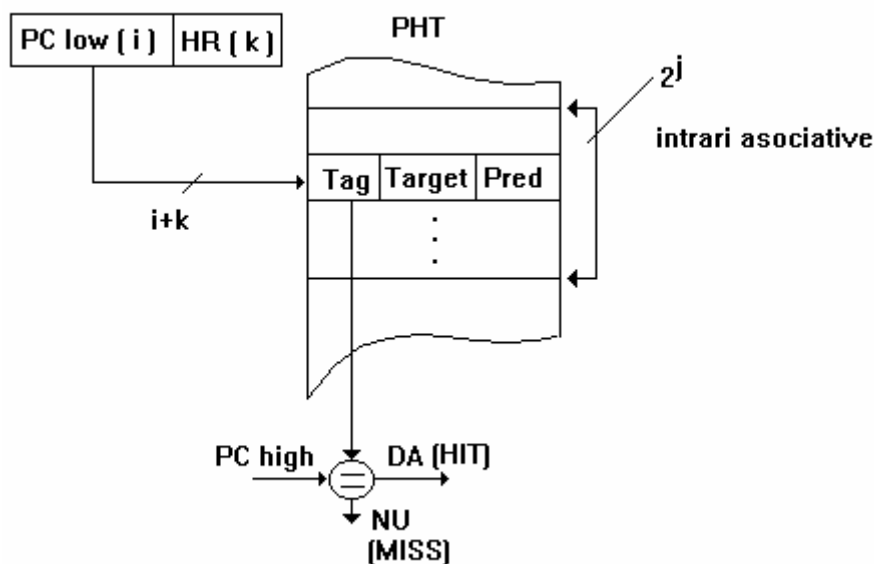


Figura 5.1. Structură de predicție de tip GAg

Privitor la schema GAg, tabela de predicții PHT (Prediction History Table) este adresată cu un index rezultat din concatenarea a două informații ortogonale: PClow (i biți), semnificând gradul de localizare al saltului, respectiv registrul de predicție (HR - History Register pe k biți), semnificând "contextul" în care se situează saltul în program. Ambele contribuții s-au făcut cu scopul eliminării interferențelor branch-urilor în tabela de predicție. Adresarea PHT exclusiv cu HR ca în articolul [Yeh92], ducea la serioase interferențe (mai multe salturi puteau accesa același

automat de predicție din PHT), cu influențe evident defavorabile asupra performanțelor. Desigur, PHT poate avea diferite grade de asociativitate. Un cuvânt din această tabelă are un format similar cu cel al cuvântului dintr-un BTB. O soluție alternativă la concatenarea HR și PClow în adresarea PHT, cu rezultate foarte bune, o reprezintă dispersia printr-o funcție de tip SAU EXCLUSIV a celor două informații, cu care se adresează apoi tabela PHT. Această compresie are o influență benefică asupra capacității tabelului PHT.

În scopul reducerii interferențelor diverselor salturi în tabela de predicții, în [Yeh92] se prezintă o schemă numită **PAG**- Per Address History Table, Global PHT, a cărei structură este oarecum asemănătoare cu cea a schemei GAg. Componenta $HR^*(k)$ a fost introdusă în [Vin00], având semnificația componentei HR de la varianta GAg, adică un registru global care memorează comportarea ultimilor k salturi. Fără această componentă, schema PAG și-ar pierde din capacitatea de adaptare la contextul programului în sensul în care schema GAg o face. *Yeh* și *Patt* renunță la informația de corelație globală (HRg) în trecerea de la schemele de tip GAg la cele de tip PAG, în favoarea exclusivă a informației de corelație locală (HrI). În schimb, componenta HrI din structura History Table, conține "istoria locală" (taken / not taken) a saltului curent, ce trebuie predicționat. Performanța schemei PAG este superioară celei obținute printr-o schemă de tip GAg, cu tributul de rigoare plătit complexității hardware. Schema de predicție de tip PAG predicționează pe baza a **3 informații ortogonale**, toate disponibile pe chiar timpul fazei de aducere a instrucțiunilor (IF): istoria HRg a anterioarelor salturi corelate (taken / not taken), istoria saltului curent HrI și PC-ul acestui salt.

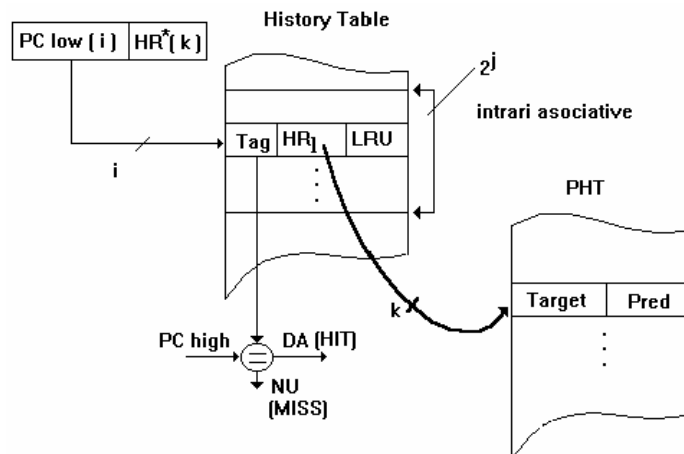


Figura 5.2. Structură de predicție de tip PAG

După cum se observă din figura 5.2, deși tabela de pe primul nivel (History Table) este specifică unui salt (PC concatenat cu HR_g), tabela de pe nivelul al doilea (PHT) este globală, fiind adresată doar cu istoria locală a unui salt (HRI), fapt ce conduce la interferențe între instrucțiunile de salt. Dacă adresarea tabelii PHT s-ar face în schema PAg cu HR concatenat cu PClow(i), atunci practic fiecare branch ar avea propria sa tabelă PHT, rezultând deci o schemă și mai complexă numită **PAp** (Per Address History Table, Per Address PHT) [Yeh92, Vin00]. Complexitatea acestei scheme o face practic neimplementabilă în siliciu la ora actuală, fiind doar un model utilizat în cercetare.

Desigur, este posibil ca o parte dintre branch-urile memorate în registrul HR, să nu se afle în corelație cu branch-ul curent, ceea ce implică o serie de dezavantaje. În astfel de cazuri pattern-urile din HR pot pointa în mod inutil la intrări diferite în tabela de predicție, fără beneficii asupra performanței predicției, separându-se astfel situații care nu trebuie separate. Mai mult, aceste situații pot conduce la un timp de "umplere" a structurilor de predicție mai îndelungat, cu implicații defavorabile asupra performanței [Eve96]. O posibilă soluție și în acest caz poate consta în păstrarea și utilizarea unei istorii cât mai „lungi” (îndepărtate) în procesul de predicție aferent instrucțiunilor de salt [Tho03]. Argumentul este „de bun simț” întrucât unele salturi corelate pot apărea la o distanță considerabilă în șirul de instrucțiuni dinamice. Acest lucru se poate întâmpla dacă două salturi corelate sunt despărțite (separate) de către un apel de funcție care conține multe branch-uri. La momentul „părăsirii” funcției, o istorie globală redusă poate conține doar comportamentul salturilor din cadrul funcției, în timp ce o istorie globală extinsă poate reține și rezultatul saltului corelat, anterior apelului funcției. De asemenea, o idee interesantă ce poate fi abordată o reprezintă studiul fezabilității unui predictor de salturi, corelat pe bază de arbori de decizie și senzitiv la informația de corelație cu adevărat utilă [Fern03]. Asupra acestei relativ recente idei se va reveni în subcapitolul 5.2.2.5.

Există însă salturi care rămân greu predictibile în ciuda păstrării istoriei locale sau globale a comportamentului lor anterior. Pot exista două cazuri. Unele salturi a căror condiție este dependentă de anumite date aleatoare vor fi întotdeauna greu de prezis. În schimb există posibilitatea ca, identificând **noi** mecanisme de corelație care să fie adăugate procesului de predicție, să favorizeze predicția cu acuratețe a unora dintre salturile greu de prezis. În [Ega03] se propune exploatarea acestei informații de corelație potențială prin intermediul unor predictoare neuronale.

O critică valabilă pentru toate schemele corelate constă în faptul că informația de corelație globală este insuficientă în predicție. Registrul de

predicție (HRg) nu poate capta întreg contextul de apariție al unui anumit salt. Simulări laborioase de tip trace - driven pe benchmark-urile Stanford [Vin99], arată că există salturi care în același context (HRg, HRl) au comportări antagoniste, adică de exemplu în 56% din cazuri s-au făcut, iar în 44% din cazuri nu s-au făcut. Prin urmare aceste salturi sunt practic nepredictibile, din motivul că "*același context*" nu este în realitate același! O soluție ar consta în asocierea fiecărui bit din HRg cu PC-ul aferent saltului respectiv și accesarea predicției pe baza acestei informații mai complexe. Astfel, ar exista siguranța că la contexte diferite de apariție a unui anumit salt, se vor apela automate diferite de predicție, asociate corect contextelor, reducându-se din efectele fenomenului de interferență a predicțiilor. Compararea acestor noi scheme de predicție trebuie făcută cu scheme clasice având aceeași complexitate structurală. Comprimarea acestui complex de informație (HRg cu PC-urile aferente) este posibilă și chiar necesară, având în vedere necesitatea unor costuri rezonabile ale acestor scheme. Ea se poate realiza prin utilizarea unor funcții de dispersie simple (de exemplu tip SAU- EXCLUSIV). Această observație simplă poate conduce la îmbunătățiri substanțiale ale acurateții de predicție, comparativ cu oricare dintre schemele actuale. În cadrul acestei lucrări am aplicat tehnica mai sus menționată la predicția salturilor indirecte (vezi subcapitolul 5.3.2).

Având în vedere complexitatea tot mai mare a acestor predictoare, cu implicații defavorabile asupra timpului de căutare în structurile aferente, se vehiculează tot mai des ideea unor predictoare hibride, constând în mai multe predictoare relativ simple asociate diferitelor tipuri de salturi în mod optimal. Aceste predictoare se activează în mod dinamic, funcție de tipul saltului care este în curs de predicționat. Subcapitolele 5.3.3.1. și 5.3.3.2. descriu implementarea a două predictoare hibride utilizate în predicția salturilor indirecte: unul cu selecție bazată pe *aritate* și altul cu selecție bazată pe *confidență*. Conform literaturii [Jim02], din punct de vedere al implementărilor comerciale existente, cel mai performant predictor hibrid este înglobat în procesorul Alpha 21264 și este în genul celui propus de McFarling [McFar93] (pentru detalii privind acest tip de predictor vezi 5.2.2.3).

5.1.2. DEPĂȘIREA PERFORMANȚELOR PREDICTOARELOR CLASICE PRINTR-O ABORDARE INTEGRATOARE *HARDWARE-SOFTWARE*

5.1.2.1. PREDICTOARE MARKOVIENE.

Se poate afirma că predicția salturilor reprezintă un caz particular al problemei predicției în general, care apare în diverse domenii ale științei și necesită folosirea unor instrumente matematice foarte puternice precum procesele Markov sau seriile de timp. Este surprinzător faptul că puțini cercetători înțeleg necesitatea unei abordări integratoare *hardware* (tehnologie respectiv arhitectură) – *software* (concepte, algoritmi și metode) cu scopul obținerii unor arhitecturi de procesare cât mai evolute, revoluționare chiar (viteză de procesare ridicată, predictive aproape perfecte etc). Printre cei care au recurs la îmbinarea ideilor din diferite domenii ale științei calculatoarelor a fost Trevor Mudge de la Universitatea din Michigan, care în 1996 a introdus conceptul de „*predictor pe bază de lanțuri Markov*”, utilizat în predicția salturilor. În [Chen96] s-a demonstrat că toate predictivele adaptive pe două niveluri implementează cazuri particulare ale algoritmului universal de compresie prin potrivire parțială (*Prediction by Partial Matching* – PPM), care s-a dovedit optim în compresii și extrageri anticipate de date și în recunoașterea vorbirii. De asemenea, PPM reprezintă un instrument de diagnoză în măsurarea limitei “inerente” de predictibilitate din programele de test.

Algoritmul PPM de ordinul m (predictor PPM complet – vezi figura 5.3) este alcătuit din $(m+1)$ predictive Markov. Un predictor Markov de ordinul j predicționează bit-ul următor pe baza pattern-urilor precedente de j biți (în concordanță cu cel mai frecvent bit care urmează pattern-ului). De câte ori apare un miss (nu regăsim pattern-ul de j biți în șirul de intrare), PPM reduce pattern-ul cu 1 bit și folosește predictorul Markov de ordin imediat inferior pentru predicție. Procedura continuă până când se identifică un pattern și se poate realiza predicția corespondentă.

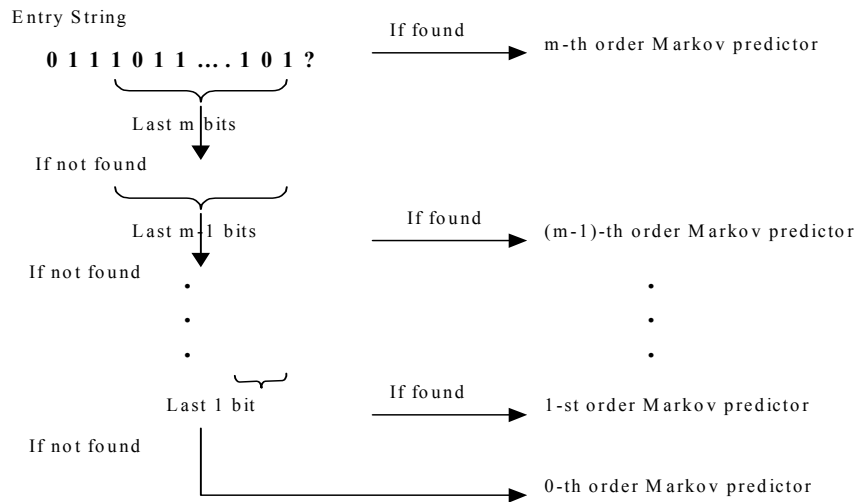


Figura 5.3. Predictor PPM de ordin m

În esență pentru predicție, se caută pattern-ul memorat în registrul HRg pe k biți într-un șir binar mai lung al istoriei salturilor anterioare. Dacă acest pattern este găsit în șirul respectiv cel puțin o dată, predicția se face corespunzător, pe baza unei statistici care determină de câte ori a fost urmat de 1 logic (taken) respectiv 0 logic (not taken) – *valoarea predicționată este aceea care a urmat cu cea mai mare frecvență contextului considerat*. Dacă însă pattern-ul din HRg nu a fost găsit în șirul de istorie, se construiește un nou pattern mai scurt prin eliminarea ultimului bit din HRg și algoritmul se reia pe căutarea acestui nou pattern, ș.a.m.d. până la găsirea unui anumit pattern în șir. Se observă că predicția este funcție și de contextul considerat, un context mai "bogat" ("lung") conducând adeseori la o acuratețe mai ridicată a predicției (nu întotdeauna însă: câteodată contextul se poate comporta ca "zgomot"). Se arată că deși complexitatea implementării acestei noi scheme crește de cca. 2 ori față de o schemă corelată, eficiența sa - la același buget al implementării - este clar superioară. La nivelul tehnologiei actuale, implementarea unui asemenea predictor PPM complet ar putea fi prohibitivă.

Există similitudini puternice între predictoarele adaptive pe 2 niveluri și predictoarele Markov [Chen96] (primul nivel este identic - ambele scheme de predicție utilizează comportarea ultimelor m salturi (T/NT) pentru căutarea în structurile de date corespondente, iar număratoarele saturate de pe al doilea nivel sunt approximate cu numărătorul de frecvențe Markov (frecvența de apariție a valorii 1 și 0). Cele două scheme de

predicție diferă doar prin informația folosită pentru actualizarea celor două numărătoare.

Un predictor PPM complet de ordinul m cuprinde $(m+1)$ predictoare Markov de ordinul $m, m-1, \dots, 0$. Datorită asemănărilor enunțate anterior, predictorul PPM poate fi văzut ca o mulțime de predictoare adaptive pe două niveluri, având nu doar un registru de istorie a salturilor pe m biți ci un set de regiștri cu dimensiunile de la m la 0. Rezultă în acest fel, generarea a $(m+1)$ tabele PHT având dimensiuni între $2^m \times k$ biți (unde k reprezintă numărul de biți pentru codificarea contorului de frecvențe Markov) și $2^0 \times k$ biți. Spațiul total de memorie necesar este, în acest sens: $2^m \times k + 2^{m-1} \times k + \dots + 2^0 \times k = k \times (2^{m+1} - 1) / (2 - 1) \approx 2^{m+1} \times k = 2 \times (2^m \times k) = 2 \times (\text{spațiul de memorie necesar unui GAg cu istorie globală pe } m \text{ biți})$. În concluzie, costul hardware al unui predictor PPM de ordinul m complet este echivalent cu dublul costului unui predictor adaptiv corelat pe două niveluri utilizând tot m biți de istorie globală, ceea ce îl face dificil de implementat în siliciu.

Încercând să evite complexitatea acestui circuit, în [Vin99a] s-a studiat comportarea unui predictor PPM simplificat, constând într-un predictor Markov de ordin m și respectiv unul de ordin 0, chiar fezabil pentru a fi implementat în hardware. Rezultatele simulărilor de tip trace driven pe benchmark-urile Stanford au arătat că acuratețea predicției obținută cu predictorul PPM este substanțial mai bună față de folosirea unui predictor GAg clasic (în medie $\approx 7\%$).

5.1.2.2. METODE NEURONALE DE PREDICȚIE. PREDICTORUL PERCEPTRON.

Limitările structurilor adaptive pe două niveluri din punct de vedere al acurateții predicției, dar și datorită complexității hardware – ceea ce ar conduce la accesarea structurilor de predicție într-un număr ridicat de perioade de t_{CPU} , plus faptul că creșterea nivelului de istorie determină o creștere exponențială a tabelelor de predicție PHT – au condus la introducerea în procesul de predicție a rețelelor neurale ca și alternativă la automatele de predicție deterministe de tip numărătoare saturate.

Din punct de vedere al scopului, domeniul rețelelor neuronale, se încadrează în domeniul mai mare al recunoașterii formelor, iar acesta în inteligența artificială prin necesitatea învățării, iar din punct de vedere al metodei aplicate, se încadrează în cel al proceselor distribuite paralel (PDP).

Rețelele neuronale încearcă să simuleze structura neurofiziologică a creierului uman. Creierul este alcătuit dintr-o mulțime de neuroni interconectați. Fiecare neuron primește informații de la alți neuroni prin

intermediul dendritelor și transmite la rândul său un semnal prin intermediul axonului. Acest model poate fi implementat pe calculator, considerând legăturile dintre neuroni numere (ponderi) care se modifică în funcție de cât de des este activată legătura.

O rețea neurală este un sistem de procesare a informației format dintr-o mulțime de neuroni (puternic) interconectați. Conceptul de rețele neurale artificiale a fost inspirat de rețelele neurale biologice. Neuronii biologici sunt considerați a fi constituanții structurali ai creierului care deși sunt mult mai înceți decât o poartă logică, implementată în siliciu, realizează inferențe și gândesc mult mai rapid și, în orice caz, mai profund și mai nuanțat decât cel mai bun calculator actual. Creierul compensează operațiile relativ încete (operații luate individual) printr-un număr imens de neuroni interconectați, prin paralelism masiv, reușind să se adapteze mediului înconjurător, să „mânuiască” informații vagi, imprecise și probabilistice, să generalizeze de la situații și exemple cunoscute la situații necunoscute, robustețe, toleranță la erori. Rețele neurale artificiale încearcă să imite o parte a caracteristicilor descrise mai sus ale rețelelor neurale biologice și constă dintr-o mulțime de neuroni artificiali interconectați, de mici procesoare care execută doar operații de bază.

Noțiunea de *perceptron* a fost introdusă de Frank Rosenblatt [Ros62], denumire dată rețelei sale neuronale. În ciuda aplicațiilor variate care puteau fi rezolvate prin intermediul rețelelor neuronale (recunoașterea formelor, clasificare, reconstrucția imaginilor), în 1969, Marvin Minsky și S. Papert au analizat posibilitățile de învățare ale perceptronului și au ajuns la concluzii destul de sceptice, dovedind imposibilitatea rezolvării de către perceptronul cu un singur strat a unor probleme simple, cum ar fi învățarea funcției XOR (funcție care nu este liniar separabilă). Ca urmare, interesul pentru acest domeniu de cercetare a scăzut dramatic. Relansarea interesului oamenilor de știință s-a produs în 1986, odată cu apariția unor modele și tehnici noi de învățare supervizată (*metodele propagării înapoi*).

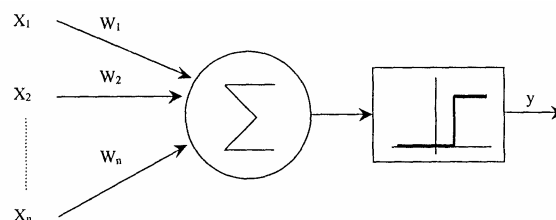


Figura 5.4. Modelul matematic al unui neuron artificial

Modelul matematic al unui neuron artificial propus de McCulloch și Pitts calculează o sumă ponderată de n semnale de intrare și generează o

singură ieșire de 1 dacă această sumă este mai mare decât o anumită valoare (numită *prag*) și 0 dacă este mai mică.

Una dintre cele mai evidente îmbunătățiri care pot fi aduse modelului McCulloch-Pitts, care de altfel reprezintă o simplificare grosolană a neuronului biologic, este folosirea unei *funcții de activare* diferite de funcția treaptă cum ar fi *funcția sigmoidă* sau *funcția Gaussiană*.

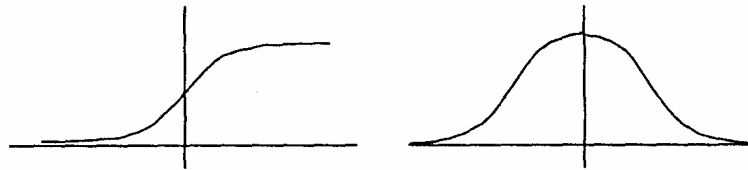


Figura 5.5. Două funcții de activare: sigmoidă și Gaussiană

Funcția sigmoidă este de departe cea mai folosită în rețelele neurale. Este strict crescătoare și asimptotică având expresia matematică $g(x)=1/(1+\exp(-(\beta x)))$. Prin folosirea unei funcții de activare neliniare pot fi rezolvate un număr mai mare de probleme decât cu funcții liniare iar neuronul artificial este adus mai aproape de cel biologic, care este analogic și nu binar.

Caracterizarea unei rețele neurale poate fi făcută după:

- **Arhitectură.** Rețelele neurale artificiale pot fi văzute ca și grafuri orientate în care neuronii artificiali sunt noduri iar muchiile sunt conexiuni între ieșirile neuronilor și intrările acestora. Astfel există două categorii de rețele:
 - ☐ **rețele feed-forward**, în a căror grafuri nu există bucle; (cu reglare înainte; cu reacție pozitivă) unde fluxul de date de la unitățile de intrare la cele de ieșire se desfășoară strict înainte. Procesarea datelor poate fi extinsă pe mai multe straturi dar, fără a fi prezente conexiuni înapoi (feed-back) de la ieșirile unităților la intrările celor din același strat sau din straturile anterioare. Într-una din cele mai cunoscute familii de rețele feed-forward numită *multilayer perceptron*, neuronii sunt organizați pe nivele care au conexiuni unidirecționale între ele. Predictoarele neurale descrise în continuare se bazează pe rețele de tip *feed-forward*. Diferite configurații determină diferite comportamente și capabilități ale rețelei.
 - ☐ **rețele recurente** (feed-back) în care apar bucle datorită conexiunii înapoi. Spre deosebire de rețele feed-forward, aici sunt importante proprietățile dinamice ale rețelei. În unele cazuri, valorile activării unităților trec printr-un proces de relaxare astfel încât, rețeaua va

evolua într-o stare stabilă în care aceste valori nu se vor mai modifica. Exemple de rețele de acest tip sunt *rețelele Hopfield* și *rețelele Kohonen*.

- **Regulile de actualizare a ponderilor (algoritm de învățare).** Abilitatea de a învăța este o caracteristică fundamentală a inteligenței. Nu există o definiție precisă a procesului de învățare însă acest proces poate fi văzut în cazul rețelelor neurale artificiale ca și o problemă de adaptare a ponderilor astfel încât rețeaua să poată rezolva o anumită problemă. Sunt trei mari paradigme de învățare: *supervizată*, *nesupervizată* și *hibridă*. În învățarea supervizată, sau învățarea cu un „profesor”, rețelei i se furnizează un răspuns pentru fiecare tip de intrare. Ponderile se vor modifica astfel încât rețeaua să dea un răspuns cât mai aproape de cel corect. Prin contrast, învățarea nesupervizată, sau învățarea fără profesor, nu necesită un răspuns corect asociat fiecărui tip de intrare din mulțimea de intrări de antrenament. În schimb rețeaua exploatează corelațiile întâlnite în cadrul datelor de intrare și organizează șabloanele în categorii rezultate din aceste corelații. Învățarea hibridă combină învățarea supervizată și cea nesupervizată. O parte dintre ponderi sunt determinate prin învățarea supervizată iar celelalte sunt obținute prin învățarea nesupervizată;
- **Funcția de activare.** Poate fi folosită o plajă întinsă de funcții de activare dar cele mai eficiente în rezolvarea problemelor dificile sunt funcțiile de activare neliniare. Studiul acestor funcții este dificil fapt pentru care nici rețelele care se bazează pe astfel de funcții nu se știe exact cum funcționează, similar cu sistemul neural uman care este necunoscut în proporție de 80%.

Domeniile de aplicabilitate / utilitate al rețelelor neurale:

- ☒ **Clasificarea pattern-urilor.** Sarcina de clasificare a pattern-urilor se referă la a cataloga un pattern reprezentat printr-un vector de trăsături la una din clasele prespecificate (recunoașterea caracterelor, recunoașterea de voce, clasificarea celulelor de sânge, etc.)
- ☒ **Clustering sau gruparea.** În grupare, care se mai numește și clasificare a pattern-urilor nesupervizată nu există date de antrenament și clase cunoscute. Un algoritm de grupare exploatează similaritățile dintre pattern-uri și plasează pattern-uri similare într-o grupare (compresia de date).
- ☒ **Aproximare de funcții.** Fiind dată o mulțime de perechi (x_1, y_1) , (x_2, y_2) , ..., (x_n, y_n) generate de o funcție necunoscută. Sarcina aproximării unei funcții este de a găsi un estimator a acesteia cu care să poate să fi

generate și alte valori cu o anumită aproximație (predicția la bursă, prognoza vremii).

- ☒ **Predicția.** Dându-se o mulțime de n exemple ($y(t_1), y(t_2), \dots, y(t_n)$) în ordinea temporală t_1, t_2, \dots, t_n problema este de a predicționa valoarea de la momentul $t+1$.
- ☒ **Optimizarea.** Scopul optimizării este de a găsi o soluție la o anumită problemă dar care să satisfacă anumite restricții cum ar fi *maximizarea* sau *minimizarea*.
- ☒ **Control.** Având un sistem caracterizat de o mărime de intrare și control $u(t)$ și una de ieșire $y(t)$ scopul este de a genera o intrare de control astfel încât sistemul urmează o anumită traiectorie după un anumit model.
- ☒ **Aplicații potențiale ale rețelelor neurale în microarhitecturi:**
 - ✓ **Predicția dinamică a salturilor** – este realizată cu ajutorul unei rețele neurale care primește ca și intrări în format binar comportamentul celor mai recent executate branch-uri. Ieșirea rețelei o reprezintă predicția (dacă saltul curent se va face sau nu). După cunoașterea rezultatului real al saltului, istoria care a condus la acest rezultat poate fi folosită la antrenarea rețelei pentru a produce un rezultat corect pe viitor.
 - ✓ **Predicția trace-ului următor.** Rețeaua neurală selectează din mai multe trace-uri posibile pe cel care să fie extras din trace-cache.
 - ✓ **Politica de înlocuire din cache** bazată pe rețele neuronale adaptabile la pattern-urile de date accesate de programe determină reducerea ratei de miss.
 - ✓ **Predicția valorilor.** Rețelele neurale pot fi utilizate la selecția cu succes a uneia dintr-un set de valori care au fost generate sau sunt probabil a fi furnizate de o anumită instrucțiune și folosită apoi speculativ.
 - ✓ Rețelele neurale pot ajuta și la **predicția target-urilor salturilor indirecte** generate la nivel low (hardware), în principal, de către apelul metodelor virtuale din programele orientate obiect de la nivel high (software). Detalii privind generarea salturilor indirecte au putut fi obținute din subcapitolul 4.2.

Rețelele neurale funcționează cu randament maxim atunci când clasifică o intrare care provine dintr-un număr restrâns de clase. Predicția neurală folosită în probleme de clasificare este utilizată cu succes [Jim02, Ega03] în predicția salturilor condiționate (selectează direcția) și necesită un singur *perceptron* pentru a clasifica un salt ca fiind *taken* sau *nottaken*. Predicția unei valori sau a unui target de branch este însă mult mai complexă. Ea necesită mai mult de un perceptron pentru fiecare predicție și se bazează în principiu pe existența unei tabele auxiliare de

opțiuni din care rețeaua va face selecția (de exemplu, ultimele k valori sau target-uri anterior generate, k – parametrizabil). Într-o manieră originală, în subcapitolul 6.2.2. am implementat un predictor neural cu rol în metapredicția valorilor regiștrilor procesorului. Rețeaua neuronală implementată selectează între trei sau mai multe predictoare clasice de valori (incremental, contextual, LastValue).

Rețelele neurale au fost folosite pentru prima dată, în procesul de predicție statică a instrucțiunilor de către Calder [Cal95]. Direcția probabilă (*taken / not taken*) aferentă unui salt static este prezisă în momentul compilării, folosind pentru antrenarea rețelei neurale informații de profil cum ar fi: opcode-ul instrucțiunilor de salt, direcția acestora (înainte sau înapoi), caracteristici ale saltului anterior și ale basic-block-urilor succesoare instrucțiunii de salt. Acuratețea predicției obținută este de doar 80%, extrem de ridicată pentru o tehnică de predicție statică dar scăzută în raport cu acuratețea unei predicții dinamice [Vin00c, Ega03]. Probabil contribuția cea mai folositoare a respectivei lucrări este de a sugera o gamă variată de intrări alternative care pot fi utile în mecanismul de corelație și pot fi cu succes adăugate predictoarelor dinamice.

Literatura de specialitate remarcă activitatea independentă și aproape simultană a doi cercetători, cu realizări importante în domeniul predictoarelor neurale. Este vorba de Lucian Vințan de la Universitatea „Lucian Blaga” din Sibiu, care a introdus conceptul de *predictor neural dinamic* și a propus două tipuri de astfel de predictoare: unul de tip *Learning Vector Quantization* – LVQ [Vin99b] și unul de tip *Multi-Layer-Perceptron* – MLP [Ega03] cu algoritmul de învățare de tip *backpropagation*. Al doilea cercetător este Daniel Jimenez de la Universitatea Rutgers SUA, al cărui perceptron s-a dovedit nu numai fezabil la nivel de implementare în Siliciu ci și cel cu acuratețea de predicție cea mai ridicată – 98.29% [Jim02]. Înainte de a prezenta mai în detaliu despre fiecare tip de predictor sunt descrise câteva asemănări și deosebiri în abordările celor doi cercetători privitor la schemele de predicție dezvoltate.

Principala deosebire între cele două studii constă în faptul că Jimenez [Jim02] utilizează doar registrul de istorie – ce reflectă comportamentul salturilor anterioare – ca intrări în predictorul său *perceptron*, în timp ce Vințan [Vin99b, Vin00c, Ega03] folosește ca intrări în cele două predictoare neurale (LVQ și perceptronul multistrat) atât registrul de istorie cât și adresele branch-urilor. O asemănare între cele două studii se referă la faptul că, în ambele, rețeaua neurală este antrenată dinamic după fiecare predicție. Vințan aplică perceptronului multistrat algoritmul complex de învățare *backpropagation* (va fi descris ulterior în cadrul acestui subcapitol). Algoritmul de învățare al lui Jimenez, mai simplu și mai rapid poate fi

rezumat pe scurt astfel: intrările în rețeaua neurală pot fi 1 sau -1. Dacă o intrare în rețea coincide cu rezultatul branch-ului (un *nottaken* este asociat cu -1 iar un *taken* cu +1) atunci ponderea aferentă respectivei intrări este incrementată. În caz contrar ponderea este decrementată. În cadrul perceptronului lui Jimenez fiecare pondere reprezintă gradul de corelație (pozitivă sau negativă) dintre comportamentul unui salt trecut (văzut de pe nivelul de intrare) și comportamentul saltului curent care trebuie prezis; cu cât este mai mare o pondere cu atât va fi influențată mai mult predicția. Concluziile ambilor cercetători sugerează faptul că rețelele neurale extrag un grad de corelație suplimentar față de schemele adaptive pe două niveluri, cu implicații benefice asupra acurateții de predicție. Din punct de vedere al performanței, perceptronul lui Jimenez înlătură până la 36% din predicțiile eronate obținute cu cel mai bun predictor hibrid existent în varianta comercială, de tip McFarling [Jim02]. Acuratețea predicției obținută de Vințan variază între 89% și 92% și echivalează cu o îmbunătățire cu 3% a predictorului neural față de structurile adaptive pe două niveluri [Ega03]. Diferența de performanță se poate datora și faptului că Jimenez folosește pe intrările perceptronului doar registrul de istorie a cărui lungime poate varia până la 66 de biți, în timp ce Vințan restricționează nivelul de istorie la doar 10 biți. De asemenea, algoritmi de antrenare diferiți pot constitui un factor critic în determinarea comportamentului rețelelor neurale utilizate.

Primul predictor neural care se prezintă este cel bazat pe modelul de rețea *Learning Vector Quantization* [Vin99b]. LVQ este o rețea neurală simplă folosită pentru clasificarea pattern-urilor. Modul de învățare este unul supervizat bazat pe competiție. În cadrul algoritmului vor fi atâția vectori binari câte clase diferite sunt. Numărul de biți folosiți la codificarea unui singur pattern determină lungimea acestor vectori. În principiu rețeaua neurală, componentă a structurii de predicție (vezi figura 5.6), este compusă din doi vectori binari: V_t - asociat unei predicții de tip *taken* respectiv V_{nt} - asociat unei predicții de tip *nottaken*. Fiecare dintre acești vectori binari are o lungime de $(i+k+1)$ biți, în concordanță cu lungimea totală a celor 3 componente (adresă branch, istorie globală, istorie locală) pe care se bazează predicția. Inițial, fiecare element al vectorului V_{nt} este inițializat cu zero logic iar fiecare element al vectorului V_t cu unu logic. În procesul de predicție, fiecare salt în curs de predicționat generează un vector binar $X(t)$ pe $(i+k+1)$ biți, în corespondență cu PC, HRg și HRl pe care le are asociate la momentul respectiv. Pentru a clasifica un patern este calculată distanța

Hamming dintre patternul de intrare și fiecare vector $HD = \sum_{p=1}^{i+k+1} (X_p - V_{w_p})^2$,

unde X este paternul de intrare și V_w este unul dintre vectori (V_{nt} sau V_t).

Paternalul X este clasificat ca făcând parte din clasa asociată vectorului care are distanța Hamming cea mai mică. Vectorul (V_{nt} sau V_t) având distanța Hamming minimă față de vectorul $X(t)$, se numește "vector învingător" (V_w - "winner"). Celălalt vector se va numi perdant (V_l - "loser").

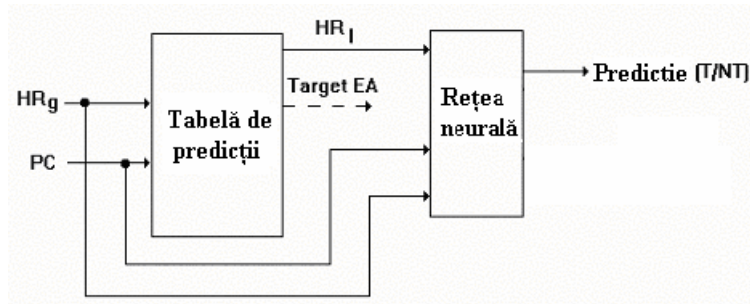


Figura 5.6. Schemă bloc a predictorului neural [Vin99b]

Algoritmul de predicție adaptiv este descris de ecuațiile de mai jos (5.1 și 5.2) Fiind un algoritm de învățare vectorii inițiali trebuie modificăți, dar într-un singur ciclu se modifică doar vectorul câștigător, cel perdant rămâne nemodificat, motiv pentru care se consideră algoritmul de învățare bazat pe competiție.

$$V_w(t+1) = V_w(t) \pm a(t)[X(t) - V_w(t)] \quad (5.1)$$

$$V_l(t+1) = V_l(t) \quad (5.2)$$

În ecuația (5.1) $a(t)$ reprezintă pasul de învățare și ia valori reale pozitive, subunitare. Semnul "+" se referă la o predicție corectă iar semnul "-" la una incorectă. Metoda propusă face pentru prima dată pe plan mondial legătura între problema predicției salturilor în microprocesoarele de mare performanță și respectiv problema recunoașterii formelor, recunoscând în prima un caz particular al celei de a 2-a probleme. Forma de intrare (X) se clasifică într-una din cele 2 forme care se modifică adaptiv, V_t respectiv V_{nt} . Practic se înlocuiesc automatele de predicție deterministe de tip numărătoare saturate (uzual de ordinul miilor sau zecilor de mii) cu o singura rețea neurală de predicție.

Acuratețea medie de predicție, obținută cu predictorul LVQ pe benchmark-urile Stanford printr-o metodologie *trace-driven*, este de 88.10% [Ega03], echivalentă unui predictor de tip BTC (vezi subcapitolul 5.1.1) dar inferioară celei produse de o structură adaptivă pe două niveluri. În ciuda modestiei rezultatelor, acestea pot fi considerate încurajatoare din cel puțin două motive: primul ar fi simplitatea rețelei neurale folosite iar, al doilea, faptul că algoritmul de învățare este complet neoptimizat – practic rețeaua învață pe toată durata procesării trace-ului. De asemenea, se remarcă

superioritatea perceptronului [Jim02] față de LVQ [Vin99b] din punct de vedere al acurateții predicției dar și al implementării la viteze de procesare ridicate datorită calculelor complexe implicate de operațiile în virgulă mobilă din cadrul LVQ.

Preluat din [Ega03] se prezintă un al doilea model de rețea neurală utilizată în predicția salturilor, și anume, perceptronul multistrat, o rețea de tip feed-forward, cu algoritm de învățare de tip backpropagation. Backpropagation definește doi pași: primul în care informația trece de la intrare către ieșire și apoi un pas înapoi de la nivelul de ieșire la nivelul de intrare. Pasul de trecere înainte propagă vectorul de intrare în primul nivel al rețelei, ieșirile din acest nivel produc un nou vector care vor fi intrări pentru nivelul următor, până când se ajunge la ultimul nivel al căror ieșiri produc ieșirile rețelei. Pasul înapoi este similar cu cel înainte exceptând faptul că erorile sunt propagate înapoi prin rețea pentru a determina adaptarea ponderilor. Algoritmul backpropagation, reprezentând de fapt o metodă de minimizare a erorii medii pătratice, poate fi formulat astfel:

1. Inițializează ponderile cu valori mici aleatoare în intervalul $[-2/N_1, 2/N_1]$ unde N_1 reprezintă numărul de noduri de pe nivelul de intrare [Gal93].

2. Plasează la intrare un vector X .

3. Propagă acest vector de intrare înainte prin rețea.

4. Calculează eroarea în nivelul de ieșire.

$$\delta_i^m = f'(h_i^m)[d_i^m - y_i^m],$$
 unde f' este derivata funcției de activare f ,

h_i^m reprezintă intrările perceptronului i din nivelul m , d este rezultatul dorit iar y este rezultatul produs de ultimul nivel al rețelei.

5. Calculează erorile pentru nivelele precedente prin propagarea erorilor înapoi:

$$\delta_i^m = f'(h_i^m) \sum_j w_{ij}^{l+1} \delta_j^{l+1},$$
 unde l ia valori de la numărul nivelului

ultim -1 până la 1.

6. Actualizează ponderile folosind $\Delta w_{ji}^l = \eta \delta_i^l y_j^{l-1}$.

7. Repetă de la pasul 2 cu următorul pattern până când eroarea din nivelul de ieșire scade sub un anumit nivel (threshold).

Cu toate că alegerea lui η influențează într-o mare măsură algoritmul de învățare backpropagation, această alegere este dependentă de specificul problemei.

Predictorul neuronal de salturi implementat în [Ega03] este constituit în principal dintr-o rețea neuronală, un registru de istorie global (HRg) și o tabelă cu regiștrii de istorie locală. Rețeaua neuronală (pe post de predictor

global al tuturor salturilor din program) este o rețea de tip Multi Layer Perceptron, de tip *feedforward* având un singur nivel ascuns, un nivel de intrare și un nivel de ieșire (vezi figura 5.7). Fiecare nivel este format dintr-un număr de noduri astfel:

- ✓ Pentru primul nivel numărul de noduri este calculat după următoarea formulă:

dimensiunea PC-ului + dimensiune HRg + dimensiunea unui HRI.

- ✓ Dimensiunea nivelului al doilea este dată de *dimensiunea primului nivel + un delta dat* ca parametru de intrare (având ca valori discrete 2,4,6,8).
- ✓ Nivelul de ieșire conține un singur nod și constituie predicția saltului (*taken / not taken*).

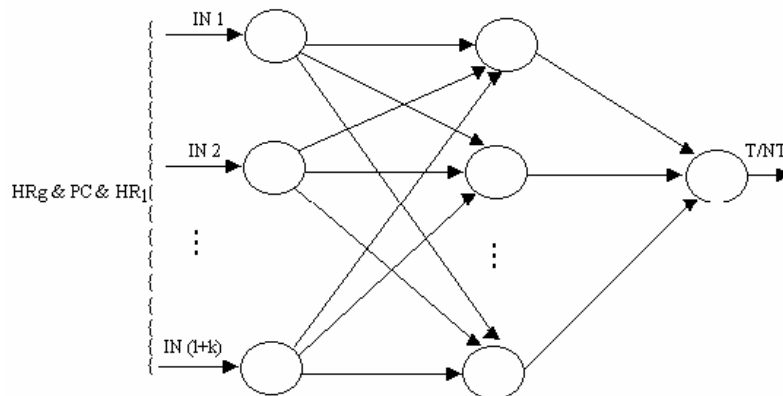


Figura 5.7. Predictor neural de ramificații program [Ega03]

Fiecare nod de pe nivelurile ascuns și de ieșire este în legătură cu nodurile nivelului care le precede prin intermediul unor ponderi. Valorile fiecărui nod de pe aceste două niveluri sunt calculate după următoarea formulă:

unde n_j este nodul a cărui valoare este calculată

$$n_j = f\left(\sum_i n_i * w_{i,j} + t_j\right)$$

n_i este unul din nodurile nivelului precedent iar $w_{i,j}$ este ponderea ce leagă nodul i de nodul j

În [Ega03] sunt studiate patru rețele neuronale diferite, rezultate în urma folosirii pe intrările rețelei a două tipuri de istorie, locală și respectiv globală dar și a două funcții de activare diferite (una de tip gaussian $\frac{2}{1+e^{-\beta \cdot x}} - 1$ și

una de tip sigmoidal $\frac{1}{1 + e^{-\beta \cdot x}}$). Utilizarea celor două versiuni de funcții se datorează în principal codificării diferite a intrărilor în rețea. La folosirea funcției sigmoidele intrările sunt 0 și 1 (binare) și o valoare mai mare decât 0.5 la ieșirea rețelei reprezintă predicție *taken* a saltului curent, în timp ce o valoare sub acest prag (0.5) semnifică predicție *nottaken*. Când se folosește funcția sigmoială bipolară (gaussiană) intrările în rețea sunt codificate -1 și respectiv +1, iar un branch este prezis *taken* dacă ieșirea rețelei exprimă o valoare mai mare sau egală cu 0. O ieșire negativă sugerează un branch *nottaken*. Jimenez utilizează în studiul său doar intrări bipolare în rețea (-1 și +1) argumentând că intrările binare având valoarea 0 (în cazul folosirii funcției sigmoidele) chiar dacă au o pondere substanțială asupra rezultatului (predicției curente), prin înmulțirea dintre pondere și 0 (valoarea de pe nivelul de intrare) nu rezultă nici un impact asupra ieșirii rețelei (vezi ecuația 5.3 care descrie modelul analitic al perceptronului [Jim02]).

$$y = w_0 + \sum_{j=1}^n w_j x_j \quad (5.3)$$

Revenind la studiul din [Ega03], perceptronul multistrat cu backpropagation, istorie locală pe intrare și funcție de activare gaussiană s-a dovedit cel mai performant (acuratețea predicției obținută fiind de 91.53%), îmbunătățind acuratețea unui predictor adaptiv corelat pe două niveluri de tip GAs cu 5.2%. Cu toate acestea, datorită complexității algoritmului de învățare backpropagation, până la ora actuală este imposibil de a realiza o predicție într-un număr restrâns (1÷4 [Jim02]) de perioade de tact procesor, conform cerințelor existente în domeniul microarhitecturilor (vezi detalii la începutul subcapitolului 5.1).

O soluție în vederea creșterii acurateții de predicție a schemelor neurale constă în combinarea predicției statice cu cea dinamică în trei moduri [Vin99a, Vin01]:

- ❖ Prima metodă constă în preînvățarea unor statistici, realizate asupra trace-ului, de către rețeaua neuronală cu o anumită eroare urmată de predicția propriu-zisă. În prima fază ponderile sunt ajustate astfel încât rețeaua “*învață*” o parte din trace, realizându-se o pre-predicție (se stabilește anterior execuției fiecare salt cum va fi predicționat atunci când va fi executat) înaintea execuției iar apoi urmează predicția dinamică din timpul execuției.
- ❖ A doua metodă constă în “*antrenarea*” rețelei folosind un număr de $n-1$ trace-uri furnizate în mod aleator până când eroarea maximă a învățării va scade sub o anumită valoare. Antrenarea este realizată printr-o

predicție “*în gol*” adică predicția unui trace fără reținerea rezultatului. După ce eroarea învățării a ajuns sub o anumită valoare urmează predicția propriu-zisă pe cel de-al n -lea trace, unde de data aceasta sunt reținute rezultatele predicției.

- ❖ A treia metodă constă în utilizarea unui algoritm genetic pentru antrenarea rețelei neurale. Algoritmii genetici constituie o parte a calculului evolutiv care la rândul său este o componentă a inteligenței artificiale. Ca și rețelele neurale, algoritmii genetici se bazează pe o metaforă biologică (asocierea soluției unei probleme unui cromozom uman și folosirea operatorilor genetici pentru determinarea de noi soluții). Prin acești algoritmi se percepe învățarea ca o competiție între membrii unei populații de soluții posibile a unei probleme în evoluție. Este utilizată o funcție de evaluare a fiecărei soluții care să estimeze dacă o soluție va contribui la următoarea generație de soluții. Apoi prin operații similare cu transferul de gene din reproducerea sexuală algoritmul creează o nouă populație de posibile soluții. Inițial se stabilește o populație de cromozomi suficient de largă pentru a asigura variabilitatea populației. Fiecare cromozom conține un număr de gene, acestea reprezentând de fapt o pondere din cadrul rețelei neurale. Genele aparținând populației inițiale de cromozomi sunt inițializate cu valori aleatoare în intervalul $[-1, 1]$. Algoritmul genetic se desfășoară astfel:

- ☐ pentru un număr de generații stabilit se repetă pașii următori
 - /// sunt evaluați toți cromozomii
 - /// este aplicat operatorul de încrucișare (Cross)
 - /// este aplicat operatorul de mutație (Mutation)
- ☐ cel mai performant cromozom din generația finală va conține prin genele sale ponderile rețelei antrenate static.

Parametrii cheie ai acestui algoritm sunt numărul de generații și dimensiunea populației. Dacă populația este prea mică nu va fi suficientă diversitate în cadrul populației pentru a obține soluția optimă. Dacă însă numărul de generații este prea mic nu vor fi destule șanse pentru a ajunge la soluția optimă. De notat că optimul nu este garantat că va fi obținut, există doar o mare probabilitate de a-l găsi.

Predictorul neural propus de Jimenez [Jim02] – va fi numit *Perceptron* pe parcursul acestui subcapitol, reprezintă o schemă de predicție corelată pe două niveluri care înlocuiește automatele de predicție deterministe (tabela PHT din figurile 5.1, 5.2) cu perceptroni cu un singur strat. Întrucât capacitatea predictorului variază liniar și nu exponențial cu dimensiunea nivelului de intrare, principalul avantaj al *Perceptronului* lui Jimenez se referă la posibilitatea exploatării unei istorii mult mai lungi în

procesul de predicție (până la 66 de salturi anterioare) în vederea creșterii acurateții de predicție. Perceptronul cu un singur strat, introdus pentru studiul funcțiilor creierului [Ros62] constă dintr-un singur neuron artificial care conectează n unități de intrare ponderate, la o singură ieșire. În cazul predictorului Perceptron intrările reprezintă conținutul registrului de istorie globală, iar ieșirea (un singur bit) specifică dacă saltul curent supus predicției se va face sau nu. Fiecare pondere reprezintă gradul de corelație dintre comportamentul unui salt trecut și comportamentul saltului supus predicției.

Figura 5.8 ilustrează modelul grafic iar ecuația 5.3 prezintă modelul analitic al perceptronului. Acesta este descris printr-un vector ale cărui elemente sunt ponderile, numere întregi cu semn reprezentate pe 8 biți. Ieșirea perceptronului se determină ca produs scalar dintre vectorul de intrare $x_{1..n} - x_0$ fiind întotdeauna 1, oferind o pondere de bază (*bias*), și vectorul de ponderi $w_{0..n}$. **Ponderea de bază (*bias*) reflectă (învață) în ce măsură (mai mare sau mai mică) comportamentul saltului curent este independent de istoria salturilor anterioare.** Intrările perceptronului sunt bipolare, o valoare -1 aferentă unui x_i semnificând salt nottaken iar un +1 însemnând salt taken. O ieșire negativă este interpretată ca predicție nottaken iar una pozitivă semnificând predicție taken.

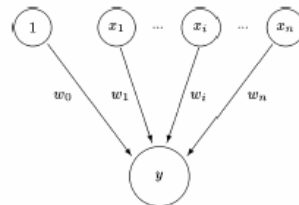


Figura 5.8. Predictorul *Perceptron* [Jim02]

În figura 5.9. se prezintă pentru o mai bună înțelegere un mic exemplu. Se consideră astfel un registru de istorie care reține comportamentul ultimelor 4 salturi. Se observă că salturile al doilea și al patrulea (corespondente biților 1 și respectiv 3) contribuie cel mai mult la predicția saltului curent. Cea de-a doua pondere arată că există o puternică corelație pozitivă între direcția celui de-al doilea branch și direcția branch-ului curent supus predicției. De asemenea, se remarcă o puternică corelație negativă între rezultatul saltului patru și rezultatul saltului curent. Întrucât al patrulea salt nu s-a făcut această corelație, sugerează faptul că saltul curent se va face.

	bit	0	1	2	3	Intrare de bază
comportament anterior al salturilor		NT	T	T	NT	
istoria salturilor		-1	1	1	-1	1
ponderile		1	30	-2	-20	10

Stabilirea predicției $-1 + 30 - 2 + 20 + 10 = 57 \geq 0 \rightarrow$ Predicție Taken

Figura 5.9. Exemplu privind mecanismul de predicție al *Perceptronului*

În cele ce urmează va fi descris mecanismul de realizare a predicției, algoritmul de învățare – actualizare a ponderilor și modul de implementare hardware al perceptronului. Procesorul încorporează o tabelă de N perceptroni implementați în memorie SRAM foarte rapidă. Parametrul N este impus de bugetul hardware și respectiv de numărul de ponderi – determinat la rândul său de cantitatea de istorie reținută. În momentul în care procesorul întâlnește un branch, pe timpul fazei de fetch instrucțiune, au loc următoarele:

1. Adresa saltului este dispersată pentru a obține un index $i \in 0..N-1$ în tabela de N perceptroni.
2. Perceptronul i este extras din tabelă într-un registru vectorial $P_{0..n}$ de ponderi.
3. Valoarea ieșirii perceptronului, y , este calculată ca produs scalar între vectorul P și registrul de istorie globală.
4. Branch-ul va fi prezis nottaken dacă y este negativ sau taken în caz contrar. Asupra punctelor 3° și 4° se va insista când se va descrie modul de implementare hardware al tabelii de perceptroni.
5. Odată ce rezultatul real al branch-ului devine cunoscut, algoritmul de învățare folosește acest rezultat și valoarea y (ieșirea prezisă) pentru actualizarea ponderilor în registrul vectorial P .
6. Ponderile actualizate vor fi înscrise în intrarea i a tabelii.


```

function prediction (pc: integer): (taken, not taken);
begin
  i := pc mod N;
   $y_{out} := W[i,0] + \sum_{j=1}^h \begin{cases} W[i,j] & \text{if } G[j] = \text{taken} \\ -W[i,j] & \text{if } G[j] = \text{not taken} \end{cases}$ 
  if  $y_{out} \geq 0$  then
    prediction = taken
  else
    prediction = not taken
  end if
end

procedure train (i, yout: integer, prediction, outcome :(taken, not taken));
begin
  if prediction ≠ outcome or  $y_{out} \leq \theta$  then
     $W[i,0] := W[i,0] + \begin{cases} 1 & \text{if } outcome = \text{taken} \\ -1 & \text{if } outcome = \text{not taken} \end{cases}$ 
    for j in 1..h parallel do
       $W[i,j] := W[i,j] + \begin{cases} 1 & \text{if } outcome = G[j] \\ -1 & \text{if } outcome \neq G[j] \end{cases}$ 
    end for
    G := (G << 1) or outcome
  end if
end

```

Selectează perceptronul corespunzător - *i*
 Calculează ieșirea respectivului perceptron folosind
 registrul de istorie globală *G* ca și intrare
 Predicționează saltul curent în funcție de semnul
 ieșirii perceptronului.
 Dacă predicția este incorectă sau ieșirea
 nu depășește un anumit prag atunci:
 Este incrementată ponderea de bază
 dacă saltul s-a făcut sau decrementată
 dacă saltul nu s-a făcut
 În paralel sunt incrementate ponderile
 pentru o corelație pozitivă (rezultatul
 saltului a fost identic cu rezultatul
 saltului anterior *j*) și decrementate
 pentru o corelație negativă
 Este actualizat registrul de istorie globală
 conform rezultatului saltului.

Figura 5.10. Implementarea software a mecanismului de predicție și a algoritmului de învățare

În [Jim02a] sunt prezentate în pseudocod (apropiat de limbajul Algol) două subprograme care permit implementarea software a mecanismului de predicție și a algoritmului de învățare (vezi figura 5.10). Lungimea registrului de istorie este considerată h . Tabela de predicție este reprezentată de o matrice $N \times (h+1)$ ponderi, numere întregi pe 8 biți – $W[0..N-1, 0..h]$. Fiecare rând al matricei este un vector de $(h+1)$ ponderi, care reprezintă ponderile unui perceptron. Prima coloană reține ponderile (de bază) independente de istoria branch-urilor, aferente celor N salturi din tabela de perceptroni. Vectorul boolean $G[1..h] \in \{1..h\} \times \{\text{nottaken}, \text{taken}\}$ reprezintă registrul global de deplasare care păstrează istoria salturilor. Expresiile *taken* și *nottaken* sunt considerate constante booleene iar cuvintele scrise cu litere aldine reprezintă cuvinte cheie.

La nivel hardware matricea ponderilor W este implementată ca o memorie mapată direct direct, fără TAG, cu N blocuri, fiecare bloc conținând $(h+1)$ ponderi aferente unui perceptron. Astfel, de fiecare dată când este necesară o predicție se citește din memorie într-un registru vectorial aceste ponderi. Circuitul care determină ieșirea perceptronului (calea critică pentru realizarea unei predicții) acceptă ca semnale de intrare tabloul de ponderi (registrul vectorial) și registrul de istorie. Întrucât elementele registrului de istorie pot fi doar -1 și +1 rezultă că nu mai este necesară implementarea unui circuit multiplicativ, acesta fiind înlocuit cu

simple adunări și scăderi (adunare cu complementul față de 2). În urma simulărilor efectuate, Jimenez [Jim02] a apreciat că o adunare cu complementul față de 1 reprezintă o bună estimare a complementului față de 2, și suplimentar, evită întârzierea introdusă de un sumator cu propagare a transportului. Astfel, dacă intrarea i a registrului de istorie este -1 rezultă că ponderea i din tablou trebuie scăzută (adunat la suma totală y a complementului față de 2 a respectivei ponderi). Prin reprezentarea lui -1 în complement față de 2 și efectuarea unui *sau exclusiv* (*xor*) cu valoarea ponderii respective, și știind că:

$$a \text{ xor } 1 = \text{not } a \quad \forall a \in \{0, 1\},$$

se obține practic complementul față de 1 al ponderii. Prin aproximarea propusă de Jimenez rezultă practic valoarea căutată. $(-1 \cdot w_i)$ care va fi adunată la suma totală y . Dacă intrarea i din registrul de istorie este 1 atunci nu se aplică nici o operație ponderii respective. După determinarea fiecărui produs $x_i w_i$ dacă este cazul ($x_i = -1$) se calculează suma tuturor produselor folosind un circuit numeric bazat pe arbori Wallace – sumatoare cu transport salvat cu 3 intrări și 2 ieșiri. Pentru suma a trei numere de câte m biți efectuată de un sumator cu transport salvat de dimensiune $\theta(m)$ este necesar un timp $\theta(1)$ pentru a ajunge la suma a două numere. Arborii Wallace reduc problema sumei a n numere codificate pe m biți la suma a două numere codificate pe $2m$ biți. Ultimele două numere rezultate sunt adunate cu ajutorul unui sumator cu transport anticipat [Cor90]. Întrucât adâncimea unui astfel de arbore este de ordinul $O(\log n)$ iar cea a unui sumator cu transport anticipat este de ordinul $O(\log m)$, iar m în [Jim02] este 8, rezultă că timpul de calcul este relativ scurt. Semnul sumei va determina dacă predicția va fi *taken* sau *nottaken*.

Cercetarea lui Jimenez [Jim02] cuprinde și o simulare la nivel de circuit a tabelii de perceptroni implementată hardware. În acest sens au fost folosite:

- ✓ Simulatorul CACTI, un instrument de modelare a cache-ului pentru evaluarea timpului necesar citirii ponderilor din tabela de perceptroni și înscrierea lor într-un registru vectorial.
- ✓ Simulatorul HSPICE, care determină timpul necesar stabilirii ieșirii perceptronului: *taken* (≥ 0) sau *nottaken* (< 0).

În condițiile unui buget hardware limitat, trei parametri pot fi variați pentru obținerea modelului optim de perceptron: *dimensiunea registrului de istorie*, numărul de biți pe care vor fi reprezentate *ponderile* și *pragul* folosit în algoritmul de învățare. Studiind influența istoriei globale asupra latenței de predicție a perceptronului se desprinde o primă concluzie privind

fezabilitatea acestuia. Astfel, pentru un buget hardware de 4KB și o lungime a istoriei de 24 rezultă o latență a predicției de 2.4ns, care reprezintă 2 perioade de $tact_{CPU}$ pentru un procesor Alpha 21264 la 833MHz, implementat într-o tehnologie de 180nm (performanță acceptabilă din punct de vedere al latenței de predicție, inacceptabilă prin prisma frecvențelor de procesare actuale). Acuratețea de predicție a respectivului perceptron este de 95.4%, superioară cu 26% față de un predictor *gshare* [McFar93, Bur97] echivalent.

Tendențele microarhitecturale curente însă, urmăresc implementarea de microprocesoare cu frecvențe de tact cât mai ridicate, chiar și în detrimentul performanței globale de procesare (măsurată în IPC – instrucțiuni per $tact_{CPU}$). În aceste condiții predictoarele care pot furniza o predicție într-un număr cât mai mic de perioade de tact (1÷2) sunt fie hibride, fie cascade (se va discuta în subcapitolul 5.2.3), fie predictoare cu suprascriere a predicției (*overriding*). Jimenez [Jim02] propune o astfel de schemă – structură ierarhică caracterizată de faptul că, în același timp este furnizată o predicție și de un prim nivel, foarte rapid, dar mai simplu (*gshare*) și de un al doilea nivel, mai complex cu latență mai mare de predicție (perceptronul). Primul nivel generează imediat o predicție care poate fi suprascrisă de cel de-al doilea nivel puțin mai târziu. Dacă se întâmplă acest lucru atunci procesul de fetch este reluat de la adresa stabilită de predictorul de pe nivelul 2, provocând o penalitate, mai mică totuși decât cea datorată unei predicții greșite. Spre exemplu, latența de suprascriere (*overriding*) a predictorului hibrid încorporat în procesorul Alpha 21264 este de 2-3 perioade de $tact_{CPU}$ iar penalitatea în cazul unui salt greșit predicționat, fără proces de suprascriere este de 7 perioade de $tact_{CPU}$ [Kes99]. În principiu, la apariția unui branch există patru posibilități:

- ✓ Predicțiile generate de cele două nivele coincid și sunt corecte. Rezultă penalitate nulă.
- ✓ Predicțiile diferă, iar cea de pe nivelul 2 este corectă. În acest caz, cel de-al doilea nivel suprascrie predicția primului nivel, cu o penalitate redusă.
- ✓ Cele două predicții generate sunt în dezacord, iar perceptronul greșește. În acest caz penalitatea este ridicată (trebuie realizată întâi suprascrierea predicției de pe primul nivel, apoi după determinarea predicției greșite a celui de-al doilea nivel prin aflarea rezultatului saltului trebuie reluat procesul de fetch de pe ramura corectă). Cu toate că reprezintă cel mai defavorabil caz, este mai puțin probabil să apară întrucât, luate individual, acuratețea predicției de pe nivelul 2 este superioară predicției de pe nivelul 1.
- ✓ Cele două predictoare generează aceeași predicție, dar greșită. Nu se realizează o suprascriere dar apare penalitatea datorată predicției greșite.

Astfel, la o frecvență de 1.76GHz oferită de procesorul Intel Pentium4, predictorul cu suprascriere având perceptron cu istorie de 24 pe nivelul al doilea, realizează predicția în patru perioade de $tact_{CPU}$: 1 $tact_{CPU}$ – pentru citirea ponderilor din tabela de perceptroni și alte 3 $tact_{CPU}$ – pentru calcularea ieșirii perceptronului folosit pentru predicție. Cele 3 perioade de $tact$ se datorează în primul rând arborelui Wallace care, pentru $n=24$, are 7 nivele (conform relației de recurență $D(n) = D\left(\left\lceil \frac{2 \cdot n}{3} \right\rceil\right) + 1$ [Cor90]). Aceste 7 nivele, plus sumatorul final cu transport anticipat (rădăcina arborelui Wallace) sunt pipelinizate astfel: primele 3 nivele sunt executate în 1 $tact_{CPU}$, următoarele 3 nivele sunt executate în 1 $tact_{CPU}$, și ultimul nivel plus sumatorul se execută în cel de-al patrulea $tact_{CPU}$. Chiar și în condițiile unei viteze de procesare ridicate (1.76GHz), un predictor cu suprascriere având pe al doilea nivel Perceptronul [Jim02] cu latență de suprascriere de 4 cicli de $tact$ conduce la o performanță globală de procesare, superioară (cu 5.7%) unui procesor Alpha 21264 care încorporează un predictor hibrid cu suprascriere cu latență de 3 cicli [Kes99]. Aceleași simulări demonstrează o superioritate a performanței globale de procesare a arhitecturii bazată pe Perceptron cu 15.8% față de o arhitectură echivalentă Pentium4 cu predictor Gshare încorporat. O soluție de îmbunătățire a timpului de efectuare a predicției de către Perceptron într-un procesor cu viteză ridicată de execuție (1.76GHz) o reprezintă predictorul neural „*path-based*” [Jim02a]. Acesta eșalonează calculele în timp permițând eliminarea arborelui Wallace în procesul de însumare și înlocuirea cu un simplu sumator. Latența de predicție devine astfel 2 perioade de $tact_{CPU}$, în concordanță cu predictoarele existente în implementările comerciale [Sez02].

În finalul acestui subcapitol vor fi prezentate câteva din avantajele (suplimentare) oferite de perceptron pe lângă acuratețea de predicție ridicată, performanță globală de procesare superioară altor implementări și fezabilitate hardware – în condiții echivalente cu structurile adaptive pe două niveluri. Astfel, unul din avantajele se referă la *independența față de istorie a timpului de antrenament* în cazul perceptronului, contrar cu schemele adaptive corelate pe două niveluri, la care timpul de antrenament depinde de lungimea istoriei, iar de la o anumită mărime poate afecta negativ performanța. Un alt avantaj îl reprezintă *asignarea unui grad de confidență predicției*. Acesta este determinat de distanța absolută dintre ieșirea perceptronului și 0 (pragul față de care se stabilește dacă saltul se face sau nu se face). Cu cât această distanță este mai mare *se asigură* faptul că acel branch va fi taken sau nottaken. O confidență redusă permite microarhitecturii execuția speculativă a ambelor ramuri ale saltului (taken

sau nottaken) în timp ce o confidență ridicată impune execuția doar a unei anumite ramuri (cea prezisă). De asemenea, Perceptronul ajută la identificarea salturilor mai importante din punct de vedere al corelației dintre acestea, pentru predicția unui anumit salt – stabilește care biți de istorie sunt mai importanți (pondere foarte mare în modul) și care nu (pondere apropiată de 0) în predicția saltului curent. Această proprietate a perceptronului poate fi folosită în determinarea informațiilor de profil a salturilor și combinată cu alte structuri de predicție cunoscute în literatură [Sta98] (vezi subcapitolul 5.2.2.4) care exploatează informația de corelație variabilă.

5.2. NECESITATEA PREDICȚIEI SALTURILOR / APELURILOR INDIRECTE. SOLUȚII.

Din punct de vedere microarhitectural, ultima perioadă este caracterizată de creșterea în importanță a salturilor indirecte comparativ cu salturile condiționate, în ciuda frecvenței destul de reduse a primelor în programele de calcul. Unul din motive îl reprezintă execuția predicativă care determină reducerea numărului de salturi condiționate (înjumătățirea acestora în cadrul arhitecturii IA'64 [Intel97]). Un alt motiv îl constituie amploarea pe care a luat-o dezvoltarea de aplicații desktop, vizuale dar mai ales obiectuale, respectiv tendința de portabilitate a cât mai multor dintre acestea. Aplicațiile desktop sunt bogate în apeluri de funcții de bibliotecă [Flo04], iar programele C++ și Java asigură un stil polimorfic de programare în care, legarea dinamică a apelurilor de subrutine (implementată la nivel hardware prin apeluri indirecte) constituie instrumentul principal de proiectare modulară a codului. Adăugând la acestea dificultatea predicției salturilor indirecte comparativ cu celelalte tipuri de instrucțiuni de salt se justifică aportul crescut al salturilor / apelurilor indirecte asupra costului global generat de predicțiile greșite și respectiv necesitatea dezvoltării de noi mecanisme de predicție (structuri hibride, predictoare cascade pe două sau mai multe niveluri [Dri98c, Dri99] sau adaptate din predicția valorilor) pentru predicția cu acuratețe a acestora. Dacă se mai pune în balanță și faptul că un număr restrâns de salturi indirecte statice sunt responsabile pentru mai bine de 90% din salturile indirecte dinamice (2 – *go.ss*, 3 – *jpeg.ss* etc, vezi benchmark-urile SPEC'95 – subcapitolul 7.1.1, respectiv o medie de 22 pe 8 benchmark-uri

obiectuale [Dri98a]), rezultă și mai clar că, performanța globală a arhitecturilor este extrem de sensibilă la predicția salturilor indirecte (o îmbunătățire a predicției pentru un singur salt static are un impact semnificativ asupra acurateții globale de predicție, respectiv asupra ratei de procesare în arhitecturile superscalare actuale cu structuri pipeline extrem de „adânci”). Creșterea adâncimii pipe-ului determină nu doar o reducere a perioadei de tact a procesorului ci și o accentuare a efectului defavorabil provocat de hazardurile de toate felurile, dar mai ales de cele de ramificație. Simulări efectuate pe o suită amplă de benchmark-uri (SPEC'95, '2k, aplicații desktop, multimedia și Internet, recunoașterea vorbirii, CAD etc.) având la bază o arhitectură superscalară echivalentă procesorului Intel Pentium 4 au demonstrat diminuarea ratei globale de procesare cu 0.45% pentru fiecare ciclu de tact suplimentar necesar în cazul unei predicții greșite a salturilor [Spra02].

Pentru creșterea acurateții predicției se pune mai întâi problema determinării caracteristicilor de program și a paradigmatelor existente în aplicațiile procedurale respectiv obiectuale care generează salturi / apeluri indirecte. Astfel, simulări laborioase (până la 6.000.000 de salturi indirecte executate / program de test) au arătat că procentajul apelurilor indirecte de funcții datorat metodelor virtuale este de peste 34% (în medie pe 8 benchmark-uri obiectuale: *porky, eqn, troff, jhm* etc este de 69%) [Dri98a].

Pe parcursul subcapitolului 5.2 este realizată o descriere a structurilor de predicție dedicate salturilor / apelurilor indirecte, dezvoltate pe parcursul ultimei decade, de la cele mai simple până la cele mai complexe, urmând ca în subcapitolul 5.3 să fie prezentate câteva cercetări proprii care urmăresc determinarea limitelor existente, implementări ale schemelor actuale și îmbunătățiri aduse acestora în vederea creșterii acurateții predicției.

5.2.1. PREDICTOARE ADAPTIVE PE DOUĂ NIVELURI. PREDICTORUL TARGET-CACHE.

Schema clasică de predicție BTB (branch target buffer) specifică în special salturilor condiționate, a fost propusă pentru început și pentru predicția salturilor indirecte (vezi figura 5.11). Au existat două variante: *standard*, în care target-ul este evacuat cu fiecare predicție greșită, respectiv *BTB-2bc*, un BTB standard suplimentat cu numărătoare saturate pe doi biți aferente fiecărei locații (câmpul 2bc) în care target-ul este actualizat cu cel corect doar după a doua predicție greșită (structură oarecum similară cu predictorul de valori LastValue descris în subcapitolul 6.1.1).

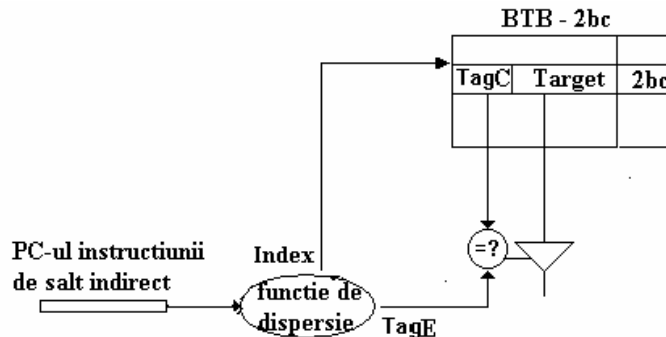


Figura 5.11. Predicția target-urilor în cazul salturilor indirecte. Structura BranchTargetBuffer [Dri98a]

În implementarea acestor scheme s-a plecat de la premisa că, chiar și în cazul salturilor polimorfice ale căror target-uri variază foarte des, în general predomină în execuție un singur target [Aig96]. Acuratețea maximă de predicție obținută pe benchmark-urile SPEC'95 a fost de 71.9% cu schema BTB standard și respectiv 75.1% cu BTB-2bc. Simulări efectuate cu cel din urmă predictor pe o suită de programe atât procedurale cât și obiectuale au relevat variația acurateții de predicție (între 62% - pe programe C și 80% - pe programe C++ și Java)[Dri98a].

Întrucât în predicția salturilor condiționate schemele adaptive, corelate pe două niveluri s-au dovedit mai eficiente decât cele de tip BTB [Yeh92], au fost propuse și pentru predicția salturilor / apelurilor indirecte (vezi figura 5.12), cu o deosebire însă. Deoarece majoritatea salturilor indirecte sunt necondiționate rezultă ineficiența păstrării unei istorii (*history pattern*) a comportamentului salturilor indirecte anterioare (Taken) alegându-se în schimb să se rețină pe nivelul de istorie comportamentul salturilor condiționate premergătoare saltului indirect (Taken / NotTaken) [Cha97].

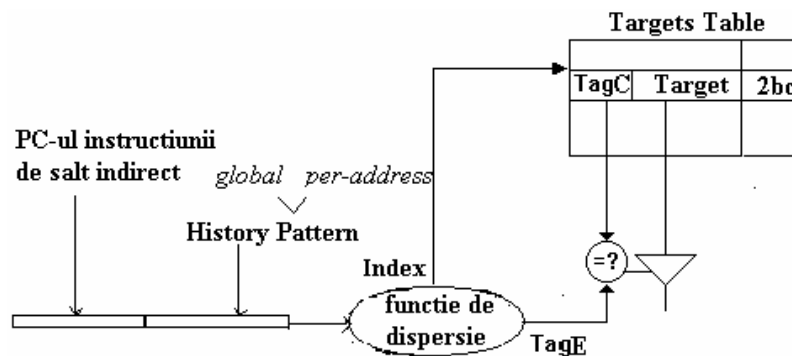


Figura 5.12. Predicția target-urilor salturilor indirecte prin scheme adaptive corelate pe două niveluri [Dri98a]

Variante de implementare ale schemei adaptive pe două niveluri folosesc pe nivelul *History Pattern* target-urile anterioare sau anumiți biți ai acestora – *path* (funcție și de dimensiunea pattern-ului), aferente respectivului salt indirect static, supus predicției [Dri98a]. În cadrul acestui predictor rezultat, informația de predicție suplimentară poate fi concatenată, comprimată prin intermediul unei funcții de dispersie sau chiar întreșută. Rezultatele raportate de cercetători [Dri98a] referitor la care din cele modalitățile de indexare ale structurii de predicție sunt mai eficiente, exprimă următoarea concluzie: *creșterea în acuratețe în cazul în care informația de predicție este concatenată nu justifică utilizarea unei tabele de target-uri foarte mari comparativ cu o tabelă de dimensiuni mai modeste, necesară dacă informația de predicție este comprimată.*

Diferența în performanță între un BTB și cel mai bun predictor adaptiv pe două niveluri, în cazul salturilor indirecte, devine semnificativă doar pentru tabele de target-uri mai mari de 64 de intrări. Pentru atingerea unei acurateți de predicție de 90% este necesar un predictor adaptiv pe două niveluri *path-based* cu o tabelă de minim 1024 intrări [Dri98a]. În urma unor simulări laborioase, Driesen a ajuns la concluzia că o cale (*path*) cu o lungime de 3 target-uri anterioare respectiv o tabelă cu grad de asociativitate 4-way s-au dovedit optime din punct de vedere al acurateții de predicție, rezultate oarecum similare cu cele obținute de autor în cercetările efectuate (vezi subcapitolul de rezultate – 7.1). Întrucât fiecare instrucțiune de salt ar necesita un număr mare de intrări în tabela de target-uri care variază exponențial cu lungimea *pattern*-ului, pentru o tabelă de capacitate limitată, creșterea dimensiunii *pattern*-ului poate conduce la un număr ridicat de miss-uri de capacitate cu implicații defavorabile asupra acurateții predicției.

Din categoria predictoarelor adaptive pe două niveluri poate fi considerat ca făcând parte și predictorul Target Cache (vezi figura 5.13), implementat și supus analizei în subcapitolul 5.3.2. Sunt discutate în literatură două variante: un predictor bazat pe *pattern* – care reține comportamentul ultimelor salturi condiționate (T / NT), respectiv unul bazat pe *path* (calea spre respectivul salt indirect, compusă atât din *pattern* cât și din adresele instrucțiunilor de salt condiționate). În acest caz predicția adresei de salt nu se mai face pe baza ultimei adrese țintă a saltului indirect, ca în schemele de predicție clasice, ci pe baza alegerii uneia dintre ultimele adrese țintă ale respectivului salt, memorate în structură. În unele implementări [Cha97, Dri98a], structura de predicție, memorează pe parcursul execuției programului pentru fiecare salt indirect ultimele *pattern* adrese țintă (sau un număr de biți din fiecare). Cele două/trei surse de informație binară (PC, *pattern* / *path*) sunt în general prelucrate prin intermediul unei funcții de dispersie (XOR), rezultând indexul de adresare

în cache și *tag*-ul aferent. În implementarea proprie s-a optat pentru un *Target Cache* *N*-way asociativ (un set conține *N* adrese destinație) cu “*TAG*” pentru evitarea interferențelor ce pot să apară atunci când o instrucțiune de salt utilizează o intrare din structură care a fost anterior accesată de către un alt salt, generând o predicție “*din start*” greșită, cu implicații defavorabile asupra timpului de execuție.

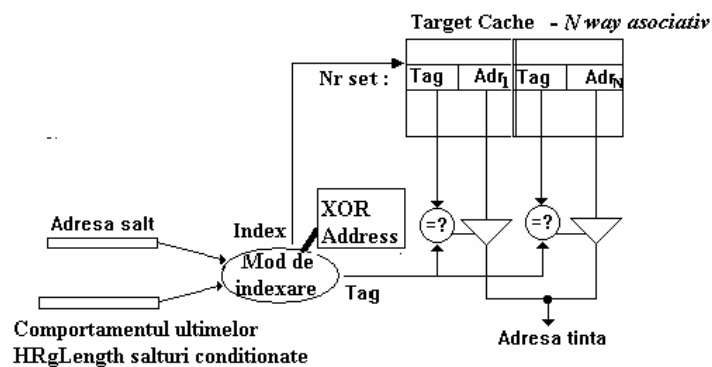


Figura 5.13. Predicția target-urilor în cazul salturilor indirecte. Structura TargetCache [Cha97]. HRgLength – lungimea *pattern*-ului de istorie globală

Modul de funcționare a structurii de predicție Target Cache este următorul: numărul de *biți mai puțin semnificativi* ai cuvântului obținut prin dispersia PC-ului saltului și a registrului de istorie globală a salturilor condiționate (globalHR) formează *Indexul* iar *Tag*-ul (reprezentând de fapt contextul de apariție al aceluia salt indirect) aferent valorii care va fi predicționată este reprezentat de *cei mai semnificativi biți*. În cazul unui acces cu hit la locația Index în TargetCache valoarea prezisă (target-ul saltului) va fi câmpul *Adr* din structură, în caz contrar putându-se alege una din următoarele variante (nu se prezice un target sau se face o *predicție implicită* – spre exemplu, cea mai recentă adresă destinație din setul respectiv). În implementarea realizată (vezi subcapitolul 5.3.2), un miss în Target Cache nu conduce la îmbunătățirea acurateții de predicție. După ce adresa țintă a saltului devine efectiv cunoscută, în situația unui hit în TargetCache dacă predicția a fost eronată, target-ul de la TAG-ul emis prin PC (și eventual globalHR) va fi actualizat cu noua adresă destinație, reprezentând de fapt contextul de apariție al aceluia salt indirect. În caz de miss în TargetCache tabela este actualizată prin introducerea valorii reale a registrului sursă aferent instrucțiunii de salt indirect, evacuând din lista de target-uri de la respectiva adresă (identificată prin set) cea mai veche dintre acestea, printr-un algoritm de tip **FIFO** (first in first out). Principiul predicției este în acest caz simplu: la același *pattern* de context, aceeași

adresă țintă. Prin astfel de scheme, măsurat pe benchmark-urile SPEC'95, acuratețea predicției salturilor indirecte crește și ca urmare, câștigul global asupra timpului de execuție este de cca 4.3% - 9%. Un TargetCache cu tag, de tip *16-way set asociativ*, folosind o *istorie globală pe 16 biți* reduce timpul de execuție cu 12,66%, respectiv 4,74% pe benchmark-urile SPEC'95 *perl* și *gcc*, în timp ce un target cache *8 way set asociativ* reduce timpul de execuție cu 10,27% respectiv 4,31% pe aceleași două programe de test [Cha97].

5.2.2. PREDICTOARE HIBRIDE ȘI METAPREDICTOARE.

5.2.2.1. CU SELECȚIE BAZATĂ PE CODUL SURSĂ.

În literatura de specialitate recentă au fost propuse câteva noi scheme de predicție bazate pe codul sursă ce stă la originea salturilor și apelurilor indirecte (principalele „*vinovate*” în acest sens fiind unele construcții *switch/case*, apelurile indirecte de funcții prin pointer și respectiv polimorfismul – apelul metodelor virtuale, fapt arătat în capitolul 4). Astfel, în urma informațiilor extrase după compilare sau pe baza informațiilor de profil, instrucțiunile de salt pot fi partiționate în clase diferite, fiecărei clase fiindu-i atașat câte un predictor specific. Aceste predictoare specifice sunt combinate sinergic într-un predictor hibrid (vezi figura 5.14). În acest scop Driesen a folosit simulatorul dedicat *shade* pentru generarea trace-urilor tuturor salturilor indirecte [Cme93].

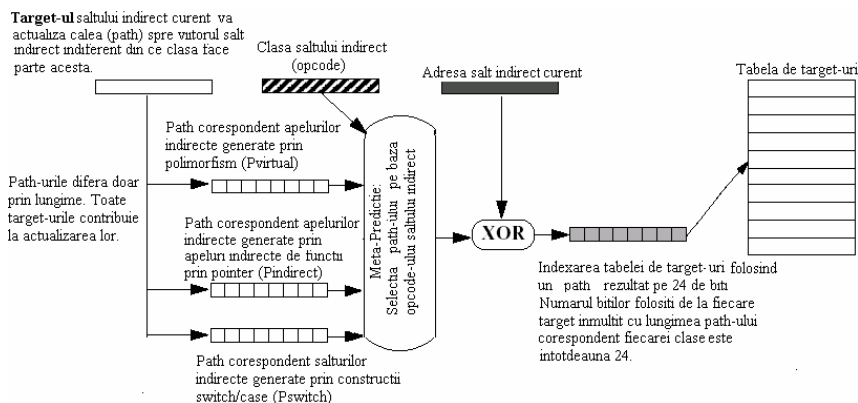


Figura 5.14. Predictorul hibrid cu mecanism de clasificare bazat pe *opcode* [Dri98b]

În experimentul propus de Driesen [Dri98b], predictoarele componente diferă doar prin lungimea căii (dependent de clasa fiecărui salt indirect). În timp ce istoria (*path*-ul) fiecărui predictor component este actualizată de către toate salturile indirecte indiferent de clasa căreia îi aparțin, doar o singură „*cale*” este selectată pentru predicție (cea aferentă clasei din care face parte branch-ul curent supus predicției). Predictorul hibrid poate folosi o tabelă de target-uri partajată de către toate clasele de salturi indirecte sau tabele separate pentru fiecare clasă. Din punct de vedere al timpului de procesare, această selecție bazată pe „*opcode*” (care identifică clasa branch-ului) pare să întârzie predicția de la finele fazei IF (instruction fetch) la finele fazei de decodificare – ID.

În continuare sunt descrise câteva aspecte legate de detecția salturilor indirecte, procentajul fiecărei clase, și numărul de target-uri distincte generat de fiecare clasă, detalii folositoare în analiza performanței predictorului hibrid. Așa cum s-a demonstrat în subcapitolul 4.2 al prezentei lucrări, în limbajul C/C++, construcțiile *switch/case* cu mai mult de 5 cazuri determină în urma compilării salturi indirecte (folosind GNU gcc 2.6.3 cu opțiunea de optimizare *-O3*). Driesen a probat și el acest lucru cu deosebirea că versiunea de compilator a fost GNU gcc 2.7.2, opțiunea de optimizare *-O2*, *-multrasparc plus static linking*, iar numărul de cazuri rezultat a fost 7. Recunoașterea acestor construcții în codul asamblare MIPS (generat după compilare, având ca platformă gazdă un procesor Intel Pentium II și sistem de operare Linux Red Hat 7.3) este relativ simplă întrucât au mnemonicica *j \$reg* (salt necondiționat la adresa specificată de registru *reg*) [Flo04]. Pentru identificarea apelurilor indirecte datorate *metodelor virtuale* Driesen folosește EEL – o bibliotecă de instrumente pentru analiza și modificarea unui cod executabil – [Lar95], care într-un mod similar cu tehnica OVFC de predicție prin precalculare a target-urilor apelurilor metodelor virtuale [Roth99], selectează o secvență de 5 instrucțiuni care conduc la apelul indirect respectiv (extragere din memorie a adresei obiectului, determinare adresă de bază a tabelii de metode virtuale, calcul adresă exactă la care se găsește adresa metodei virtuale, extragere adresă metodă, salt necondiționat pe prima instrucțiune aferentă metodei virtuale). Celelalte tipuri de salturi indirecte găsite în program, diferite de cele mai sus descrise (*switch* și *virtual*) au fost considerate de Driesen ca fiind apeluri indirecte (datorate apelurilor de funcții prin pointer, funcții de bibliotecă, DLL-uri).

În timp ce în programele obiectuale salturile indirecte datorate apelurilor de metode virtuale predomină (61% - vezi tabelul 5.1), în aplicațiile procedurale, care nu conțin astfel de salturi, dominante sunt apelurile indirecte de funcții prin pointer (69%). Cu toate că salturile

indirecte datorate instrucțiunii de selecție multiplă *switch / case* prezintă cele mai multe adrese destinație comparativ cu celelalte tipuri de salt indirect (12.7), comportamentul salturilor din clasa *switch* din punct de vedere al acurateții predicției este oarecum similar cu cel al salturilor generate de apeluri *indirecte* de funcție prin pointeri (dll-uri), acuratețea maximă de predicție obținându-se pentru monopredictor cu *path* de lungime 3. Apelurile *virtuale* diferă de salturile indirecte din celelalte clase, prin aceea că lungimea ideală a *path*-ului este ușor mai scurtă, fiind corelate probabil cu cele mai recente 2 salturi indirecte anterior executate. Predictorul hibrid cu tabele separate prezintă o acuratețe de predicție inferioară fiecărui monopredictor în parte. Aceasta se datorează în primul rând faptului că predictoarele componente ale hibridului nu utilizează eficient spațiul din tabelele de target-uri, întrucât procentajul fiecărei clase de salturi indirecte variază în funcție de benchmark. De exemplu, programele C nu conțin apeluri virtuale, astfel că cel puțin un sfert din spațiul total al tabelii de target-uri este neutilizat. În cazul predictorilor cu tabelă partajată are loc o balansare dinamică a adreselor de instrucțiuni pentru toate clasele de salturi componente.

Clasa	Structura sursă	Mod de detecție a clasei	Programe obiectuale			Programe procedurale		
			D [%]	S [%]	T	D [%]	S [%]	T
<i>switch</i>	Construcție <i>switch</i> cu mai mult de 7 [Dri98b]	Prin opcode: j – MIPS, jmp – SPARC	22.5	2.6	12.7	31.4	35.6	4.9
<i>virtual</i>	Apeluri de metode virtuale în C++	Folosind EEL [Lar95]; OVFC [Roth99]	61.2	69.4	2.1	–	–	–
<i>indirect</i>	Restul salturilor indirecte	Prin opcode: jal – MIPS, jmp l – SPARC	16.3	28.0	2.2	68.6	64.4	5.1

Tabelul 5.1. Procentajul din totalul salturilor indirecte (Dinamice – D, Statice – S) care îl reprezintă fiecare clasă și numărul mediu de target-uri (T) distincte generat de fiecare clasă [Dri98b]

Câștigul în acuratețe obținut printr-un predictor hibrid bazat pe clasificare statică a opcode-ului este destul de redus, sugerând o prea mare generalitate a claselor de salturi și totodată faptul că informațiile semantice obținute de la nivel înalt, sunt insuficiente pentru o acuratețe de predicție ridicată a salturilor indirecte (cunoaștere exactă a comportamentului acestora). Pentru o tabelă de target-uri cu 1024 intrări, 4-way asociativă, partajată de cele 3 clase și *path* (3 – indirecte, 2 – virtuale, 3 – *switch*)

acuratețea obținută este de 90.5% superioară totuși acurateții obținute cu un monopredictor two-level adaptiv cu $path = 3$ (90.2%). În general lungimea $path$ -ului optim variază cu capacitatea tabelii și gradul de asociativitate pentru fiecare clasă.

5.2.2.2. CU SELECȚIE BAZATĂ PE ARITATE.

Aritatea salturilor indirecte [Dri98b] reprezintă numărul target-urilor distincte generate de fiecare salt indirect. Aceasta poate fi fie determinată în urma simulării și cunoscând informații de profil (identificarea claselor de branch-uri) fie estimată printr-o analiză la nivelul codului sursă. În urma studiului efectuat, Driesen [Dri98b] a clasificat instrucțiunile de salt indirect în trei categorii: cu un singur target, cu două target-uri și respectiv cu mai mult de două target-uri. Salturile monomorfe (cu un singur target) sunt executate 34% din totalul salturilor dinamice indirecte iar din cadrul celor polimorfe, cele care au 2 target-uri sunt executate 17% din totalul salturilor dinamice indirecte iar cele cu trei sau mai multe adrese destinație 49%. Rezultate aproape identice, obținute de autorul acestei lucrări în urma simulării benchmark-urilor SPEC'95 pot fi sesizate și în subcapitolul de rezultate – 7.1.1, figura 7.16. În urma clasificării bazate pe aritate, salturile monomorfe pot fi cu succes predicționate de un monopredictor fără istorie (BTB, LastValue). O „cale lungă” crește numărul predicțiilor greșite întrucât fiecare cale diferită spre un același salt cauzează un miss de start rece suplimentar față de cazul în care istoria ar fi nulă. Această situație este profund dăunătoare în cazul salturilor monomorfe deoarece, prin inserarea inutilă și repetată în tabela de target-uri a aceleași adrese destinație aferente unui singur salt, dar cu fiecare nou $path$ întâlnit, cresc chiar și miss-urile de capacitate cu repercursiuni negative asupra acurateții predicției (informația suplimentară acționează ca zgomot). În schimb, salturile polimorfe beneficiază de pe urma istoriei memorate ($path$): pentru predicția optimă a salturilor cu două target-uri (duomorfe) este suficientă o cale ($path$) de lungime 2, iar pentru celelalte o cale de lungime 3 și o tabelă de 1024 intrări.

Comportamentul diferit al acestor clase de branch-uri poate fi exploatat pentru atingerea unei mai bune acurateți de predicție, cu ajutorul unui predictor hibrid care folosește predictoare componente diferite pentru fiecare clasă în parte. Astfel, dacă salturile monomorfe ar fi prezise de un BTB și nu ar mai accesa o structură de predicție mai complexă ($path$ -based) s-ar reduce un număr semnificativ de miss-uri de start rece și capacitate determinând o creștere însemnată a acurateții de predicție. De exemplu, în

cazul benchmark-ului *jhm* – *Java High-level Class Modifier* (benchmark obiectual cu 6.000.000 de instrucțiuni de salt indirect executate), un predictor cu 256 de intrări generează o rată de miss de 2.3% după predicția tuturor branch-urilor, iar dacă salturile monomorfe sunt înlăturate rata de miss scade la 0.9% [Dri98b]. Cu toate acestea, pot exista și salturi greșit clasificate. De exemplu, un branch duomorf poate efectua salturi preponderent la una din adresele destinație, aspect care conduce la concluzia că ar fi fost poate mai eficient dacă saltul ar fi fost clasificat ca monomorf. În subcapitolul 4.3. sunt prezentate astfel de cazuri ce pot să apară în execuția programelor, cu exemplificări pe benchmark-urile SPEC'95 (SPEC 2000) cu număr bogat de salturi indirecte. De concluzia generată prin exemplul anterior ar trebui să țină cont cercetătorii în alegerea mecanismului de clasificare. Astfel, ar fi mai indicată poate o clasificare a branch-urilor după numărul de situații când acestea suferă o modificare a target-ului.

Predictorul hibrid cu mecanism de clasificare bazat pe aritate [Dri98b] deși este superior ca și acuratețe unui monopredictor în aceleași condiții de cost hardware (mai ales pentru tabele de capacitate redusă), și competitiv (echivalent ca performanță – 91% acuratețe pentru o tabelă de 1024 intrări) cu un predictor hibrid *dual-path* (vezi figura 5.16), prezintă și câteva dezavantaje:

- Instrucțiunile de salt indirect trebuie să cuprindă în câmpul *opcode* (să fie adnotate) un contor de aritate, ceea ce presupune o extensie a arhitecturii setului de instrucțiuni.
- Întrucât aritatea se determină folosind informații de profil s-ar putea ca aceasta să varieze funcție de fișierele de intrare folosite, sau estimările făcute în timpul analizei codului să nu fie cele mai corecte.

5.2.2.3. CU SELECȚIE BAZATĂ PE CONFIDENȚĂ.

Predictoarele hibride cu selecție bazată pe confidență folosesc două sau mai multe predictoare simple (P1 și P2) drept componente, multiplexate printr-un mecanism de selecție corespondent fiecărei instrucțiuni statice de salt indirect. McFarling [McFar93] a propus pentru salturi condiționate o tabelă de selecție a predictorului folosit care asociază fiecărui PC un număr saturat pe doi biți ($P1c \in \{0,1\}$, respectiv $P2c \in \{0,1\}$) pentru a păstra comportamentul relativ al celor două predictoare – care dintre ele a prezis cu acuratețe mai mare (vezi figura 5.15). Astfel, dacă ($P1c-P2c$) este 0 atunci predicția va fi făcută de structura care a prezis și ultima oară, dacă ($P1c-P2c$) este 1 atunci va prezice P1, altfel P2. După rezolvarea saltului

numărătorul este actualizat pentru a ilustra acuratețea relativă a celor două predictoare componente. Din punct de vedere al implementărilor comerciale existente, cel mai performant predictor hibrid este înglobat în procesorul Alpha 21264 și este în genul celui propus de McFarling [McFar93], unde P1 este un predictor foarte simplu pe un singur nivel „*line predictor*” – tabelă de numărătoare saturate pe doi biți indexată cu PC-ul saltului [Kes99], iar P2 este un predictor hibrid la rândul său, mai complex [Kes99]. P2 este compus dintr-un predictor pe două niveluri de tip GAg cu 4192 intrări și un predictor tot pe două niveluri cu 1024 intrări care rețin o istorie locală, fiecare astfel de intrare indexând o tabelă de 1024 numărătoare saturate pe 3 biți [Kes99, Sez02, Jim02b].

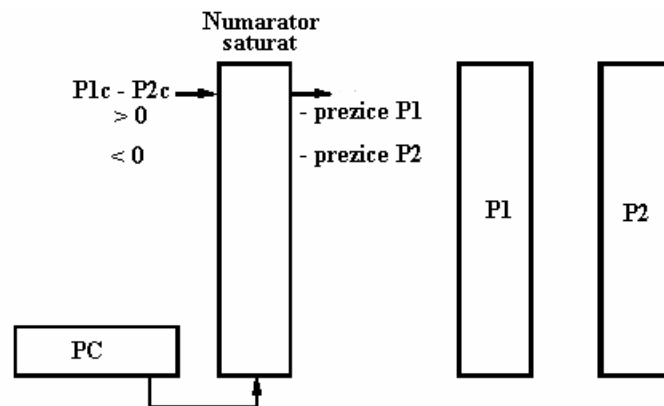


Figura 5.15. Structură hibridă de predictoare [McFar93]

Predictorul hibrid implementat de Driesen, folosește două monopredictoare (două predictoare adaptive pe două niveluri *path-based* a căror nivel *History Pattern* are dimensiunile $k1$ respectiv $k2$, cu $k1 \neq k2$). Fiecare intrare în tabela de target-uri are atașat un numărător de confidență pe n biți, $n \in \{1 \div 4\}$, care exprimă de câte ori predictorul respectiv a generat o predicție corectă în ultimele $2^n - 1$ accese ale respectivei intrări. La înlocuirea unei intrări numărătorul aferent va fi resetat. Valoarea va fi prezisă de monopredictorul care va avea o confidență mai mare pe locația corespunzătoare din tabela proprie. Fiecare salt indirect va actualiza ambele monopredictoare (path-ul respectiv numărătorul de confidență aferent).

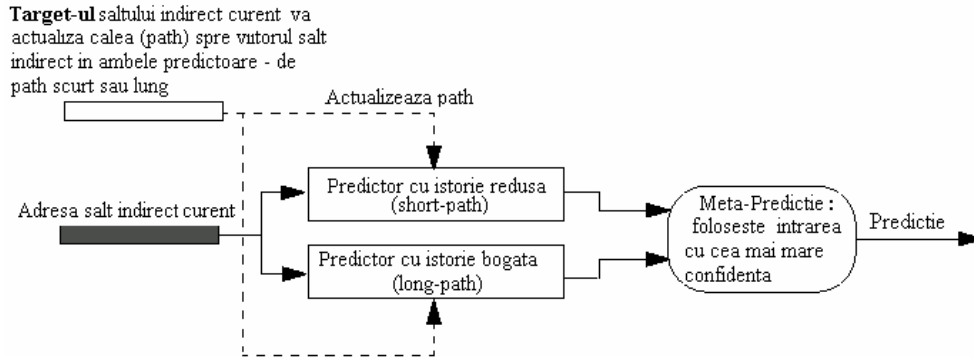


Figura 5.16. Predictor hibrid dual-path [Dri98a]

Schema reprezintă o generalizare naturală a celei propusă de McFarling. Dacă în cazul unui BTB (pattern-ul fiind de lungime 0) informația de confidentă poate fi considerată „*per-branch*”, în cazul unui predictor *path-based* pe două niveluri cu *pattern* > 0 informația de confidentă corespunde mai degrabă fiecărui path, decât fiecărui branch, procesul de selecție (*meta-predicție*) a target-ului fiind mai rafinat decât în cazul lui McFarling. Meta-predicția lui Driesen [Dri98a] evită situații de genul: în lipsa unei informații de context (path) un salt indirect la care se ajunge pe două căi diferite, să partajeze un același numărator de confidentă. Pentru o tabelă 4-way asociativă acuratețea optimă de predicție obținută cu un predictor hibrid *dual-path* variază între 91.02% pentru 1024 intrări și 94.05% pentru 8192 intrări. Simulările [Dri98a] au dovedit că număratoarele de confidentă pe 2 biți sunt suficiente pentru o selecție optimă a monopredictoarelor componente, iar cea mai bună performanță se obține când un monopredictor folosește o cale scurtă ($k1 \in \{1 \div 3\}$) iar celălalt o cale lungă ($k2 \in \{5 \div 12\}$). Predictorul hibrid a dovedit o acuratețe de predicție superioară unui predictor *non-hibrid* în aceleași condiții de cost hardware, în ciuda faptului că cele două monopredictoare suferă mai mult datorită miss-urilor de capacitate și conflict. Mai mult, pentru predictoare non-hibride cu asociativitate 2-way sau 4-way de capacitate mai mare de 2048 intrări se obține o acuratețe mai mare de predicție prin hibridizare decât prin dublarea capacității tablei. Pentru tablele de dimensiuni reduse (64÷512 intrări) efectul creșterii gradului de asociativitate rămâne mai puternic decât efectul hibridizării.

5.2.2.4. STRUCTURĂ DE PREDICȚIE PENTRU EXPLOATAREA CORELAȚIEI VARIABLE.

Din cadrul predictoarelor hibride poate face parte, datorită mecanismului de clasificare al salturilor pe care îl presupune, și schema de predicție propusă de Stark [Sta98] – vezi figura 5.17, care, bazat pe informații de profil, exploatează corelația, variabilă ca și lungime, între instrucțiunea de salt indirect curentă și ultimele salturi condiționate sau indirecte. Implementarea hardware se bazează pe un set de funcții de dispersie care combină un număr variabil de target-uri (de la 1 la N , unde N reprezintă numărul maxim de salturi care sunt corelate cu saltul indirect curent). Acesta constituie primul nivel de istorie (buffer de istorie globală a target-urilor) și cuprinde ultimele N adrese destinație (complete sau comprimate pe k biți) aferente salturilor condiționate, indirecte sau reveniri din subrutină. Funcțiile de dispersie generează un set de N indici care atacă un multiplexor cu rol de selecție. Multiplexorul este controlat fie de compiler, în urma unor consistente informații de profil aferente fiecărui salt indirect, fie de către hardware, sau chiar o combinație dintre cele două. Compilerul și link-editorul sunt responsabili de comunicarea funcției de dispersie potrivite structurii hardware de predicție a salturilor, soluție realizabilă prin modificarea arhitecturii setului de instrucțiuni. Formatul instrucțiunii procesorului Alpha AXP asigură un număr de biți care pot fi utilizați în stocarea indicelui funcției de dispersie necesară la indexarea structurii de predicție. Predictorul indexat cu un „*path*” de lungime variabilă necesită două accese secvențiale la tabelele componente pentru a face predicție: tabela indicilor funcțiilor de dispersie adresată cu cei mai puțin semnificativi biți ai PC-ului saltului indirect curent și respectiv tabela de predicție indexată cu informația emisă la ieșirea multiplexorului.

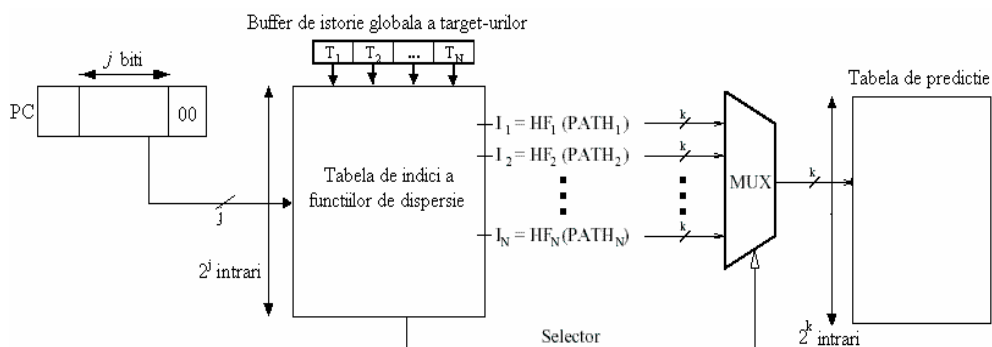


Figura 5.17. Structură de predicție indexată cu un *path* de lungime variabilă [Sta98]

Simulări laborioase pe 8 programe de test din care 4 SPEC'95 (*gcc*, *li*, *m88ksim*, *perl*) și respectiv 4 non-SPEC'95 (interpretor de Ghostscript - *gs*, interpretor de limbaj de programare obiectual Python – *python*, un program de desenare – *plot* și un procesor de documente – *groff*, ultimele două sub GNU) au dovedit o acuratețe medie de predicție pentru salturile indirecte de 87%, dar într-o plajă de valori de la 71% (*python*) până la 99% (*perl*) [Sta98]. Comparativ cu Chang [Cha97], la un cost de implementare de 2k octeți predictorul dezvoltat de Stark reduce numărul predicțiilor greșite cu 36.4% chiar și fără folosirea informațiilor de profil, pentru un N fixat, determinat ca fiind *path*-ul optim (furnizează cel mai mic număr mediu de predicții greșite pe benchmark-urile mai sus amintite). Conștienți de rezultatele totuși inferioare în ceea ce privește acuratețea predicției salturilor indirecte, proiectanții structurii de predicție cu *path* de lungime variabilă au propus și câteva idei noi care să vină în sprijinul cercetătorilor, pentru obținerea de performanțe superioare în domeniul paralelismului la nivelul instrucțiunilor și firelor de execuție. O primă posibilitate interesantă ar fi salvarea istoriei, premergătoare unor structuri de control cum ar fi bucle de program sau apeluri de subrutine, și restaurarea acesteia după încheierea respectivelor structuri. Astfel, înaintea fiecărui apel de procedură este stocată în stivă istoria globală a salturilor anterioare apelului. În acest fel, vechea istorie, preluată din stivă la revenirea din procedură, poate fi combinată cu istoria recentă, obținută pe timpul apelului proceduri, pentru identificarea de corelații între salturi mai îndepărtate [Tho03]. O altă încercare ar putea fi îndreptată spre înlocuirea informațiilor de profiling cu o euristică generată de compilator pentru selecția funcției de dispersie și implicit a dimensiunii *path*-ului, sau utilizarea perceptronului în obținerea informațiilor de profil.

O soluție proprie, pornită de la premisa că nu toate salturile (indirecte) sunt corect determinate de pattern-uri de aceeași lungime [Tho03, Sta98], și preluată din predicția valorilor și a salturilor condiționate [Dri98a], se referă la înlocuirea unui predictor PPM complet, mai performant decât unul adaptiv pe două niveluri în condiții echivalente de cost hardware dar mai „scump de implementat”, cu un predictor hibrid a cărui componente le reprezintă două predictoare contextuale cu pattern-uri de lungime fixă, diferite ($\mathbf{P1} = \text{Markov}(m)$ și $\mathbf{P2} = \text{Markov}(n)$, cu $m \neq n$). În acest sens a se vedea subcapitolul de cercetări proprii 5.3.3.2.

5.2.2.5. SELECȚIA DINAMICĂ A CONTEXTULUI RELEVANT PENTRU PREDICȚIE.

Selecția dinamică a caracteristicilor cu adevărat relevante pentru predicție, bazat pe arbori de decizie evită creșterea exponențială a structurilor de predicție în cazul utilizării unei bogate informații la intrare [Fern03]. Acuratețea predicției obținută de un predictor de salturi bazat pe arbori de decizie, folosind ca și caracteristici istoria salturilor, este comparabilă cu acuratețea predicției furnizată de un predictor convențional, facilitând asemeni *Perceptronului* lui Jimenez [Jim02] utilizarea unui număr ridicat de caracteristici la un cost liniar. De exemplu, un arbore de decizie de adâncime 0 (un singur bit din 64 de biți de istorie locală + globală stabilește predicția) atinge o performanță egală sau ușor superioară decât cele mai neperformante versiuni de predictoare (capacitate redusă) GAP și PAP cu o istorie pe 16 biți.

Arborii de decizie frecvent utilizați în aplicațiile de inteligență artificială (*machine learning*) sunt utili în domenii în care, acuratețea predicției poate fi realizată din reguli care implică un număr restrâns de caracteristici, selectate dintr-un set lărgit al acestora. Un arbore de decizie de adâncime d efectuează predicția bazat pe reguli ce folosesc cel mult $d+1$ caracteristici pentru fiecare predicție, în timp ce predictoarele adaptive, corelate pe două niveluri utilizează un număr considerabil mai mare de caracteristici.

Arborii de decizie sunt constituiți din noduri interne de test, care examinează câte o caracteristică a obiectului, și frunze, care indică clasificarea obiectului. Arborii de decizie sunt folosiți pentru a clasifica obiectele prin testarea inițial a caracteristicii specificate în nodul rădăcină, și apoi urmărind direcția indicată de rezultatul testului în jos, către următorul nivel al arborelui. Rezultatul testelor de pe fiecare nivel determină o cale unică până la o frunză, pentru fiecare obiect. Când se atinge un nod frunză, clasificare asociată respectivei frunze este asignată obiectului.

Caracteristica folosită pentru împărțire producătoare a celor mai omogene partiții este utilizată pentru testul nodului rădăcină. Deoarece este posibil ca partițiile astfel rezultate să fie încă neomogene, se creează recursiv sub-arbori pentru fiecare partiție creată prin aplicarea testului asupra partiției anterioare. Recursivitatea se oprește în momentul în care toate partițiile din frunzele arborelui aparțin aceleiași clase, când nu mai sunt caracteristici care să fie testate, sau când dimensiunea subpartițiilor este prea mică încât este puțin probabil ca o continuare a procesului de separare să fie utilă [Cal97].

Există câteva explicații potențiale a superiorității predictorului bazat pe arbori de decizie:

- ☐ Prima dintre acestea o reprezintă *abilitatea arborilor de decizie de a selecta cele mai importante (relevante) caracteristici pentru predicție*, aferente fiecărei instrucțiuni de salt, coroborată cu faptul că unele dintre acestea sunt predicționate mai bine păstrând o istorie globală, altele reținând o istorie locală, iar pentru anumite salturi fiind utilă păstrarea unei istorii combinate (locală + globală).
- ☐ O a doua explicație se referă la *dimensiunea istoriei folosită pentru predicție*. Dacă în cazul predictorilor adaptive, corelate pe două niveluri istoria este limitată la 16 biți datorită costului hardware exponențial, predictorii bazate pe arbori de decizie obțin o performanță ridicată dacă baza de selecție a caracteristicilor este suficient de mare (≥ 32 de biți), la un cost liniar însă.
- ☐ A treia justificare constă în *timpul inferior de "warm-up" aferent predictorului bazat pe arbori de decizie*, datorită abilității de ignorare a caracteristicilor irelevante pentru predicție. Prin păstrarea informațiilor de predicție irelevante în cazul predictorilor adaptive corelate pe două niveluri se dublează numărul automatelor care trebuie aduse într-o stare corespunzătoare pentru o predicție corectă ("warm-up").

Predictorul bazat pe arbori de decizie operează într-o manieră dinamică în două etape, similar cu predictorii clasice:

- Prima, o reprezintă **predicția**, realizabilă în faza de aducere a instrucțiunii și necesită ca informații la intrare un vector de caracteristici (*istorie locală + istorie globală*) iar la ieșire va genera pe un bit dacă saltul curent supus predicției se va face (1) sau nu se va face (0).
- În a doua etapă, se realizează **actualizarea** stării predictorului bazat pe o pereche formată dintr-un vector de caracteristici și rezultatul real al saltului, pentru îmbunătățirea predicțiilor viitoare bazate pe aceleași informații de intrare.

Salvarea de resurse în cazul arborilor de decizie se datorează mecanismului de selecție al vectorilor de trăsături (caracteristici). Astfel, decât să memoreze câte o predicție pentru fiecare vector de caracteristici, prin partiționarea mulțimii de vectori de caracteristici într-un număr mai mic de subseturi, este stocată câte o predicție pentru fiecare subset. Metoda dă randament maxim când un procentaj ridicat de vectori din fiecare subset predicționează același rezultat; este mai probabil ca predicția unui subset de vectori să fie mai uniformă decât cea a setului care include respectivul subset.

O **definiție** sumară a arborilor de decizie este următoarea: *un arbore de decizie este un arbore binar ale cărui noduri conțin o caracteristică (un bit sau mai multe informații) sau negația acesteia. Adâncimea unui nod al arborelui este dată de numărul de arce care traversează calea de la rădăcină spre respectivul nod. Adâncimea arborelui constituie adâncimea celui mai îndepărtat nod de rădăcină.* În aceleași condiții primare (dându-se un arbore de decizie și un vector de caracteristici) predicția arborelui poate fi definită recursiv, astfel:

- Dacă rădăcina arborelui este o frunză (arborele este constituit de fapt dintr-un singur nod) atunci valoarea prezisă va fi rezultatul testului nodului rădăcină (1 sau 0), pentru vectorul de caracteristici dat.
- În caz contrar, este evaluat nodul rădăcină pentru vectorul de caracteristici dat și funcție de rezultat (1 sau 0), va prezice subarborele drept sau stâng al arborelui.

Predicția hardware bazată pe arbori de decizie presupune antrenarea acestora folosind un mecanism de selecție dinamică a corelației caracteristicilor (banc de numărătoare cu semn, câte unul aferent fiecărei caracteristici de intrare). Acest mecanism urmărește alegerea celei „mai corelate” caracteristici dintr-un set lărgit al acestora. Spre deosebire de predictorile bazate pe tabele în care numărătoarele erau saturate și semnul acestora indica predicția, în cazul arborilor de decizie este foarte importantă și amplitudinea (*valoarea absolută*), care stabilește gradul de corelație al respectivei caracteristici cu rezultatul saltului curent, supus predicției. După observarea unei perechi de tipul – vector de caracteristici / rezultat salt, în faza de actualizare a predictorului sunt incrementate toate contoarele aferente caracteristicilor cu valori identice cu cea a rezultatului saltului și decrementate celelalte (vezi figura 5.18).

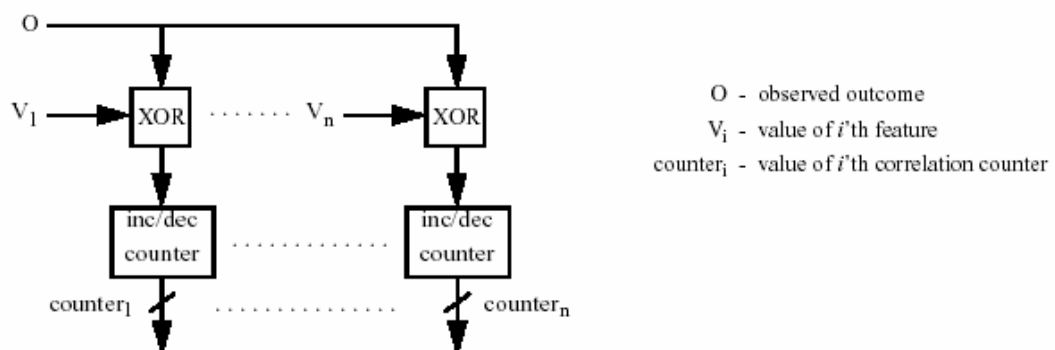


Figura 5.18. Actualizarea contoarelor de corelație

Astfel, după întâlnirea unui număr ridicat de perechi formate din vector de caracteristici și rezultat salt, un contor cu amplitudine ridicată dovedește o puternică corelație a caracteristicii aferente cu saltul curent (pozitivă sau negativă, în funcție de semnul contorului).

Caracteristica cu valoarea contorului cea mai mare reprezintă *cea mai corelată*¹ dintre acestea și va avea un rol hotărâtor pentru predicție (vezi figura 5.19). La finele acestui subcapitol este descrisă și o altă modalitate de selecție a caracteristicii optime după care se face predicția la nivelul fiecărui nod.

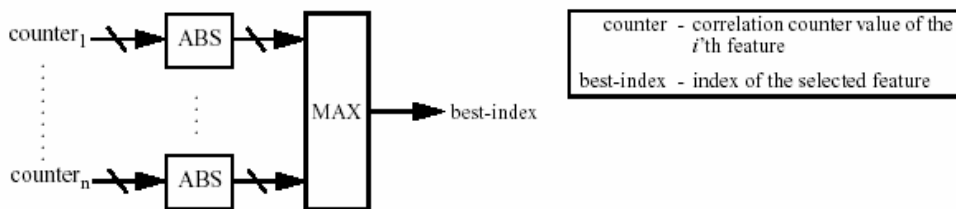


Figura 5.19. Determinarea celei mai predictibile caracteristici

În continuare este descrisă structura arborelui de decizie – datele componente și funcționalitatea fiecărui nod precum și modul de combinare al nodurilor individuale într-un arbore capabil să furnizeze predicția. Se disting două tipuri de noduri: *frunze* (noduri terminale) și noduri interne, care au doi „copii” (subarbore stâng respectiv drept) cu rol de predicție fiecare. Funcționalitatea celor două tipuri de noduri este aproape identică și se bazează pe cele două moduri de operare, anterior descrise (predicție și actualizare). Starea internă a fiecărui nod este ilustrată prin selectorul de context relevant pentru predicție (corelația caracteristicilor de intrare), la care se mai adaugă încă două informații suplimentare (o caracteristică constantă și predicția provenită de la nivelul „copiilor”).

Fiecare nod intern realizează o funcție de „spargere” respectiv „combinare” a două predictoare „copii” pentru a prezice cu o acuratețe mai mare bazat pe șirul de perechi – vector de caracteristici / rezultate salt, observate anterior de acel nod. Prin „spargere” se înțelege o divizare a șirului de perechi în două subșiruri, conform caracteristicii celei mai predictibile (f_{split}). Fiecare din cele două subșiruri este transmis predictorului „copii” corespunzător; întrucât target-urile (rezultatele salturilor) vor fi mai uniforme, cele două subșiruri vor fi mai ușor de prezis decât șirul original. Nodul părinte reține predicția generată de unul din cele două predictoare „copii” și poate să o folosească sau să o ignore. La nivelul nodului părinte,

¹ corelată cu comportamentul saltului (T/NT) pe o „lungă” perioadă de timp real

pe lângă celelalte caracteristici de intrare, se mai adaugă una, f_{sub} , care reprezintă recomandarea predictorului „copil”. Dacă contorul atașat va avea amplitudinea mai mare decât a celei mai corelate caracteristici, atunci nodul părinte va ține cont de recomandarea predictorului „copil” (va folosi respectiva predicție). Cea de-a doua caracteristică suplimentară celor de intrare, o reprezintă o caracteristică *constantă* – f_c , care identifică dacă șirului de vectori de caracteristici aferente unui nod, îi corespunde target-uri aproape uniforme. Figura 5.20 ilustrează cum un nod divide un șir de perechi – vector de caracteristici / rezultate salturi în funcție de caracteristica cea mai predictibilă. Practic, toate perechile de vectori / target-uri ($F_1 \dots F_6$) au fost anterior întâlnite astfel încât la momentul predicției fiecare nod își cunoaște starea internă a contoarelor de corelație.

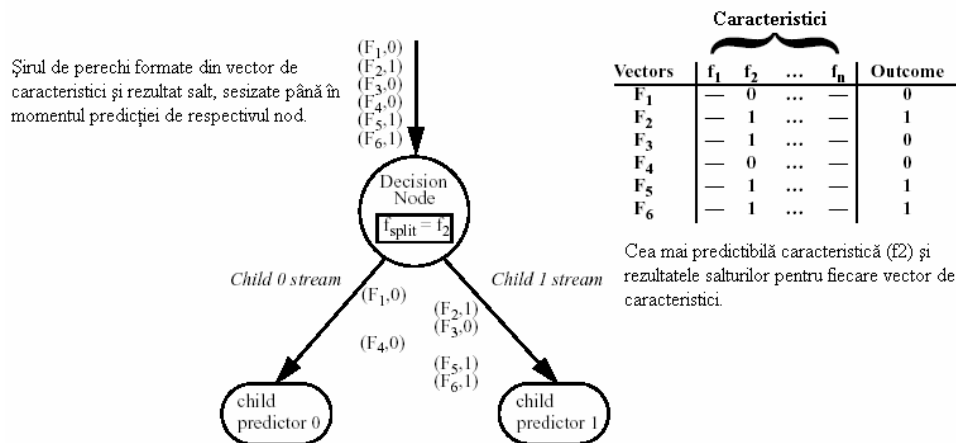


Figura 5.20. Divizarea unui șir de perechi – vector de caracteristici / rezultat salt în funcție de cea mai predictibilă caracteristică (f_{split})

În figura 5.21 sunt descrise faza de predicție și *informațiile sumare* din nodul intern care concură la aceasta. Pe lângă caracteristicile de intrare $f_1 \dots f_n$ și cele suplimentare f_c , respectiv f_{sub} , pentru realizarea predicției mai sunt utile următoarele informații binare, rezultate ca efect al unor funcții aplicabile caracteristicilor de mai sus:

- ✓ **sign**(f_{split}) – reprezintă semnul corelației celei mai corelate caracteristici (1 – dacă corelația este pozitivă, 0 – dacă corelația este negativă).
- ✓ **best**(f_1, \dots, f_n) returnează indicele celei mai corelate caracteristici pe baza valorilor exprimate de contoarele asociate caracteristicilor. Astfel, **best-index** reprezintă indicele caracteristicii f_{split} .
- ✓ **use- f_c** := ($f_c = \text{best}(f_c, f_1, \dots, f_n, f_{sub})$).
- ✓ **use- f_{sub}** := ($f_{sub} = \text{best}(f_c, f_1, \dots, f_n, f_{sub})$).

✓ $\text{sign}(f_c)$ – indică semnul caracteristicii constante.

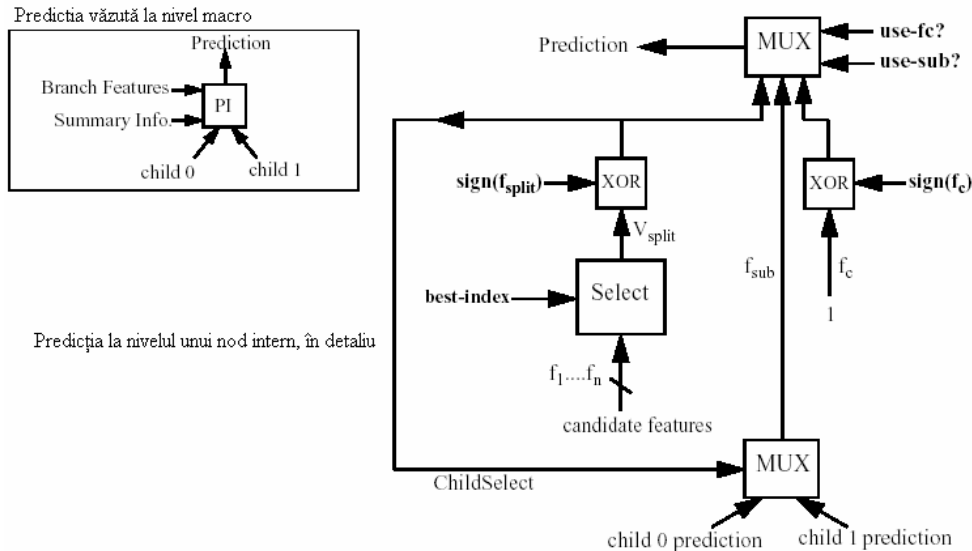


Figura 5.21. Informațiile necesare și modul de realizare a predicției la nivelul unui nod intern

Rolul multiplexorului din partea superioară a figurii este de a selecta pe baza valorilor biților $\text{use-}f_c$ și $\text{use-}f_{sub}$ care caracterică va stabili predicția: f_{split} , f_c sau f_{sub} . Algoritmul de predicție poate fi descris prin următoarea secvență:

```

/* se consideră o funcție predict care întoarce o valoare booleană și
   care primește ca parametru un nod al arborelui de decizie */
if (use- $f_c$  = 1) then
    predicție = sign( $f_c$ )    /* întrucât  $f_c=1$  rezultă că dacă
                               există o corelație pozitivă atunci
                               predicție = 1, altfel predicție = 0. */
else
    if (use- $f_{sub}=0$ ) then    /* se ignoră recomandarea
                               predictorului "copil" */
        predicție =  $V(f_{split}) \oplus \text{sign}(f_{split})$ 
    else
        if ( $V(f_{split}) \oplus \text{sign}(f_{split})=0$ ) then // predicția va fi dată de
                                                       // predictorul copil stâng
            predicție = predict(subarbore_stâng)
        else // predicția va fi dată de
              // predictorul copil drept
            predicție = predict(subarbore_drept)

```


Blocul **Select** generează caracteristica cea mai predictibilă de indice **Best-index**. Rolul celor două blocuri **XOR** este de a furniza valoarea caracteristicii sau negația acesteia, în funcție de gradul de corelație existent (mai precis semnul acesteia). Rolul multiplexorului din partea inferioară a figurii este de a actualiza caracteristica f_{sub} cu predicția subarborelui stâng sau drept în funcție de caracteristica f_{split} și semnul acesteia.

În faza de actualizare, pe baza perechii curente – vector de caracteristici / rezultat salt, sunt modificate contoarele de corelație (incrementate / decrementate), aferente fiecărei caracteristici. În plus, nodul curent activează predictorul „copil” – doar cel care furnizează caracteristica f_{sub} , pentru actualizarea contoarelor aferente caracteristicilor acestuia. Actualizarea trebuie să aibă loc chiar dacă în procesul de predicție, nodul părinte ignoră recomandarea predictorului „copil”. Cu noile contoare se trece la actualizarea informațiilor sumare (**sign**(f_{split}), **sign**(f_c), **best-index**, **use-f_c** și **use-f_{sub}**). De asemenea, trebuie specificat că **actualizarea se aplică în paralel tuturor nodurilor de-a lungul căii formate din nodurile copil selectate de la rădăcină până la frunze, dar nu și celorlalte noduri.**

În acest moment sunt cunoscute toate informațiile și funcționalitatea unui nod intern. Spre deosebire de acestea, nodurile terminale (*frunze*) diferă prin faptul că nu rețin și actualizează caracteristica f_{sub} , neexistând predictive „copil” care să o genereze. Nodurile interne și cele terminale sunt conectate într-un arbore de decizie cu rol de predictor hardware dinamic. Conexiunile realizate nu alterează operațiile de bază ale fiecărui nod, cu o singură excepție: divizarea nodurilor trebuie făcută de fiecare dată după o altă caracteristică. Un nod nu trebuie „spari” după o caracteristică care a fost anterior folosită la divizarea unui nod „strămoș”.

În continuare este descris **modul de implementare hardware**, la nivel de cip, a structurilor de date arborescente, anterior prezentate. Cheia implementării realiste a predictorului hardware bazat pe arbori de decizie este de a organiza informațiile colectate în nodurile arborelui sub forma unor tabele care ocupă un spațiu acceptabil și necesită timpi de acces redus.

În figura 5.22 este prezentată structura de predicție, compusă din logica de decizie (arborele – DDT), o tabelă de predicție care stochează informațiile sumare aferente fiecărui nod al arborelui și tabela de contoare de corelație aferente caracteristicilor din fiecare nod.

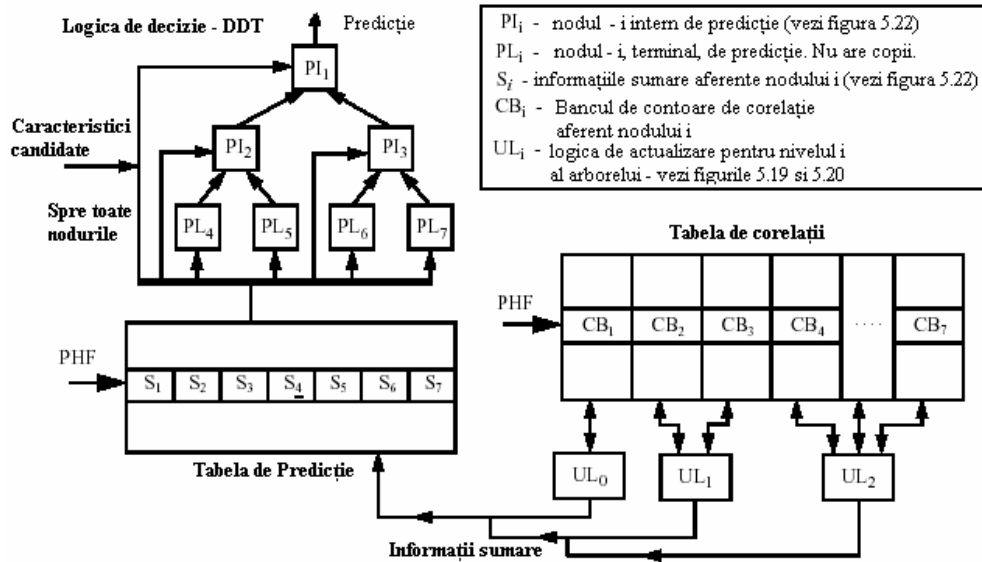


Figura 5.22. Implementarea predictorului hardware bazat pe arbori de decizie - DDT

Tabela de corelații este un masiv bidimensional, indexat după o dimensiune cu cei mai puțin semnificativi biți ai adresei instrucțiunii de salt (PHF – *prediction hash function*) iar după cealaltă dimensiune cu nodul arborelui. Practic, în cazul unei instrucțiuni de salt sunt selectate toate contoarele de corelație aferente caracteristicilor din fiecare nod, pentru a putea fi procesate la nevoie, în paralel de logica de decizie. Contoarele de corelație sunt incrementate / decrementate la cunoașterea rezultatului real al saltului în faza de actualizare.

În faza de predicție, PC-ul instrucțiunii de salt și valorile vectorului de caracteristici constituie informația de intrare în structura hardware de predicție. Tabela de predicție este indexată cu cei mai puțin semnificativi biți ai PC-ului și sunt selectate informațiile sumare aferente fiecărui nod al arborelui de decizie. Acestea, folosite în conjuncție cu caracteristicile de intrare, permit fiecărui nod să emită decizia: fie să predicționeze bazat pe o singură caracteristică (constantă, sau cea mai predictibilă), fie să preia predicția generată de predictorul „copil” cel mai potrivit. Cu toate că selecția (decizia) poate fi făcută în **paralel** de către toate nodurile, predicția globală (a arborelui) se obține din combinarea deciziei tuturor nodurilor pentru a selecta o „cale” prin arbore – practic un proces **secvențial** în adâncimea arborelui. După stabilirea deciziei la nivelul fiecărui nod, informația circulă în sens invers, dinspre frunze spre rădăcină, pe o singură cale – anterior

stabilită, **pentru a genera predicția întregului arbore. Timpul de execuție** a acestei operațiuni **este proporțional cu adâncimea arborelui.**

Pentru evitarea complexității structurii de predicție, o parte din mecanismele implementate la nivelul fiecărui nod (determinarea contorului de corelație maxim, folosirea multiplexorului pentru selecția caracteristicii) sunt partajate între mai multe noduri (întrucât se alege o singură cale de la frunze la rădăcină) și de asemenea, partajează caracteristicile de intrare. **Logica pentru determinarea contorului de corelație maxim trebuie multiplicată astfel cu adâncimea arborelui de decizie.**

În continuare este realizată o estimare a dimensiunii celor două tabele hardware, ca o funcție de adâncimea arborelui (fie d), numărul de caracteristici de intrare monitorizate (fie n) și numărul de biți pe care sunt reprezentate contoarele de corelație (fie b). Astfel, dimensiunea fiecărei intrări în tabela de corelație este dată de formula 5.4:

$$\begin{aligned} \text{SizeOfCTentry} &= \text{nr_frunze} \times \text{nr_biți_per_frunză} + \\ &\text{nr_noduri_interne} \times \text{nr_biți_per_nod_intern} = \\ &2^d \times [\text{nr_contoare_per_frunză} \times b] + (2^d - 1) \times [\text{nr_contoare_per_nod_intern} \times b] = \\ &2^d \times [(n+1) \times b] + (2^d - 1) \times [(n+2) \times b] \quad (5.4) \end{aligned}$$

În estimarea făcută s-a ținut cont că în nodurile terminale există $(n+1)$ contoare de corelație aferent caracteristicilor f_c, f_1, \dots, f_n , iar nodurile interne mai adaugă câte un contor pentru caracteristica f_{sub} – predicția subarborelui copil.

Dimensiunea fiecărei intrări în tabela de predicție este dată de formula:

$$\begin{aligned} \text{SizeOfPTentry} &= \text{nr_frunze} \times \text{nr_biți_per_frunză} + \\ &\text{nr_noduri_interne} \times \text{nr_biți_per_nod_intern} = \\ &2^d \times \text{nr_biți_info_sumare_per_frunză} + (2^d - 1) \times \\ &\text{nr_biți_info_sumare_per_nod_intern} = 2^d \times (n+2) + (2^d - 1) \times (n+4) \quad (5.5) \end{aligned}$$

În această ultimă estimare s-a ținut cont că fiecare informație privitoare la frunze din tabela de predicție conține pe lângă caracteristicile de intrare f_1, \dots, f_n și un bit care indică dacă va fi folosită caracteristica constantă f_c sau caracteristica cea mai predictibilă f_{split} , precum și un bit pentru semnul caracteristicii alese. Se observă că dimensiunea unei intrări în tabela de predicție este mai mică decât dimensiunea unei intrări în tabela de corelații cu un factor egal cu b , favorizând timpi de acces foarte rapizi, esențiali în faza **critică** de predicție.

Cu toate că articolul lui Fern [Fern03] este criticabil, întrucât nu prezintă care ar fi limita superioară a adâncimii arborelui de decizie pentru ca ideea să rămână fezabilă, și de asemenea nu expune câștigul global de performanță al unei microarhitecturi speculative care înglobează predictorul

hardware bazat pe arbori de decizie față de una standard, articolul este remarcabil prin faptul că oferă perspectiva utilizării în procesul de predicție a unui bogat număr de caracteristici (context amplu) la un cost hardware liniar. Rămâne însă de explorat dacă latența de predicție este comparabilă cu cea a predictoarelor moderne utilizate în implementările comerciale.

Dacă mecanismul de predicție propus de Fern [Fern03] se bazează pe contoare asociate fiecărei caracteristici de intrare pentru a determina caracteristica optimă după care se face predicția la nivelul fiecărui nod, în [Des04a] se propune folosirea unei metrici (indexul *Gini*) pentru a determina **caracteristicile** – individuale sau pattern-uri – (informații statice sau dinamice aferente instrucțiunilor de salt condiționat) **cu bune capacități discriminatorii**, utile apoi în procesul de predicție bazat pe arbori de decizie. Indexul *Gini* pentru un set de date S se definește astfel:

$$Gini(S) = 1 - \sum_{i=0}^{c-1} p_i^2 \quad (5.6)$$

unde,

c – numărul de clase posibile,

C_i – clasa pentru $i=0, \dots, c-1$,

s_i – numărul de exemple în C_i ,

$p_i = \frac{s_i}{S}$ frecvența relativă a clasei i în setul S

Suma pătratelor probabilităților $\sum_{i=0}^{c-1} p_i^2$ este cunoscută în literatură

sub numele de **energie informațională** – noțiune introdusă de matematicianul român Octav Onicescu pentru aprecierea calității, capacității și eficienței unui mediu informațional [Oni66].

Practic metrica *Gini* indică *puritatea* (omogenitatea) partițiilor în setul de date S . În cazul predicției salturilor se disting două clase (corespunzătoare salturilor *care se fac*, respectiv *care nu se fac*); rezultă că indexul *Gini* ia valori în intervalul $[0, 0.5]$. Dacă toate datele din S aparțin aceleiași clase, *Gini*(S) ia valoarea minimă – 0, ceea ce înseamnă că toate salturile care formează setul S au comportament similar: *taken* sau *nottaken* – clase *pure*. Dacă *Gini*(S) este egal cu 0.5, salturile sunt distribuite uniform între clasa salturilor care se fac, respectiv clasa salturilor care nu se fac. Cu cât valoarea lui *Gini* este mai mică cu atât capacitatea sa discriminatorie este mai mare.

Calitatea de split (divizare) [Des04a] după o anumită caracteristică în k subseturi S_i se calculează ca o medie ponderată a indicilor Gini aferenți subseturilor:

$$Gini_{split} = \sum_{i=0}^{k-1} \frac{n_i}{n} Gini(S_i) \quad (5.7)$$

unde,

n_i este numărul de exemple din subșetul S_i

$$n = \sum_{i=0}^{k-1} n_i \quad (5.8)$$

În cercetările sale Desmet calculează metrica Gini pe baza unor prealabile execuții a programelor de test folosind o serie de predictive clasice din literatură (bimodal, GAg, PAg, Gshare) [Bur97], furnizând la ieșire statistici laborioase referitoare la comportamentul salturilor (Taken / NotTaken) pentru fiecare din pattern-urile de caracteristici apărute.

Pentru o mai bună înțelegere, în continuare se prezintă detaliat metodologia de calcul a metricii *Gini*. Considerându-se un *pattern de caracteristici* alcătuit din 10 biți de istorie globală rezultă un număr de $k=2^{10}=1024$ *seturi* distincte de caracteristici (situații ce pot să apară în execuția programelor). Astfel, făcând referire la formula 5.7 rezultă că:

n = numărul total de instrucțiuni dinamice de salt din benchmark-ul analizat

$i \in \{0 \div 1023\}$ – indicele subșetului

n_i = numărul de salturi care apar în “*contextul*” de istorie globală având codificarea binară valoarea i . Întrucât este posibil ca unele contexte din totalul de 1024 să nu apară pe parcursul execuției programului de test atunci valorile n_i corespondente sunt nule. Pentru restul de pattern-uri (la care $n_i \neq 0$) se calculează

$$Gini(S_i) = 1 - p_{i_1}^2 - p_{i_2}^2 \quad \text{unde } p_{i_1} = \text{proporția de instrucțiuni}$$

Taken apărute în respectivul context (*pattern*) respectiv $p_{i_2} =$

procentajul de instrucțiuni *NotTaken* apărute în același context.

În final se determină $Gini_{split}$ ca fiind suma ponderată a indicilor $Gini(S_i)$ aferenți tuturor subșeturile care apar, conform formulei 5.7.

Funcția $Gini_{split}$ este calculată pentru toate caracteristicile posibile (folosind un singur bit de istorie, apoi 2 biți, șamd), iar caracteristica având $Gini_{split}$ minim este aleasă ca fiind cea după care se poate face predicția în nodurile arborelui de decizie [Des04a].

În implementările sale Desmet analizează mai multe tipuri de caracteristici: dinamice – istoria globală, locală, o dispersie a adresei salturilor cu istoria globală și respectiv statice – direcția saltului (înainte / înapoi), tipul saltului (bne, bge, blt, etc.). Rezultatele obținute, din punct de

vedere al indicelui Gini, pentru istoria locală sunt *superioare* celor calculate pentru restul tipurilor de caracteristici, oarecum firesc întrucât istoria locală este reținută per branch. Relativ la acest tip de istorie, un număr redus de biți determină o capacitate de discriminare ridicată, adăugarea de biți suplimentari neducând la o îmbunătățire a indicelui Gini. În ce privește însă celelalte caracteristici, o creștere a numărului de biți conduce la un Gini mai performant (cu o capacitate discriminatorie mai bună). În fine, o ultimă concluzie desprinsă din [Des04a] exprimă faptul că schemele de predicție care folosesc ca intrări caracteristicile amintite anterior având valori foarte mici pentru indicele Gini sunt caracterizate de acurateți ridicate de predicție, dovedind că respectiva metrică constituie un bun indicator pentru studiul caracteristicilor de predictibilitate ale salturilor.

O idee relativ interesantă o reprezintă ***utilizarea arborilor de decizie pentru îmbunătățirea predicției statice a branch-urilor***. În [Des04], extinzând ideea originală a lui Ball și Larus [Ball93] de predicție statică bazată pe euristici, Desmet arată cum arborii de decizie pot fi folosiți pentru a îmbunătăți respectivul mecanism de predicție, pe două căi: determinând în mod automat o ordine optimă de aplicare a euristicilor, și găsind în mod automat euristici suplimentare.

Practic, având un salt caracterizat de un vector de caracteristici, se urmărește determinarea cărei clase aparține respectivul salt: clasa *taken* (clasa salturilor care se fac) sau clasa *not taken* (clasa salturilor care nu se fac), bazat pe clasificarea anterioară a unor salturi similare din setul de antrenament. Arborii de decizie pot fi folosiți pentru determinarea apartenenței salturilor la o clasa sau alta.

Algoritmul lui Ball și Larus [Ball93] folosește în cadrul euristicilor lor informații privitoare la opcode-ul salturilor, operanzi ai acestora testați în condiția de salt precum și caracteristici ale *basic block*-urilor succesoare instrucțiunii de salt supusă predicției. Respectivul algoritm poate fi îmbunătățit prin utilizarea arborilor de decizie pentru clasificarea salturilor, optimizând secvența de aplicare a euristicilor și adăugând două noi informații euristice: una bazată pe relația de post-dominare dintre *basic-block*-ul curent și succesorul lui, și respectiv, una bazată pe distanța dintre instrucțiunea de salt și cea de care aceasta depinde (instrucțiunea care definește operandul testat în opcode-ul respectivului salt) [Des04].

Modelul propus de Ball și Larus[2] începe prin a clasifica salturile ca fiind salturi de tip *loop*, respectiv *non-loop*. Spre exemplu, instrucțiunile repetitive de tip *for*, *while* sunt prezise foarte bine de euristica de tip *loop*, predicția fiind că saltul (înapoi) se face (se reia bucla, iar condiția de ieșire

forțată din buclă nu se face). În continuare sunt prezentate euristicele se ocupă de salturile *non-loop*:

- ✓ euristica *pointer* – prezice că, în majoritatea cazurilor, pointerii sunt nenuli, și că doi pointeri sunt egali doar ocazional; astfel, dacă un branch compară un pointer cu *null*, sau doi pointeri între ei, se prezice că saltul se face pe condiția *false*.
- ✓ euristica *opcode* – pleacă de la premiza că multe programe folosesc numere negative pentru a codifica valori de eroare; astfel, un branch care testează dacă un întreg este mai mic sau egal cu zero, este prezis *taken* (saltul se face) pe condiție *false*.
- ✓ euristica de tip *început de bucla* (Loop Header) – prezice că buclele sunt mai degrabă executate decât evitate.
- ✓ euristica *call* – prezice că un salt mai curând evită un apel de funcție decât să-l execute. Deși surprinzător, un basic block succesori unui branch, care nu este *post-dominant* și care conține un *Call*, are tendința de a nu fi executat. Apelurile de cele mai multe ori sunt necondiționate și apar din funcții de bibliotecă (*printf*, *qsort*, *DLL*-uri).
- ✓ euristica *store* – prezice că saltul nu se face dacă blocul succesori care nu este *post-dominant* conține o instrucțiune de tip *Store*. Introdusă mai mult din curiozitate [Ball93], respectiva euristica și-a dovedit utilitatea mai ales în cazul benchmark-urilor în virgulă mobilă.
- ✓ euristica *return* – prezice că un succesori care conține un *Return* nu va fi executat. S-a observat că aceste basic block-uri sunt fie reveniri forțate din recursivitate, condiții de eroare mai rar întâlnite, întreruperi sau excepții și atunci de cele mai multe ori este mai probabil să nu se ajungă în aceste basic bloc-uri.

O metrică folosită în evaluarea performanței fiecărei euristici o reprezintă *acoperirea* (*coverage*). Aceasta constituie numărul de salturi dinamice cărora li se aplică respectiva euristica. Tabelul următor prezintă *acoperirea* și *rata de miss* aferente fiecărei euristici obținute de Ball în simulările proprii realizate pe o arhitectură MIPS [Ball93]:

Euristica	Acoperirea	Rata de miss
Loop	35%	12%
Pointer	21%	40%
Opcode	9%	16%
Loop Header	26%	25%
Call	22%	22%
Return	22%	28%
Store	25%	45%

Tabelul 5.2. Statistici privind performanța euristicilor propuse de Ball și Larus [Ball93]

În cazul în care se pot aplica mai multe euristici, apare o problema când predicțiile generate de respectivele euristici nu coincid. O ordonare în aplicarea euristicilor rezolvă această problemă: de îndată ce o euristică se aplică, saltul este prezis de către aceasta și toate celelalte euristici care mai pot fi aplicate se ignoră. Dacă nu se aplică nici o euristică, se face o predicție aleatoare. Ordonarea euristicilor poate avea un impact important asupra ratei de miss globală. Ball și Larus au determinat următoarea ordine fixă de aplicare a euristicilor: *Loop* → *Pointer* → *Call* → *Opcode* → *Return* → *Store* → *LoopHeader* [Ball93]. Pentru identificarea ordinii optime, s-au evaluat toate combinațiile posibile, procedeu însă mare consumator de timp. În plus, *chiar dacă se determină ordinea optimă pentru o anumită arhitectură a setului de instrucțiuni (ISA), compiler și limbaj de programare, nu înseamnă că pe o altă ISA ordinea optimă este aceeași* [Ball93, Flo03a]. Pentru a determina ordinea optimă în [Des04] se propune folosirea arborilor de decizie.

Este necesar un proces de învățare al arborilor de decizie. Pentru aceasta, Desmet [Des04] propune utilizarea unui instrument software creat de J. Quinlan [Qui93], numit *C4.5* – o aplicație de „*machine learning*” scrisă în limbajul C pentru Linux (aproximativ 8800 linii). Algoritmul folosit de *C4.5* este următorul: se pornește de la un set mare de cazuri (de antrenament) care aparțin unor clase cunoscute. În clase, care sunt descrise de anumite proprietăți, sunt căutate patern-uri care permit separarea lor. Clasele sunt exprimate apoi ca modele, sub formă de arbori de decizie sau seturi de reguli *if-then*, care pot fi folosite să clasifice noi cazuri, cu scopul de a face modelele inteligibile și precise. Sistemul a fost aplicat cu succes în procese care implica zeci de mii de cazuri, descrise de sute de proprietăți [Qui93]. Ca intrare în acest proces, se propune *evaluarea fiecărei euristici pentru fiecare salt static, împreună cu direcția cel mai des urmată de acesta (T/NT)* [Des04]. Aplicând apoi algoritmul de învățare *C4.5* asupra acestui set de date, se obține următoarea ordine: *Loop* → *Opcode* → *Call* → *Return* → *LoopHeader* → *Store* → *Pointer*. Când nici o euristică nu poate fi aplicată, se alege calea *NotTaken*. Noua ordine e superioară cu 2.5% față de ordinea inițială propusă de Ball și Larus [Ball93]. O euristică neamintită aici (*Guard*) și care avea cea mai mică prioritate în schema propusă de Ball este complet ignorată de arborele de decizie [Des04].

Având o metodă automată de alegere a euristicilor, se pot investiga dacă euristici suplimentare ar ajuta la îmbunătățirea acurateții de predicție. În acest sens, s-au evaluat câteva noi elemente de informație disponibile la nivelul structurii programului. În urma acestei evaluări a rezultat că există 2 euristici care, adăugate la setul propus anterior pot crește acuratețea de predicție. Prima dintre acestea privește relația de post-dominare dintre basic

block-ul succesor și respectiv cel curent. Cel mai simplu exemplu în care această euristică „*predict non-postdominating-successor*” se aplică este o instrucțiune *if*, fără un bloc *else*; în acest caz, euristica prezice că instrucțiunile aflate în blocul *if* se vor executa. A doua euristică este bazată pe distanța dintre un branch și instrucțiunea producătoare a operandului testat în condiția de salt – numărul de instrucțiuni dintre generarea valorii unui registru și folosirea acesteia de către un branch dependent. Euristica bazată pe *distanța de dependență* susține că o distanță mică se manifestă printr-o probabilitate mai mare a saltului de a fi făcut [Des04]. Figura următoare ilustrează arborele de decizie final pentru setul extins de euristici.

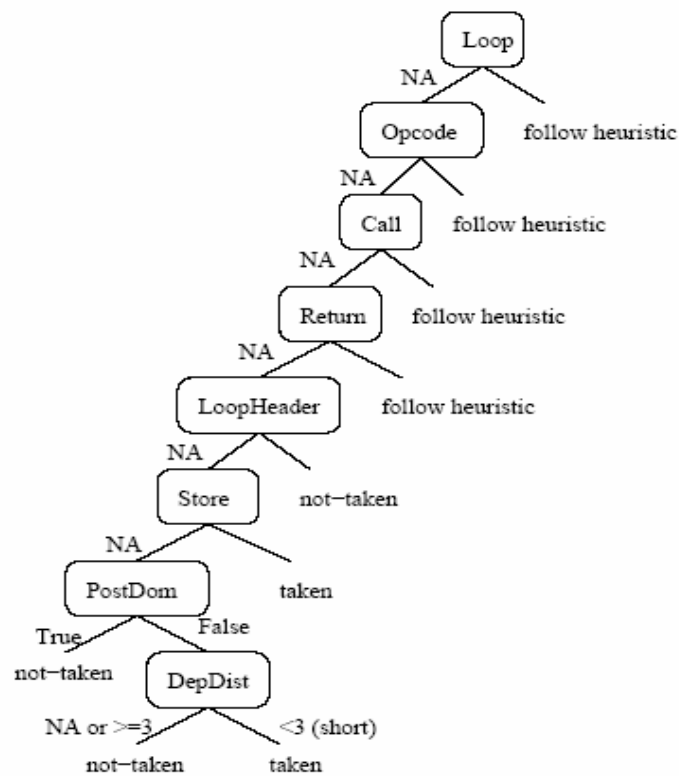


Figura 5.23. Arborele de decizie final, pentru setul extins de euristici [Des04]

Adăugând euristica „*predict non-postdominating successor*” metrica IPM (număr mediu de instrucțiuni per branch greșit predicționat) crește de la 31.3 la 34.7; euristica bazată pe distanța de dependență îmbunătățește și mai mult IPM-ul, până la 37.1 ceea ce înseamnă de fapt o creștere de 18.5% peste modelul propus de Ball și Larus [Des04].

5.2.3. PREDICTOARE CASCADATE PE DOUĂ SAU MAI MULTE NIVELURI.

Predictoarele **cascadate pe două niveluri** [Dri98b,c] (vezi figura 5.24) clasifică dinamic salturile / apelurile indirecte prin observarea comportamentului acestora pe un *predictor simplu de prim nivel* (BTB [Dri98b], LastValue [Flo03]). Doar când acest predictor eșuează în a predicționa salturile corect, un predictor mai puternic de pe al doilea nivel permite stocarea informației de predicție (target-ului) pentru pattern-urile noi de istorie ale aceluiași branch. Prin faptul că previne ocuparea (inutilă) a spațiului în tabela de target-uri de pe nivelul 2 pentru salturile ușor de prezis, predictorul de pe primul nivel funcționează practic ca un *filtru*. Astfel, tabela de target-uri de pe nivelul 2 nu se saturează la fel de repede ca și în cazul în care n-ar exista filtru, reducându-se practic miss-urile de capacitate și crescând implicit acuratețea predicției. Spațiul tabelii de target-uri aferent predictorului de pe al doilea nivel este utilizat doar în procesul de predicție al salturilor care necesită reținerea unui pattern de istorie (polimorfe). Figura 5.24 descrie modul de funcționare (predicție și actualizare) al unui predictor cascadat.

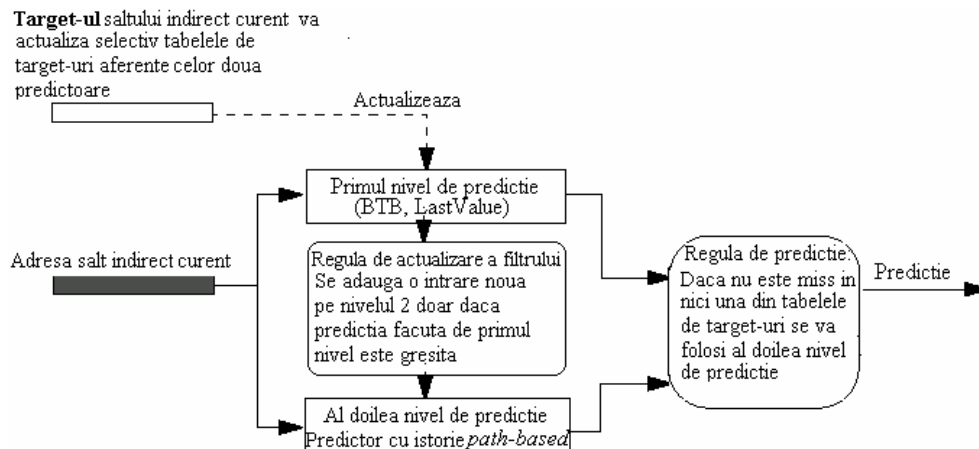


Figura 5.24. Structură de predicție în cascadă pe două niveluri [Dri98c]

Dacă ambele predictoare ar furniza o predicție (nu există miss în nici una din cele două tabelle de target-uri) atunci se va utiliza valoarea generată de către predictorul *path-based* (cel de pe nivelul 2), motiv pentru care tabela de target-uri aferentă acestuia trebuie să conțină un câmp de *tag* necesar la detectarea acceselor cu miss.

Diferența principală dintre predictoarele hibride și cele cascade constă în modul de actualizare al tabelelor de target-uri aferente monopredictoarelor componente. În cazul predictoarelor cascade, dacă structura de pe primul nivel generează o predicție corectă, atunci predictorului de pe nivelul 2 nu îi este permis să insereze în tabela sa de target-uri o (nouă) intrare pentru acest salt. Cu toate acestea, dacă intrarea respectivă este deja prezentă în tabela de target-uri atunci target-ul va fi actualizat. În schimb, tabela predictorului de pe primul nivel va fi întotdeauna actualizată.

În experimentul propus de Driesen [Dri98c] predictorul de pe primul nivel este un BTB (fără istorie practic și suficient pentru predicția salturilor monomorfe), în timp ce al doilea nivel, necesar în cazul salturilor polimorfe este constituit dintr-un predictor *path-based*, optimul determinat în simulările anterioare [Dri98a] din punct de vedere al lungimii pattern-ului și al gradului de asociativitate. Au fost studiate două tipuri de predictoare cascade, în funcție de regula de actualizare folosită în implementarea efectului de filtrare:

- **Cu filtrare strictă (etansă)** – care permit predicția branch-urilor cu structura de pe nivelul 2 doar când predictorul de pe primul nivel a prezis greșit (nu la miss-uri de start rece). Altfel spus, salturile indirecte avansează spre nivelul 2 doar dacă se dovedesc a fi polimorfe. Predictoarele cascade cu filtrare strictă devin eficiente abia pentru tabele de target-uri, aferente primului nivel, de dimensiuni ridicate. Pentru capacități mai mici sau egale cu 16 intrări acuratețea predicției este inferioară celei obținute cu un predictor cascade fără filtrare (salturile sunt inserate în ambele predictoare iar la hit în ambele, prezice cel de pe nivelul 2) în aceleași condiții de cost hardware. În cazul filtrării stricte, predictorul de pe nivelul 1 recunoaște branch-urile polimorfe doar dacă acestea rămân suficient timp în tabela de target-uri pentru a fi sesizată o modificare de target-uri (și astfel o predicție greșită). Dar pentru tabele de capacități reduse, multe salturi polimorfe sunt înlocuite înainte de a apărea o predicție greșită (target-ul nu coincide) și astfel ele nu vor putea accesa niciodată predictorul de pe al doilea nivel și nu vor fi inserate în tabela de target-uri aferente acestuia. Evident că, odată cu creșterea capacității tablei de pe primul nivel vor fi evitate majoritatea miss-urilor de capacitate, efectul de filtrare devine eficient conducând la creșterea acurateții de predicție. Simulări efectuate pentru 3 predictoare de nivel 2 de capacități diferite (256, 1k, 4k) intrări, au demonstrat că filtrarea strictă generează rezultate mai bune comparativ cu un predictor

cascatat fără filtrare, pentru tabele de target-uri aferente primului nivel de dimensiuni mai mari sau egale cu 256 intrări.

- **Cu filtrare mai puțin etanșă (*leaky*)** – care permite inserarea salturilor, la miss în tabela de target-uri de pe primul nivel, în tabelele ambelor structuri de predicție. Astfel, tabela de pe nivelul 2 poate conține intrări aferente salturilor monomorfe, dar doar dacă acestea nu se regăsesc în BTB (pe primul nivel). Pentru a preveni problemele legate de miss-urile de capacitate apărute în cazul predictoarelor cascadate cu filtrare strictă, filtrele mai puțin etanșe (*leaky*) inserează câte o intrare în tabelele de target-uri aferente ambelor predictoare. Fiecare salt este introdus cel puțin o dată în predictorul de pe nivelul 2. Salturile corect predicționate sunt reținute prin filtrare astfel încât o nouă inserare în tabela de pe nivelul 2 va putea avea loc doar la un miss în primul nivel (fie saltul a fost evacuat din tabela de target-uri fie target-ul diferă posibil și datorită faptului că s-a ajuns la salt pe o altă *cale*).

Spre deosebire de filtrarea strictă, filtrarea mai puțin etanșă furnizează rezultate satisfăcătoare din punct de vedere al acurateții predicției pentru toate dimensiunile de tabele de target-uri aferente primului nivel. Predictoarele cascadate cu filtrare mai puțin etanșă ating acuratețile de predicție obținute prin schemele anterior descrise (BTB, Two-Level, hibride) dar la jumătate cost hardware (tabele de target-uri de dimensiuni înjumătățite) [Dri98b,c]. În timp ce predictoarele necascadate necesită tabele de target-uri de cel puțin 1024 de intrări pentru a atinge acurateți de predicție de 91%, un predictor cascatat cu filtrare mai puțin etanșă compus dintr-o tabelă BTB cu 32 intrări pe primul nivel și un predictor hibrid dual-path de 512 intrări situat pe nivelul 2 obține o performanță echivalentă. Pentru un filtru BTB cu 64 intrări, 4 – way asociativ și un predictor adaptiv pe două niveluri situat pe nivelul 2 de 1024 intrări acuratețea predicției este de 92.2%, aproximativ aceeași performanță obținută cu un monopredictor, necascadat, two-level, de capacitate de 4 ori mai mare. Prin combinarea aceluiași tip de filtru cu un predictor de pe nivelul 2 dual-path cu 1024 intrări acuratețea predicției crește până la 92.7% superioară unui predictor necascadat hibrid dual-path cu 2048 intrări.

- Predictoarele **casadate pe mai multe niveluri** ($n \geq 3$) au fost introduse de Driesen [Dri99] și reprezintă o generalizare a schemei de predicție în cascadă pe două niveluri [Dri98b] (figura 5.24). Fiecare nivel este constituit dintr-un predictor adaptiv de tip *Two-Level* [Dri98a] cu tabelă de predicție și istorie proprie. Pe măsură ce se adâncește nivelul ($n \nearrow$) corelația cu celelalte salturi indirecte crește (*path-ul* \nearrow) – vezi figura

5.25. Prin folosirea separată a resurselor este permisă predicția în paralel a fiecărui nivel (predictor aferent). În caz de hit în toate tabelele de predicție va fi *folosit speculativ* target-ul generat de structura cu „*calea*” cea mai lungă (istoria cea mai bogată) reținută. Analizând după rezultatele exprimate de cercetători la ora actuală în ceea ce privește acuratețea predicției salturilor indirecte, un predictor cascadat pe 3 niveluri (vezi figura 5.25) generează un maxim de predicție de 94.8% folosind o tabelă de predicție cu 4096 intrări, iar cu o tabelă de doar 1536 intrări acuratețea obținută este de 94%, maximum obținut de o schemă ipotetică de predicție adaptivă corelată pe două niveluri, nelimitată ca și capacitate. Predictoarele cascadeate multinivel constituie o variantă fezabilă din punct de vedere hardware (în contextul unui buget limitat de tranzistori asigură o acuratețe de predicție superioară schemelor adaptive corelate pe două niveluri la un sfert din costul de implementare al acestora) [Dri99].

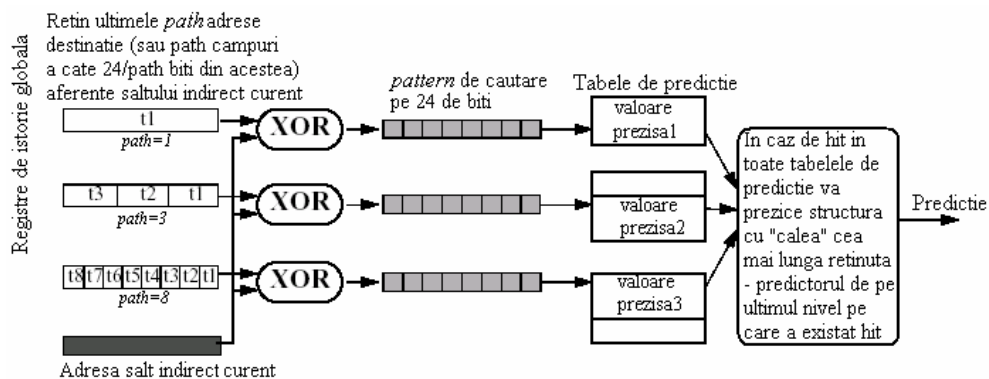


Figura 5.25. Predictor cascadat pe trei niveluri: componentele sunt predictoare *path-based* cu *path* diferit (1, 3, 8) [Dri99]

Un predictor cascadat multinivel care are drept componente, pe fiecare nivel, predictoare adaptive de tip two-level cu *path*-uri de căutare succesive începând de la 0 se numește predictor cascadat complet [Dri99]. Una din concluziile generate în urma simulărilor laborioase [Dri99] a fost că predictoarele cascadeate sunt superioare din punct de vedere al acurateții predicției celor adaptive pe două niveluri (*two-level*), în aceleași condiții de cost hardware, indiferent de dimensiunea istoriei păstrate. În studiul efectuat Driesen a verificat și fezabilitatea implementării hardware a predictoarelor cascadeate multinivel. Pornind de la structuri ideale de predicție (tabele de capacitate nelimitate) sunt adăugate pe rând constrângeri privitoare la capacitățile tabelor, grade reduse de asociativitate și respectiv un număr limitat (optim) de niveluri de predicție. Pentru acuratețea maximă de

predicție (94.2%) obținută de un predictor ideal de tip two-level folosind o istorie de 6 target-uri anterioare, este necesară utilizarea unei tabele de predicție care să rețină până la 13210 de pattern-uri aferente salturilor indirecte supuse predicției. Cum stocarea fiecărui pattern poate fi făcută pe aproximativ 60 de biți (24 de biți tag-ul, 32 de biți adresa destinație – vezi figura 5.25) costul total se ridică la 100k octeți de memorie [Dri99], prea mult pentru capacitățile tehnologice actuale. Astfel, eforturile cercetătorilor au fost canalizate pe reducerea costului memoriei, timp rezonabili de căutare în tabele – grade de asociativitate limitate care să permită implementarea în Siliciu dar păstrarea acurateții de predicție la valori cât mai apropiate de 100%.

Așa cum s-a precizat anterior, tabelele de predicție aferente fiecărui nivel sunt separate în caz contrar fiind necesară suportarea unui număr ridicat de accese simultan (complexitate sporită). Se pune problema: *cum este mai eficient să fie folosite mai multe niveluri cu tabele de capacități reduse sau mai puține niveluri cu tabele de dimensiuni mai mari*? În prima variantă rezultă un număr crescut de miss-uri de capacitate care, similar cu predictorul cascadat pe două niveluri cu regulă *strictă* de filtrare și tabele de dimensiuni reduse, diminuează din eficiența filtrării pattern-urilor (întrucât doar pattern-urile corect predicționate nu ocupă spațiul tabelor aflate pe nivelurile superioare) ocupându-se de multe ori inutil locațiile nivelurilor ulterioare chiar și pentru salturi monomorfe sau duomorfe, dar evacuate prematur (înainte de a fi accesate cu hit din nou) din tabela de pe primul nivel. Această presupunere s-a concretizat în practică întrucât rezultatele simulărilor lui Driesen au arătat că un predictor cascadat complet cu 9 niveluri (*path*-uri de lungime de la 0 până la 8) cu tabele de 128 intrări per nivel (buget total de 1152 intrări) atinge o acuratețe de predicție de 92.3% în timp ce un predictor cascadat cu doar două niveluri (*path*-uri de lungime 1 și 8), cu tabele de 512 intrări per nivel (buget total 1024 intrări) obține o acuratețe de 93.2% [Dri99]. Concluzia cercetătorilor a fost că, în condițiile unui buget limitat de intrări ale tabelii de predicție, un predictor cascadat multinivel cu un număr de niveluri între 3 și 8 (probabil mai apropiat de limita inferioară) se dovedește a fi optim din punct de vedere al acurateții de predicție și superior celorlalte structuri dezvoltate în literatura de specialitate în aceleași condiții de complexitate hardware. Diferența dintre un predictor cascadat cu două și respectiv unul cu trei niveluri este destul de mică: acuratețea unui predictor cu trei niveluri având un buget total de intrări egal cu $3 \cdot X$ este egală sau ușor superioară celei obținute de predictorul cascadat pe două niveluri având un buget total de $4 \cdot X$ intrări, pentru $X \geq 512$. Cu toate acestea, câștigul de acuratețe obținut pare să nu justifice complexitatea

hardware suplimentară introdusă de încă un nivel de predicție. O altă concluzie certă ar fi că bugetul total de intrări să fie împărțit echitabil (în părți egale) între tabelele aferente fiecărui nivel.

În cadrul predictoarelor pe mai multe niveluri ar putea fi încadrat și predictorul PPM (*partial prefix matching*) complet, implementat de Kalamatianos [Kal98], cu rezultate optime din punct de vedere al acurateții predicției de 90.53%. Predictorul PPM complet cuprinde $N+1$ predictoare Markov (contextuale) de la ordinul 0 la ordinul N . Ordinul fiecărui predictor reprezintă dimensiunea pattern-ului de căutare în tabelele de predicție aferente fiecărui nivel. Dacă predictorul Markov de ordinul N produce un rezultat valid atunci procesul se termină dacă nu se activează predictorul de ordin imediat inferior. În [Kal98] capacitatea tabeli de predicție de pe fiecare nivel (privind descrescător de la ordinul N la 0) este dublul celei aferente nivelului imediat inferior. Deși există unele păreri că un predictor Markov de ordin 0 nu își are sensul întrucât ultimul target al saltului dinamic nu se regăsește în istoria memorată, neexistând deci practic repetiție de context pentru a putea predicționa pe baze rezonabile ce va urma după acesta, în cercetările efectuate Kalamatianos a implementat un predictor PPM complet de ordinul 3 având drept componente predictoare Markov de ordin 3 până la 0 inclusiv [Kal98].

Cu toate că funcționarea mecanismului de predicție este asemănătoare cu regula de filtrare folosită în cadrul predictoarelor cascade multinivel, există și deosebiri între cele două structuri de predicție. Prima dintre acestea se referă la faptul că predictoarele cascade folosesc buffer-e de istorie a salturilor diferite, câte unul aferent fiecărui nivel. Numărul de niveluri este independent de lungimea pattern-ului de căutare iar tabelele aferente fiecărui nivel pot avea orice capacitate. Îmbunătățirea performanței obținută prin predictorul PPM complet față de scheme clasice (BTB, Two-Level) poate fi datorată și modului de clasificare dinamică a salturilor indirecte. Astfel, într-o clasă se regăsesc salturile care sunt într-o mai bună corelație atât cu branch-urile condiționate cât și cu target-urile salturilor indirecte iar în a doua clasă sunt clasificate salturile aflate în corelație doar cu target-urile salturilor indirecte anterioare. Predictoarele cascade multinivel păstrează în registrul de istorie globală a salturilor doar adresele destinație aferente salturilor indirecte anterioare (vezi figura 5.25).

O concluzie ce se desprinde atât din cercetările efectuate de Driesen cât și ale altor cercetători [Kal98] este că „*lupta este extrem de aprigă, la nivel de procent*” în încercarea de a predicționa cu acuratețe salturile indirecte, în sensul că fiecare câștig potențial (introdus de o nouă tehnică sau

îmbunătățiri aduse structurii de predicție), oricât de mic (1, 2%) trebuie exploatat. Dacă adăugăm la acestea și faptul că, într-un procesor superscalar care poate lansa simultan 4 instrucțiuni independente în execuție, o predicție greșită a salturilor de doar 2.3% față de ideal conduce la o diminuare a performanței procesorului cu până la 40% [Vin00], rezultă și mai mult importanța fiecărui procent suplimentar de predicție corectă și necesitatea stringentă de a găsi și implementa mecanisme de predicție eficiente – cu acurateți de aproape 100%, reducându-se astfel din efectul defavorabil al ramificațiilor de program asupra performanței procesoarelor avansate. O critică însă, ce poate fi adusă activității lui Driesen constă în faptul că, rezultatele sale remarcabile în ceea ce privește acuratețea predicției salturilor / apelurilor indirecte și respectiv costul hardware al implementării schemelor de predicție propuse, nu sunt dublate de informații legate de timpul total de execuție, rată de procesare și speed-up-ul obținut prin implementarea predictoarelor cascade și hibride. Bazat pe rezultatele excelente obținute [Dri98a,b] se impune introducerea și exploatarea predictoarelor cascade și în alte domenii de cercetare în vederea creșterii paralelismului la nivelul instrucțiunilor și a firelor de execuție: *predicția salturilor condiționate și predicția valorilor instrucțiunilor*.

5.3. CERCETĂRI PROPRII PRIVITOARE LA PREDICȚIA SALTURILOR / APELURILOR INDIRECTE.

5.3.1. PREDICTORUL PPM COMPLET.

Multe din predictoarele existente, deși capabile să prezică cu acuratețe direcția salturilor condiționate (BTB, adaptive corelate pe două nivele – GAg, PAg) până la 97,7% [Yeh92], sunt ineficiente în determinarea corectă a „*target-urilor*” salturilor în mod de adresare indirectă (datorită modificării adreselor destinație cu fiecare instanță dinamică a aceluiași salt static). Acuratețea predicției este deosebit de scăzută la ora actuală (cca. 75.1%) [Dri98]. În [Cha97] se arată bazat pe simulări laborioase că o schemă de predicție de tip *Branch Target Buffer* cu automat de predicție pe doi biți a îmbunătățit acuratețea de predicție pentru benchmark-urile SPEC'95

compress, *gcc*, *jpeg* și *perl*, în același timp înrăutățind acuratețea de predicție pentru *m88ksim*, *vortex* și *xlisp (li)*.

Trebuie specificat faptul că, exceptând situațiile când se precizează explicit, afirmațiile, simulările și rezultatele grafice se referă doar la instrucțiuni de *salt indirect pure*, nu la instrucțiuni de revenire din subrutină – *return*. Deși tehnic și acestea din urmă sunt salturi indirecte, ele pot fi tratate folosind alte structuri hardware: stive de tip CALL/RETURN. Rezultatele simulărilor pe benchmark-urile SPEC'95 au evidențiat că o pereche de stive de tip CALL/RETURN cu 5 intrări fiecare, atașată unei structuri de predicție de tip BTB reduce numărul predicțiilor incorecte cu **31.5%** față de situația când nu s-ar folosi stivele [Kae97].

Pentru determinarea acurateții de predicție în cazul instrucțiunilor de salt indirecte s-a implementat un predictor **PPM complet** (vezi figura 5.26) [Vin02, Kal98].

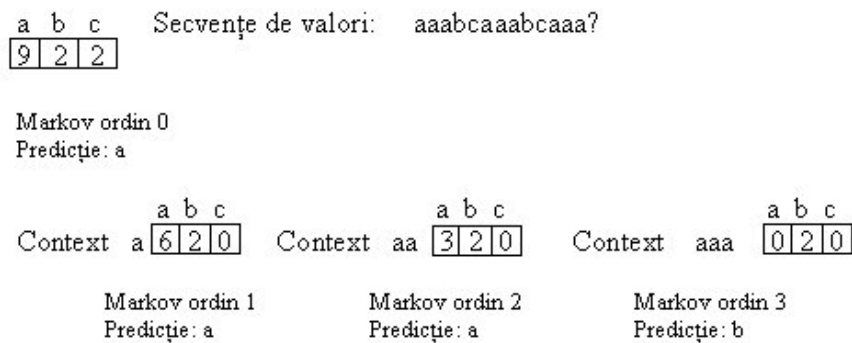


Figura 5.26 Predictor contextual PPM (*Prediction by Partial Matching*)

Valoarea predicționată este aceea care a urmat cu cea mai mare frecvență contextului considerat. După cum se observă în figura 5.26, ea este funcție și de contextul considerat, un context mai “bogat” (“lung”) conducând adeseori la o acuratețe mai ridicată a predicției (nu întotdeauna însă – vezi rezultatele simulărilor din capitolul 7.1.1 și 7.2). În exemplul considerat, abia predictorul *Markov* de ordinul 3 generează o predicție “corectă”. Dacă predictorul Markov de ordinul N produce un rezultat valid atunci procesul se termină dacă nu se activează predictorul de ordin imediat inferior. În figura 5.27 se prezintă schema bloc a unui predictor contextual de valori (Markov de ordinul k) – implementată în cadrul simulatorului *simvpred.exe* folosit la predicția target-urilor instrucțiunilor de salt indirect.

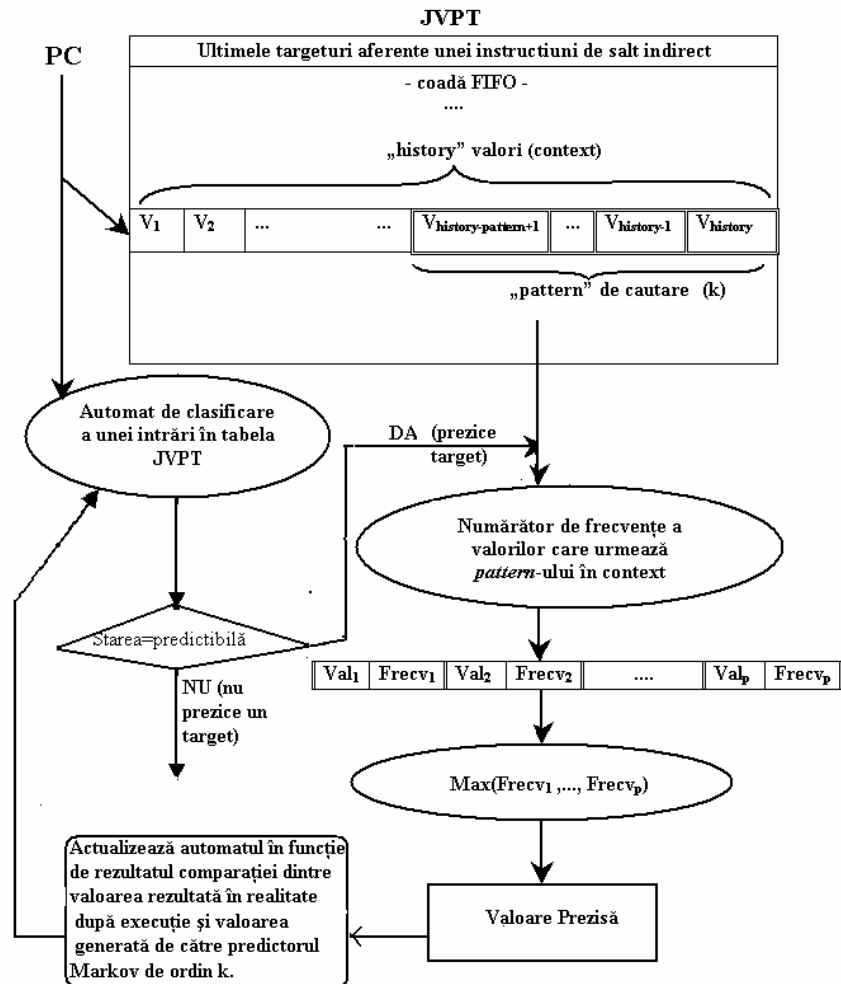


Figura 5.27. Schema unui predictor contextual de ordin k (Markov_K)

Tabela JVPT (*Indirect Jump Value Prediction Table*) – care cuprinde ultimele *history* adrese destinație aferente fiecărei instrucțiuni de salt indirect (*contextul*), este indexată cu PC-ul instrucțiunii, pe timpul fazei de aducere. Pe baza unui *pattern* de căutare (ultimele k valori din context) predictorul Markov complet de ordinul k va predicționa noul target. Se verifică de fapt, în lista de *history* valori generate de către respectiva instrucțiune de salt indirect ce valori urmează respectivului context cu o frecvență mai mare. Valoarea prezisă va fi folosită în execuție (stabilind practic noul PC) doar dacă automatul de clasificare aferent instrucțiunii de salt indirect (vezi figura 5.28) de la respectiva adresă (PC-ul curent) se va afla în starea *predictibil*. În cazul în care respectivul *pattern* a apărut în *context* cel puțin o dată, atunci lista de valori nu va fi vidă, valoarea prezisă

fiind cea cu frecvența maximă, în caz contrar fiind necesară activarea predictorului Markov de ordin $k-1$. În funcție de rezultatul comparației dintre valoarea reală (target-ul rezultat după execuție) și cea ce a predicționat predictorul contextual Markov de ordin k se va actualiza corespunzător automatul de clasificare.

Practic, schema predicționează pe baza probabilității statistice ca o anumită valoare să urmeze unui anumit context dinamic, comportându-se excelent pentru secvențe repetitive de valori.

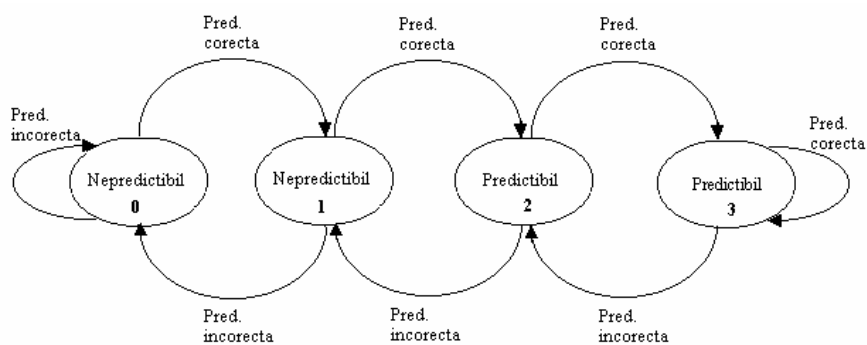


Figura 5.28. Automatul de clasificare aferent unei intrări în tabela JVPT

Simulatorul dezvoltat este de tip *execution-driven* și extinde mediul SimpleScalar 3.0 cu structuri noi de date respectiv funcții specifice vecinătății și predicției valorilor. Vecinătatea valorii pentru un benchmark este calculată ca o medie a numărului de instrucțiuni de salt indirect care vor efectua saltul la o aceeași adresă ca și una din anterioarele k instanțe ale respectivei instrucțiuni de salt și totalul de instrucțiuni de salt indirect dinamice existente în benchmark. Întrucât este necesară memorarea adresei fiecărei instrucțiuni de salt indirect (jal \$reg sau j \$reg) din acel benchmark, deci capacități mari de memorare, am folosit în implementare structuri dinamice de date (liste simplu înlănțuite). Pentru **determinarea localității valorilor** sunt folosite aceleași structuri de date pentru toate tipurile de instrucțiuni (salt indirect, ALU, Load).

Tabela de predicție este similară cu cea aferentă instrucțiunilor Load (vezi subcapitolul 6.1.1), cu mici completări.

```

typedef struct JVPTelement *JVPTvalueList;
struct JVPTelement
{
    sword_t      value;
    JVPTvalueList  nextValue;
}
  
```

```

        int                count; /*reține frecvența de apariție a
    respectivei valori după          contextul dat*/
};

typedef struct JVPTlocation *JVPTaddrList;
struct JVPTlocation
{
    md_addr_t            addr;
    JVPTaddrList        nextAddress;
    JVPTvalueList        values;
    int                  automat;
};

```

Identificarea câmpurilor din structură este următoarea:

addr – adresa (PC) instrucțiunii de salt indirect (jal \$reg sau j \$reg);
values – reprezintă lista ultimelor k valori dinamice pentru saltul respectiv (practic ultimele k target-uri aferente instrucțiunii de salt indirect);

automat – este un automat cu patru stări (vezi fig. 5.28). Inițial un salt indirect este nepredictibil dacă automatul acestuia se află în starea 0 sau 1 și este predictibil dacă automatul se află în starea 2 sau 3. În subcapitolul 5.3.2 va fi studiat comportamentul unui automat cu rol de confidență pe un număr variabil de biți (≥ 2).

Alți parametri importanți ai simulatorului care pot fi modificați de către utilizator, sunt: tipul simulării (**1**: determinarea localității sau **0**: predicția valorilor: **-pred**), “adâncimea” istoriei: **-history**, tipul tabelii de predicție (**0**: mapată direct sau **1**: asociativă: **-assoc**), dimensiunea tabelii de predicție (numărul de locații: **-jvpt**), tipul predictorului utilizat (**-contextual**; **0**: TargetCache sau **1**: *PPM complet*) și dimensiunea contextului (**-pattern**) în cazul predictorilor contextuale de tip *Markov*.

Indiferent de tipul tabelii utilizate (mapată direct sau asociativă), în cazul în care se obține un hit în tabela JVPT, se face o predicție. În funcția *predictValue* din programul *vpred.c* este implementat predictorul contextual de tip PPM (*Prediction by Partial Matching*) complet. În această funcție este folosit predictorul definit prin parametrii introduși de către utilizator. În cazul în care automatul de clasificare aferent unei intrări din tabela JVPT se află într-o stare de tip “predictibil”, se înaintează valoarea prezisă (practic noul PC) și aceasta este preluată prin *bypassing* și încărcată în registrul de adresă a următoarei instrucțiuni (PC), salturile indirecte făcându-se

necondiționat. La determinarea valorii reale a registrului care codifică adresa pentru instrucțiunea de salt indirect, în urma fazei de decodificare/execuție, aceasta este comparată cu valoarea prezisă, și instrucțiunile dependente executate speculativ fie urmează parcursul normal – până la nivelul *Write Back* al structurii *pipeline* - fie sunt retrimise spre execuție. Tabela JVPT este actualizată prin incrementarea automatului în cazul unei predicții corecte sau decrementarea acestuia în cazul unei predicții incorecte și introducerea valorii reale a registrului sursă aferent instrucțiunii de salt în noul context de la intrarea corespunzătoare din JVPT, evacuând din locația respectivă cea mai veche valoare printr-un algoritm de tip FIFO (*First In First Out*).

Exemplificăm în continuare prezentând antetele principalelor funcții ce intervin în cadrul predictorului JIndirValue: **predictValue**, **foundAssociativeJVPTAddress**, **insertJVPTValue**, **foundAddress_INDIR**, **foundValue_INDIR**. Descrierea completă a acestor funcții poate fi regăsită atât în referatul de doctorat nr. 2 [Flo03a] cât și pe CD-ul cu surse și simulatoare în format executabil ce însoțește această lucrare. Pentru început am descris nucleul de execuție al simulatorului (fazele *Fetch instrucțiune*, *Decodificare* și *Execuție* – aferente instrucțiunilor de salt indirect) rezident în rutina **sim-main** din modulul **sim-vpred.c**. Funcțiile apelate în interiorul acestei rutine au fost definite în modulul **vpred.c**.

```

/*****sim-main*****/
MD_FETCH_INST(inst, mem, regs, regs_PC); /* extragerea instrucțiunilor din
cache/memorie */
MD_SET_OPCODE(op, inst); /* decodificarea instrucțiunii */

/* execuția instrucțiunii */
switch (op){
    #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)
    \
    #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
    #define CONNECT(OP)
}

// tratarea funcțiilor indirecte pure în faza de execuție
if((MD_OP_FLAGS(op)& F_INDIRJMP)&& !MD_IS_RETURN(op))
{
    sim_indir_refs++;
    if(predict == 1) /* JIndir value prediction */
    {
        if(isAssoc) // Tabela JVPT asociativă
        {

```

```

if(!foundAssociativeJVPTAddress(regs.reg PC, regs.reg R[in1], history, &jvpt,
JVPTdim, contextual, pattern))
{
    jvpt = pushJVPTAddress(jvpt, regs.reg PC);
    insertJVPTValue(jvpt, regs.reg R[in1], history, contextual);
}
}
else // Tabela JVPT mapată direct
    insertIntoDirrectMappedJVPT(regs.reg PC, regs.reg R[in1], history, &jvpt,
contextual, JVPTdim, pattern);
}
else /* JIndir value locality verification */
{
    if(l_INDIR == NULL)
    {
        l_INDIR = pushAddress(l_INDIR, regs.reg PC, regs.reg R[in1]);
        nr_salturi_stactice_indir++;
        fprintf(stderr, "\n PC:%x op:%d %-10s in1:%d in2:%d out1: %d \n",
            regs.reg PC, op, MD_OP_NAME(op), in1, in2, out1);
    }
    else if(!foundAddress_INDIR(l_INDIR, regs.reg PC, regs.reg R[in1], history))
    {
        l_INDIR = pushAddress(l_INDIR, regs.reg PC, regs.reg R[in1]);
        nr_salturi_stactice_indir++;
        fprintf(stderr, "\n PC:%x op:%d %-10s in1:%d in2:%d out1: %d\n",
            regs.reg PC, op, MD_OP_NAME(op), in1, in2, out1);
    }
} // de la localitate
}

```

```

/*****/
sword_t predictValue(JVPTAddrList p, int history, int contextual, int pattern);

```

✓ Identifică structura de predicție corespunzătoare prin intermediul parametrilor de intrare și prezice valoarea pentru resursa în cauză (target-ul saltului indirect).

```

int foundAssociativeJVPTAddress(md_addr_t addr, sword_t value, int history,
JVPTAddrList *jvpt, int JVPTdim, int contextual, int
pattern);

```

✓ Stabilește dacă instrucțiunea curentă de salt indirect a mai fost sau nu prezisă anterior și returnează rezultatul corespunzător. În caz afirmativ verifică dacă valoarea prezisă de structură este chiar cea rezultată în urma execuției instrucțiunii de salt indirect și actualizează automatul de clasificare conform rezultatului predicției. De asemenea, noua adresă destinație este inserată în lista de target-uri cu ajutorul funcției *insertJVPTValue*. Dacă instrucțiunea respectivă nu a fost supusă predicției anterior atunci ea este inserată în structura de predicție prin intermediul funcției *pushJVPTAddress*.

void **insertJVPTValue**(JVPTAddrList **ad**, sword_t **value**, int **history**, int **contextual**);

- ✓ Inserează un target în JVPT la o adresă (PC) pe prima poziție în lista de valori. Se determină dacă lista are exact *history* valori în acest caz fiind nevoie de eliminarea ultimei valori din listă (implementarea FIFO a listei de valori – cea mai recentă valoare se află în fața listei iar cea mai demult introdusă valoare se află pe ultima poziție în listă). Dacă în listă nu există încă *history* valori nu se va elimina nici o locație din listă.

void **insertIntoDirrectMappedJVPT**(md_addr_t **addr**, sword_t **value**, int **history**, JVPTAddrList ***jpvt**, int **contextual**, int **JVPTdim**, int **pattern**);

- ✓ În cazul structurii Target Cache determină locația unde se va efectua inserarea. În caz de miss în JVPT se actualizează adresa cu valoarea corespunzătoare (noul PC), se șterge fizic lista de valori de la respectiva adresă și se inițializează automatul de predicție. În caz de hit se verifică dacă predicția este corectă (target-ul prezis coincide cu adresa reală de salt obținută în urma execuției) și se incrementează contoarele corespunzătoare. După efectuarea predicției are loc inserarea noului target în lista de valori.

int **foundAddress_INDIR**(addrList l, md_addr_t **addr**, sword_t **value**, int **history**);

- ✓ Este utilizată în determinarea gradului de localitate existent pe resursele folosite. Se verifică apariția anterioară a instrucțiunii de salt indirect. În caz afirmativ se verifică prin intermediul funcției *foundValue_INDIR* dacă în lista de target-uri se află și noua valoare (*value*). Dacă noua valoare nu a existat atunci este inserată ca și ultim target în respectiva listă cu ajutorul funcției *pushValue*.

int **foundValue_INDIR**(addrList l, sword_t **value**, int **history**);

- ✓ Funcția este apelată și pentru actualizarea statisticilor privind localitatea valorilor aferent target-urilor instrucțiunilor de salt indirect. Caută target-ul curent în lista de valori deja reținute.

Rezultatele simulării împreună cu parametrii arhitecturii vor fi scriși într-un fișier (*simout.res*) din directorul curent. Dintre rezultatele ce caracterizează localitatea valorii se amintesc:

- ▣ *sim_indir_refs* – numărul total de salturi indirecte executate dinamic dintr-un benchmark.
- ▣ *sim_num_jumps* – numărul total de instrucțiuni de salt executate.
- ▣ *sim_num_static_indir* – numărul total de salturi indirecte statice găsite într-un program sursă asamblare și librăriile aferente.
- ▣ *sim_num_static* – numărul total de branch-uri statice aflate într-un program sursă asamblare și librăriile aferente.
- ▣ *JindirValueLocality* – numărul de instanțe dinamice aferente instrucțiunilor de salt indirect care au avut ca *target*

aceeași valoare ca una din precedentele *history* instanțe dinamice ale aceluiași salt indirect static.

Rezultate ce exprimă cât de eficient a fost automatul implementat:

- *valuePrediction* – numărul total de adrese (target-uri) prezise corect de către automat: (automatul a clasificat valoarea predictibilă iar valoarea propusă de automat a fost exact cea care a rezultat în urma decodificării / execuției instrucțiunii), practic rezultată în urma predicției.
- *classifiedPred* – numărul de instrucțiuni de salt indirect clasificate predictibile de către automat.
- *classifiedUnpred* – numărul de instrucțiuni de salt indirect clasificate nepredictibile de către automat.
- *wpredicted* – numărul de instrucțiuni de salt indirect greșit predicționate (automatul a clasificat valoarea ca nepredictibilă iar valoarea propusă a fost într-adevăr diferită de cea care a rezultat în urma decodificării / execuției instrucțiunii).

Cunoscând aceste informații, se poate determina acuratețea predicției pentru arhitecturile simulate, precum și gradul de încredere al automatului de clasificare.

5.3.2. ÎMBUNĂTĂȚIREA PERFORMANȚEI PREDICTORULUI TARGET CACHE. IMPLEMENTAREA UNUI MECANISM DE CONFIDENȚĂ.

În primă fază, rezultatele obținute cu predictorul PPM au fost comparate cu cele generate de predictorul *TargetCache*, schemă propusă de Chang în 1997 (vezi figura 5.13) [Cha97].

În [Cha97] sunt descrise trei modalități de indexare ale Target Cache-ului: *HistoryXOR*, *Address* și *HistoryConcatenate* dintre care au fost implementate primele două. Atât prima schemă cât și cea de-a doua utilizează cei mai puțini semnificativi biți ai adresei instrucțiunii de salt indirect pentru identificarea setului. În timp ce la schema *Address* cei mai semnificativi biți ai adresei și istoria globală a salturilor condiționate sunt dispersați printr-o funcție XOR (sau exclusiv) formând Tag-ul emis pentru comparare, la schema *HistoryConcatenate* aceste informații sunt concatenate.

La un grad de asociativitate redus, prima schemă generează un număr semnificativ de miss-uri de conflict în TargetCache întrucât toate țintele aferente unui salt indirect static sunt mapate într-un același set. Creșterea gradului de asociativitate conduce însă la îmbunătățiri ale performanței Target Cache-ului prin reducerea unui procentaj semnificativ de miss-uri de conflict.

Pentru un target cache cu tag, numărul de biți care memorează istoria instrucțiunilor de salt nu este limitat de dimensiunea structurii întrucât biții suplimentari de istorie pot fi stocați în câmpul de tag. Utilizând o istorie bogată poate fi identificată cu ușurință apariția instrucțiunii de salt indirect în cadrul fluxului de instrucțiuni. Cu toate acestea, pentru target cache-urile cu asociativitate scăzută performanța se degradează (pierderea datorată miss-urilor de conflict depășește câștigul realizat printr-o mai bună identificare a salturilor indirecte).

Complexitatea arhitecturală a procesoarelor actuale dar și tehnologică agravează impactul negativ asupra performanței cauzat de o predicție greșită. Pentru predictoarele de capacitate redusă ($\leq 16\text{Ko}$ [Tho03]) a căror acuratețe este în principal limitată de interferențe destructive, o mică creștere în dimensiune determină o îmbunătățire substanțială a acurateții. Predictoarele globale de salturi îmbunătățesc acuratețea predicției prin corelarea comportamentului saltului curent (T / NT) cu istoria celor mai recente salturi dinamice precedente acestuia, o creștere liniară a lungimii istoriei (vezi în continuare parametrul HRgLength) cauzând o creștere exponențială a capacității predictorului. Din fericire, chiar și aceste structuri „imense” de predicție pot fi implementate în Siliciu, bugetul de tranzistoare existent la ora actuală permițând acest lucru, singurul neajuns constituindu-l întârzierea predicției.

Un argument „de bun simț” în favoarea păstrării și utilizării unei istorii cât mai „lungi” (îndepărtate) în procesul de predicție aferent instrucțiunilor de salt se referă la faptul că unele salturi corelate pot apărea la o distanță considerabilă în șirul de instrucțiuni dinamice. Acest lucru se poate întâmpla dacă două salturi corelate sunt despărțite (separate) de către un apel de funcție care conține multe branch-uri. La momentul „părăsirii” funcției, o istorie globală redusă poate conține doar comportamentul salturilor din cadrul funcției, în timp ce o istorie globală extinsă poate reține și rezultatul saltului corelat, anterior apelului funcției.

Se nasc astfel două întrebări: *i) Cât de „departe” trebuie căutate salturi corelate ? (HRgLength - maxim) și ii) Ce modalitate optimă preț / performanță permite utilizarea unei istorii vaste în procesul de predicție ?* În ce privește răspunsul la prima întrebare, la ora actuală (2003) în majoritatea cercetărilor lungimea istoriei globale se presupune a fi $\leq \log_2$ din

numărul de intrări în tabela de predicție. La a doua întrebare, pe lângă soluția evidentă (dar nu întotdeauna acceptabilă) de creștere corespunzătoare (HRgLength) a tabelii de predicție, ar mai exista posibilitatea dispersiei istoriei globale în vederea accesării unei tabele de predicție de dimensiune acceptabilă (64ko [Tho03]). De asemenea, unii cercetători au propus păstrarea unei istorii variabile ca lungime pentru fiecare instrucțiune de salt dinamică [Sta98].

Cu privire la modificările aduse structurii de predicție Target Cache în vederea îmbunătățirii acurateții predicției aferente instrucțiunilor de salt indirect o **primă etapă a constituit-o studiul influenței istoriei globale (globalHR) a salturilor condiționate asupra predicției**. Au fost folosite două moduri de indexare a structurii de predicție Target Cache:

a1) modul XOR

Nr set=LeastSignificantBits of (PC_instr **xor** globalHR)=(PC_instr **xor** globalHR) **mod** nr_seturi;

Tag = MostSignificantBits of (PC_instr **xor** globalHR)=(PC_instr **xor** globalHR) **div** nr_seturi

a2) modul Address

Nr set=LeastSignificantBits of PC_instr=PC_instr **mod** nr_seturi;

Tag = (MostSignificantBits of PC_instr) **xor** globalHR=(PC_instr **div** nr_seturi) **xor** globalHR

Schema **Address** utilizează cei mai puțini semnificativi biți ai adresei instrucțiunii de salt indirect pentru identificarea setului. Cei mai semnificativi biți ai adresei și istoria globală a salturilor condiționate sunt dispersați printr-o funcție XOR (sau exclusiv) formând Tag-ul emis. Studiind cele două moduri de indexare se poate intui superioritatea schemei de predicție TargetCache adresată în modul **XOR** față de aceeași structură dar indexată în modul **Address**, din punct de vedere al acurateții predicției instrucțiunilor de salt indirect. O explicație ar putea fi următoarea: la un grad de asociativitate redus, schema **Address** generează un număr semnificativ de miss-uri de conflict în TargetCache întrucât toate țintele aferente unui salt indirect static sunt mapate într-un același set (indiferent de contextul – globalHR - din care el apare 100, 010 etc).

Următorul pas făcut pentru a crește acuratețea de predicție a structurii TargetCache a constat în **extinderea informației de corelație** prin asocierea fiecărui bit din *globalHR* cu PC-ul aferent saltului condiționat respectiv și determinarea predicției pe baza acestei informații mai complexe [Nai95, Vin99] – vezi figura 5.29.

O critică generală ce poate fi adusă schemelor de predicție adaptive corelate pe două niveluri (aplicate atât salturilor condiționate cât și celor

indirecte) constă în faptul că folosesc insuficientă informație de corelație pentru identificarea cu precizie a contextului de apariție a saltului de prezis (globalHR pe HRgLength biți). Există situații în care două pattern-uri diferite de salturi condiționate dar cu comportament identic conduc la aceeași instrucțiune de salt indirect, target-urile acestuia fiind însă diferite de la un caz la altul – vezi și anexa 1 de la sfârșitul lucrării. În continuare am dezvoltat un predictor *bazat pe calea până la saltul indirect* care folosește drept informație de predicție pe lângă PC-ul saltului indirect și istoria globală a salturilor condiționate și PC-urile corespondente acestor salturi, în vederea reducerii coliziunilor în tabela Target Cache determinând creșterea acurateții predicției salturilor indirecte. Extinzând informația de corelație rezultă că la contexte diferite de apariție a unui salt indirect (PC₁, PC₂, PC₃ vs. PC₄, PC₅, PC₆ caracterizate de același comportament - 110), se vor accesa seturi diferite din cadrul structurii Target Cache, asociate corect contextelor, reducându-se astfel o parte din interferențe. Comprimarea acestui complex de informație este posibilă și chiar necesară, având în vedere necesitatea unor costuri rezonabile pentru aceste scheme. Funcția de dispersie utilizată este simplă (XOR). S-au păstrat cele două modalități de indexare ale structurii (XOR și Address) descrise anterior. În subcazul XOR rezultă cele două formule de identificare a setului și tag-ului, astfel:

$$\text{Nr set} = \text{LeastSignificantBits of } (\text{PC_instr} \text{ xor } \text{PC}(1) \text{ xor } \dots \text{ xor } \text{PC}(k) \text{ xor } \text{globalHR}) = (\text{PC_instr} \text{ xor } \text{PC}(1) \text{ xor } \dots \text{ xor } \text{PC}(k) \text{ xor } \text{globalHR}) \text{ mod nr_seturi};$$

$$\text{Tag} = \text{MostSignificantBits of } (\text{PC_instr} \text{ xor } \text{PC}(1) \text{ xor } \dots \text{ xor } \text{PC}(k) \text{ xor } \text{globalHR}) = (\text{PC_instr} \text{ xor } \text{PC}(1) \text{ xor } \dots \text{ xor } \text{PC}(k) \text{ xor } \text{globalHR}) \text{ div nr_seturi}$$

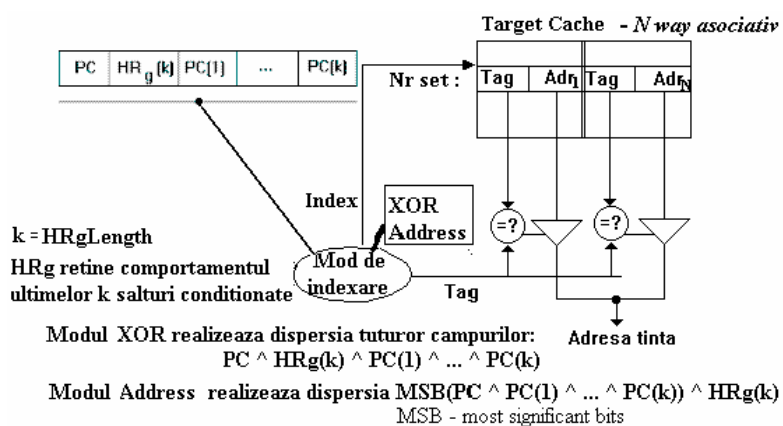


Figura 5.29. Extinderea informației de corelație pentru o structură de tip Target Cache

Pentru benchmark-urile caracterizate de un procentaj ridicat de salturi indirecte extinderea informației de corelație ($PC_1, PC_2, \dots, PC_{HRgLength}$), la costuri identice de implementare (structură Target Cache de aceeași capacitate) determină creșterea acurateții predicției acestora. Efectul pozitiv apare practic pe acele programe de test pentru care istoria salturilor condiționate influențează predicția (pentru detalii a se vedea capitolul de rezultate 7.1.1).

Cu toată îmbunătățirea adusă de extinderea corelației, acuratețea predicției instrucțiunilor de salt indirect este totuși inferioară celei obținute cu un predictor PPM complet (**89.83%**), în ciuda unor rezultate realmente extraordinare, care de fapt reprezintă excepția care confirmă regula. Astfel, acuratețea obținută pe *hydro2d.ss* este **99.74%** egală cu cea obținută cu predictorul PPM complet. Un exemplu care evidențiază limitarea avantajului introdus de tehnica de extindere a informației de corelație pentru pattern-uri de salturi condiționate de istorie redusă este prezentat în Anexa 1 de la sfârșitul lucrării. De asemenea, în urma simulărilor se observă incapacitatea tehnicii de extindere a informației de corelație de a depăși efectul negativ al miss-urilor de conflict, pentru un grad scăzut de asociativitate a tabelii Target Cache, în modul de indexare Address.

În continuare se exemplifică structurile de date noi introduse, opțiunile de simulare, principalele modificări aplicate funcției **sim-main** din modulul **sim-vpred.c**, care descriu funcționarea predictorului TargetCache (modalitatea de indexare – XOR sau Address, precum și păstrarea istoriei globale, respectiv a contextului - $PC_1, PC_2, \dots, PC_{HRgLength}$). Pentru început am descris nucleul de execuție al simulatorului (fazele *Fetch instrucțiune*, *Decodificare* și *Execuție* – aferente instrucțiunilor de salt indirect) rezident în rutina **sim-main** din modulul **sim-vpred.c**. Funcțiile apelate în interiorul acestei rutine au fost definite în modulul **vpred.c**. Dintre acestea **insertPC** înserează în coada de PC-uri adresa fiecărei instrucțiuni de salt condiționat întâlnit înaintea instrucțiunii de salt indirect. Funcția **hash_1** realizează dispersia printr-o instrucțiune de *SAU Exclusiv* între PC-urile salturilor condiționate anterior reținute în listă cu ajutorul funcției **InsertPC**.

/ Coada cu PC-uri extinsă */*

```
typedef struct PClocation *ExtendPC;
struct PClocation {
    md_addr_t addr;
    ExtendPC nextPC;
};
```

Dintre parametrii importanți ai simulatorului (**sim-vpred**) suplimentari celui prezentat la predictorul PPM complet, și care pot fi și modificați de către utilizator, sunt: numărul de seturi din TargetCache (**-dimcache**), dimensiunea listei de adrese destinație aferente unei instrucțiuni de salt indirect (gradul de asociativitate: **-poscache**), **-HRgLength** – reprezentând numărul de biți pentru reprezentarea istoriei globale a salturilor condiționate, **-XOR** – o opțiune pentru stabilirea modului de indexare a structurii TargetCache {**0** - Address mode | **1** - XOR mode}, **-Extend** – stabilește utilizarea (**1**) sau nu (**0**) a informației de corelație extinse reprezentată de PC-urile ultimelor HRgLength salturi condiționate.

Pentru fiecare instrucțiune de salt indirect întâlnită, se construiește un *Index* cu care se va „ataca” tabela de predicție (identificarea setului) și respectiv un *Tag* care va fi cheia de căutare pe nivelul pointat prin *Index* (vor avea loc cel mult *poscache* căutări pe nivel). De fapt, *Indexul* reprezintă restul împărțirii dintre PC-ul instrucțiunii și numărul de seturi din tabela TargetCache (*dimcache*), iar *tag-ul* reprezintă câtul aferent acestei împărțiri.

În funcția *TCpredictValue* din modulul *VPred.c* este implementat procesul de căutare a valorii care va fi prezisă (destinația instrucțiunii de salt indirect). Dacă în respectivul set se găsește o locație cu *tag-ul* căutat (**hit în TargetCache**), se înaintează valoarea prezisă (practic noul PC) și aceasta este preluată prin *bypassing* și încărcată în registrul de adresă a următoarei instrucțiuni (PC) - salturile indirecte făcându-se necondiționat. La determinarea valorii reale a registrului care codifică adresa pentru instrucțiunea de salt indirect, în urma fazei de decodificare/execuție, aceasta este comparată cu valoarea prezisă, și instrucțiunile dependente executate speculativ fie urmează parcursul normal – până la nivelul *Write Back* al structurii *pipeline* - fie sunt retrimise spre execuție de la adresa corectă. În situația unui **hit în TargetCache dacă predicția a fost eronată**, target-ul de la TAG-ul emis prin PC (și eventual HRg) va fi actualizat cu noua adresa destinație. În caz de **miss în TargetCache** tabela este actualizată prin introducerea valorii reale a registrului sursă aferent instrucțiunii de salt indirect, evacuând din lista de target-uri de la respectiva adresă (identificată prin set) cea mai veche dintre acestea, printr-un algoritm de tip FIFO.

```

/*****sim-main*****/
MD_FETCH_INST(inst, mem, regs regs_PC); /* extragerea instrucțiunilor din
cache/memorie */
MD_SET_OPCODE(op, inst); /* decodificarea instrucțiunii */

/* execuția instrucțiunii */
switch (op){
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3) \

```

```

#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
#define CONNECT(OP)
}

...
/* introducerea istoriei globale a salturilor conditionate in indexarea tabelii Target
Cache */
if (MD_OP_FLAGS(op) & F_COND){
    /*fprintf(stderr, "\n PC: %x NPC: %x op: %d %-10s in1: %d in2: %d \n",
    regs.regs_PC, regs.regs_NPC, op, MD_OP_NAME(op), in1, in2);*/
    isTaken=(regs.regs_NPC != (regs.regs_PC + sizeof(md_inst_t)));
    globalHR = (globalHR*2) % putere(2, HRgLength) + isTaken;
    /*fprintf(stderr, " HR = %d ",globalHR);*/
    /* inserare PC in coada de PC-uri (FIFO) */
    queue_PC = insertPC(queue_PC, regs.regs_PC,HRgLength);
}
// tratarea funcțiilor indirecte pure în faza de execuție
if((MD_OP_FLAGS(op)& F_INDIRJMP)&& !MD_IS_RETURN(op)){
    sim_indir_refs++;
    ...
    if(contextual == 0) /*target cache predictor*/
    {
        if(XORfunction) /* indexarea TC este XOR mode */
        {
            if(HRgLength == 0) /* nu tin cont de context */
                adresa_emisa = regs.regs_PC;
            else{
                /*show(queue_PC);*/
                if(Extend)
                    /* utilizarea istoriei extinse din PC-urile branch-
                    urilor conditionate */
                    PC_rezultat = hash_1(queue_PC);
                    adresa_emisa = PC_rezultat ^ regs.regs_PC ^
                    globalHR;
                }
            else
                adresa_emisa = regs.regs_PC ^ globalHR;
        }
    }
    else /* indexarea TC este Address mode */
    {
        if(HRgLength == 0)
            adresa_emisa = regs.regs_PC; /* PClow - setul; PChigh -
            tagul*/
        else /* HRgLength > 0 */
        {
            if(Extend) /* Folosesc informatie de corelatie extinsa */
            {
                PC_rezultat = hash_1(queue_PC);
                adresa_emisa = PC_rezultat ^ regs.regs_PC;
            }
        }
    }
}

```

```

        adresa_emisa_low = adresa_emisa & masca_nesem;
        adresa_emisa_high = ((adresa_emisa &
            masca_sem)>>biticache) ^ globalHR;
    }
    else /* Exista istorie dar nu folosesc informatie
        extinsa */
    {
        adresa_emisa_low = regs.reg PC & masca_nesem;
        adresa_emisa_high = ((regs.reg PC &
            masca_sem)>>biticache) ^ globalHR;
    }
}
if(!XORfunction && (HRgLength > 0))
    /*mod de indexare Address, si pattern de salturi
    conditionate > 0 */
    insertIntoDirrectMappedTargetCache(regs.reg R[in1],
        adresa_emisa_high,
        adresa_emisa_low, dimcache,
        nposcache);
else /* Fie mod de indexare XOR fie nu exista istorie */
    insertIntoDirrectMappedTargetCache(regs.reg R[in1],
        (adresa_emisa & masca_sem) >>
        biticache, adresa_emisa &
        masca_nesem, dimcache,
        nposcache);
}
/*****
sword_t TCPredictValue(IVPTCache p,md_addr_t tag ,int nposcache);

```

✓ Funcția de predicție folosită în cazul structurii Target Cache. În cazul în care există o singură locație per set, aceasta nefiind nulă, și cu tag corespunzător se efectuează predicția, returnându-se valoarea target-ului instrucțiunii. În cazul unui grad de asociativitate ridicat se parcurge lista target-urilor, comparându-se tagul din cache cu cel construit din PC-ul instrucțiunii; în cazul în care se găsește un tag corespunzător se efectuează predicția, altfel este considerată o predicție greșită.

Poate modificarea cea mai însemnată și originală în același timp referitoare la structură TargetCache a reprezentat-o **ignorarea selectivă a efectuării unor predicții** cu scopul de a îmbunătăți acuratețea predicției. Structura de predicție a fost îmbogățită (vezi figura 5.30) prin introducerea unui grad de confidență aferent fiecărei locații, folosindu-se și un mecanism de inserare / evacuare în / din set bazat pe:

- i. LRU(least recently used).
- ii. Confidență

- iii. Mecanism adaptiv ce se bazează pe superpoziția celor două anterioare (MPP – minim de performanță potențial).

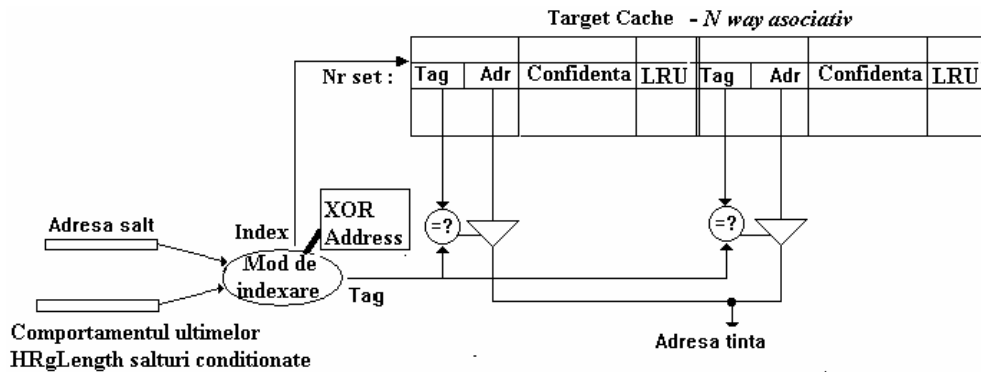


Figura 5.30. Îmbunătățirea structurii TargetCache cu un mecanism de confidență

Insuficiența informației de predicție poate conduce la costuri suplimentare în cazul unei predicții eronate, datorate refacerii contextului procesorului și reexecuției unui eventual lanț de instrucțiuni dependente de cea greșit predicționată, de multe ori fiind preferabil chiar să nu se facă predicția (evitându-se astfel execuția speculativă). Bazat pe simulare se va demonstra că acuratețea predicției valorilor poate fi influențată pozitiv printr-o ignorare selectivă a efectuării unor predicții. Pentru realizarea acestei selecții este asociat fiecărei predicții un *grad de încredere* sau **confidență**, stocat în tabela de predicție corespunzător fiecărei locații. O confidență ridicată semnifică încredere în predicție. În situația în care, unei intrări aferente predictorului de valori îi este asignată o confidență ridicată, valoarea prezisă va fi folosită. Dacă încrederea este însă sub un prag prestabilit (**threshold**) se renunță la predicție, execuția făcându-se nespeculativ.

Există în literatura de specialitate [Des02] trei tipuri de informație de confidență: *numărătoare saturate*, identificare de *tag*-uri și respectiv recunoașterea *pattern*-urilor. Dintre acestea am implementat software și simulat doar numărătoarele saturate. Am propus trei variante de mecanism de inserare / evacuare în / din set: prima bazată pe LRU, a doua determinată de confidență respectiv o a treia adaptivă, dinamică – numită minim de performanță potențială și bazată pe superpoziția celor două informații ortogonale amintite la variantele 1 și 2 (LRU respectiv confidență). Se determină practic cărui target din set îi corespunde cea mai mică valoare determinată de produsul $LRU \cdot Confidență$. Revenind la numărătorul saturat, trebuie remarcat că dacă pragul (**threshold**) este mai mare atunci mecanismul de confidență devine mai selectiv. Fără a depinde de confidența

deja asignată – starea în care se află numărătorul, acesta este actualizat imediat ce valoarea calculată (target-ul în cazul instrucțiunilor de salt indirect) devine disponibilă în urma fazei de execuție. Astfel, numărătorul este incrementat (cu 1 în general, dar pot exista și variații) în cazul unei predicții corecte și decrementat dacă predicția este greșită. Se poate afirma că printr-o mediere, numărătoarele saturate exprimă istoria recentă a predicției. Există și variante care, în cazul unei predicții greșite, resetează numărătorul sau decrementează cu 2 etc.

În [Des02] sunt propuse două metrici utile de comparare a mecanismelor de confidență: **acuratețea predicției și sensibilitatea**. Fiecare predicție în parte, poate proveni dintr-una din stările mecanismului de confidență: încredere ridicată (*high-confident*) sau încredere scăzută (*low-confident*). În acest sens, poate fi făcută următoarea clasificare a proceselor de predicție (vezi tabelul 5.3).

	Corect prezise	Inc corect prezise
Încredere ridicată	HCcorr	HCntcorr
Încredere scăzută	LCcorr	LCntcorr

Tabelul 5.3. Clasificarea proceselor de predicție

Un mecanism perfect de confidență ar avea completate doar celulele (HCcorr respectiv LCntcorr). În realitate însă, datorită limitărilor mecanismului de confidență, toate cele 4 celule ale tabelului 5.3 sunt completate (chiar și LCcorr respectiv HCntcorr). Trebuie observat că cele două clase nefavorabile, tocmai amintite, nu sunt echivalente din punct de vedere al impactului creat asupra funcționării microarhitecturii, deosebirea dintre ele constând în faptul că prima poate genera o creștere de performanță prin predicție corectă iar cea de-a doua - o penalitate prin refacerea contextului procesorului. Performanța scade drastic, fiind mult mai dezavantajos în cazul unei predicții făcute greșit (HCntcorr) decât atunci când se „ratează” (nu se face o predicție din lipsă de încredere care ar fi totuși corectă). Tehnologia hyper-pipeline a microarhitecturii Intel NetBurst determină creșterea în adâncime a structurii pipeline asigurând o sporire a performanței, frecvenței și scalabilității procesorului. Mecanismul de refacere a contextului procesorului și structurii pipeline implementat la procesorul Intel Pentium 4 consumă 31 de perioade de tact în tehnologie de 90 nm comparativ cu 20 de perioade de tact în tehnologie 0.13 microni.

După unii cercetători [Cal99, Rych98], un aspect important în creșterea performanței procesoarelor prin predicția valorilor îl constituie **eficiența**. Astfel, se consideră că doar predicțiile corecte de pe calea critică de program (*data-flow*) pot îmbunătăți performanța în timp ce predicțiile

greșite nu sunt dramatice dacă nu apar în acest graf al dependențelor de date, dar și invers, o predicție corectă care nu aparține căii critice nu se dovedește esențială din punct de vedere al creșterii performanței. Întrucât cazul nostru, al predicției valorilor aplicate instrucțiunilor de salt indirect – predicția **target-urilor**, se referă la fluxul de control al programului (*control-flow*), se poate spune că **eficiența** intervine și aici cu consecințele sale pozitive dar mai ales negative.

Revenind la metricile propuse în [Des02] **acuratețea predicției (A_p) reprezintă probabilitatea ca predicția generată de o stare ridicată de încredere să fie corectă.**

$$A_p = \text{Prob}(\text{predicția corectă} \mid \text{Incredere ridicată}) = \frac{HC_{\text{corr}}}{HC_{\text{corr}} + HC_{\text{ntcorr}}} \quad (5.9)$$

După părerea autorului acestei lucrări, un singur neajuns ar fi referitor la această metrică, și anume, **în ce măsură este relevantă o predicție foarte mare** (de exemplu 99% - vezi rezultatele grafice din capitolul 7.1.1) **atunci când procentajul cazurilor în care se face predicție – confidența ridicată** (vezi ecuația 5.10) **este, spre exemplu, mai puțin de 50% din totalul salturilor indirecte din program.** În acest sens, a fost introdus-ă o metrică proprie, vezi ecuația 5.11.

$$\text{Usage (grad de realizare a predicției)} = \frac{HC_{\text{corr}} + HC_{\text{ntcorr}}}{\text{Total_instrucțiuni_de_salt}} \quad (5.10)$$

Definim ca *performanță globală al predictorului* produsul:

$$\mathbf{P} = \mathbf{A}_p \cdot \mathbf{Usage} \quad (5.11).$$

Din (5.9), (5.10) și (5.11) rezultă că:

$$P = \frac{HC_{\text{corr}}}{\text{Total_instrucțiuni_de_salt}} \quad (5.11')$$

Senzitivitatea [Des02] reprezintă fracțiunea de predicții corecte identificate printr-o confidență ridicată.

$$\text{Senzitivitatea} = \text{Prob}(\text{Incredere ridicată} \mid \text{Totalul valorilor corect prezise}) = \frac{HC_{\text{corr}}}{HC_{\text{corr}} + LC_{\text{corr}}}$$

Având în vedere cele descrise anterior, se poate spune că **un mecanism de confidență bun este caracterizat prin tendința celor două metrici (acuratețea predicției și performanță globală) astfel: „cu cât mai mari cu atât mai bine”.**

Câmpul **Confidența** reprezintă un numărător saturat pe un număr de biți (parametrizabil) și care va fi incrementat de câte ori predicția s-a dovedit corectă sau decrementat de fiecare dată când predicția s-a dovedit eronată. O stare de confidență ridicată (peste un anumit prag – *threshold*, parametrizabil) îndreptățește procesorul să facă predicție dovedind practic **predictibilitatea corectă continuă într-o istorie dată a respectivei instrucțiuni de salt indirect**. În caz contrar nu se face predicție. Concretizând, rezultă că o confidență mică semnifică un salt indirect nepredictibil, nefiind necesar să-l rețin în tabele.

Se pune problema, în condițiile existenței câmpului de confidență, dacă este necesară prezența câmpului LRU, și nu cumva confidența să substituie sau să preia din sarcinile LRU-ului, existând astfel o oarecare dependență între ele. Răspunsul este unul afirmativ și exemplificăm printr-o situație ce poate apărea: Fie un salt indirect (cu *target*-ul aferent) ajuns la o confidență ridicată și care ulterior nu mai este accesat. Fără a exista câmpul LRU, rezultă că acest salt nu ar fi evacuat din structura Target Cache niciodată (absurd!) – pe viitor, evident, cu toate că el nu va fi de nici un folos și datorită gradului de asociativitate scăzut sunt necesare înlocuiri în setul respectiv - *miss*. Se poate ca această situație ipotetică să nu apară în practică niciodată, în funcție de pattern-urile urmate de benchmark-uri. Rezultatele exacte vor fi determinate însă abia după simulare.

Câmpul **LRU**, este caracterizat tot printr-un numărător saturat pe un număr de biți (parametrizabil și independent de reprezentarea confidenței) și reprezintă **gradul de activitate al saltului** respectiv. Cu fiecare *hit* în structura Target Cache (coincide tag-ul instrucțiunii de salt indirect cu unul din cele *poscache* existente în setul determinat) este incrementat câmpul LRU al respectivei locații (salt) și decrementat pentru toate celelalte salturi din setul respectiv. La un *miss* în structură, se va insera noul salt în tabelă având tag-ul și target-ul corespunzătoare execuției iar câmpurile LRU și Confidența vor fi 0. Inserarea se va face conform principiului LRU: dacă mai există locații libere în set atunci se va insera pe prima poziție liberă, în caz contrar fiind necesară înlocuirea locației având câmpul LRU cel mai mic cu valorile actuale ale câmpurilor Tag, Adr (Target), iar LRU și Confidența vor fi 0. În cazul în care a fost hit în tabela Target Cache, dar Confidența a fost sub pragul impus, nu s-a folosit predicția. Dacă în urma execuției s-a dovedit că predicția dacă s-ar fi făcut ar fi fost greșită atunci respectivei locații îi vor fi substituite câmpurile Adr (target-ul rezultat), Confidența (0) și LRU-ul (0). Rezultă astfel că, cel puțin la nivel teoretic, rezultatele vor fi slabe pe benchmark-uri cu o dispersie foarte mare a target-urilor (dacă același salt schimbă target-urile unul după altul fără a căpăta o oarecare încredere); astfel, de fiecare dată se va substitui vechiul target cu noul target,

scăzând practic în final acuratețea predicției - fapt ce se întâmplă și în cazul "fără confidență".

Având în vedere toate aceste informații privind mecanismul de funcționare al schemei de predicție Target Cache modificată consider că se poate afirma că cele două câmpuri care intervin în procesul de predicție (Confidența și LRU) sunt practic **ortogonale**. Mai mult, în cadrul modificărilor făcute se va introduce și un mecanism adaptiv de inserare (evacuare a locațiilor conflictuale) în tabelă. Astfel, evacuarea poate fi făcută și dacă LRU este mic – „slabă activitate” în ultimul timp, comparativ cu celelalte salturi, dar confidență ridicată, respectiv și dacă LRU este mare – foarte activ respectivul salt dar confidență scăzută (a prezis greșit de prea multe ori).

Practic acuratețea predicției [Des02] crește substanțial prin restrângerea cazurilor în care se face predicție. Bazat pe simulare se observă că pragul (*threshold*) reprezintă obstacolul care determină selecția și cu cât acesta este mai mare acuratețea predicției tinde spre absolut, dar procentajul cazurilor în care se face predicție din totalul instrucțiunilor de salt indirect scade semnificativ. Este de dorit totuși ca un procentaj cât mai ridicat de instrucțiuni de salt indirect să fie supuse predicției și acuratețea acestora să fie foarte mare. Rezultă că, ar fi ideal dacă s-ar putea printr-o metodă oarecare să reducem pragul dar să păstrăm acuratețea de predicție cât mai ridicată. În acest sens se va extinde informația de corelație pentru indexarea structurii Target Cache. Adresarea unei scheme de predicție de tip Target Cache îmbogățită cu mecanism de confidență (figura 5.30) cu informație de corelație extinsă își dovedește și de această dată eficacitatea atât din punct de vedere al acurateții predicției cât și din punct de vedere al performanței globale a predictorului (vezi formula 5.11') – obținându-se rezultate comparabile cu cele furnizate de un predictor PPM complet (vezi subcapitolul 7.1.1).

5.3.3. PREDICȚIA SALTURILOR INDIRECTE:

5.3.3.1. IMPLEMENTAREA PREDICTORULUI HIBRID CU SELECȚIE BAZATĂ PE ARITATE.

Ținând cont de experimentul lui Driesen dar mai ales de rezultatele limitate, din punct de vedere al acurateții predicției salturilor indirecte pe o serie de benchmark-uri indiferent de îmbunătățirile arhitecturale aduse, pe baza comportamentului dinamic am realizat o statistică (vezi capitolul de

rezultate 7.1.1) privind aritatea salturilor (*monomorfe* – generează dinamic un singur target, *duomorfe* – generează dinamic 2 target-uri distincte, respectiv *polimorfe* – salturile se efectuează la mai mult de două adrese destinație distincte).

Mediile aritmetice obținute pe benchmark-urile SPEC ('95, 2000) bogate în salturi indirecte sunt în concordanță cu rezultatele obținute de Driesen și exprimă faptul că deși salturile monomorfe sunt prezente într-o proporție covârșitoare din punct de vedere static comparativ cu cele duomorfe și respectiv polimorfe, ele reprezintă doar aproximativ o treime din totalul branch-urilor indirecte dinamice executate. Salturile polimorfe, deși puține din punct de vedere static (după cum s-a putut observa și în subcapitolul 4.2), dinamic constituie aproape jumătate din total.

Pornind de la predictoarele cascade pe mai multe niveluri și respectiv hibride [Dri98b, Dri98c], am implementat software și simulat o structură hibridă de predicție (vezi figura 5.31), compusă dintr-un predictor de tip LastValue și cel mai bun predictor contextual determinat în urma simulărilor cu istorie și pattern fix, în două ipostaze: (istorie redusă – 32 și pattern 3, sau istorie bogată – 256 și pattern 6) selecția făcându-se pe bază de aritate, după cunoașterea în prealabil a informațiilor de profil aferente fiecărui salt indirect.

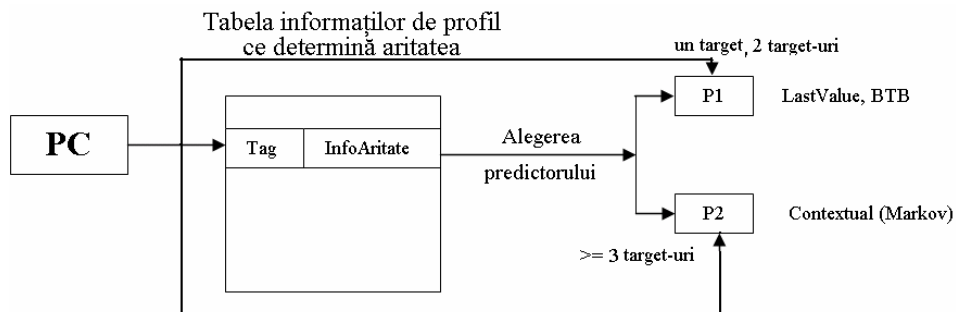


Figura 5.31. Predictor hibrid cu selecție bazată pe aritate.

Tabela informațiilor de profil este complet asociativă și reține informații privind aritatea fiecărui salt indirect, determinate în urma unei anterioare simulări privind studiul localității valorilor. Predictorul Contextual P2 este similar celui din subcapitolul 5.3.1 (vezi figura 5.27) iar P1 este identic cu cel folosit în predicția valorilor și va fi descris în subcapitolul 6.1.1. Ambele predictoare au un grad parametrizabil de asociativitate și sunt indexate cu ajutorul adresei instrucțiunii de salt indirect (PC).

Predictorul hibrid – o singură structură face predicție la un moment dat - **(LastValue+Contextual)**, cu selecție bazată pe aritate îmbunătățește acuratețea predicției salturilor indirecte cu procentaje cuprinse între 2.44% și 5.42% în funcție de structura de predicție cu care se face comparația (cel mai performant predictor contextual sau respectiv structură de predicție de tip TargetCache îmbogățită cu mecanism de confidență și indexată cu informație de corelație extinsă). În medie aritmetică pe benchmark-urile SPEC'95 acuratețea maximă de predicție obținută este de **93.77%**, apropiată de valorile maxime raportate în literatura de specialitate (94.8% cu predictor cascadat pe 3 niveluri). Procentajul substanțial de salturi polimorfe dinamice și dispersia ridicată a target-urilor anumitor salturi – vezi subcapitolul 4.3 – stă la baza limitării acurateții predicției. Rezultatele excelente obținute obligă la introducerea și **exploatarea predictoarelor hibride și cascadeate și în alte domenii de cercetare în vederea creșterii paralelismului la nivelul instrucțiunilor și a firelor de execuție: *predicția salturilor condiționate și predicția valorilor instrucțiunilor.***

În continuare se exemplifică structurile de date noi introduse, opțiunile de simulare, principalele modificări aplicate funcției **sim-main** din modulul **sim-vpred.c** precum și celorlalte funcții (**foundValue_INDIR**, **aritate_Address** din **vpred.c**), care descriu funcționarea predictorului cu selecție bazată pe aritate.

În implementare s-au folosit următoarele trei structuri, extinse din modelul implementării pentru localitatea valorii:

```
typedef struct element *valueList;
struct element
{
    sword_t value;
    int freq;      /* a fost adăugat pentru determinarea arității
                  fiecărui salt indirect */
    valueList nextValue;
};

typedef struct location *addrList;
struct location
{
    md_addr_t addr;
    addrList nextAddress;
    valueList values;
};
```

```
typedef struct typelocation *addrType;
struct typelocation
{
    md_addr_t addr;
    addrType nextAddress;
    int type;          //4-monomorfe;2-duomorfe;1-polimorfe
};
```

Identificarea câmpurilor din structură este următoarea:

addr – adresa (PC) instrucțiunii de salt indirect (jal \$reg sau j \$reg);

values – reprezintă lista ultimelor k valori dinamice pentru saltul respectiv (practic ultimele k target-uri aferente instrucțiunii de salt indirect);

Prezentăm în continuare principalele funcții ce intervin în cadrul procesului de determinare a arității: **foundValue_INDIR**, **aritate_Address**, definite în modulul **vpred.c**.

```
int foundValue_INDIR(addrList I, sword_t value, int history);
```

✓ Funcția *foundValue_INDIR* anterior descrisă la secțiunea 5.3.1 este îmbogățită cu numărătoare de frecvență necesare în calculul arității fiecărui salt indirect.

```
void aritate_Address(addrList I);
```

✓ Parcurge lista de adrese de salturi indirecte și preia informațiile de aritate anterior calculate stabilind procentajul de instrucțiuni statice și dinamice monomorfe / duomorfe / polimorfe existent în benchmark-ul respectiv. De asemenea, adresa și aritatea fiecărei instrucțiuni de salt indirect este înscris într-un fișier care va fi folosit ulterior pe post de informații de profil în procesul de predicție.

În continuare este prezentată modificarea adusă nucleului de execuție al simulatorului (fazele *Fetch instrucțiune*, *Decodificare* și *Execuție* – aferente instrucțiunilor de salt indirect) rezident în rutina **sim-main** din modulul **sim-vpred.c**. Primul pas este constituit din preluarea informațiilor de aritate din fișierul *PCType.txt*, având rolul tabelii informațiilor de profil (obținute prin simularea pe același benchmark cu flagul **pred** pe **0**) și introducerea acestora în lista *PCType*.

```
/******sim-main*****/
if (predict == 1)
{
    /* Se creează o listă dinamică cu informațiile de aritate stocate în fișier pentru a reduce
    accesul la discul magnetic pe măsură ce se întâlnește fiecare instrucțiune de salt
    indirect și trebuie prezisă */
```

```

    ....
    În caz că nu sunt colectate aceste informații sunt afișate mesaje de eroare care impun
    mai întâi generarea acestora.
    }
    ...
    MD_FETCH_INST(inst, mem, regs.reg_PC); /* extragerea instrucțiunilor din
    cache/memorie */
    MD_SET_OPCODE(op, inst); /* decodificarea instrucțiunii */
    /* execuția instrucțiunii */
    switch (op){
        #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)\
        #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
        #define CONNECT(OP)
    }
    ...
    // tratarea instrucțiunilor de salt indirecte pure în faza de execuție
    if((MD_OP_FLAGS(op)& F_INDIRJMP)&& !MD_IS_RETURN(op)){
        sim_indir_refs++;
        ...
        artateType = PCtype;
        bContinua = 1;
        nContor = 0;
        while ((artateType != NULL) && (bContinua) ){
            if(artateType->addr == regs.reg_PC){
                arietate = artateType->type;
                bContinua = 0;
            }
            artateType = artateType->nextAddress;
        }
        ....
        if((arietate == 4) || (arietate == 2)) //pentru monomorfe și duomorfe
        {
            history = 1; /* salturile monomorfe și duomorfe sunt predicționate de un
            predictor LastValue */
            auxvpt = lvpt;
        }
        else{
            auxvpt = jvpt; /* salturile polimorfe sunt predicționate de un
            predictor contextual */
            history = historyBackup;
        }
    }
    /***/

```

5.3.3.2. PREDICTOR HIBRID CU SELECȚIE BAZATĂ PE CONFIDENȚĂ.

În continuare sunt prezentate modificările aduse pentru implementarea predictorului hibrid cu selecție bazată pe confidență (numărătoare saturate).

Arhitectura predictorului este similară cu cea prezentată în figura 5.15. Se disting următoarele etape în realizarea predicției:

- ✓ Accesarea tabelii de tip selector pentru determinarea predictorului utilizat (cel cu confidența mai mare).
- ✓ Simularea (generarea predicției) cu predictorul determinat.
- ✓ Actualizarea tabelii *Selector* în funcție de rezultatul simulării.

În această versiune de simulator sunt disponibile următoarele combinații pentru cele două tipuri de predictoare componente:

- ✓ P1 – TargetCache ; P2 – LastValue
- ✓ P1 – Contextual (Markov ordin m); P2 - Contextual (Markov ordin n), cu $m > n$.
- ✓ P1 – TargetCache ; P2 – Contextual (Markov ordin k).
- ✓ P1 - Contextual (Markov ordin k); P2 – LastValue.

Selectorul este implementat sub forma unei liste înlănțuite, având ca elemente obiecte din structura următoare:

```
typedef struct _SELaddrList{
    md_addr_t addr;      //adresa instructiunii de salt indirect
    sword_t counter1;   //numărătorul aferent primului predictor (P1).
    sword_t counter2;   /*numărătorul aferent celui de-al doilea
                        predictor (P2).*/
    struct _SELaddrList *nextAddress;
}SELaddrList;
```

Pentru implementarea selectorului s-au folosit funcțiile **foundAssociativeSELAddress**, **pushSELAddress**, **updateSelector**, toate implementate în fișierul **vpred.c**.

```
int foundAssociativeSELAddress(md_addr_t addr, SELaddrList* sel);
```

- ✓ Se determină dacă instrucțiunea a mai fost anterior predicționată și în caz afirmativ se preia confidența fiecăruia (care din cele două a prezis cu succes mai mult timp). Valoarea prezisă de componenta cu confidența cea mai ridicată va fi cea folosită în continuare.

```
SELaddrList* pushSELAddress(SELaddrList* sel, md_addr_t addr);
```

- ✓ Dacă instrucțiunea nu a mai fost anterior predicționată atunci este inserată în lista de confidențe și sunt asignate grade de încredere egale ambelor predictoare folosite.

```
void updateSelector(SELaddrList* sel, md_addr_t addr, int predNr, int predOk),
```

- ✓ Funcția se apelează după verificarea predicției, penalizând în cazul unei predicții greșite predictorul care a generat valoarea prezisă, sau mărindu-i confidența acestuia în cazul unei predicții corecte.

5.3.4. JIndirSim – SIMULATOR FUNCȚIONAL PENTRU PREDICȚIA SALTURILOR INDIRECTE.

5.3.4.1. ARHITECTURA APLICAȚIEI.

În acest subcapitol se va prezenta structura bazei de date dezvoltate și utilizate, modul de acces la aceasta, modul de desfășurare al simulării din punct de vedere al firelor de execuție folosite, precum și tehnica folosită pentru raportare.

Pentru salvarea parametrilor și rezultatelor simulărilor s-a folosit o bază de date Microsoft Access 2000, și anume *JindirPred.mdb*. Această bază de date conține două tabele: *SimCaract*, în care se memorează caracteristicile simulărilor realizate (tipul predictorului, tipul benchmarkului, benchmarkul, parametrii de simulare, ... etc); *SimResult*, în care sunt memorate rezultatele simulărilor (timpul în care a fost realizată simularea, numărul de salturi indirecte, numărul de salturi indirecte predicționate corect). Cele două tabele sunt legate prin câmpul *Id*, având proprietatea autonumber în *SimCaract* și necesar în *SimResult*.

Ca modalitate de conectare la baza de date s-a ales tehnologia ODBC, pe de o parte pentru a face mai accesibilă conectarea (nu mai este nevoie de scrierea *stringului* de conectare), iar pe de altă parte pentru transparența serverului de baze de date. Open Database Connectivity (ODBC) este o componentă a Microsoft Windows Open Services Architecture (WOSA). Interfețele ODBC fac posibil accesul din aplicații la aproape orice date relaționale stocate în aproape toate sistemele de gestiune a bazelor de date (DBMS). ODBC este larg acceptat ca API (*application programming interface*) pentru accesul la bazele de date. Se bazează pe specificația *Call-Level Interface* (CLI) din X/OPEN și ISO/IEC pentru API-uri de baze de date și folosește SQL ca limbaj de acces la bazele de date. Interfața ODBC utilizează drivere pentru conversia sintaxei SQL de la un produs la altul. Microsoft a implementat un număr de drivere ODBC pentru a accesa diverse date stocate (Access, Foxpro, MS SQL Server, Oracle, Paradox).

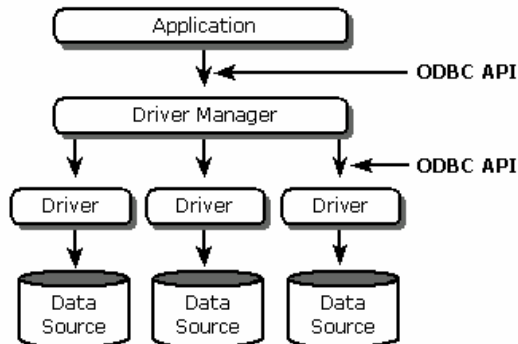


Figura 5.32. Interfața ODBC cu aplicația.

Pentru procesul de simulare, după colectarea datelor necesare, se creează un fișier *sim.bat* în cadrul căruia este înregistrată comanda de execuție pentru simulatorul solicitat, împreună cu toți parametrii acestuia din linia de comandă. Se creează un proces care să ruleze acest fișier (*sim.bat*). Apoi este creat un fir de execuție care, din timp în timp (inițial este setat la o secundă, dar, pentru simulări mai laborioase este recomandată mărirea acestui interval) verifică dacă simularea s-a încheiat (vezi figura 5.33).

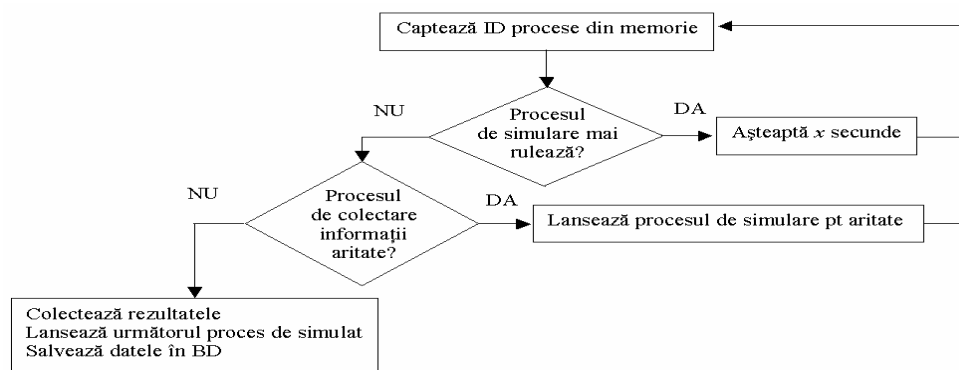


Figura 5.33. Firul de execuție pentru monitorizare.

Această verificare este realizată prin captarea numelor și identificatorilor proceselor care rulează la acel moment, iar apoi căutarea procesului cu același identificator (*ID*) cu procesul care rulează fișierul *sim.bat*. În cazul în care procesul s-a încheiat, tot în cadrul firului se preiau datele din fișierul rezultat în urma simulării (*simout.res*) și se introduc în baza de date. Înainte de introducerea datelor rezultate în baza de date, se lansează următoarea simulare, dacă aceasta există. Deoarece cele două tabele sunt legate printr-un *ID* cu proprietatea *autonumber*, s-a folosit

metoda *AddNew*, care adaugă o înregistrare goală în setul de înregistrări (RecordSet), care se actualizează cu valorile dorite. În acest fel este posibilă obținerea Identity-ului ID, fără a fi în pericol de a pointa rezultatele la altă simulare, cum s-ar fi întâmplat în cazul în care ID-ul s-ar fi preluat cu funcția *Max(ID)*. Apoi se scriu valorile rezultate în tabela *SimResult*.

Pentru obținerea statisticilor s-a folosit mediul de raportare *Seagate Crystal Reports*, mediu capabil să creeze rapoarte și din baze de date prin conexiune ODBC. *Crystal Reports* este o modalitate rapidă și eficientă de a ordona datele, de a evidenția anumite caracteristici etc. În ultima vreme, *Crystal Reports* este tot mai întâlnit în rândul aplicațiilor, de la soluții ERP până la softuri de dezvoltare precum *Visual Studio .Net*. Pentru integrarea rapoartelor în cadrul aplicației s-a folosit unul dintre utilitarele de vizualizare (*view-er*) existente, încorporat în componentele CRPE. S-a creat clasa *CrapView* derivată din *CView*, clasa ce va fi suportul pentru raportul de afișat. Afișarea raportului se face prin funcția *AfxRaport(CString szRaport, CString szWhere)*, funcție ce permite filtrarea datelor din raport și din program, nu numai din *design*-ul raportului. Un exemplu de clauză SQL folosită este următorul: *WHERE Simulator = 'Markov' AND DimensiuneTabelă = 256 AND Istorie = 256*. Se configurează apoi datele care se doresc afișate sub forma grafică: *Acuratețe, AVERAGE(Acuratețe)*, rezultatul putând fi urmărit în figura 5.41.

În ceea ce privesc dezvoltările viitoare asupra aplicației, s-ar putea avea în vedere extinderea domeniului de statistici prin crearea de noi rapoarte care să evidențieze diverse situații, cum ar fi determinarea ordinului optim aferent predictoarelor Markov din cadrul predictorului hibrid cu două predictoare Markov. O altă direcție de dezvoltare ar fi implementarea unei strategii de introducere dinamică în aplicație a unui nou predictor, fără a fi necesară scrierea de cod. O idee în această direcție ar fi stocarea datelor caracteristice predictoarelor într-o tabelă (executabilul cu care se execută, parametrii de care are nevoie, parametrii pe care nu îi folosește etc.), din care să se colecteze datele înainte de simulare. În acest caz, operația de adăugare a unui predictor s-ar reduce la adăugarea unei linii într-o tabelă. De asemenea, implementarea de noi statistici s-ar putea implementa dinamic, aproximativ prin aceleași metode. Altă direcție de dezvoltare este filtrarea datelor corespunzătoare simulărilor din aplicație, proces început dar adus doar în stadiul în care datele pot fi ordonate după orice câmp, coloanele pot fi rearanjate după preferințele utilizatorului. Ar putea urma implementarea posibilității de ștergere a articolelor din lista caracteristicilor simulărilor. Având funcția *AfxRaport* care afișează raportul cu posibilitatea introducerii unei clauze SQL (cel mai des folosită este *WHERE*), până la implementarea unui mecanism de filtrare dinamică a datelor din rapoartele

existente mai este doar un mic pas, respectiv crearea unui ecran care să permită filtrarea datelor și apoi reorganizarea datelor din ecran într-o clauză Where de exemplu. Poate cea mai importantă dezvoltare ulterioară care poate fi realizată în scopul eficientizării procesului de simulare pe benchmark-urile actuale existente (SPEC 2000 înglobează instrucțiuni dinamice de ordinul sutelor de miliarde – vezi tabelul 3.2) se referă la distribuirea proceselor de predicție în cadrul unei rețele. Avantajul evident ar consta în generarea mult mai rapidă a acurateții medii de predicție pentru simulările multiple, utile în statisticile care stau la baza emiterilor de concluzii privind eficiența schemelor hardware implementate.

5.3.4.2. *JIndirSim*. INTERFAȚĂ GRAFICĂ. GHID DE UTILIZARE.

Privind din punctul de vedere al utilizatorului, se consideră imperios necesară o interfață vizuală prietenoasă, bazată pe meniuri, ferestre de dialog, imagini grafice edificatoare etc. Avantajul utilizării imaginilor și a butoanelor grafice, constă în caracterul internațional al imaginilor, spațiu redus de stocare față de echivalentele textuale. Principalul scop al utilizării imaginilor grafice este să ajute utilizatorul să recunoască un program sau o funcție mai rapid decât prin parcurgerea unui text descriptiv. Interfața trebuie să fie simplă de utilizat, să permită utilizatorului manevrarea ușoară a simulatorului, interpretarea și prelucrarea eficientă a rezultatelor, extinderea ulterioară cu noi opțiuni de meniu sau salvarea diverselor rezultate sub diferite forme.

Implementarea interfeței simulatorului în mediul Developer Studio din Visual C++ (versiunea 6.0) s-a făcut datorită faptului că limbajul C++ oferă un suport puternic pentru programarea orientată pe obiecte: încapsulare, moșteniri multiple, redefinirea operatorilor, funcții și clase prieten etc. Conceptele de moștenire și polimorfism creează premisele dezvoltării ulterioare (extinderii) a variantei actuale de simulator. De asemenea, implementarea trebuie realizată în așa manieră încât, orice modificare (adăugare) în hardware sau software să fie făcută cu minim de efort. Pe lângă compilator, pachetul Visual C++ conține biblioteci, exemple și documentația necesară pentru crearea aplicațiilor în sistemele de operare Windows 9x, Windows 2000 sau Windows XP.

Pentru a porni simulatorul este nevoie de un sistem pe care să fie instalat un sistem de operare pe platforma NT (Windows NT 4.0, Windows 2000, Windows XP, Windows 2003) și să existe benchmark-urile (subdirectoarele) SPEC'95 și 2k cu programele de test aferente în locația stabilită (inițial **C:\Benchmark**). Datorită incompatibilităților dintre

sistemele de operare pe platforma NT și sistemele Windows 9x (funcții precum *EnumProcesses*, folosite în aplicație, nu au corespondent pe sisteme de operare Windows 9x) aplicația nu funcționează corect, neputând surprinde momentul în care simularea a luat sfârșit (vezi subcapitolul anterior). Pentru funcționarea corectă a aplicației "*JIndirSim*" procesul care stă în spatele aplicației (**sim-vpred.exe**) și care este vital acesteia, poate fi compilat/asamblat/linkeditat atât sub sistemul de operare Windows NT/Windows 2000/Windows XP cât și sub Windows 9x.

Aplicația este alcătuită din punct de vedere al interfeței dintr-un formular (FormView) ce conține ecranul și două ferestre *dockabile*, menite să ofere utilizatorului posibilitatea adaptării interfeței în funcție de preferințele proprii. (vezi figura 5.34).

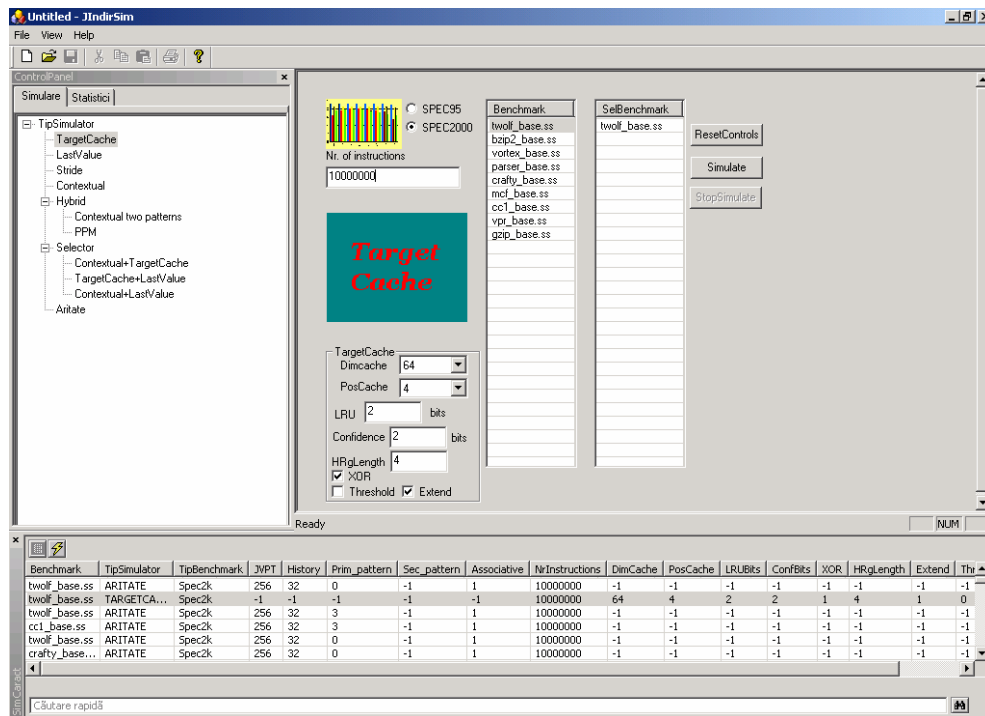


Figura 5.34. Selecția tipului de simulare

Pentru ca zona vizibilă a aplicației să cuprindă cât mai multe controale și ferestre necesare unei simulări chiar și la rezoluții mai mici, atât fereastra de configurare a tipului simulării sau a statisticilor dorite cât și fereastra ce conține caracteristicile simulărilor anterioare pot fi pliate pe oricare din cele patru margini a aplicației, sau chiar închise. Aceste ferestre pot fi închise sau vizualizate din meniul principal *View->ControlPanel* (pentru fereastra de

configurare) sau *View->SimCaract* (pentru fereastra ce conține caracteristicile simulărilor). Din aceleași considerente, pentru a evita încărcarea ecranului cu controale nefolosite la acel moment, controalele necesare simulării au fost separate de cele necesare vizualizării statisticilor prin folosirea unui control cu foi de proprietăți.

În continuare sunt prezentați pașii necesari procesului de simulare și de vizualizare a rezultatelor:

- Se alege tipul simulatorului dorit din fereastra *ControlPanel* (simulatorul implicit cu care pornește aplicația este de tip *TargetCache*) (vezi figura 5.35); în funcție de tipul predictorului ales, în ecran se încarcă controalele corespunzătoare parametrilor acestuia, cu eventuale valori prestabilite (de exemplu, la predictorul de tip *LastValue* parametrul *history* este întotdeauna 1). Sunt disponibile următoarele tipuri de predictoare: *TargetCache*, *LastValue*, *Stride*, *Contextual* (Markov de ordin (k)), predictor *hibrid alcătuit din două predictoare Markov* (unul de ordin mai mare, iar dacă acesta nu reușește să predicționeze, se încearcă cu unul de ordin mai mic), *PPM complet*, cu selecție bazată pe confidență (combinații între *TargetCache*, Markov și *LastValue*), precum și *predictor cu selecție bazată pe aritate*.

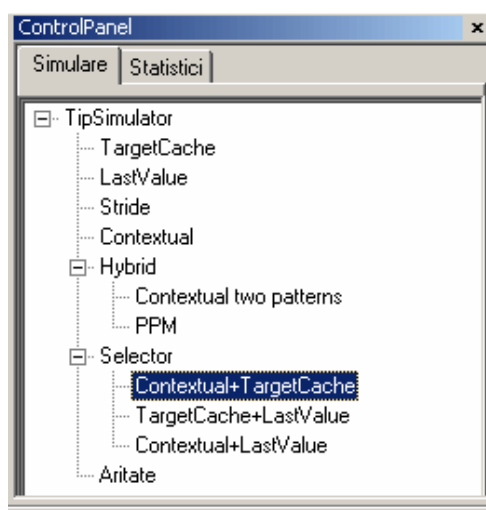


Figura 5.35. Fereastra de alegere a tipului de simulator

- Se aleg valorile parametrilor corespunzători simulatorului ales în fereastra ecran (vezi figura 5.36)

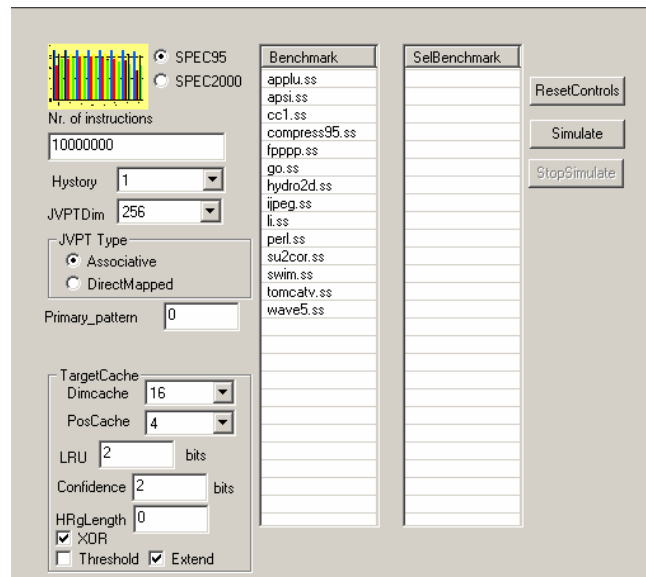


Figura 5.36. Ecranul cu parametrii simulatorului

- Se aleg benchmark-urile pentru care se dorește simularea prin **dublu click** pe numele acestora din lista *Benchmark*, sau prin selectarea mai multor benchmarkuri în lista și apăsarea tastei **Enter**; în urma acestei operații, benchmark-urile selectate vor fi introduse în lista *SelBenchmark*. Dacă se dorește renunțarea simulării pe unul sau mai multe benchmarkuri selectate, se marchează acestea în lista *SelBenchmark* și se apasă tasta **Delete**.
- Se apasă butonul *Simulate* pentru a începe simularea. Dacă se dorește oprirea simulării, se poate apăsa butonul *StopSimulate*, disponibil pe durata în care simularea este în curs de desfășurare.
- Pentru a vizualiza rezultatul simulării, se selectează linia dorită în lista simulărilor efectuate *SimCaract* (figura 5.37). În urma acestei operații va apărea în locul ecranului cu parametri raportul corespondent simulării selectate (vezi figura 5.38)

Ben...	TipSimul...	TipB...	JVPT	History	Prim_patern	Sec_patern	Associative	NrInstructions	DimCache	PosCache	LRUBits	ConfBits	XOR	HRgLength	Extend	Threshold	
twolf...	ARITATE	Spec2k	256	32	0	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	972
twolf...	TANKS2k	Spec2k	256	32	1	-1	1	10000000	34	4	2	1	1	1	1	1	974
twolf...	ARITATE	Spec2k	256	32	3	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	970
cc1_b...	ARITATE	Spec2k	256	32	3	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	969
twolf...	ARITATE	Spec2k	256	32	0	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	968
craft...	ARITATE	Spec2k	256	32	0	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	967
li.ss	ARITATE	Spec95	256	32	3	-1	1	10000000	-1	-1	-1	-1	-1	-1	-1	-1	966

Figura 5.37. Lista parametrilor pentru simulările anterioare

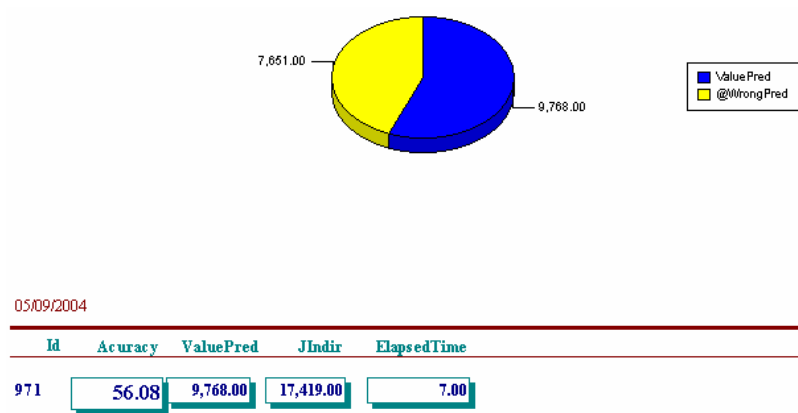


Figura 5.38. Raportul afișat după simulare

- După efectuarea mai multor simulări pot fi vizualizate diverse statistici, disponibile după activarea foii de proprietăți *Statistici* din *ControlPanel* (vezi figura 5.39). Aceste statistici sunt împărțite în funcție de tipul de benchmark, respectiv SPEC'95 și SPEC2k; există statistici exprimate în funcție de tipul de predictor folosit, respectiv evidențierea performanțelor acestuia pe anumite benchmark-uri (vezi figura 5.40); există și statistici pe mai multe benchmark-uri, cum ar fi găsirea ordinului optim al predictorului Markov (vezi figura 5.41).

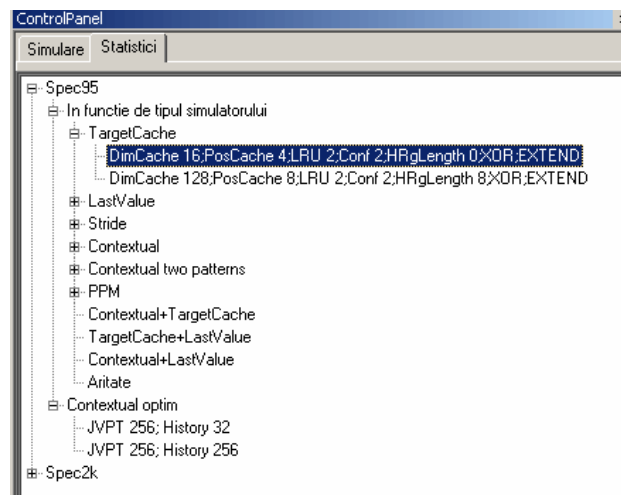


Figura 5.39. Statisticile disponibile în versiunea curentă de simulator

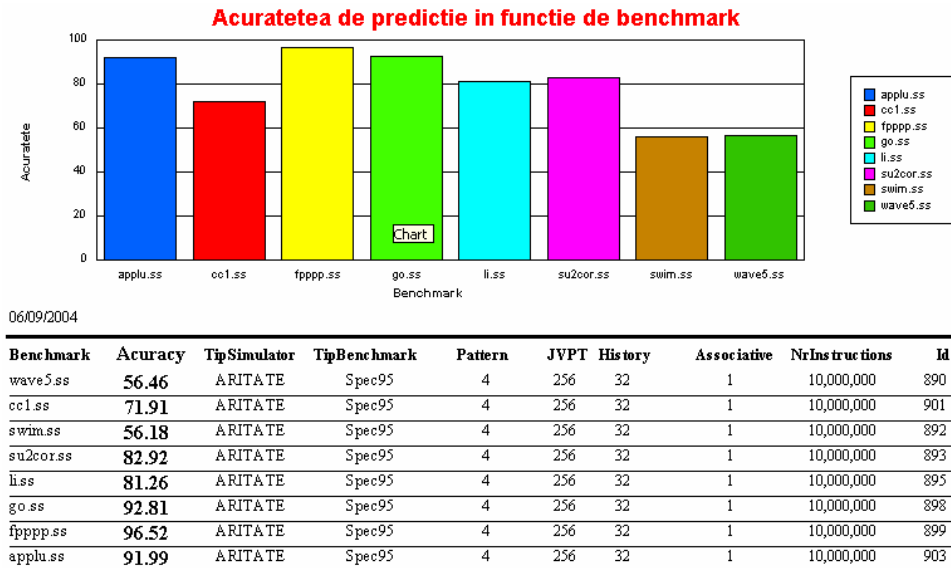


Figura 5.40. Raport statistic pentru un tip de predictor (TargetCache)

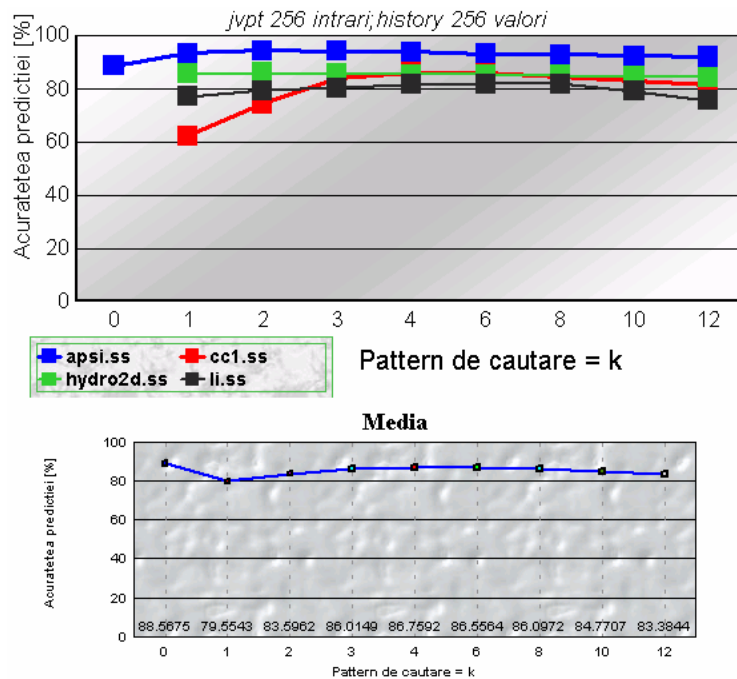


Figura 5.41. Raport statistic pentru determinarea ordinului optim al predictorului Markov

Problema salturilor indirecte este de mare actualitate, cu precădere în contextul programelor obiectuale, legat mai ales de implementarea polimorfismelor. În acest caz, pe baza adreselor de început ale diferitelor obiecte vizate se determină inițial adresa tabelii metodelor virtuale apoi adresa funcției (metodei virtuale) care va fi înscrisă dinamic în registrul de indirectare al saltului implementându-se astfel polimorfismul [Roth99].

Pe baza schemelor de predicție descrise în acest subcapitol (5.3) și totodată implementate software se pot urmări în capitolul 7.1 rezultatele obținute în urma simulărilor pe programe de test reprezentative, sub formă grafică sau tabele, ajutând la înțelegerea particularităților fiecărei scheme în parte dar și la determinarea unei configurații optime din punct de vedere al performanței (*acuratețea predicției* în acest caz).

6. CERCETĂRI CU PRIVIRE LA PREDICȚIA DINAMICĂ A VALORILOR INSTRUCȚIUNILOR

6.1. PREDICȚIA VALORILOR INSTRUCȚIUNILOR (LOAD, ALU).

6.1.1. DESCRIEREA SIMULATORULUI “VALUE PREDICTOR”.

Pentru extinderea setului de instrumente SimpleScalar cu modulul **VPred.c** (**Value Predictor** – care exploatează gradul de localitate existent în programele de calcul – pe tipuri de instrucțiuni, pe regiștri etc. și implementează o mulțime de 4 predictoare de la simple la cât mai complexe) s-a pornit de la scheletul simulatorului funcțional **sim-bpred.c**. După descărcarea de pe Internet de la adresa "<http://www.cs.wisc.edu/~mscalar/simplescalar.html>" a fișierului *simplesim-3.0b.tar.gz*, funcția *myrand()* din *misc.c* a fost modificată pentru eliminarea erorii apărute la compilare; astfel apelul funcției *random()* a fost condiționat și de *defined(_CYGWIN32_)*. Pentru compilarea surselor s-a folosit aplicația **Cygwin** (comanda *make*), acesta fiind un emulator de Linux, care permite generarea executabilului pentru sistemul Windows. Pentru implementarea noului simulator **VPred** (**Value Predictor**), a fost necesară modificarea fișierului *Makefile* astfel încât la comanda *make* să fie compilat și acesta. Structurile utilizate, precum și declarațiile funcțiilor se află în fișierul *vpred.h*, iar definițiile funcțiilor pot fi găsite în *vpred.c*. Motorul simulatorului îl reprezintă funcția *sim_main()* din *sim-vpred.c*, aici sunt folosite atât structurile cât și funcțiile amintite.

Față de arhitectura clasică a unui procesor superscalar implementată în simulatorul *sim-bpred*, se introduc structurile de date necesare pentru determinarea localității valorilor existentă la nivelul instrucțiunilor Load/ALU sau a regiștrilor din benchmark-urile SPEC. Localitatea (vecinătatea) valorilor descrie probabilitatea statistică de referire a unei

valori anterior folosite și stocată în aceeași resursă (locație de memorie, registru). Localitatea valorii pentru un benchmark este calculată ca raport dintre numărul de instrucțiuni Load *dinamice* care regăsesc o aceeași valoare în memorie ca și precedentele k accese și numărul de instrucțiuni Load *dinamice* existente în benchmark-ul respectiv, lucru ce presupune memorarea adresei fiecărei instrucțiuni Load dinamice din acel benchmark. Au fost abordate două moduri de exploatare a localității valorilor din programele de calcul. Una, propusă de Lipasti, în care s-a “forțat” puțin în obținerea localității, prin păstrarea în istoria de valori aferente unei instrucțiuni doar a valorilor distincte, care pot apare nu neapărat la fiecare instanță a respectivei instrucțiuni [Lip96] și a doua, în care, în lista de valori rezultate pentru o anumită instrucțiune s-au păstrat ultimele valori, indiferent dacă unele dintre acestea s-au repetat.

De asemenea se introduc și structurile necesare pentru implementarea tehnicilor de predicție a valorilor prezentate în subcapitolul 2.2.2. Utilizând această tehnică hardware speculativă [Lip96] se urmărește exploatarea redundanței existente în programe prin comprimarea dinamică a dependențelor de date. Tehnica Load Value Prediction predicționează rezultatele instrucțiunilor Load la expedierea spre unitățile funcționale de execuție exploatând corelația dintre adresele respectivelor instrucțiuni (sau date) și valorile citite din memorie de către acestea, permițând deci instrucțiunilor Load/ALU etc. să se execute înainte de calculul adresei și îmbunătățind astfel performanța. Ca și consecință a predicției valorilor se reduc efectele defavorabile ale hazardurilor RAW, prin reducerea așteptărilor instrucțiunilor dependente ulterioare. Este evident că acest câștig este mult mai mare atunci când predicția se face pe calea critică de program [Cal99].

Un alt deziderat al cercetării l-a constituit exploatarea gradului de localitate exprimat și de alte tipuri de instrucțiuni (aritmetico-logice – ALU și de salt indirect - JIndir). Asupra acestora din urmă s-a insistat pe parcursul subcapitolelor 5.2÷5.3. Avantajele predicției valorii pentru instrucțiunile ALU (rezultatele acestora) constau atât în reducerea timpului de execuție pentru instrucțiunile consumatoare de timp (DIV, MULT) dar și în reducerea efectelor defavorabile ale hazardurilor RAW, prin reducerea așteptărilor instrucțiunilor dependente ulterioare.

Pentru determinarea localității / predicției valorilor sunt folosite următoarele structuri:

```

typedef struct VHTElement *VHTvalueList;
struct VHTElement
{
    sword_t value;
    VHTvalueList nextValue;
    int count;        // este folosit doar de către predictorul contextual
};

typedef struct VHTlocation *VHTaddrList;
struct VHTlocation
{
    md_addr_t addr;
    VHTaddrList nextAddress;
    VHTvalueList values;
    int automat;
    sword_t stride[2]; // este folosit doar de către predictorul incremental
};

```

Fiecare locație din **VHT** (tabela de predicție) conține următoarele câmpuri:

- ☞ **addr** – adresa instrucțiunii sau adresa datei;
- ☞ **values** – reprezintă lista celor mai recente valori regăsite de respectivul Load în memorie;
- ☞ **automat** – este un automat cu patru stări. Un Load este nepredictibil dacă automatul acestuia se află în starea 0 sau 1 și este predictibil dacă automatul se află în starea 2 sau 3;
- ☞ **stride** – reprezintă două câmpuri în care sunt memorați cei doi pași utilizați în predicția incrementală.

Presupunând că pe lângă corelația dintre adresele instrucțiunilor Load și valorile citite din memorie de către acestea, există o corelație și între adresele datelor aferente instrucțiunilor Load și valorile de la acele adrese, simulatorul a fost proiectat în așa fel încât să permită utilizarea în predicție atât a adresei instrucțiunii cât și a adresei datei (flag-ul **-memaddr**). Alți parametri importanți ai simulatorului care pot fi modificați de către utilizator, sunt: *tipul simulării* (**-pred** - determinarea localității (**0**) pe tipuri de instrucțiuni / regiștrii procesorului MIPS sau predicția valorilor (**1**)), *gradul de localizare* (**-history** – istoria folosită), *tipul tablei de predicție* (**-assoc** - mapată direct (**0**) sau asociativă (**1**)), *dimensiunea tablei de predicție* exprimată în număr de locații (**-lvpt**), tipul predictorului utilizat: – **contextual** (incremental - **0**, contextual - **1** sau hibrid - **2**) și dimensiunea

contextului (*-pattern*) în cazul predictorilor contextuale și hibride. Selecția tipului de predictor *LastValue* se face impunând parametrul *-history* pe 1.

Datorită similitudinilor dintre între principalele funcții ce intervin în cadrul predictorului “*Value Predictor*” și cele aferente predictorului PPM prezentat în cadrul subcapitolului 5.3.1 referitor la salturi indirecte: **predictValue**, **foundAssociativeLVPTAddress** și **insertLVPTValue**, s-a renunțat la prezentarea conținutului acestora.

La citirea unei instrucțiuni Load din cache sau memoria centrală, în cazul unei tabele VHT (Value History Table) mapată direct, cu cei mai puțin semnificativi biți ai adresei instrucțiunii Load (PC_{LOW}) se adresează tabela VHT (vezi figurile 2.14, 2.17, 2.18). Maparea se face după următoarea formulă:

$$\text{index} = \text{addr} \bmod \text{VHTdim},$$

unde *addr* reprezintă adresa instrucțiunii sau adresa datei, iar *VHTdim* reprezintă dimensiunea tabelii de predicție. Se verifică dacă *addr* este egală cu adresa de la indexul respectiv, caz în care avem hit. În cazul în care valorile nu sunt egale vom avea miss în VHT, nu se poate face predicție, iar locația corespunzătoare indexului calculat va fi actualizată cu noua adresă și cu valoarea adusă din memorie de instrucțiunea Load.

Dacă se folosește o tabelă VHT asociativă, *addr* este comparat cu adresa din fiecare locație a tabelii și va fi hit în cazul în care adresa este găsită. Dacă avem miss în VHT, pe baza algoritmului LRU (Least Recently Used) implementat, se evacuează din tabelă cel mai puțin recent accesat Load și se introduce noua adresă și valoarea citită din memorie. S-a dorit evaluarea performanțelor diferitelor predictoare în condițiile utilizării unui algoritm LRU “*perfect*”, care e mai greu de implementat în hardware. Tabela VHT poate fi privită ca o listă în care cele mai recent accesate Load-uri se află la început, iar cele mai puțin recent accesate Load-uri se află la sfârșit. În cazul unui hit în VHT Load-ul găsit trece pe prima poziție din listă. În cazul unui miss în VHT, se evacuează Load-ul aflat pe ultima poziție, iar noul Load se inserează la începutul listei.

Indiferent de tipul tabelii utilizate (mapată direct sau asociativă), în cazul în care avem hit în VHT se face predicție. În funcția *predictValue* din *vpred.c* sunt implementate: predictorul de tip “*Last Value*” (vezi figura 2.14), predictorul incremental de tip “2-delta” (vezi figura 2.17), predictorul contextual de tip PPM complet (vezi figura 2.18) și predictorul hibrid (incremental, contextual – figura 2.20). În această funcție este folosit predictorul corespunzător parametrilor introduși de către utilizator. În cazul în care automatul din locația VHT se află în starea *predictibil*, se înaintează valoarea precisă și aceasta este preluată prin *bypassing* de către instrucțiunile dependente aflate în așteptare în stațiile de rezervare. La

returnarea datei reale din memorie, aceasta este comparată cu valoarea prezisă, și instrucțiunile dependente executate speculativ fie urmează parcursul normal - nivelul Write Back al structurii pipe - fie sunt retrimise spre execuție. Tabela VHT este actualizată prin incrementarea automatului în cazul unei predicții corecte respectiv decrementarea acestuia în cazul unei predicții greșite și introducerea datei citite din memorie, evacuând din locația respectivă cea mai veche valoare. Lista valorilor implementează o structură de tip coadă (pentru reținerea celor mai recente *history* valori). În cazul predictorului *Last Value*, istoria fiind egală cu 1, se evacuează de fapt ultima valoare.

Rezultatele simulării (gradul de vecinătate al valorii, acuratețea predicției) împreună cu parametrii arhitecturii vor fi scriși într-un fișier (*simout.res*) în directorul curent (unde se află programele de test și simulatorul *sim-vpred.exe*). În cazul simulării unui predictor de valori este afișat *numărul Load-urilor a căror valoare a fost prezisă corect*. Alte date care arată *eficiența automatului* implementat sunt (încrederea oferită de acesta): numărul Load-urilor **clasificate predictibile** - generate de automatul de predicție, numărul Load-urilor **clasificate nepredictibile** - generate de automatul de predicție, numărul Load-urilor **predictibile** care au fost **prezise corect** (automatul a clasificat valoarea predictibilă iar valoarea propusă de automat a fost exact cea care se aducea din memorie). Analog pentru cele **nepredictibile prezise greșit** (în sensul că automatul a clasificat valoarea ca nepredictibilă iar valoarea propusă a fost într-adevăr diferită de cea adusă din memorie).

În ce privește interfața grafică a simulatorului, pentru lansarea acestuia în execuție este nevoie de un sistem pe care să fie instalat Windows 9x sau Windows 2000 și să existe benchmark-urile SPEC (programele de test). Din motive ce țin de sistemul de operare Windows 9x (preluarea identicatorului unui proces părinte - lucru care se face diferit în Windows NT) aplicația nu poate fi rulată pe Windows NT. Aceasta se datorează funcțiilor API folosite (*CreateToolhelp32Snapshot* - care creează o listă cu toate procesele aflate în execuție în momentul respectiv, *Process32First* - care determină primul proces din lista anterior creată și *Process32Next* - care determină următorul proces din listă ce respectă o anumită condiție), funcții care nu au corespondent în Windows NT. Pentru funcționarea corectă a aplicației "*Value Predictor*" procesul care stă în spatele aplicației (**sim-vpred.exe**), de importanță vitală, poate fi compilat / asamblat / link-editat atât sub sistemul de operare Windows NT / Windows 2000 cât și sub Windows 9x. Practic toate opțiunile de simulare aferente procesului *sim-vpred.exe* își au corespondent în controalele de tip editare, checkbox, listă din aplicația vizuală. De asemenea, rezultatele generate de *sim-vpred.exe*

sunt preluate din fișierul *simout.res* în controalele aplicației *Value Predictor* pentru a înlesni înțelegerea și prelucrarea ulterioară a acestora. Trebuie specificat că, din punct de vedere cronologic, cercetarea aferentă acestui capitol a fost făcută anterior celei din subcapitolul 5.3; se observă astfel o îmbunătățire a instrumentelor, modului de simulare și exploatare a rezultatelor în cazul salturilor indirecte.

La lansarea în execuție a aplicației, pe ecran apare o fereastră de configurare înzestrată cu un meniu principal (figura 6.1) și se poate trece la introducerea parametrilor simulării.

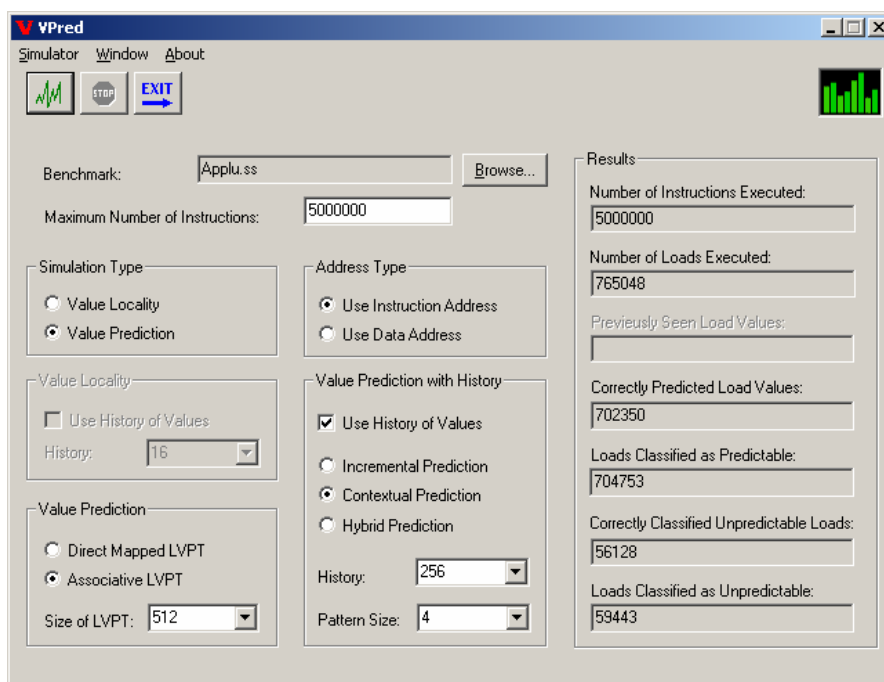


Figura 6.1. Fereastra de configurare a simulatorului *Value Predictor*

Unul din cei mai importanți parametri este benchmark-ul simulat. La apăsarea butonului *Browse*, se deschide o fereastră de dialog obișnuită (similară cu *OpenFile* la Microsoft Word) care permite selecția și deschiderea unui benchmark. Limitarea duratei simulării sau a numărului de instrucțiuni care să fie executate se poate face prin introducerea unui număr maxim de instrucțiuni. În cazul în care această limitare nu este făcută, simularea poate dura mai multe ore, sau chiar zile, în funcție de configurația calculatorului pe care se lucrează, până la execuția tuturor instrucțiunilor din benchmark-ul selectat. Utilizatorul poate să aleagă tipul de simulare (determinarea localității valorilor sau predicția valorilor), adresa care să fie

utilizată (adresa instrucțiunii sau adresa datei), “adâncimea” istoriei, tipul tabelii de predicție (mapată direct sau asociativă) și dimensiunea acesteia. În cazul în care se face predicție fără istorie, predictorul simulat este de tip *last value*. Dacă predicția se face cu istoria valorilor utilizatorul trebuie să aleagă tipul predictorului care poate fi incremental, contextual sau hibrid. În cazul unei predicții contextuale sau hibride, trebuie aleasă dimensiunea *pattern*-ului de căutare. Trebuie precizat că predictorul incremental implementat este de tip “2-delta”, iar cel contextual este de tip PPM complet (Prediction by Partial Matching).

Bazat pe simulări laborioase se poate observa că, rezultatele obținute prin predicție sunt mai slabe decât cele generate prin localitate și datorită faptului că tabelele de predicție sunt limitate dar și datorită modului de implementare al arhitecturilor de predicție (mapate direct, asociative).

6.1.2. IMPLEMENTAREA TIMING-ULUI ÎN CADRUL SIMULATORULUI.

În acest subcapitol se propune dezvoltarea unui model teoretic și practic de evaluare a performanței arhitecturilor pipeline cu execuții multiple, bazat pe implementarea schemelor de predicția valorilor. Practic se urmărește determinarea câștigului de performanță obținut (“*speed-up*”) de către o microarhitectură speculativă care înglobează tehnica de predicție a valorilor instrucțiunilor de tip Load. S-a ales acest tip de instrucțiune datorită latenței sale ridicate de execuție în condițiile accesului la memoria principală. Pentru obținerea *speed-up*-ului sunt parcurse următoarele etape:

- ☐ Este simulată execuția unui număr N (foarte mare) de instrucțiuni (poate fi chiar numărul total de instrucțiuni de pe un anumit benchmark), pe o arhitectură standard (generică) și se determină numărul total de cicluri de execuție – fie acesta T_1 . Rata de procesare (performanța globală a arhitecturii) este dată de ecuația: $IR_1 = \frac{N}{T_1}$.
- ☐ Este simulată execuția aceluiași număr de instrucțiuni (N), pe același benchmark, însă pe o microarhitectură speculativă care implementează predicția valorilor aferentă instrucțiunilor Load. Evident că prin predicția cu acuratețe a unui procentaj din instrucțiunile Load (în funcție de tipul de predictor folosit acesta este mai mult sau mai puțin semnificativ) timpul total de execuție al instrucțiunilor, de această dată T_2 , este cu siguranță mai mic decât T_1 . Rezultă că rata de procesare devine

$IR_2 = \frac{N}{T_2} > \frac{N}{T_1}$ și deci o îmbunătățire a performanței globale de procesare.

▣ Speed-up-ul obținut este: $S = \frac{IR_2 - IR_1}{IR_1} \cdot 100 [\%] = \frac{T_1 - T_2}{T_2} \cdot 100 [\%]$

Tipurile de predictoare de valori folosite (*Last Value, Incremental 2-delta, PPM complet și hibrid*) au fost cele prezentate în subcapitolul 2.2.2 și implementate în modulul *Value Predictor* descris în subcapitolul anterior (6.1.1). Unul din obiectivele cercetării l-a constituit determinarea corelației existente între acuratețea de predicție a valorilor și speed-up prin implementarea predictoarelor mai sus menționate, și de asemenea, stabilirea tipului de predictor (configurație optimă) pentru care se obține cel mai mare câștig de performanță.

Din punct de vedere al parametrilor din linia de comandă, față de cei prezentați în subcapitolul 6.1.1 se mai introduce o singură opțiune de simulare (**-speedup**), care indică dacă este **0** – o arhitectură standard iar dacă este **1** – o arhitectură care înglobează predicția valorilor.

Principalele modificări în implementare față de modulul *Value Predictor* s-au realizat la sfârșitul modulului *sim-main.c* în momentul în care simularea s-a încheiat și devin cunoscute numărul total de instrucțiuni Load, proporția de Load-uri corect și respectiv greșit predicționate. Sunt introduși câțiva parametri care reflectă realismul implementării. Astfel, este binecunoscut faptul că o structură asociativă este mai complexă, deci impune un timp de căutare mai mare decât în cazul celor mapate direct. În acest sens, s-a convenit ca timpii de căutare în orice tip de predictor asociativ să fie dublul timpilor de căutare în predictoarele cu arhitectură mapată direct. De asemenea, întrucât complexitatea predictorului incremental este inferioară celui contextual, care la rândul său este inferioară complexității celui hibrid rezultă că și timpii necesari furnizării predicției de către fiecare structură în parte, respectă aceeași relație de ordine. Valorile implicite pentru acești timpi sunt 2 pentru incremental, 5 pentru contextual și 7 (2+5) pentru cel hibrid. În cazul predictorului hibrid am ales ca implicit cazul cel mai defavorabil, întrucât în realitate predictorul hibrid nu chiar serializează cei doi timpi ($T_1=2$ și $T_2=5$). Timpul de acces la memoria principală, și el parametrizabil, preia ca valoare implicită 18.

Factor_Complexitate_Arhitectura $\in \{ 1, 2 \}$

T_{acces_DRAM}=18;

T_{acces_incremental} = **Factor_Complexitate_Arhitectura** * 2;

T_{acces_contextual} = **Factor_Complexitate_Arhitectura** * 5;

```

Tacces_hibrid = Factor_Complexitate_Aritectura * 7;
Tacces_predictor ∈ { Tacces_incremental, Tacces_contextual, Tacces_hibrid }

/*s-a simulat execuția în întregime a benchmark-ului s-au a cel puțin a
max_inst instrucțiuni*/
if (max_insts && sim_num_insn >= max_insts)
{
    if(isAssoc) /****** Structură asociativă de predicție *****/
        Factor_Complexitate_Aritectura = 2;
    else /****** Structură de predicție mapată direct *****/
        Factor_Complexitate_Aritectura = 1;

    /* Indiferent dacă este vorba despre simularea microarhitecturii standard – fără
    predicția valorilor sau despre microarhitectura speculativă care înglobează
    modulul de predicție – trebuie calculat timpul de referință T1 */
    npenload=sim_num_loads* Tacces_DRAM; /* execuția instrucțiunilor Load presupune
    acces la memoria principală, celelalte instrucțiuni
    fiind executate într-un timp unitar */
    T1=(sim_num_insn - sim_num_loads) * 1 + npenload;

    if(vsppedup) /* simularea microarhitecturii speculative – predicția valorilor
    instrucțiunilor de tip Load */
    {
        /* Determinarea timpului de acces per instrucțiune în funcție de tipul
        arhitecturii – mapată direct sau asociativă și respectiv în funcție de
        tipul de predictor selectat */
        if(contextual==0)
            Tacces_predictor=Tacces_incremental=Factor_Complexitate_Aritectura*2;
        else
            if(contextual==1)
                Tacces_predictor=Tacces_contextual=Factor_Complexitate_Aritectura*5;
            else
                Tacces_predictor = Tacces_hibrid = Factor_Complexitate_Aritectura * 7;
            npenload=0;
            if (notfoundA!=0) /* numărul instrucțiunilor Load care au accesat cu miss
            tabela de predicție */
                npenload=npenload+notfoundA* Tacces_DRAM;
            if(foundA!=0) /* numărul instrucțiunilor Load găsite în tabelă și
            predicționate corect */
                npenload=npenload+foundA * Tacces_predictor;
            if(foundA_miss!=0) /* numărul instrucțiunilor Load găsite în tabelă și
            greșit predicționate */
                npenload=npenload+foundA_miss * Tacces_DRAM;
            T2=(sim_num_insn-sim_num_loads)*1+ npenload;
            S=(T1-T2)/T2*100;
        }
    }
}

```

Algoritmul descris anterior face posibilă calcularea latențelor de execuție $T1$ și $T2$ esențiale în determinarea speed-up-ului S și implicit generarea concluziilor privitor la fiecare tip de predictor, arhitectură.

Se poate afirma că, cercetarea efectuată reprezintă mai mult decât o estimare teoretică întrucât se bazează atât pe calcul analitic cât și pe simulare, însă metodologia este doar în parte realistă. Practic, timpul total de execuție se bazează pe simularea unei microarhitecturi speculative, care înglobează un predictor de valori, și determinarea acurateții de predicție, a numărului de accese cu miss la tabela VHT (vezi figurile 2.14, 2.17, 2.18), a numărului de Load-uri care accesează cu hit tabela dar sunt greșit predicționate. Fiecărui dintre acești parametri li se asignează câte un timp estimativ de execuție (în caz de *hit* sau *recovery*) conform tipului de predictor (contextual / incremental / hibrid) și modelului arhitectural (mapat direct / asociativ) implementat cu care contribuie la timpul total de execuție. Evident că astfel, predictorul cel mai performant din punct de vedere al acurateții predicției se va dovedi optim și din punct de vedere al ratei globale de procesare. Rezultatele simulărilor (după cum se va putea vedea în capitolul 7.2) indică, indiferent de tipul arhitecturii, superioritatea predictorului hibrid și din punct de vedere al câștigului de performanță obținut (până la 20% speed-up).

În finalul acestei modeste cercetări referitoare la introducerea timing-ului la nivelul simulatorului „*Value Predictor*” trebuie ca, toate concluziile teoretice să fie comparate cu cele obținute pe bază de simulare complexă (prin folosirea ca bază de cercetare a simulatorului superspeculativ și cu execuție *out-of-order* și nu a unuia funcțional *sim-bpred*, cum a fost cazul de față – vezi justificarea în capitolul 3). Astfel, se va avea un control asupra acestor metode teoretice dar se va putea stabili și până la ce punct rămân ele realiste.

6.2. VECINĂTATEA ȘI PREDICȚIA VALORILOR CENTRATĂ PE CONTEXTUL CPU.

O evaluare originală, prezentată inițial în [Flo02] pune în evidență **conceptul de vecinătate a valorilor asociate regiștrilor generali aferenți procesorului MIPS**. Rezultate statistice bazate pe simulare au arătat că programele de uz general sunt caracterizate de repetiția valorilor [Lip96, Sod00]. Principalele cauze ale acestui fenomen le constituie: redundanța

datelor și codului, constantele din program, subrutinele compilatorului destinate rezolvării apelurilor virtuale de funcții, alias-urilor de memorie. Localitatea valorilor regiștrilor este frecvent întâlnită în programe și exprimă raportul dintre numărul situațiilor în care un registru este scris cu o valoare care a fost stocată anterior (într-o istorie dată) în același registru și numărul total de instrucțiuni care au acest registru ca destinație. Ecuația următoare (6.1) descrie metrica VL (localitatea valorii) utilizată în simulările efectuate:

$$VL_j(R_k) = \frac{\sum_{i=1}^n VL_j^k(i)}{\sum_{i=1}^n VRef^k(i)} \quad (6.1)$$

n = numărul de benchmark-uri utilizate în simulare (8 for SPEC'95 respectiv 7 for SPEC2000)

j = lungimea istoriei (4, 8, 16 respectiv 32)

k = numărul regiștrului

$VL_j^k(i)$ = Numărul situațiilor în care registru R_k este scris cu o valoare care a fost anterior înregistrată (egală cu una din ultimele j valori) în respectivul registru (pe benchmark-ul i).

$VRef^k(i)$ = Numărul total de instrucțiuni dinamice care au registru R_k ca și câmp destinație pe benchmark-ul i .

În [Vin05] autorii extind conceptul de predicție dinamică a valorilor, până atunci centrată pe instrucțiuni sau memorie, introducând conceptul de **predicție a valorilor focalizată pe regiștrii procesorului**. Localitatea valorii pe anumiți regiștri speciali ai arhitecturii MIPS este remarcabilă (cca. 90%), conducând în mod evident la ideea predicției valorilor cel puțin pentru acești regiștri. La baza acestor afirmații stau caracteristicile setului de regiștri generali ai procesorului MIPS [Flo02].

Arhitectura SimpleScalar este derivată din MIPS-IV ISA. Unitatea centrală de procesare a MIPS conține 32 de regiștri de uz general numerotați de la 0 la 31. Regiștrul n este desemnat ca $\$n$.

☐ Regiștrul $\$0$ este întotdeauna cablat la valoarea 0.

☐ Regiștrul $\$at$ (1) este rezervat asamblorului în timp ce $\$k0$ (26) și $\$k1$ (27) sunt dedicați sistemului de operare, neputând fi folosiți în programe utilizator. Regiștrul $\$at$ este des folosit în calcularea adresei de memorie pentru diverse structuri de date (instrucțiunea **lui** (*load upper immediately*)- încarcă doar partea semnificativă a unui cuvânt de 4 octeți

dintr-un registru sursă în $\$at$) și generează rar mai mult de o valoare unică (a se vedea harta de memorie a procesorului MIPS din subcapitolul 3.4).

- ☐ Regiștrii $\$a0$ - $\$a3$ (4-7) sunt folosiți pentru a transmite primele patru argumente rutinelor apelate (argumentele rămase fiind transmise stivei). Regiștri $\$v0$ și $\$v1$ (2, 3) păstrează rezultatele returnate de funcții. MIPS asigură un set redus de servicii ale sistemului de operare prin instrucțiuni. Pentru a apela un serviciu, programul încarcă codul apelului în registrul $\$v0$ și argumentele în registrele $\$a0$ - $\$a1$.
- ☐ Registrul $\$gp$ (28) este un pointer global care indică spre mijlocul blocului de 64k de memorie în segmentul de date statice.
- ☐ Registrul $\$sp$ (29) este pointerul de stivă, care indică spre prima locație liberă din stivă - variază într-un domeniu redus de valori. Registrul $\$fp$ (30) este "*pointer de cadru*". Un *cadru* constă în memoria dintre indicatorul de cadru ($\$fp$), care pointează la cuvântul imediat următor ultimului argument transmis pe stivă, și indicatorul de stivă ($\$sp$), care pointează la primul cuvânt liber pe stivă. Tipic pentru sistemele UNIX, stiva crește în jos de la adrese de memorie mai mari, astfel încât indicatorul de cadru este deasupra indicatorului de stivă. Instrucțiunea *jal* (*jump and link*) setează registrul $\$ra$ (31) cu adresa de revenire dintr-un apel de procedură fiind predictibilă dacă apelul din același punct se repetă cel puțin o dată. Gradul de localitate ridicat al ultimului registru se datorează numărului redus de apeluri de subrutine existent în programele procedurale de calcul (benchmark-uri) [Flor04] – vezi și figurile din subcapitolul 7.3.1.

Convenția de utilizare a regiștrilor MIPS furnizează regiștri "*de salvare*" *callee*- (apelați) și *caller*- (apelanți), avantajosi în circumstanțe diferite.

- Regiștrii de salvare (*callee*) $\$s0$ - $\$s7$ (16-23) sunt utilizați cu precădere pentru a reține valori de lungă durată care trebuie păstrate de-a lungul apelului, ca de exemplu variabile globale dintr-un program utilizator. Acești regiștri sunt salvați în timpul unui apel de procedură doar dacă procedura apelată dorește să utilizeze valorile reținute de aceștia.
- Pe de altă parte, regiștri de salvare (*caller*) $\$t0$ - $\$t9$ (8-15, 24, 25) sunt utilizați în special pentru a reține valori temporare care trebuie păstrate în timpul unui apel, ca de exemplu valori imediate în calculul unei adrese, parametrii efectivi de apel. În timpul unui apel, procedura apelantă poate folosi acești regiștri pentru stocarea valorilor temporare de scurtă durată.

Ideea asocierii câte unui predictor de valori pentru anumiți regiștri – deci predictoare centrate pe regiștri și nu pe instrucțiuni – originală după știința autorului, ar putea implica tehnici arhitecturale novatoare – structuri

de predicție mult mai simple (32-128 de locații), și, în consecință, performanțe îmbunătățite și costuri mai reduse ale microarhitecturilor speculative. Până acum, predicțiile se făceau centrat pe adresa instrucțiunii sau adresa datei în cazul instrucțiunilor de tip Load, cu implicații defavorabile asupra costurilor și complexității. Tehnica de predicție a valorii regiștrilor constă în predicția următoarei valori a unui registru destinație, în timpul celei de a doua jumătăți a fazei de decodificare, bazat pe valorile sale anterioare (cele mai recente într-o istorie dată). Instrucțiunile dependente succesoare folosesc valoarea prezisă iar execuția speculativă este validată când valoarea corectă (reală) a registrului devine cunoscută, după faza de execuție (vezi figura 6.2). Dacă valoarea prezisă este corectă atunci calea critică ar putea fi redusă, cu implicații benefice asupra timpului total de execuție, respectiv a performanței globale de procesare (IPC). În caz contrar, instrucțiunile executate cu intrări eronate trebuie reluate din nou (proces de recovery).

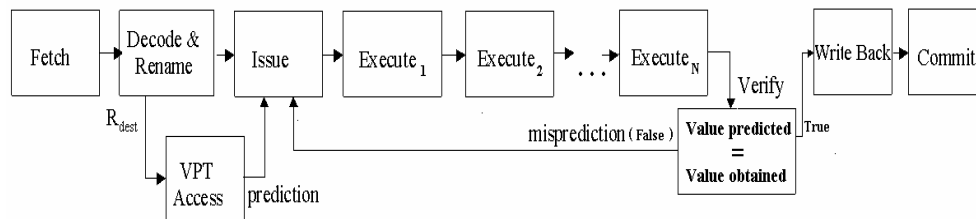


Figura 6.2. Implementarea mecanismului de predicție a valorilor regiștrilor în structura pipeline aferentă unei microarhitecturi generice

În [Vin05] au fost examinate submulțimi a regiștrilor favorabili folosindu-se diferite structuri de predicție a valorilor pentru a exploata în mod optim caracteristicile de predictibilitate a valorilor din benchmark-urile SPEC ('95 și 2000) – vezi subcapitolul 6.2.1. Rezultatele obținute demonstrează existența unei corelații dinamice între numele regiștrilor folosiți ca destinație și valorile înmagazinate în acești regiștri. Există însă și unele dezavantaje: adresarea tabelii de predicție cu numele registrului destinație (în timpul fazei de decodificare) în locul adresei instrucțiunii (PC) va determina o serie de interferențe.

6.2.1. PREDICTOARE CLASICE (LASTVALUE, INCREMENTAL, CONTEXTUAL, HIBRID)

Pe parcursul acestui subcapitol vor fi prezentate structurile de predicție dezvoltate pentru exploatarea conceptului nou introdus. În realitate respectivele scheme sunt cele utilizate în predicția valorilor centrată pe instrucțiuni și prezentate în subcapitolul 2.2.2, fiind doar adaptate la predicția valorilor regiștrilor. Principala deosebire între cele două tipuri de structuri se referă la capacitatea de memorare a acestora. De asemenea, în timp ce schemele de predicție a valorii instrucțiunilor sunt adresate în timpul fazei de aducere a instrucțiunilor prin intermediul PC-ului acestora, schemele de predicție a valorii regiștrilor, sunt indexate în cea de-a doua jumătate a fazei de decodificare, după cunoașterea numelor regiștrilor destinație.

Valoarea prezisă de o structură de tip *Last Value* (figura 6.3) pentru un registru este identică cu ultima valoare stocată în respectivul registru. Fiecare intrare în tabela de predicție VHT (figura 6.3) deține propriul automat, care este incrementat cu fiecare predicție corectă și decrementat în caz contrar (istorie locală a comportamentului fiecărui registru). Utilitatea tehnicii de predicție a valorilor este evidentă doar în cazul unei predicții corecte, altfel putând cauza hazarduri structurale cu repercusiuni negative asupra timpului total de execuție al instrucțiunilor. Bazat pe comportamentul dinamic al conținutului regiștrilor a fost dezvoltat un mecanism de clasificare (implementat prin automatul mai sus amintit) a stării regiștrilor: predictibili și nepredictibili. Prin tratarea separată a fiecărui grup de regiștri poate fi evitat costul unor predicții greșite. Tranzițiile automatului vor fi realizate în concordanță cu rezultatul comparației dintre valoarea prezisă de tabela VHT și valoarea reală rezultată în urma execuției. Dimensiunea tabelii de predicție este egală cu numărul regiștrilor logici ai procesorului.

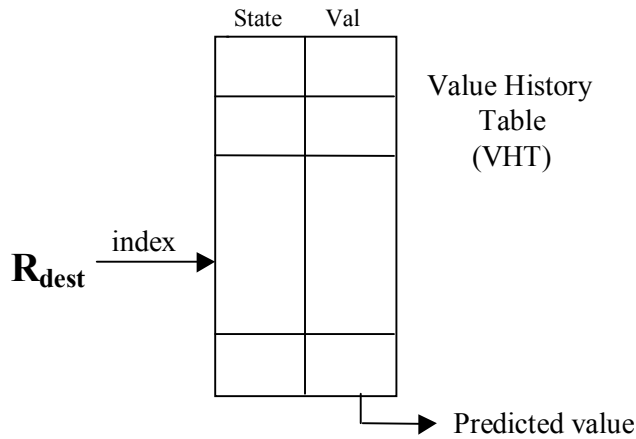


Figura 6.3. Predictor de tip *Last Value*

În cazul predictorului incremental, considerând că v_{n-1} și v_{n-2} sunt cele mai recente valori produse, noua valoare v_n va fi calculată după formula de recurență: $v_n = v_{n-1} + (v_{n-1} - v_{n-2})$, unde $(v_{n-1} - v_{n-2})$ este pasul secvenței. Predictorul implementat și prezentat în figura 6.4 este incremental de tip „2 – delta” (strategia de actualizare a pasului fiind bazată pe histerezis). În cazul acesteia sunt memorați doi pași (str_1 și str_2), $str_1 = v_n - v_{n-1}$. Numai atunci când aceeași valoare str_1 a apărut de două ori consecutiv, se face transferul $str_2 \leftarrow str_1$. Astfel, numărul predicțiilor eronate este redus de la două, la doar una, în cazul secvențelor incrementale repetitive.

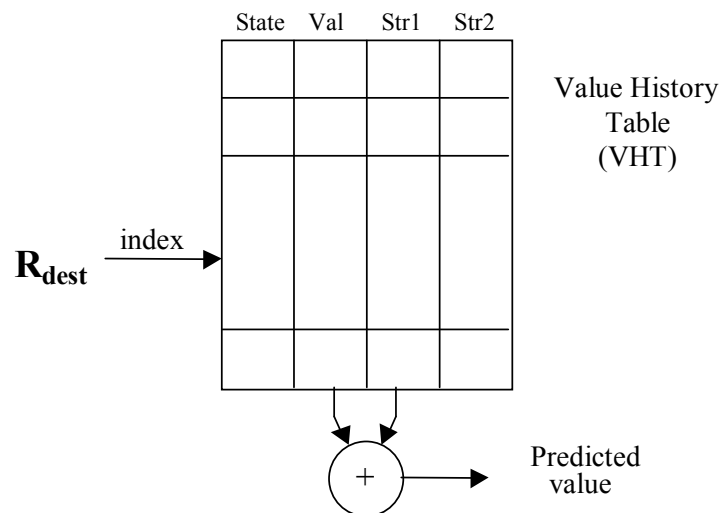


Figura 6.4. Structură de predicție incrementală – „2-delta”

Când un registru este folosit pentru prima dată ca și destinație a unei instrucțiuni va rezulta un *miss* în tabela de predicție VHT, și nu se va face nici o predicție. Valoarea registrului rezultat va fi stocată în câmpul „Val” al tabelii iar automatul trece în starea inițială „*nepredictibilă*”. Automatul de stare folosit (câmpul *State* din figura 6.4) este cel descris în cadrul capitolului referitor la predicția valorilor instrucțiunilor (vezi figura 2.16). Dacă respectivul registru este folosit din nou ca destinație nu se face nici o predicție dar se calculează pasul $S_1=V_1\text{-Val}$, și se introduc V_1 și S_1 în câmpurile Val și Str₁. Dacă pasul S_1 este diferit de 0 atunci automatul trece într-o stare tranzitorie, altfel rămâne în aceeași stare inițială. Dacă același registru este rescris din nou nu se face nici o predicție, dar se calculează pasul $S_2=V_2\text{-Val}$. Valoarea din câmpul „Str₁” este introdusă în câmpul „Str₂” iar V_2 și S_2 în câmpurile Val și Str₁. Dacă pasul nou calculat (S_2) coincide cu cel anterior S_1 atunci automatul va trece într-o stare de stabilitate (*predictibilă*), altfel va rămâne în starea tranzitorie dar tot (*nepredictibilă*). Din acest moment, la o nouă re folosire a respectivului registru cu rol de destinație, dacă Str₁= Str₂ se face predicție, valoarea prezisă fiind calculată ca sumă dintre ultimul pas memorat Str₂ și valoarea din câmpul Val. Automatul rămâne în starea de stabilitate atâta timp cât pasul este constant, altfel trecând în starea de tranziție.

Predictoarele bazate pe context (contextuale) predicționează valoarea care va fi stocată într-un registru în funcție de ultimele valori reținute de către respectivul registru. Contextul este reprezentat de o secvență finită de valori cu apariție repetată asemenea unui lanț Markov. Predictoarele care implementează algoritmul de predicție prin potrivire parțială – PPM complet constituie o importantă clasă de predictoare contextuale. Blocul PPM din figura 6.5 este identic cu cel descris în figura 5.26 și a fost utilizat cu succes și în cazul predicției target-urilor salturilor indirecte (vezi subcapitolele 5.2÷5.3). Predictorul PPM complet cuprinde N+1 predictoare Markov (contextuale) de la ordinul 0 la ordinul N. Dacă predictorul Markov de ordinul N produce un rezultat valid atunci procesul se termină dacă nu se activează predictorul de ordin imediat inferior.

Valoarea prezisă reprezintă valoarea care urmează contextului cu cea mai mare frecvență de apariție. Din acest considerent se poate concluziona că valoarea prezisă depinde de context. Cu cât acesta este mai „bogat” („lung”) cu atât mai mult sunt șanse de a obține o acuratețe de predicție mai ridicată. Există cazuri însă când, un context „exagerat” de mare poate conduce la diminuarea acurateții de predicție, comportându-se practic ca zgomot.

În figura 6.5 este prezentată structura predictorului implementat bazat pe context. Fiecare intrare din tabela VHT are asignată un automat de

clasificare, incrementat în cazul unei predicții corecte și decrementat în cazul unei predicții greșite. Câmpurile V_1, V_2, \dots, V_4 rețin ultimele patru valori (contextul), asociate registrului cu care se indexează structura la un moment dat. Dacă automatul – similar cu cel prezentat în figura 5.28 – se află într-una din cele două stări predictibile, atunci structura va predicționa valoarea care urmează cu cea mai mare frecvență contextului.

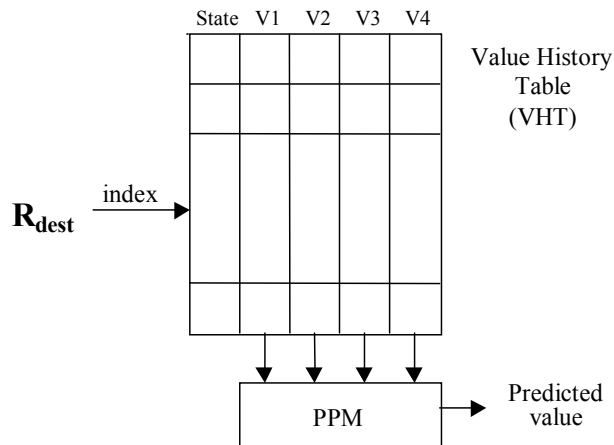


Figura 6.5. Predictor contextual de tip PPM complet de ordin 4

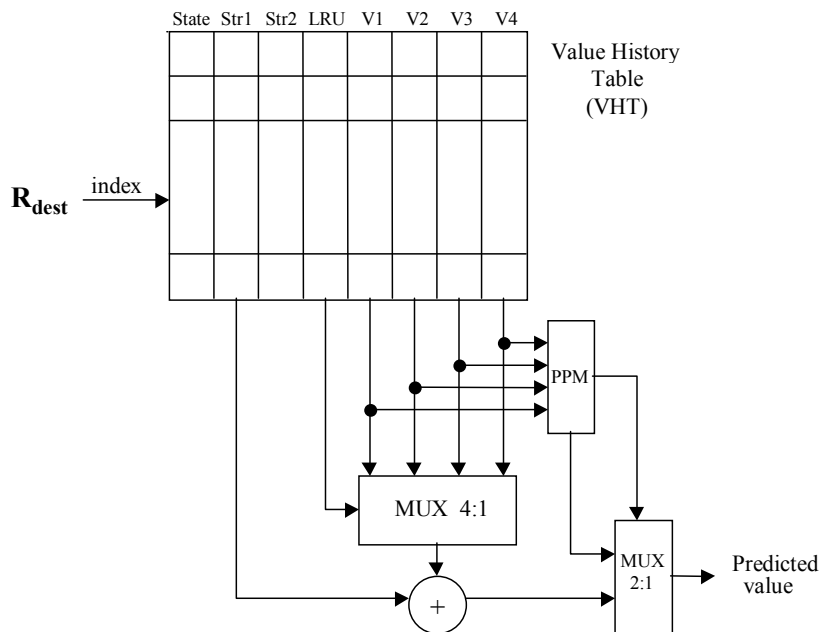


Figura 6.6. Predictor hibrid (contextual & incremental)

Studiile cercetătorilor din domeniul predicției valorilor [Wang97], dar și implementările comerciale existente la nivelul predicției salturilor condiționate [Sez02, Kes99] arată în mod clar că un singur tip de predictor nu dă în general rezultatele cele mai bune. Acest lucru se datorează în primul rând faptului că anumite tipuri de secvențe de valori generate prin program sunt prezise mai bine de un anumit predictor, iar altele, de către un alt tip de predictor particular [Saz99]. Din acest motiv a apărut ca naturală ideea predicției hibride, adică două sau mai multe predictoare de valori să conlucreze în mod dinamic în procesul de predicție. În figura 6.6 este prezentată structura unui predictor hibrid compus dintr-un predictor bazat pe context și unul incremental. Predictorul contextual are întotdeauna prioritate, astfel încât predictorul incremental va putea genera o predicție doar dacă cel bazat pe context nu poate realiza acest lucru. Cu toate că, acuratețea predicției valorilor regiștrilor procesorului generată de predictorul hibrid implementat este superioară celor obținute cu celelalte structuri de predicție descrise anterior pe parcursul acestui subcapitol, această prioritizare statică, neadaptivă, în alegerea tipului de predictor ce urmează a fi folosit, pare a fi neoptimală. O soluție pentru rezolvarea acestei probleme ar putea consta în implementarea unor metapredictoare (vezi subcapitolul următor – 6.2.2), care selectează dinamic, bazat pe diverse grade de încredere, structura care să fie utilizată la un moment dat pentru predicție.

O exemplificare convingătoare pe o secvență de program MIPS, însoțită de un predictor de tip "*LastValue*" modificat astfel încât să memoreze ultimele (2 sau 4) valori ale fiecărui registru este realizată în **Anexa 2**.

Ecuția următoare (6.2) descrie metrica VP (acuratețea de predicție a valorii) utilizată în simulările efectuate:

$$PA(R_k) = \frac{\sum_{i=1}^n CPV^k(i)}{\sum_{i=1}^n VRef^k(i)} \quad (6.2)$$

n = numărul de benchmark-uri utilizate în simulare (8 for SPEC'95 respectiv 7 for SPEC2000)

k = numărul registrului

$CPV^k(i)$ = Numărul valorilor corect predicționate ale registrului R_k (pe benchmark-ul i).

$VRef^k(i)$ = Numărul total de instrucțiuni dinamice care au registrul R_k ca și câmp destinație pe benchmark-ul i .

Utilizându-se predictorul cel mai performant dintre toate cele descrise în cadrul acestui subcapitol, și anume, cel hibrid, s-a realizat o selecție a regiștrilor procesorului în funcție de un anumit prag de acuratețe (60% respectiv 80%). Prin acest mecanism s-a încercat o reducere a numărului de regiștri asupra cărora poate fi aplicat conceptul de predicție dinamică a valorilor. Acest lucru este justificat în principal de posibile și probabile interferențe ale mai multor instrucțiuni la un anumit registru ceea ce determină o vecinătate scăzută a valorilor asociate acestuia și, în consecință, o predictibilitate de asemenea redusă.

Bazat pe simulări laborioase, s-a arătat că, beneficiind de o istorie suficient de bogată (ultimele 256 de valori), un predictor hibrid elimină problema interferențelor și poate atinge acurateți de predicție de 85.44% (medie aritmetică realizată pe 8 regiștrii MIPS folosind pentru test benchmark-urile SPEC'95), existând și cazuri particulare în care acuratețea ajunge la 96%, și respectiv 73.52% (medie aritmetică obținută pe 16 regiștri MIPS folosind benchmark-urile SPEC 2000). În ciuda rezultatelor relativ modeste privind acuratețea, principalul avantaj al tehnicii de predicție a valorii centrată pe regiștrii procesorului îl reprezintă deblocarea instrucțiunilor dependente de date aflate în coadă de așteptare.

6.2.2. PREDICTOR HIBRID CU SELECȚIE BAZATĂ PE METAPREDICTOARE. DESCRIERE SIMULATOR.

Predictoarele hibride implementate până în prezent (vezi 6.2.1) acordau întotdeauna prioritate predictorului contextual, cel incremental fiind folosit doar atunci când acesta nu putea genera predicție, iar cel de tip Last Value fiind utilizat doar ca ultimă soluție. Se poate afirma că funcționarea acestor tipuri de predictoare se bazează pe o prioritizare statică, neoptimală, în alegerea tipului de predictor ce urmează a fi folosit în procesul de predicție. O soluție pentru rezolvarea acestei probleme ar fi implementarea diferitor metapredictoare (figura 6.7), care selectează dinamic, bazat pe diverse grade de încredere, structura care să fie utilizată la un moment dat pentru predicție. Va avea prioritate predictorul care va fi declarat învingător de către metapredictor.

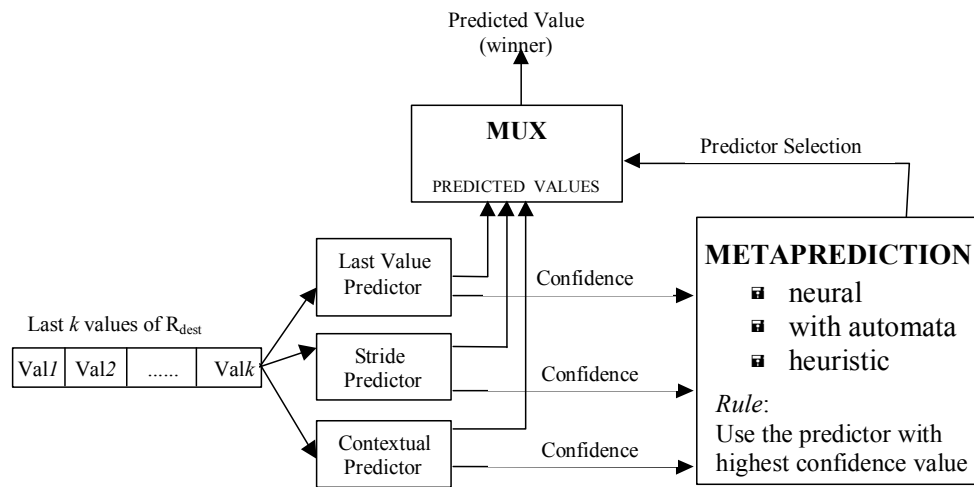


Figura 6.7. Arhitectura folosită pentru metapredicție

Astfel, metapredicțiile reprezintă un nivel suplimentar în procesul de predicție, responsabil în alegerea tipului de predictor folosit la un moment dat în procesul de predicție. Următorul paragraf încearcă să explice noțiunea de *metapredicție*.

Un predictor hibrid de valori centrat pe instrucțiune combină două sau mai multe predictoare componente, care predicționează fiecare la rândul său rezultatul instrucțiunii curente. Structura hibridă necesită un mecanism de selecție pentru a alege (*predictiona* – întrucât nu știe în acel moment dacă a făcut alegerea corectă) dintre predictoarele componente pe cel mai potrivit pentru generarea rezultatului instrucțiunii în fiecare ciclu de aducere a acesteia. Această predicție (selecție / influență asupra) a acurateții de predicție este denumit în literatură *metapredicție*.

În acest subcapitol au fost propuse două tipuri de metapredictoare: ne-adaptive (stabile), prin atașarea unui automat de confidență fiecărui predictor, și respectiv unul adaptiv (dinamic), care utilizează o rețea neurală de tip feedforward. Cele trei predictoare de valori componente, fiecare centrate pe regiștrii procesorului sunt: un predictor *last value*, unul *incremental* și unul *bazat pe context*. Fiecare predictor component generează două informații la ieșire: *valoarea prezisă* (rezultatul instrucțiunii care se înscrie în registrul supus predicției) și *confidența* acestuia. Mecanismul de confidență introdus impune o **ignorare selectivă a efectuării unor predicții** de către anumite predictoare componente (cele cu confidența mai mică decât un prag apriori stabilit) și efectuarea predicției de către cel mai potrivit la momentul respectiv, cu scopul de a îmbunătăți acuratețea globală

de predicție. O confidență ridicată semnifică încredere în predicție (șanse mari ca predicția să fie corectă întrucât și în precedentele iterații predictorul component a prezis corect – chiar dacă valoarea nu a fost neapărat înaintată instrucțiunilor dependente favorizând execuția speculativă). Metapredictorul reprezintă un al doilea nivel în procesul de predicție, responsabil în alegerea tipului de predictor folosit la un moment dat, bazat pe comportamentul anterior al fiecăruia, ilustrat prin confidență.

Rezultatele experimentale bazate pe simulări laborioase (vezi capitolul 7.3.2) arată că printr-o astfel de structură de metapredicție, aplicată la patru din cei mai favorabili regiștri ai procesorului, se obține o creștere a acurateții predicției de 2.27%. Acuratețea maximă de predicție obținută pe regiștrii MIPS a fost 95.64% pentru regiștrul care păstrează adresa de revenire din proceduri și funcții (\$ra).

Metapredictor empiric neadaptiv

Primul metapredictor neadaptiv care se prezintă, numit de autor *empiric*, selectează unul din predictoarele componente bazat pe comportamentul fiecăruia din ultimele k predicții (corecte respectiv eronate). Istoria comportării fiecărui tip de predictor poate fi privită sub forma unui vector ale cărui elemente pot lua valori logice de 0 sau 1 (regiștru binar de deplasare). Un element de 0 indică faptul că predictorul care are asociat acest vector a prezis greșit, iar un element de 1 arată că predictorul asociat a prezis corect. Dimensiunea acestui vector reprezintă cât de multă istorie va fi analizată în procesul de predicție.

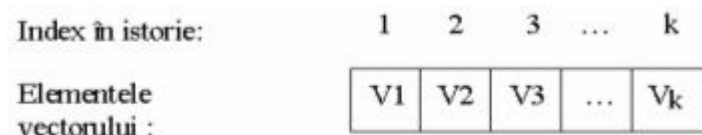


Figura 6.8. Istoria ultimelor k comportări asociată unui predictor component

Pentru a selecta predictorul component care va face predicția, sunt comparați după un anumit criteriu, trei astfel de vectori, câte unul pentru fiecare tip de predictor: last value, incremental și contextual. Criteriul folosit în acest caz îl constituie numărul maxim de elemente de 1 din cadrul vectorului de istorie, adică maximum de predicții corecte efectuate. Câștigă astfel predictorul al cărui vector asociat conține mai mulți de 1. După identificarea vectorului cu număr maxim de 1, predicția propriu-zisă se realizează doar dacă numărul de 1 este mai mare decât un anumit prag impus prin program, prag care constituie nivelul de încredere.

Metapredictor neadaptiv empiric cu automat pe nivelul doi de încredere

Acest tip de metapredictor folosește două nivele de încredere. Primul este identic cu cel prezentat anterior, selectându-se astfel predictorul care are vectorul de istorie cu cei mai mulți de 1. În acest moment se verifică cel de-al doilea nivel de încredere pentru predictorul care a fost declarat învingător la pasul precedent. Nivelul al doilea de încredere îl reprezintă automatul de stare asociat fiecărui predictor component selectat, structura automatului fiind identică cu cea prezentată în figura 5.28. Predictorului ales anterior i se va verifica starea automatului, urmând să se efectueze predicție doar dacă această stare este mai mare sau egală cu un *threshold* impus prin program, cu alte cuvinte dacă automatul este într-o stare predictibilă. Practic cel de-al doilea nivel de încredere este global, per predictor și nu local per registru, de unde poate și sursa unor rezultate mai modeste. Cele două niveluri de încredere par să aibă rezultatele favorabile în același timp și nu prin completare. Când istoria comportamentului unui predictor conține puțini biți de 1 este clar că și starea automatului va fi una nepredictibilă, iar când predictorul conține mulți de 1 este destul de probabil ca aceștia să se afle pe poziții succesive și atunci și starea automatului să fie una predictibilă. Intuiția autorului este că, un al doilea nivel de încredere local (automat per registru) ar îmbunătăți acuratețea globală de predicție. Există situații când un tip de predictor să prezică bine doar anumiți regiștri și în rest să prezică eronat. În acest caz numărul de biți de 1 al vectorului asociat respectivului predictor va fi redus dar starea automatului de predicție asociat registrelor în cauză să fie predictibilă.

Metapredictor neadaptiv cu automat de stare

În cadrul acestui tip de metapredictor fiecare predictor component (last value, incremental, contextual) are asociat un automat de stare pe 2 biți (structură identică cu cea descrisă în figura 5.28). Automatul va avea patru stări și se va afla inițial în starea 0, fiind incrementat la fiecare predicție corectă a predictorului asociat, respectiv decrementat la fiecare predicție incorectă a acestuia. Pentru început se verifică starea fiecărui automat și se alege pentru predicție predictorul care are această stare a automatului maximă. Efectuarea predicției se va realiza doar dacă starea automatului asociat este mai mare sau egală cu un anumit *threshold* impus prin program.

Metapredictor adaptiv neuronal

În continuare este prezentată o idee nouă, un metapredictor adaptiv, implementat printr-o rețea neurală nerecurentă (*feedforward*) care va avea rolul de a selecta dinamic tipul de predictor care va realiza predicția (last

value, incremental și contextual). Avantajul major al rețelelor neurale constă în capacitatea acestora de a învăța din exemple (învățare supervizată). Prin capacitatea lor de sinteză, rețelele neurale reușesc ca pe baza exemplilor anterior învățate să dezvolte și să rezolve un model de problemă.

S-a ales tipul de rețea *Perceptron multistrat* (MLP) cu un strat ascuns și algoritmul de învățare *back-propagation* doar pentru a determina potențialul de performanță al ideii de metapredictor neuronal. Stratul de ieșire al rețelei va conține astfel trei neuroni ($P = 3$) câte unul corespunzător fiecărui tip de predictor. Valorile de ieșire sunt cuprinse în intervalul real $[0, 1]$, predicția curentă realizându-se de către predictorul a cărui valoare a neuronului asociat este maximă, dar doar în condițiile în care această valoare depășește un anumit prag apriori stabilit.

Stratul de intrare este format din N neuroni, $N = 3 \cdot k$, unde:

3 - numărul predictoarelor utilizate;

k - istoria (ultimele k comportări) aferente predictorului component respectiv (1 - predicție corectă, 0 - predicție greșită);

Stratul ascuns va conține M neuroni, unde $M = N + 1$, adică $3 \cdot K + 1$ neuroni. S-a ales această valoare întrucât în urma a laborioase simulări efectuate, pentru $M = N + 1$ se obțin cele mai bune rezultate din punct de vedere al acurateții de predicție. Figura 6.9 prezintă structura rețelei neurale implementate.

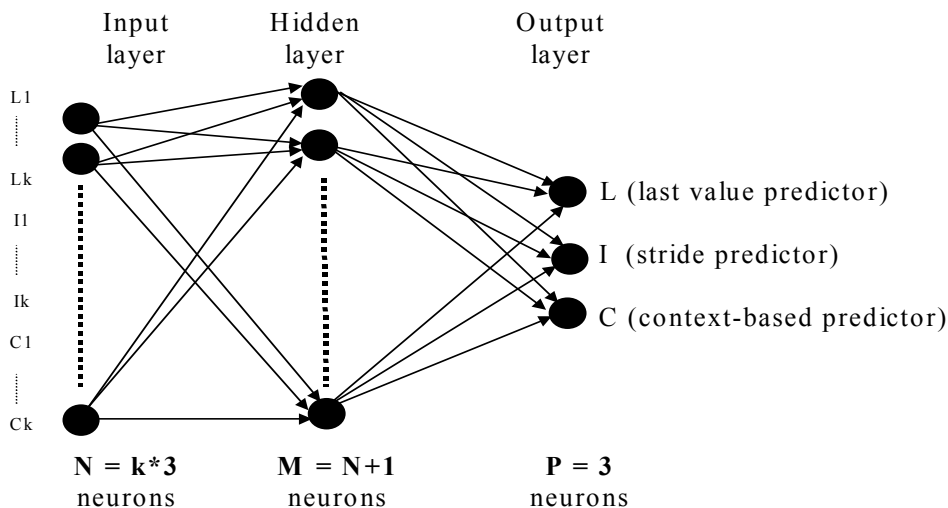


Figura 6.9. Structura rețelei neuronale

Modelul de învățare *backpropagation* este un algoritm de învățare folosit în rețelele de tip feedforward. Backpropagation definește doi pași:

primul (*forward*) în care informația trece de la intrare către ieșire și apoi un pas de la nivelul de ieșire la nivelul de intrare. Pasul de trecere înainte propagă vectorul de intrare în primul nivel al rețelei, ieșirile din acest nivel produc un nou vector care vor fi intrări pentru nivelul următor, până când se ajunge la ultimul nivel ale cărui ieșiri reprezintă ieșirile rețelei. Pasul înapoi (*backward*) este similar cu cel înainte, exceptând faptul că erorile sunt propagate înapoi prin rețea pentru a determina adaptarea ponderilor.

Întrucât caracteristicile algoritmului de învățare *backpropagation* au fost descrise în subcapitolul 5.1.2.2 în continuare vor fi prezentate doar etapele acestuia, adaptate la problema în cauză:

1. Se creează o rețea feedforward cu N intrări, un singur nivel intermediar cu $M = N+1$ neuroni și cu P neuroni în stratul de ieșire.
2. Se inițializează toate ponderile $W_{i,j}^1; i = \overline{1, N}; j = \overline{1, M}$ și $W_{i,j}^2; i = \overline{1, M}; j = \overline{1, P}$ cu valori mici aleatoare situate în intervalul $[0.3, 0.7]$. Funcția de activare utilizată este sigmoida: $F(x) = \frac{1}{1+e^{-x}}$.

În continuare termenul t_k reprezintă valoarea dorită la ieșire a neuronului k din stratul de ieșire, iar O_k este ieșirea obținută a aceluiași neuron.

3. *Until* $E(\overline{W}) = \frac{1}{2} \sum_{k \in \text{Outputs}(P)} (t_k - O_k)^2 \leq T$ (*threshold*), *do*:

- 3.1. Se plasează vectorul \overline{X} la intrările rețelei și se propagă înainte prin rețea, calculându-se vectorul de ieșire \overline{O} după formula de produs matriceal.

$$\overline{O} = \overline{X} \cdot \overline{W}^1 \cdot \overline{W}^2 \quad (6.3)$$

- 3.2. Pentru fiecare neuron din stratul de ieșire $k, k \in \overline{1, P}$ se calculează termenul de eroare δ_k :

$$\delta_k = O_k(1 - O_k)(t_k - O_k) \quad (6.4)$$

- 3.3. Pentru fiecare nod din stratul intermediar $h, h \in \overline{1, M}$ se calculează termenul de eroare δ_h :

$$\delta_h = O_h(1 - O_h) \sum_{k \in \text{Outputs}(P)} W_{k,h}^2 \cdot \delta_k \quad (6.5)$$

- 3.4. Se ajustează fiecare pondere $W_{i,j}$ a rețelei:

$$W_{i,j} = W_{i,j} + \Delta W_{i,j} \quad (6.6)$$

$$\Delta W_{i,j} = \alpha \cdot \delta_i \cdot X_{i,j} \quad (6.7)$$

unde α este pasul de învățare iar $X_{i,j}$ reprezintă nivelul de intrare dacă se dorește reglarea ponderilor dintre nivelul de intrare și cel intermediar, δ_i fiind δ_h , respectiv constituie nivelul intermediar dacă se dorește actualizarea ponderilor între nivelul intermediar și cel de ieșire, δ_i fiind δ_k .

Alegerea lui α influențează în mare măsură algoritmul de învățare backpropagation bazat pe minimizarea erorii medii pătratice. Totuși alegerea lui α este dependentă de specificul problemei. Deși nu există o metodă universală de alegere a lui α într-o problemă dată se recomandă că acesta să fie subunitar sau, eventual, descrescător odată cu creșterea numărului iterației. De regulă, cea mai convenabilă valoare este aleasă după laborioase simulări.

Confidența metapredictorului adaptiv neuronal este dată de valoarea pragului (*threshold*) impusă prin program. Predicția propriu-zisă se va realiza doar dacă valoarea maximă de pe ieșirile rețelei este mai mare sau egală decât acest prag. În caz contrar se poate afirma că nu se are suficientă încredere în acest predictor și astfel se va evita o predicție. Nivelul de încredere poate lua valori în intervalul real $[0, 1]$, datorită utilizării funcției de activare sigmoidă.

Metapredictor adaptiv neuronal cu automat pe nivelul doi de încredere

Rețeaua neuronală folosită este identică cu cea descrisă anterior. Confidența acestui metapredictor este dată de o structură implementată la nivel de registru (locală per registru și nu globală per predictor component). Structura conține un automat de stare asociat fiecăreia din cele 32 de intrări aferente celor trei predictoare componente: last value, incremental, contextual. Automatul va avea patru stări și se va afla inițial în starea 0, fiind incrementat la fiecare predicție corectă a predictorului asociat, respectiv decrementat la fiecare predicție incorectă a acestuia. Rețeaua neurală indică predictorul cu probabilitatea cea mai ridicată de a prezice corect fără a impune un anumit prag rețelei. Pentru respectivul tip de predictor se verifică starea automatului (nivelul de încredere) corespunzător registrului pentru care se realizează predicția. Predicția propriu-zisă se va face doar dacă starea acestui automat este mai mare sau egală decât un prag impus prin program. În caz contrar nu se va realiza predicție.

Simulator funcțional pentru predicția valorilor centrată pe regiștri

S-a considerat că simulatorul *sim-vpred*, destinat predicției valorilor centrate pe instrucțiuni descris la 6.1.1, este cel mai potrivit pentru a fi

extins prin implementarea metapredictoarelor propuse în această secțiune, pentru creșterea acurateții de predicție a regiștrilor procesorului. Pentru compilarea surselor s-a folosit aplicația Cygwin (comanda *make*). Structurile utilizate, precum și declarațiile funcțiilor se află în fișierul *vpred.h*, iar definițiile funcțiilor pot fi găsite în *vpred.c*. Motorul simulatorului îl reprezintă aceeași funcție *sim_main()* din *sim-vpred.c*, aici fiind folosite atât structurile cât și funcțiile amintite.

Parametrii importanți ai simulatorului care pot fi modificați de către utilizator, sunt “adâncimea” istoriei, dimensiunea contextului (*pattern*) în cazul predictoarelor contextuale, hibride, metapredictoarelor neadaptiv cu automat, neadaptiv cu număr de biți de 1 și adaptiv neuronal. Poate fi stabilit de asemenea, tipul predictorului utilizat și nivelul de încredere în cazul metapredictoarelor (**-threshold**). În cazul metapredictorului neural este permisă antrenarea: **-trainingType** (dinamică - **0**, statică - **1**). Antrenarea statică presupune actualizarea ponderilor până când eroarea medie pătratică scade sub pragul impus – „predicție în gol”. Antrenarea dinamică se bazează pe folosirea unor ponderi inițiale, anterior salvate în urma execuției altor benchmark-uri, sau efectuării predicției asupra altor regiștri. Necesari unei simulări sunt și benchmark-ul, registrul supus predicției și eventual numărul de instrucțiuni executate dacă nu se rulează în întregime programul de test. Se poate stabili ca acuratețea predicției calculată să fie globală, pentru toți regiștrii favorabili (vezi subcapitolul precedent) sau aferentă doar unui singur registru.

Predictoarele *last value*, incremental, contextual și hibrid (descrise în 6.2.1) sunt implementate în fișierul *vpred.c* prin intermediul funcțiilor *predictLastValue*, *predictStride*, și respectiv *predictContextual*, care vor întoarce valoarea precisă de fiecare tip de predictor. În momentul execuției va fi folosit predictorul corespunzător parametrilor introduși de către utilizator.

Întrucât structurile de date și funcțiile ce descriu predictoarele clasice de valori (*last value*, incremental, contextual și hibrid) au mai fost prezentate, în continuare sunt descrise cele mai importante secțiuni de cod prin care este implementat algoritmul de învățare backpropagation aplicat problemei de față precum și structurile de date care intervin în implementarea perceptronului multistrat din figura 6.9. De asemenea, este prezentat și modul în care se realizează procesul de predicție propriu-zis.

Informațiile care definesc perceptronul multistrat din figura 6.9 sunt:

N – Numărul de noduri de pe nivelul de intrare.

M , $hidd0[M]$ – Numărul de noduri din nivelul ascuns și valorile acestora.

P , $out[P]$ – Numărul de noduri de pe nivelul de ieșire al rețelei respectiv valorile acestora.

$wohi[P][M]$ – Matricea ponderilor conexiunilor dintre nivelul de ieșire și cel ascuns.

$bohi[P]$ – Deplasările de scală pentru neuronii din nivelul de ieșire (bias).

$whin[M][N]$ – Matricea ponderilor conexiunilor dintre nivelul ascuns și cel de intrare.

$bhin[M]$ – Deplasările de scală pentru neuronii din nivelul ascuns.

α = Rata de învățare = 0.3; F = funcția de activare

În cadrul metapredictorului adaptiv neuronal cu automat, întrucât confidența este la nivel de registru, pentru fiecare dintre aceștia va exista un automat (32 de automate per predictor: incremental, contextual și hibrid. Aceste automate în cod sunt reprezentate sub forma a trei vectori de tip întreg: $state_lv[32]$, $state_stride[32]$, $state_context[32]$).

Se consideră următoarele valori intermediare, care reprezintă, pentru fiecare neuron suma ponderată a tuturor intrărilor sale (vor fi numite valori *net*):

$$neto[i] = \sum_{j=0}^{M-1} wohi[i][j] * hidd0[j] + bohi[i] \quad i=0 \dots P-1. \quad (6.8)$$

$$neth0[j] = \sum_{l=0}^{N-1} whin[j][l] * inp[l] + bhin[j] \quad j=0 \dots M-1. \quad (6.9)$$

$$out[i] = F(neto[i]) \quad i = 0 \dots P-1 \quad (6.10)$$

$$hidd0[j] = F(neth0[j]) \quad j=0 \dots M-1 \quad (6.11)$$

Funcția de eroare medie pătratică este:

$$e^2 = \sum_{i=0}^{P-1} (tp[i] - out[i])^2 = \sum_{i=0}^{P-1} (tp[i] - F[neto[i]])^2 \quad (6.12)$$

În cadrul procesului de antrenare cele mai importante funcții care intervin sunt *forward* și respectiv *backward*. Detalii de implementare pot fi urmărite pe CD-ul ce însoțește această lucrare.

Pasul forward

În acest pas se realizează propagarea intrării spre ieșire, nivel cu nivel. Se aplică intrarea din perechea de antrenament la intrarea rețelei. Se

calculează valorile $neth0[j]$ pentru fiecare neuron din primul nivel ca și sume ponderate ale intrărilor în neuronul respectiv. Valoarea calculată se transformă în ieșire a stratului de intrare ($hidd0[j]$) prin aplicarea funcției de activare (F) pentru fiecare neuron din primul nivel. În continuare, ieșirile neuronilor din primul nivel servesc ca intrări în neuronii nivelului imediat următor. Procesul descris mai sus se repetă nivel cu nivel până se ajunge la nivelul de ieșire și se determină valorile ieșirii rețelei (fie $out[i]$)

```
void forward(int in[], float out[])
{
    int j,l;
    /* propagarea vectorului de intrare înspre stratul ascuns. Calcularea valorilor din
       nodurile stratului ascuns – formulele 6.9 și 6.11. */
    for(j=0; j<nHiddenLayerNeurons; j++)
    {
        neth[j] = bhin[j];
        for(l=0; l<nInputLayerNeurons; l++)
            neth[j] += whin[j][l] * in[l];
        hidd[j] = F(neth[j]);
    }
    /* propagarea informației înmagazinate în stratul ascuns spre ieșirea rețelei.
       Calcularea valorilor din nodurile stratului de ieșire – formulele 6.8 și 6.10. */
    for(j=0; j<nOutputLayerNeurons; j++)
    {
        neto[j] = bohi[j];
        for(l=0; l<nHiddenLayerNeurons; l++)
            neto[j] += wohi[j][l] * hidd[l];
        out[j] = F(neto[j]);
    }
}
```

Pasul backward

În acest pas are loc ajustarea propriu-zisă a ponderilor rețelei. Se știe că o pereche de antrenament este formată dintr-un vector de intrare (fie inp) și un vector de ieșire (fie tp), care se dorește a fi răspunsul rețelei când i se aplică la intrare vectorul inp . Diferența dintre tp și out reprezintă eroarea ieșirii rețelei la momentul respectiv. Conform acestei erori, cu ajutorul unei reguli bine stabilite, se ajustează ponderile rețelei nivel cu nivel astfel încât eroarea să tindă spre zero, adică la aplicarea vectorului inp din perechea de antrenament, ieșirea obținută out să fie cât mai apropiată (în cazul ideal identică), de vectorul de ieșire tp din perechea de antrenament. Această ajustare a ponderilor se realizează printr-un proces iterativ, adică se aplică secvențial fiecare pereche din setul de antrenament și se reactualizează

ponderile, aceasta de câte ori este necesar pentru ca eroarea să se situeze sub un anumit prag minim.

```

void backward(int tp[], int in[], float out[])
{
    double deltaout[32];
    double deltain[32];
    int j,k,l;
    /* Actualizează ponderile dintre stratul de ieșire și cel intermediar folosind formulele
       6.4, 6.6 și 6.7 */
    for(j=0; j<nOutputLayerNeurons; j++)
        for(l=0; l<nHiddenLayerNeurons; l++)
        {
            deltaout[j] = (tp[j] - out[j]) * dF(neto[j]);           – echivalent 6.4
            wohi[j][l] += learningRate*deltaout[j]*hidd[l];       – echivalent 6.6 și 6.7
            bohi[j] += learningRate*deltaout[j];
        }
    /* Actualizează ponderile dintre stratul de intermediar și cel de intrare folosind
       formulele 6.5, 6.6 și 6.7 */
    for(j=0; j<nHiddenLayerNeurons; j++)
        for(l=0; l<nInputLayerNeurons; l++)
        {
            deltain[j] = 0;
            for(k=0; k<nOutputLayerNeurons; k++)
                deltain[j] += deltaout[k]*wohi[k][j]*dF(neth[j]);
            whin[j][l] += learningRate*deltain[j]*in[l];
            bhin[j] += learningRate*deltain[j];
        }
    }

double F(double val)
{
    /* Funcția de activare (F) */
    return 1/(1 + exp(-1 * val));
}

double dF(double val)
{
    /* Derivata funcției de activare (dF) */
    return F(val)*(1 - F(val));
}

```

În continuare sunt prezentate etapele procesului de predicție:

1. Inițializarea ponderilor rețelei cu valori mici aleatoare.

În cazurile în care antrenarea este dinamică (învățare în timpul rulării), respectiv se dorește o antrenare efectivă în vederea salvării ponderilor

rezultate în urma acesteia, se folosește următoarea funcție care generează ponderi aleatoare în intervalul [0.3, 0.7], ponderi care vor fi și salvate într-un fișier pe harddisk (*RandomWeights.txt*).

```
void generateRandomWeights()
{
    int j, k;
    Weights = fopen("RandomWeights.txt", "w");
    for(j=0; j<nHiddenLayerNeurons; j++)
    {
        bhin[j] = ((rand() % 4000)*1.0)/10000.0 + 0.3;
        fprintf(Weights, "%f\t", bhin[j]);
        for(k=0; k<nInputLayerNeurons; k++)
        {
            whin[j][k] = ((rand() % 4000)*1.0)/10000.0 + 0.3;
            fprintf(Weights, "%f\t", whin[j][k]);
        }
    }
    for(j=0; j<nOutputLayerNeurons; j++)
    {
        bohi[j] = ((rand() % 4000)*1.0)/10000.0 + 0.3;
        fprintf(Weights, "%f\t", bohi[j]);
        for(k=0; k<nHiddenLayerNeurons; k++)
        {
            wohi[j][k] = ((rand() % 4000)*1.0)/10000.0 + 0.3;
            fprintf(Weights, "%f\t", wohi[j][k]);
        }
    }
    fclose(Weights);
}
```

În cazul în care se dorește salvarea ponderilor rezultate în urma antrenării statice pentru a putea fi folosite drept ponderi inițiale în cadrul unei antrenări dinamice ulterioare se va folosi funcția *saveWeights* care va salva aceste ponderi rezultate, într-un fișier pe disc (*Weights.txt*). Pentru a se încărca ponderile salvate la o antrenare statică anterioară, în vederea efectuării unei antrenări dinamice se va folosi funcția *loadWeights*. Ponderile vor fi încărcate din fișierul existent pe disc cu numele de *Weights.txt*. Cele două funcții sunt exemple banale de scriere, respectiv citire de tablouri de numere flotante în fișiere.

2. Pentru fiecare pereche de antrenament sunt executate operațiile:

- 2.1. Stabilirea vectorului de intrare din perechea de antrenament curentă la intrarea rețelei și propagarea acestuia nivel cu nivel până la nivelul de ieșire; Se consideră vectorul $in[m]$ ca fiind vectorul de intrare al

rețelei conținând m elemente, iar $out[3]$ reprezintă vectorul ce conține valorile nodurilor de ieșire.

forward (in, out);

- 2.2. Se calculează valoarea maximă dintre ieșirile rețelei, reținându-se totodată și indexul acesteia.

```
max_index = 0;
max_val = out[0];
for(t = 1; t < 3; t++)
    if(out[t] > max_val)
    {
        max_val = out[t];
        max_index = t;
    }
```

- 2.3. Dacă valoarea maximă găsită la pasul anterior este mai mare decât o valoare de prag impusă, atunci se va face predicție, incrementându-se și contorul corespunzător.

```
if(max_val > threshold)
    NeuralTotalPrediction ++;
```

- 2.4. Dacă se realizează predicție și aceasta este corectă, se va incrementa contorul aferent predicțiilor corecte:

```
if((max_val > threshold) && (((max_index == 0) && (lvalue == 1)) || ((max_index == 1) && (svalue == 1)) || ((max_index == 2) && (cvalue == 1))))
    neuralValuePrediction ++;
```

- 2.5. Se formează vectorul cu valorile dorite ale nodurilor de ieșire – $tp[]$.

```
if(lvalue == 1) tp[0] = 1;
else tp[0] = 0;
if(svalue == 1) tp[1] = 1;
else tp[1] = 0;
if(cvalue == 1) tp[2] = 1;
else tp[2] = 0;
```

- 2.6. Este calculată erorii pe baza ieșirii obținute și a celei dorite și propagarea ei înapoi nivel cu nivel până la intrare ajustându-se ponderile astfel încât eroarea să scadă:

backward(tp, in, out);

- 2.7. În cazul în care se dorește antrenare statică se repetă procesul de învățare atâta timp cât valoarea maximă curentă de la ieșirea rețelei se situează sub un nivel de confidență impus de utilizator. Ca o dezvoltare ulterioară, durata procesului de învățare poate fi limitată

prin modificarea numărului de iterații, printr-o iterație înțelegând un ciclu *forward-backward*.

```
if( (trainingType == 1)&&(train == 1) ) // antrenare statică
{
    ct_iterations = 0;
    while( (max_val < threshold)&&(ct_iterations<iterations))
    {
        forward(in, out);
        max_val = out[0];
        for(t = 1; t<3; t++)
            if(out[t]>max_val)
            {
                max_val = out[t];
                max_index = t;
            }
        backward(tp, in, out);
        ct_iterations ++;
    }
}
```

Rezultatele simulării împreună cu parametrii arhitecturali vor fi scriși într-un fișier (simout.res) în directorul curent, de unde pot fi preluați pentru eventuale prelucrări.

Prezentarea interfeței grafice și a modului de utilizare

Privind din punctul de vedere al utilizatorului s-a considerat necesară o interfață vizuală prietenoasă, bazată pe meniuri, ferestre de dialog, imagini grafice edificatoare etc. Interfața trebuie să fie simplu de utilizat, să permită utilizatorului manevrarea ușoară a simulatorului, interpretarea și prelucrarea eficientă a rezultatelor. Implementarea simulatorului s-a făcut în limbajul Visual C++ 6.0.

Pentru a porni simulatorul este nevoie de un sistem pe care să fie instalat Windows 9x sau Windows 2000 și să existe benchmark-uri (programe de test). La lansarea în execuție a aplicației, pe ecran apare fereastra de configurare din figura 6.10 și se poate trece la introducerea parametrilor simulării.

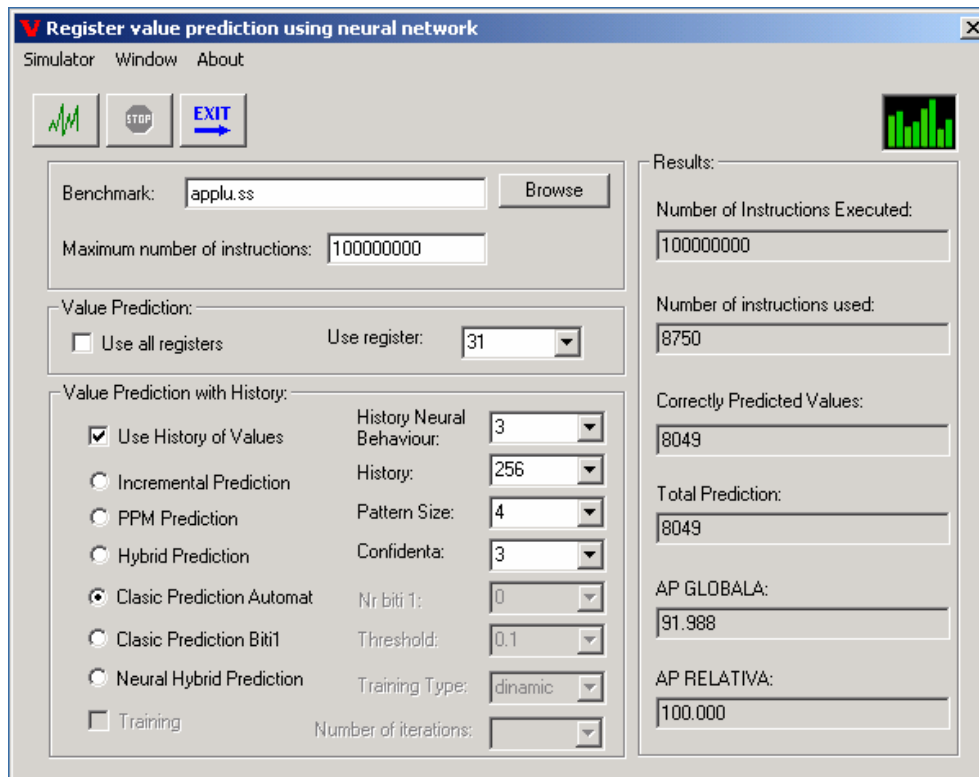


Figura 6.10. Fereastra de configurare a simulatorului

Utilizatorul poate să aleagă “adâncimea” istoriei și dimensiunea pattern-ului. În cazul în care se face predicție fără istorie, predictorul simulat este de tip *last value*. Dacă predicția se face cu istoria valorilor utilizatorul trebuie să aleagă tipul predictorului care poate fi incremental, contextual, hibrid, metapredictor neadaptiv cu automat, neadaptiv cu număr de biți de 1 sau neuronal. În cazul unei predicții contextuale sau hibride, trebuie aleasă dimensiunea contextului (*pattern*). Trebuie precizat că predictorul incremental implementat este de tip “2-delta”, iar cel contextual este de tip PPM complet.

În cazul unei metapredicții neadaptive cu automat, cu număr de biți de 1 sau adaptive neuronale trebuie ales și pragul începând de la care se va accepta predicția. În cazul unei predicții neuronale se poate alege tipul de antrenare care poate fi statică sau dinamică. Prin antrenarea dinamică se pornește simularea cu ponderi prestabilite anterior, salvate în urma execuției altor benchmark-uri, sau efectuării predicției asupra altor regiștri. Antrenarea statică presupune actualizarea ponderilor până eroarea medie pătratică scade sub pragul impus – „*predicție în gol*”. Prin modificarea

threshold-ului este limitat mai mult sau mai puțin timpul de învățare al rețelei.

Un parametru important care poate fi ales din interfață este registrul pentru care se dorește a se realiza simularea. Există posibilitatea alegerii unui anumit registru dintre cei favorabili (cei care prezintă o localitate a valorii mai mare de 60% sau 80%) sau ca simularea să se realizeze pe toți acești regiștri. La sfârșitul simulării, în partea dreaptă a ferestrei (figura 6.10) apar rezultatele: numărul de instrucțiuni executate, numărul de instrucțiuni dinamice executate care au drept destinație registrul supus predicției, numărul de predicții corecte și numărul total de predicții efectuate, acuratețea globală și relativă a predicției pentru arhitecturile simulate.

7. REZULTATE OBȚINUTE PRIN SIMULARE. INTERPRETĂRI.

Acest capitol prezintă și analizează probleme legate de vecinătatea valorilor (*value locality*) și predicția target-urilor salturilor / apelurilor de funcții indirecte, respectiv predicția valorilor instrucțiunilor și regiștrilor procesorului, cu consecința execuției speculative a instrucțiunilor și influențe benefice asupra timpului de procesare. Au fost realizate investigații cantitative privind gradele de localitate a valorii cu ajutorul setului standardizat de instrumente SimpleScalar 3.0b, dezvoltat la Facultatea de Calculatoare din Wisconsin, Madison USA și unanim acceptat de cercetătorii în arhitectura calculatoarelor din întreaga lume (spre exemplu, în anul 2000, mai mult de o treime din lucrările publicate în conferințele de top dedicate arhitecturii calculatoarelor au folosit setul SimpleScalar pentru simularea / evaluarea propriilor idei de proiectare). Evaluările arhitecturilor propuse au fost făcute folosind o colecție de simulatoare execution-driven (dezvoltate din setul de instrumente **SimpleScalar 3.0**). S-a început cu arhitectura cea mai simplă, predictorul de tip “*last value*” și s-a continuat cu predictoare de tip Target Cache, contextuale, hibride etc, ale căror scheme hardware au fost descrise pe parcursul subcapitolelor de contribuții proprii – 5.3, respectiv 6.1 și 6.2. Simularea a fost realizată pe cele două versiuni ale benchmark-urilor SPEC ('95 și 2000). De asemenea, pentru generarea codului la nivel limbaj de asamblare MIPS și a codului obiect specific arhitecturii SimpleScalar 3.0. a fost nevoie de recompilarea instrumentelor setului (utilitarele GNU, compilatorul Gcc). Cu ajutorul acestora au fost recompilate o parte din propriile programe de test folosite (plus benchmark-urile Stanford) în vederea studierii legăturii calitative și cantitative existente între paradigmele actuale de programare (programe procedurale vs. obiectuale) și respectiv generarea valorilor de anumite tipuri de instrucțiuni (salturi indirecte, Load, ALU, etc). Mai precis, s-a încercat investigarea legăturii existente între moștenire, polimorfism și alocarea dinamică a memoriei pe de o parte, și comportamentul salturilor indirecte (JR *reg*) de cealaltă parte, fiind cunoscută ca o problemă dificilă predicția branch-urilor codificate în moduri de adresare indirecte. Se urmărește practic transmiterea de informații relevante de la nivel software către proiectanții de arhitecturi (hardware).

Simulările au fost efectuate pe procesoare Pentium III la 500 MHz parțial efectuate sub sistem de operare Microsoft Windows98, 2000, sau NT având la dispoziție emulatorul Cygwin și parțial sub sistemul Linux RedHat 7.3. Având în vedere că timpul de simulare a 5.000.000 instrucțiuni aferente unui benchmark SPEC'95 variază între 30 de secunde (*swim*) și 1h30' (*cc1*) s-a optat să nu fie simulate în întregime aceste benchmark-uri cu până la 2 miliarde și jumătate instrucțiuni. Deși în cele mai multe din experimente au fost simulate doar 5 milioane de instrucțiuni dinamice, pentru predicția valorilor centrată pe regiștii procesorului unele statistici au fost realizate pentru 500 milioane de instrucțiuni dinamice în timp ce pentru predicția target-urilor salturilor indirecte au fost simulate între 100 milioane de instrucțiuni pe SPEC'95 și respectiv 500 milioane pe SPEC2000. În reprezentările grafice următoare acolo unde nu se specifică se consideră execuția a 5.000.000 de instrucțiuni dinamice.

7.1. CERCETĂRI PRIVIND VECINĂTATEA ȘI PREDICȚIA APELURILOR INDIRECTE

În cadrul acestui subcapitol, pe baza benchmark-urilor standardizate SPEC și a programelor de test propuse în subcapitolul 4.2, sunt determinate gradele de localitate existente în aplicații la nivelul *salturilor indirecte* (jr – altele decât return-uri) ca o limită ultimativă a acurateții de predicție. De asemenea, s-a realizat o analiză statistică a procentajului de salturi indirecte statice și dinamice pe benchmark-urile SPEC, urmată de determinarea unor analogii privind rezultatele obținute din punct de vedere statistic între programele procedurale și cele obiectuale.

În ce privește predicția adreselor destinație a instrucțiunilor de salt indirect, s-a început cu arhitectura cea mai simplă, predictorul de tip “*last value*” și s-a continuat cu predictoare contextuale – PPM complet, de tip Target Cache, hibride etc. Cu ajutorul predictorului contextual de tip PPM complet s-a încercat determinarea pattern-ului optim de căutare (influențat de *history* și localitatea anterior obținută), stabilirea corelației existente între salturi în funcție de context. În vederea îmbunătățirii acurateții predicției aferente instrucțiunilor de salt indirect au fost aduse câteva modificări structurii de predicție Target Cache originare propuse de [Cha97]. Mai întâi a fost studiată influența istoriei globale a salturilor condiționate asupra predicției, urmată de extinderea informației de corelație. Un ultim experiment privitor la această structură l-a constituit încercarea de îmbunătățire a acurateții de predicție printr-o ignorare selectivă a efectuării

unor predicții. În final a fost exploatată o schemă de predicție hibridă (LastValue+Contextual) cu selecție bazată pe aritate.

7.1.1. REZULTATE OBTINUTE PRIN SIMULAREA BENCHMARK-URILOR SPEC

În vederea exprimării concluziilor pe baza simulărilor efectuate, după varierea diversilor parametri și execuția unui număr prestabilit de instrucțiuni dinamice din fiecare benchmark, s-a folosit media aritmetică pe toate testele avute la dispoziție sau (în anumite condiții precizate) doar pe cele bogate în instrucțiuni dinamice de salt indirect, pentru care îmbunătățirile arhitecturale pot determina un impact pozitiv major asupra performanței. Cel mai indicat poate ar fi fost folosirea unei medii ponderate, însă datorită simulării unui număr fix de instrucțiuni și, întrucât consorțiul SPEC (*Standard Performance and Evaluation Corporation*) nu a oferit niște criterii explicite după care anumite benchmark-uri ar avea anumite ponderi în exprimarea unei metrici, am propus ponderi egale pentru fiecare test rezultând de fapt ca metrică globală media aritmetică.

	Instrucțiuni statice de salt indirect	Totalul de instrucțiuni statice de salt	% instrucțiuni statice de salt indirect	Instrucțiuni dinamice de salt indirect	Totalul de instrucțiuni dinamice de salt	% instrucțiuni dinamice de salt indirect
applu	30	848	3.538	986	2238776	0.044
apsi	58	3027	1.916	97142	6469134	1.502
cc1	213	25150	0.847	634418	20149893	3.148
fpppp	39	1103	3.536	34227	1521447	2.250
go	1	288	0.347	6272	22595640	0.028
hydro2d	54	1490	3.624	1176581	20880702	5.635
li	17	1408	1.207	1074663	22794725	4.715

Tabelul 7.1. Procentajul instrucțiunilor dinamice de salt indirect generate în urma execuției a 100.000.000 instrucțiuni din benchmark-urile SPEC'95

	Instrucțiuni statice de salt indirect	Totalul de instrucțiuni statice de salt	% instrucțiuni statice de salt indirect	Instrucțiuni dinamice de salt indirect	Totalul de instrucțiuni dinamice de salt	% instrucțiuni dinamice de salt indirect
gcc	230	33758	0.6813	1863928	91457321	2.0380
crafty	11	1899	0.5793	617615	118719981	0.5202
gap	14	1849	0.7572	36137	69420381	0.0521

mcf	14	831	1.6847	1239176	119256622	1.0391
twolf	15	2601	0.5767	197274	77237383	0.2554

Tabelul 7.2. Procentajul instrucțiunilor dinamice de salt indirect generate în urma execuției a 500.000.000 instrucțiuni din benchmark-urile SPEC2000

Procentajul instrucțiunilor dinamice de salt indirect extras din simulările efectuate atât pe benchmark-urile SPEC ('95 și 2000) cât și pe propriile aplicații propuse (vezi tabelul 7.9) este în concordanță cu rezultatul obținut de Roth privitor la frecvența apelurilor indirecte de metode virtuale din programele obiectuale. **Astfel, o instrucțiune dinamică de apel indirect apare aproximativ de la 200 până la 1000 de instrucțiuni dinamice (de 4 până la 20 de ori mai puțin decât apelurile directe și de 30 până la 150 de ori mai puțin decât salturile condiționate)** [Roth99]. O remarcă extrem de interesantă, de care poate programatorii din limbajele de nivel înalt ar trebui să țină cont, se referă la faptul că **o singură instrucțiune statică de salt indirect generează în momentul execuției peste 6200 de astfel de instrucțiuni** (vezi cazul benchmark-ului go). În ciuda procentului scăzut de instrucțiuni de salt indirect dinamice prezent în programele procedurale testate (sub **5.64%**), în cadrul arhitecturilor moderne de procesare superspeculative și cu structuri pipeline extrem de complexe, predicția eronată a unei singure instrucțiuni de salt (chiar și indirect) determină stagnări și penalități substanțiale din punct de vedere al timpului de procesare cu consecințe defavorabile asupra ratei de procesare.

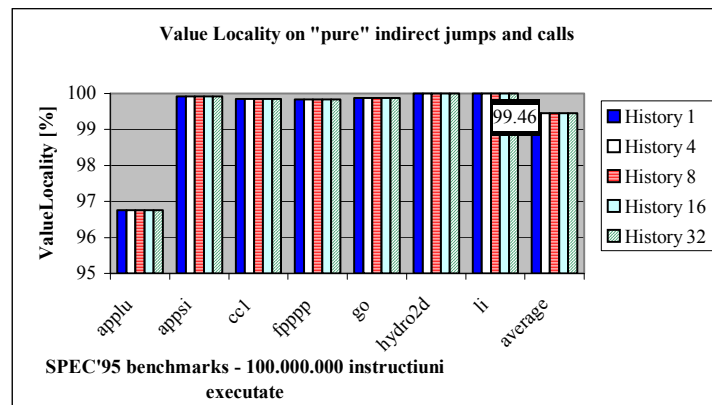


Figura 7.1. Localitatea valorilor pentru instrucțiuni de salt indirect ($j \neq x$, unde $x \neq 31$)

Un rezultat cantitativ ce se desprinde din simulările efectuate referitor la localitatea valorilor aferentă instrucțiunilor de salt indirect „pure” exprimă faptul că, indiferent de numărul instrucțiunilor executate (5.000.000

respectiv 100.000.000) pentru un context (istorie) mai mare de 4 adrese destinație reținute, se obține un grad de localitate de peste 91%, fapt remarcabil, ce sugerează că target-urile salturilor indirecte sunt foarte predictibile. Rezultatele altor simulări efectuate dar care nu vor fi ilustrate aici reflectă de asemenea că *gradul ridicat de localitate exprimat în medie de instrucțiunile de revenire din proceduri sunt în corelație cu gradul ridicat de localitate (90%) observat pe registrul \$31 (return address register) al procesorului MIPS [Flo03a].*

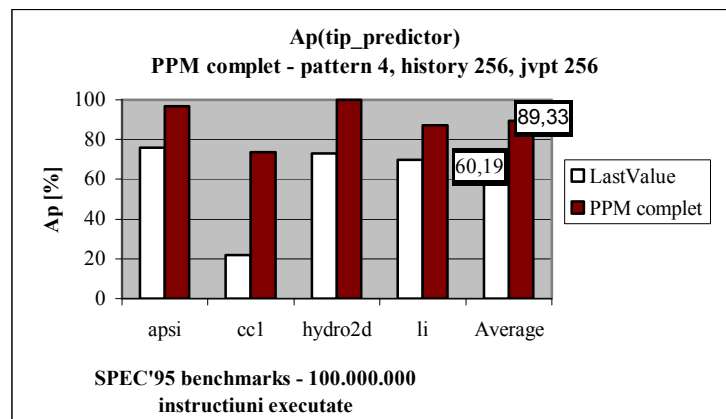


Figura 7.2. Superioritatea predictorului PPM complet față de cel de tip LastValue dpdv al acurateții predicției salturilor indirecte

În figura 7.2 este prezentat comparativ comportamentul din punct de vedere al acurateții predicției salturilor indirecte a două din structurile propuse (LastValue – foarte simplu și respectiv predictorul PPM complet – mult mai complex). Atât în figura 7.2 cât și în toate experimentele care urmează, dacă nu se specifică altfel, tabela de predicție *JVPT* este complet asociativă. Avantajul predictorului PPM complet de a avea la dispoziție un context lărgit de valori spre deosebire de predictorul Last Value se traduce într-o acuratețe de predicție cu **48.41%** mai mare în favoarea primului.

Ca și consecință a rezultatelor exprimate în figura 7.2 în continuare exploatarea gradului ridicat de localitate aferent instrucțiunilor de salt indirect obținut pe benchmark-urile SPEC'95 s-a realizat prin intermediul unui predictor contextual de tip PPM complet (vezi figurile 7.3 și 7.4). S-au realizat două statistici: pentru o istorie mare (menținerea ultimelor 256 de *target-uri*) dar și pentru o istorie dovedită optimă din punct de vedere al localității pe aceste tipuri de instrucțiuni (32). Rezultatele sunt aproximativ identice, în medie pentru un context de 3 valori obținând acuratețea de predicție cea mai mare. **Acuratețea în medie aritmetică** obținută este de **88.6%** (satisfăcător ținând cont de rezultatele anterior obținute de Chang

[Cha97]). În medie rezultatele raportate de alți cercetători indică o acuratețe de predicție pentru salturile indirecte de doar 75% [Dri98].

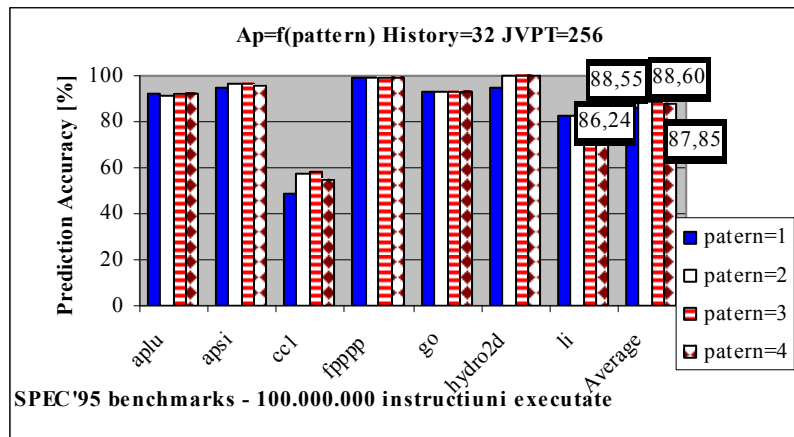


Figura 7.3. Acuratețea predicției aferentă instrucțiunilor de salt indirect utilizând un predictor de tip PPM complet, în funcție de lungimea *pattern*-ului. JVPT – asociativă (**istorie redusă**)

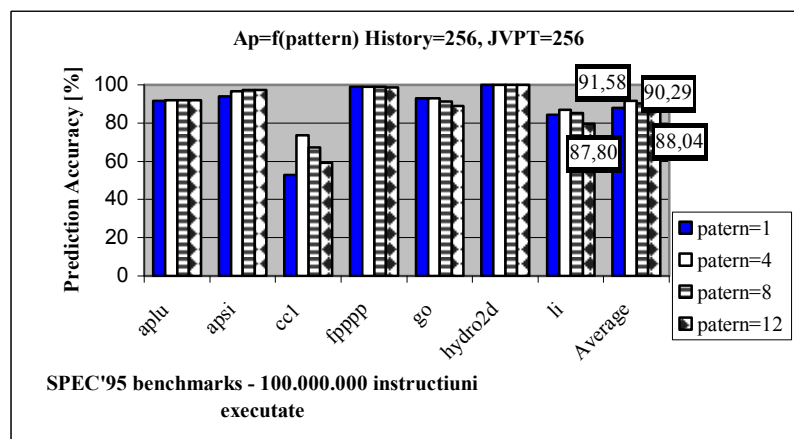


Figura 7.4. Acuratețea predicției aferentă instrucțiunilor de salt indirect utilizând un predictor de tip PPM complet, în funcție de lungimea *pattern*-ului. JVPT – asociativă (**istorie bogată**)

Rezultatul grafic din figura 7.4 evidențiază că predicția optimă (91.58%) se obține pentru un *pattern* de căutare de 4, îndreptățind afirmațiile și altor cercetători care afirmă că: "un *pattern* de căutare mai bogat poate conduce teoretic la o acuratețe mai ridicată a predicției, dar începând cu o anumită dimensiune a *pattern*-ului acesta se comportă ca

zgomot și acuratețea începe să scadă" [Vin02]. De asemenea, coroborând rezultatele obținute în figurile 7.3 și 7.4 rezultă că și *history*, care în acest caz reprezintă întregul context în care se caută apariția pattern-ului, influențează acuratețea de predicție. Rezultatele sunt ușor mai mari (cu **3.36%**) din punct de vedere al acurateții de predicție pentru *history* = 256 de valori (ultimele 256 de target-uri aferente unei anumite instanțe de salt indirect static).

Procentajul mediu al instrucțiunilor de salt indirect clasificate predictibile și predicționate corect de către automatul de clasificare (vezi figura 5.28) este destul de ridicat (90÷94%). Diferența se observă studiind coloanele **Unpred** (clasificate nepredictibile și predicționate greșit), unde procentajele din tabelul 7.3 sunt foarte reduse ($\leq 27\%$). În sprijinul acestor valori extrem de mici aș aduce o concluzie proprie, generată mai mult pe baza tabelului 7.10 referitor la programele de test proprii: *nu este obligatoriu ca rezultatul din coloana Unpred să fie mare (poate fi chiar foarte mic) dacă, procentajul de salturi indirecte dinamice clasificate nepredictibile din totalul salturilor indirecte este nesemnificativ* ($\frac{\text{classifiedUnpred}}{\text{sim_indir_refs}} < 1 \div 4\%$, unde *sim_indir_refs* și *classifiedUnpred* au fost descrise în 5.3.1).

Una din problemele care pot fi îmbunătățite pe viitor se referă la determinarea și a altor mecanisme de clasificare, care prin comportamentul lor să ajute la creșterea acurateții de predicție (vezi Perceptronul sau rețelele MLP folosite în metapredicția valorilor regiștrilor).

SPEC benchmarks	History = 32, JVPT=256							
	pattern=1		pattern=2		Pattern=3		pattern=4	
	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]
aplu	99.67	8.33	99.56	6.25	99.67	4.35	99.67	4.35
apsi	96.18	42.91	97.19	38.65	97.46	43.47	97.80	31.95
cc1	70.97	50.44	76.13	52.88	79.96	58.93	80.48	63.39
fpppp	99.19	19.12	99.17	17.19	99.18	18.33	99.15	17.74
go	92.85	0.00	92.84	0.00	92.85	0.00	92.82	0.00
hydro2d	94.60	16.80	100.00	13.95	100.00	16.92	100.00	20.59
li	87.55	37.90	88.39	42.26	88.42	43.43	88.30	44.66
Average	91.57	25.07	93.32	24.45	93.93	26.49	94.03	26.10

Tabelul 7.3. Procentajul de instrucțiuni de salt indirect predictibile și nepredictibile identificate corect de către automatul de clasificare aferent predictorului PPM complet

În vederea înlocuirii unui predictor PPM complet, mai performant decât unul adaptiv pe două niveluri în condiții echivalente de cost hardware dar mai „scump de implementat”, cu un predictor hibrid a cărui componente le reprezintă două predictoare contextuale cu pattern-uri de lungime fixă, diferite ($P1 = \text{Markov}(m)$ și $P2 = \text{Markov}(n)$, cu $m \neq n$), am simulat o tabelă de predicție *qvpt* (vezi subcapitolul 5.3.1) cu 256 intrări (vezi figurile 7.5 și 7.6) în două ipostaze: cu istorie redusă (sunt reținute ultimele 32 de target-uri) respectiv cu o istorie bogată (256 de target-uri).

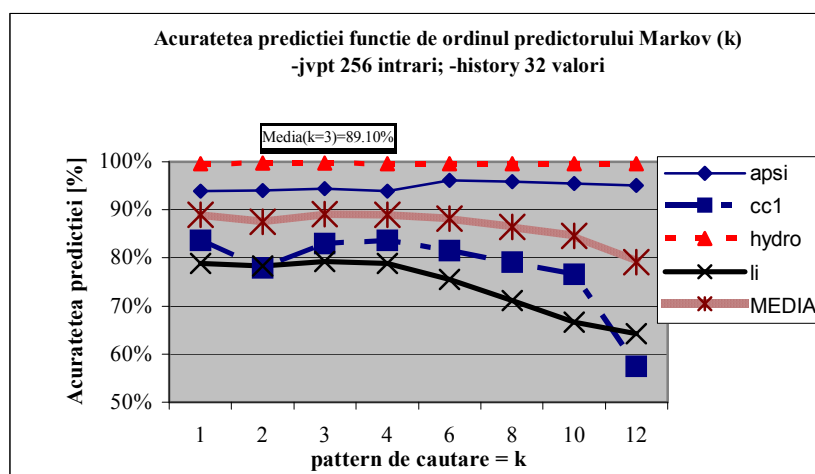


Figura 7.5. Determinarea predictorului Markov de ordin optim (istorie redusă - 32)

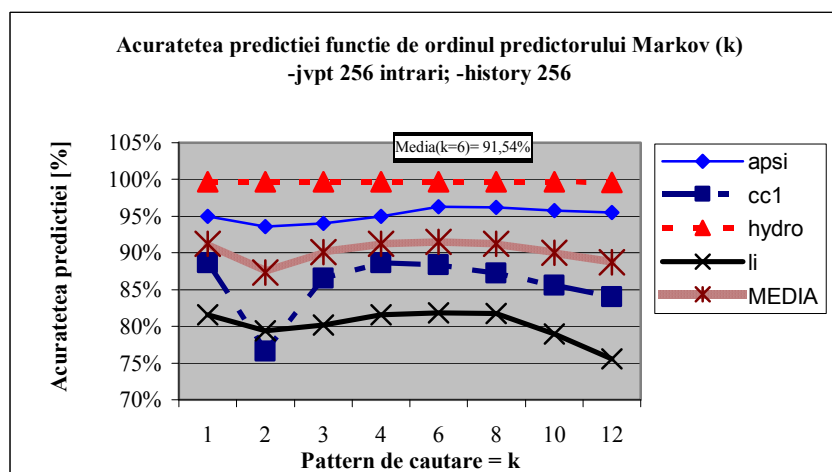


Figura 7.6. Determinarea predictorului Markov de ordin optim (istorie bogată - 256)

Oarecum firesc, creșterea istoriei valorilor reținute (a contextului în care se caută) implică o acuratețe de predicție mai mare prin creșterea pattern-ului de căutare (se distinge o corelație între salt-uri mai îndepărtate dacă contextul ar permite acest lucru). Astfel, acuratețea de predicție maximă pentru o istorie de 32 valori este obținută pentru un pattern de căutare $k=3$ (89.10%) respectiv pentru o istorie de 256 valori pentru un pattern $k=6$ (91.54%). Rezultatele par să întărească afirmația cercetătorilor [Tho03] care susțin păstrarea și utilizarea unei istorii cât mai „lungi” (îndepărtate) în procesul de predicție aferent instrucțiunilor de salt întrucât unele salturi corelate pot apărea la o distanță considerabilă în șirul de instrucțiuni dinamice. Acest lucru se poate întâmpla dacă două salturi corelate sunt separate de către un apel de funcție care conține multe branch-uri. La momentul „părăsirii” funcției, o istorie globală redusă poate conține doar comportamentul salturilor din cadrul funcției, în timp ce o istorie globală extinsă poate reține și rezultatul saltului corelat, anterior apelului funcției.

Rezultatele exprimate în continuare se referă la structura de predicție Target Cache. Următorul grafic, figura 7.7 prezintă comparativ acuratețea predicției în funcție de asociativitatea structurii Target Cache. Fiecare set este propriu fiecărei instanțe de instrucțiuni de salt indirect, iar în cadrul fiecărui set pot fi reținute **assoc** target-uri distincte.

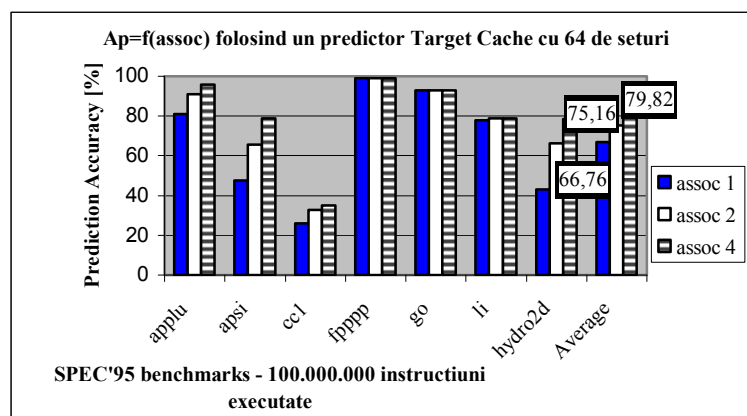


Figura 7.7 Acuratețea predicției aferentă instrucțiunilor de salt indirect în funcție de gradul de asociativitate al structurii Target Cache implementate

Identic ca și la cache-urile de date sau instrucțiuni, rezultatele evidențiază faptul că acuratețea predicției crește odată cu gradul de asociativitate. Rezultatele altor simulări efectuate dar care nu vor fi ilustrate aici reflectă de asemenea că, prin creșterea capacității tablei Target Cache

(a numărului de seturi) este îmbunătățită acuratețea predicției. Cu toate acestea, **pentru un TargetCache mare (256 de seturi) se observă o saturare a acurateții de predicție (practic tabela poate fi și mapată direct în acest caz)**. Rezultatele mai puțin satisfăcătoare obținute în cazul folosirii schemelor de predicție mapate direct sau semiasociative cu grad de asociativitate redus, se datorează și PC-urilor salturilor indirecte, multiple de 4 (instrucțiuni codificate pe 32 de biți), ceea ce face ca doar anumite locații (seturi) să fie accesate. Cu toate că gradul de utilizare al tabelor este destul de redus, numărul de interferențe ce apar este foarte ridicat diminuând în final acuratețea predicției. O soluție posibilă ar putea fi indexarea structurilor de predicție eliminându-se ultimii doi biți din adresa instrucțiunii de salt.

Rezultatele obținute, mai slabe decât cele generate de predictorul PPM complet, pot fi datorate și faptului că în implementarea făcută de autor tabela TargetCache nu a fost indexată cu adresa instrucțiunii concatenate (sau dispersată printr-o funcție "hash") cu istoria globală a salturilor condiționate, ci doar cu PC-ul saltului indirect. Pentru a înlătura acest neajuns și pentru a putea răspunde la întrebarea „Cât de *departe* trebuie să căutăm salturi corelate ? (*HRgLength* maxim = ?) **s-a introdus în procesul de predicție comportamentul ultimelor *HRgLength* salturi condiționate**, anterioare saltului indirect de prezis. Dispersia istoriei și a adresei saltului se face printr-o funcție XOR.

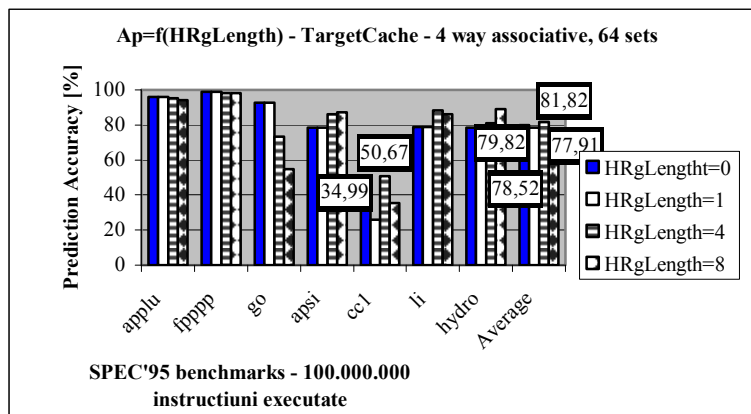


Figura 7.8. Influența istoriei globale a salturilor condiționate asupra predicției

Pentru benchmark-urile cu o dispersie ridicată a target-urilor (apsi, cc1, li, hydro – vezi subcapitolul 4.3), utilizarea istoriei globale a salturilor condiționate în indexarea Target Cache-ului joacă un rol important în creșterea acurateții predicției salturilor indirecte (în medie cu până la

16.93% iar în cazuri particulare - cc1 - chiar și cu 45%). În medie aritmetică pe toate cele 7 benchmark-uri SPEC'95, acuratețea optimă de predicție se obține prin reținerea comportamentului global al ultimelor 4 salturi condiționate. Repetând experimentul pentru valori ale parametrului HRgLength de 12 respectiv 16 acuratețea predicției scade pe măsură ce istoria crește. Rezultă că un pattern format din mai mult de 8 salturi condiționate se comportă practic ca zgomot pentru predicția salturilor indirecte.

Următorul pas făcut pentru a crește acuratețea de predicție a structurii TargetCache a constat în **extinderea informației de corelație** prin asocierea fiecărui bit de istorie din *globalHR* cu PC-ul aferent saltului condiționat respectiv și determinarea predicției pe baza acestei informații mai complexe și mai complete [Nai95, Vin99] – vezi figura 5.29.

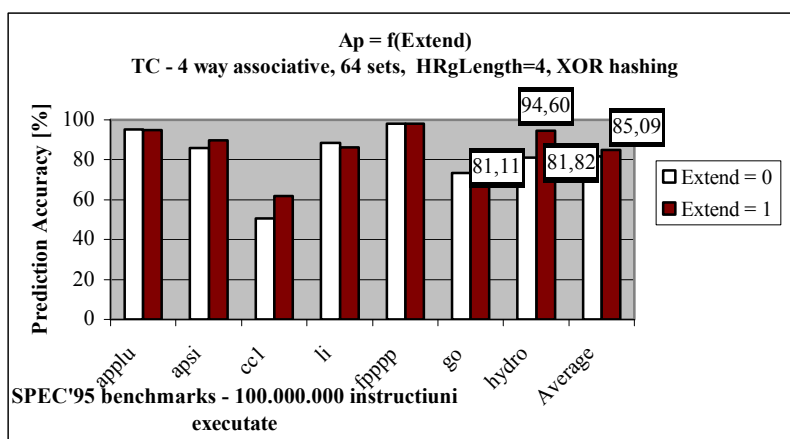


Figura 7.9. Acuratețea predicției instrucțiunilor de salt indirect folosind o tabelă TargetCache în funcție de extinderea sau nu a informației de corelație

După cum rezultă din figura 7.9 pentru benchmark-urile caracterizate de un procentaj ridicat de salturi indirecte **extinderea informației de corelație** ($PC_1, PC_2, \dots, PC_{HRgLength}$), la costuri identice de implementare (structură Target Cache de aceeași capacitate) **determină creșterea acurateții predicției** acestora. Efectul pozitiv apare practic pe acele programe de test pentru care istoria salturilor condiționate influențează predicția. Creșterea acurateții de predicție prin folosirea unei informații mai bogate de context este de **8.64%** pentru $HRgLength=4$ respectiv de **15.16%** când $HRgLength=8$ – vezi tabelul 7.4. Tabelul următor (7.4) ilustrează acuratețea de predicție în medie aritmetică doar pe 4 din benchmark-urile SPEC'95 (cc1, li, apsi, hydro – cele mai semnificative din punct de vedere al salturilor indirecte).

XOR mode; HRgLength = 4; TC – 4 way associative; 64 de seturi

	HRgLength = 4	HRgLength = 8
Extend = 0	76.52%	74.56%
Extend = 1	83.13%	85.86% (respectiv 88.21% pentru un TC – 8 way associative)

Tabelul 7.4. Creșterea acurateții de predicție prin folosirea unei informații mai bogate de context pe benchmark-urile bogate în salturi indirecte dinamice

Cu toată îmbunătățirea adusă de extinderea corelației, acuratețea predicției instrucțiunilor de salt indirect este totuși inferioară celei obținute cu un predictor PPM complet (**89.33%** - vezi figura 7.2). Și totuși există și rezultate realmente extraordinare: acuratețea obținută pe hydro.ss este **99.98%** (Target Cache 4 way associative, 64 de seturi, HRgLength=8, folosind informație de corelație extinsă - $PC_1, PC_2, \dots, PC_{HRgLength}$) egală cu cea obținută cu predictorul PPM complet. Un exemplu care evidențiază limitarea avantajului introdus de tehnica de extindere a informației de corelație pentru pattern-uri de salturi condiționate de istorie redusă este prezentat în Anexa 1 de la sfârșitul lucrării.

Un ultim experiment realizat în încercarea de a atinge acuratețea de predicție a predictorului PPM complet cu o structură de tip Target Cache – mai simplă ca implementare, s-a bazat pe **introducerea unui mecanism de confidență care să asigure o ignorare selectivă a efectuării unor predicții**. Pe lângă modificările structurale s-a folosit și un mecanism de inserare / evacuare în / din set bazat pe *LRU*(least recently used), *Confidență* și pe superpoziția celor două (*MPP* – minim de performanță potențial).

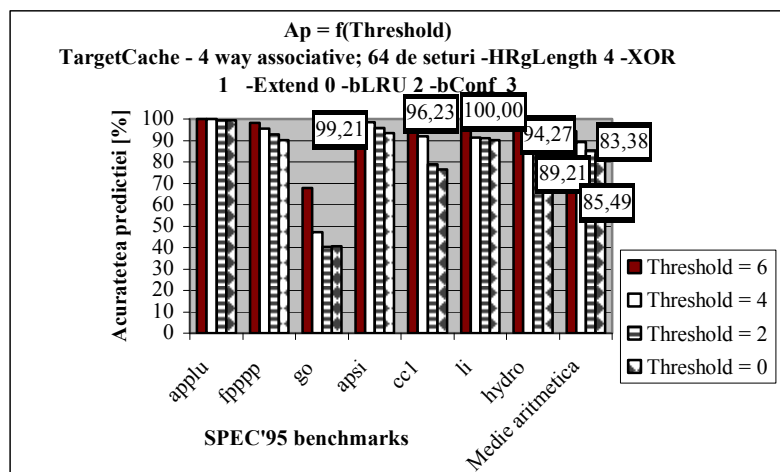


Figura 7.10. Acuratețea predicției [Des02] target-urilor instrucțiunilor de salt indirect în funcție de prag, folosind un mecanism de confidență

Practic acuratețea predicției [Des02] (vezi formula 5.6 din capitolul 5) crește substanțial prin restrângerea cazurilor în care se face predicție (cu 3.57% până la 11.45% în funcție de prag – medii statistice realizate pe programele de test bogate în instrucțiuni dinamice de salt indirect). Maximul de performanță s-a obținut utilizând un automat de confidență pe 3 biți și un prag de 6 (de cel puțin 6 ori saltul s-a prezis corect anterior – chiar dacă nu s-a folosit predicția), rezultând astfel în medie armonică pe cele 4 benchmark-uri semnificative (apsi, cc1, li și hydro) o acuratețe de predicție de 98.43%.

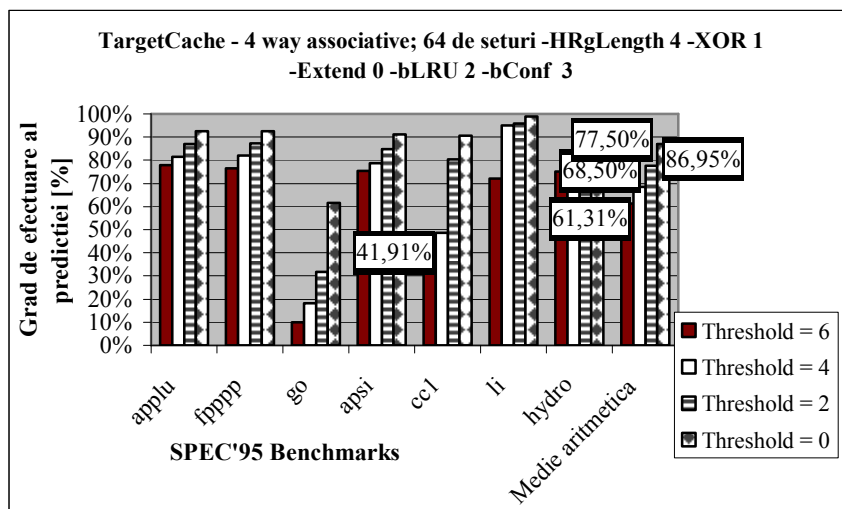


Figura 7.11. Procentajul cazurilor în care se face predicție (fracțiunea de salturi având *confidența* > *Threshold*)

Pentru a nu fi înșelați de acuratețile „foarte mari” obținute în figura 7.11 este ilustrat un grafic cu **procentajul cazurilor în care se face predicție** (vezi formula 5.7 din capitolul 5). Se observă că pragul (*threshold*) reprezintă obstacolul care determină selecția și cu cât acesta este mai mare acuratețea predicției tinde spre absolut, dar procentajul cazurilor în care se face predicție din totalul instrucțiunilor de salt indirect scade semnificativ.

Este de dorit totuși ca un procentaj cât mai ridicat de instrucțiuni de salt indirect să fie supuse predicției și acuratețea acestora să fie foarte mare. Rezultă că, ar fi ideal dacă s-ar putea printr-o metodă oarecare să reducem pragul dar să păstrăm acuratețea de predicție cât mai ridicată. Întrucât ponderile de 77.50% (media aritmetică pe cele 7 benchmark-uri SPEC'95) respectiv 84.17% (media aritmetică pe doar cele 4 benchmark-uri semnificative - apsi, cc1, li și hydro) din totalul instrucțiunilor de salt

indirect predicționate (pentru $\text{threshold}=2$) constituie totuși procente „decente“, încercăm să obținem o acuratețe a predicției sporită pentru acest prag. În acest sens se va extinde informația de corelație pentru indexarea structurii Target Cache.

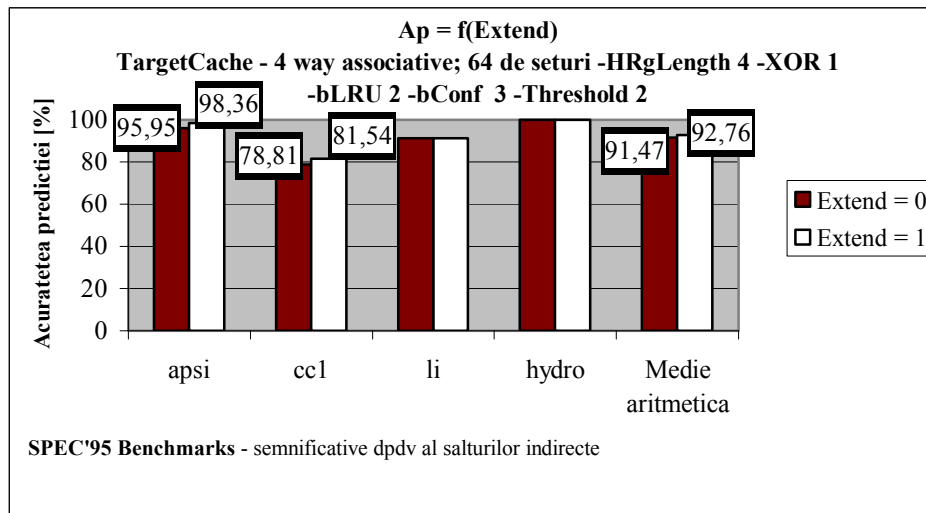


Figura 7.12. Influența informației de corelație extinsă asupra acurateții de predicție a target-urilor instrucțiunilor de salt indirect, folosind și un mecanism de confidență

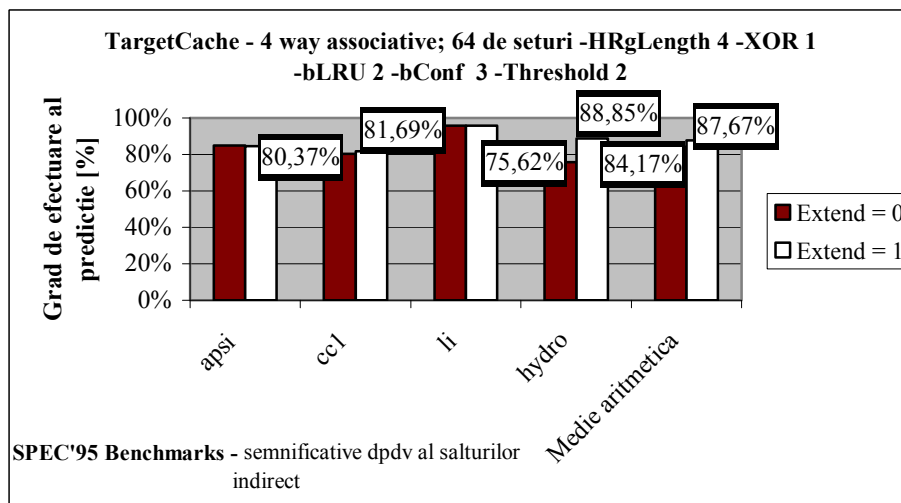


Figura 7.13. Creșterea procentajului cazurilor în care se face predicție prin extinderea informației de corelație

Privind comparativ, rezultatele grafice din figurile 7.10÷7.13 exprimă **eficacitatea unei scheme de predicție de tip Target Cache modificată care este indexată cu informație de corelație extinsă**. Astfel, în acest caz se obține o acuratețe de predicție superioară **cu 1.41%**, **dar și mai important, procentajul instrucțiunilor de salt indirect care sunt supuse predicției crește cu 4.16%**, **ajungând la aproape 88%** pe benchmark-urile care au un număr semnificativ de astfel de instrucțiuni. **Privind din punct de vedere al performanței globale a predictorului** (vezi formula 5.8' din capitolul 5) **superioritatea tehnicii de extindere a informației de corelație devine și mai accentuată** (creștere de **5.62%**).

Repetând experimentul anterior de extindere a informației de corelație, pentru parametrii Target Cache (a căror semnificație a fost descrisă în subcapitolul 5.3.2) - 4 way asociative; 64 de seturi -HRgLength 4 -XOR 1 -bLRU 2 -bConf 3, și utilizând un prag mai mare (automat de confidență mai selectiv – **threshold = 4**) s-a obținut următorul tabel de rezultate.

	Acuratețea Predicției [Des02]		Grad de realizare a predicției	
	Threshold=2	Threshold=4	Threshold=2	Threshold=4
Extend = 0	91.47%	95.49%	84.17%	74.39%
Extend = 1	92.76%	96.10%	87.67%	79.19%

Tabelul 7.5. Influența combinată a informației extinse de context cu automatul de confidență asupra acurateții de predicție, pe benchmark-urile bogate în salturi indirecte dinamice

Influența informației de corelație extinsă asupra unui predictor bazat pe un automat de confidență mai selectiv (threshold = 4) este nesemnificativă ($\approx 0.64\%$). Avantajul în această situație îl constituie totuși creșterea cu **6.45%** a cazurilor când se face predicție, ajungându-se până la **79.19%**. Se impune totuși o comparație între performanța globală generată în cele două cazuri din punct de vedere al automatului de confidență. Revenind la rezultatele din tabelul 7.5, ultima linie, și ținând cont de definiția *performanței globale*, rezultă că:

$$P(\text{Extend}=1 \text{ and Threshold}=2) = 92.76\% \cdot 87.67\% = \mathbf{81.32\%} \text{ și}$$

$$P(\text{Extend}=1 \text{ and Threshold}=4) = 96.10\% \cdot 79.19\% = \mathbf{76.10\%}$$

Practic performanța globală a predictorului cu automat de confidență pe 3 biți este mai bună când acesta este mai puțin selectiv (threshold = 2 abia la a patra predicție corectă se utilizează predictorul aferent, vezi și algoritmul implementat - - *if(confidența > threshold) valuePrediction++*).

De asemenea, **în condițiile noii metrici introduse, se observă că prin adăugarea unui automat de confidență nu întotdeauna** (depinde de

pragul impus) se îmbunătățește performanța globală a predictorului (practic ceea ce îl interesează în mod direct pe utilizatorul de programe) - vezi tabelul 7.6. Performanța globală a predictorului crește de îndată ce gradul de asociativitate al tabelii Target Cache crește (între 2.84% și 4.27% - vezi tabelul 7.6 coloana Threshold=1), optimul de performanță obținându-se pentru o **asociativitate 8-way** a tabelii Target Cache, inferior totuși cazului când se păstrează comportamentul pattern-ului de 8 salturi condiționate (vezi tabelul 7.7. coloana Threshold=1, linia asociativitate=8).

	Ap (fără confidență)	Performanța globală a predictorului cu confidență			
			Threshold = 1		Threshold = 2
asociativitate=2	78.15%	<	79.39%	>	77.91%
asociativitate=4	82.27%	<	82.78%	>	81.32%
asociativitate=8	82.35%	<	85.13%	>	83.87%

Tabelul 7.6. Superioritatea predictorului cu confidență pentru o asociativitate mai mare a tabelii Target Cache (TC - **64 de seturi -HRgLength 4 -XOR 1 -Extend 1 -bLRU 2 -bConf 3**) - I

	Ap (fără confidență)	Performanța globală a predictorului cu confidență		
		Threshold=0	Threshold=1	Threshold=2
asociativitate=4	86.40%	< 87.39%	85.66%	84.98%
asociativitate=8	86.47%	< 88.88%	> 87.17%	> 86.51%

Tabelul 7.7. Superioritatea predictorului cu confidență pentru o asociativitate mai mare a tabelii Target Cache (TC - **128 de seturi -HRgLength 8 -XOR 1 -Extend 1 -bLRU 2 -bConf 3**) - II

În condițiile extinderii informației de corelație superioritatea predictorului cu confidență față de unul care nu are implementat acest mecanism (vezi tabelul 7.7 linia cu asociativitate=8) este de **2.79%** pentru un **threshold=0** (nu se utilizează predicția decât a doua oară). *Acuratețea de predicție a Target Cache-ului cu confidență (88.88%) este încă inferioară celei obținute cu cel mai performant predictor PPM complet (89.33% - vezi figura 7.2) dar se apropie semnificativ (diferența fiind sub 0.51%), făcând posibilă înlocuirea unei scheme extrem de complexe (PPM complet) cu una fezabilă hardware (Target Cache).*

Pentru mecanismul de confidență (indiferent de pragul impus), creșterea gradului de asociativitate al structurii Target Cache joacă un rol pozitiv, ceea ce nu se poate spune și în cazul predictorului fără confidență (vezi tabelele 7.6 și 7.7). **Creșterea în acuratețe a predicției salturilor indirecte devine asimptotică pentru o tabelă de capacitate mai mare**

decât 128 de seturi, 8-way asociativă, folosind pentru indexare un pattern de 8 salturi condiționate și informație de corelație extinsă ($A_p=88.97\%$ dacă structura Target Cache are 256 de seturi, păstrând în rest condițiile precizate mai sus).

Se pune problema stabilirii influenței câmpului LRU asupra performanței și a determinării valorii sale optime. Întrucât pentru o asociativitate de 4 a tabelului Target Cache procentajul cazurilor în care se inserează în set după principiul LRU este **sub 1%** (exceptând cele două benchmark-uri *apsi* și *hydro*, vezi tabelul 7.8), rezultă că, crescând asociativitatea și implicit scăzând procentajul miss-urilor de conflict, influența câmpului LRU tinde să devină insignifiantă (≈ 0). Într-un fel, acest fapt este benefic întrucât s-ar putea implementa un algoritm trivial de evacuare (FIFO etc.) conducând la scăderea complexității, idee care ar fi părut „naivă” în lipsa simulărilor efectuate.

	apsi	hydro
asociativitate=2	10.18%	13.37%
asociativitate=4	3.71%÷6.26% (funcție de numărul de biți pe care se reprezintă câmpul LRU: 4÷1)	5.38%

Tabelul 7.8. Procentajul cazurilor când se inserează într-un set conform principiului LRU

Simulări efectuate pentru două grade de asociativitate diferite (2-way respectiv 4-way) și variind parametrul LRU au condus la observația că optimul acurateții de predicție se obține pentru $LRU=\log_2(\text{gradul de asociativitate})$.

Modalitatea de inserare / evacuare în / din set după LRU minim s-a dovedit mai eficientă decât după confidență minimă, cu 2.43%, și chiar cu 0.34% față de modalitatea MPP (vezi figura 7.14), pentru structuri Target Cache identice (64 seturi, 4 way asociativă, $HRgLength=8$, $XOR=1$, $Extend=1$, $bLRU=2$, $bConf=2$, $Threshold=0$). Rezultatele statistice obținute pe o tabelă Target Cache 2 way asociativă arată o superioritate și mai accentuată a acurateții de predicție dacă inserarea se face după LRU (6.88% față de inserarea după confidență, respectiv 1.02% față de inserarea după MPP). Cu toate acestea, bazat și pe rezultatele anterioare, (vezi figura 7.7 și [Flo04]), o structură Target Cache este relativ eficientă ($A_p\approx 82.65\%$) pentru un grad de asociativitate ≥ 4 . O primă concluzie ar fi că **anumite salturi indirecte sunt evacuate din structura Target Cache înainte de a fi căpătat o anumită confidență, și care ulterior s-ar dovedi corect predicționate**. Superioritatea modului de inserare / evacuare după LRU minim față de modul după confidență minimă se atenuază odată cu creșterea gradului de asociativitate al structurii Target Cache și implicit cu

diminuarea miss-urilor de conflict (câștigul în acuratețe scade la 1.63%). De asemenea, deși se aștepta ca evacuarea după MPP să genereze o acuratețe de predicție mai mare decât în celelalte două situații, se pare că valorile minime (posibil 0) ale câmpului MPP specific fiecărei locații din set să fie influențate de confidența situată în mai multe cazuri pe 0.

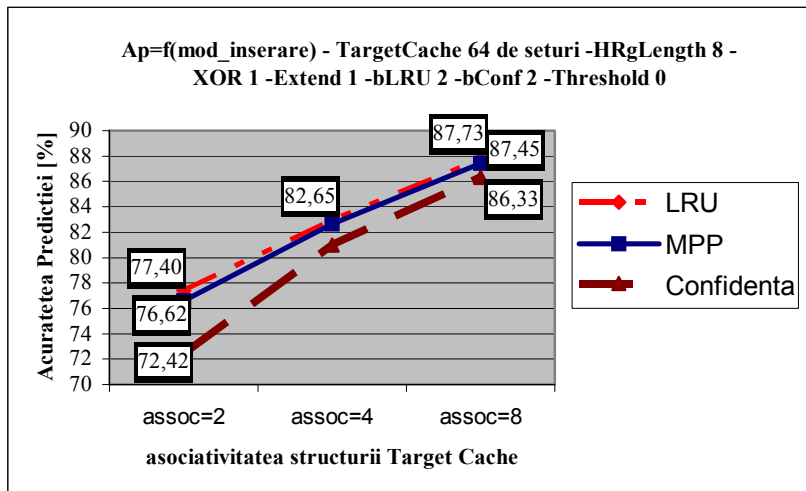


Figura 7.14. Influența modului de inserare / evacuare în / din set asupra acurateții de predicție a salturilor indirecte. Statisticile reprezintă mediile aritmetice obținute pe benchmark-urile SPEC'95 bogate în salturi indirecte.

Ținând cont de experimentul lui Driesen dar mai ales de rezultatele limitate, din punct de vedere al acurateții predicției salturilor indirecte pe o serie de benchmark-uri indiferent de îmbunătățirile arhitecturale aduse, pe baza comportamentului dinamic am realizat o statistică (vezi figura 7.15) privind aritatea salturilor (*monomorfe* – generează dinamic un singur target, *duomorfe* – generează dinamic 2 target-uri distincte, respectiv *polimorfe* – salturile se efectuează la mai mult de două adrese destinație distincte).

Din punct de vedere dinamic pe 7 benchmark-uri SPEC'95 aritatea salturilor indirecte este ilustrată în figura 7.15. Din punct de vedere static, pe aceleași programe de test, se poate spune că, 67% din salturile indirecte sunt monomorfe, 7% sunt duomorfe și 26% sunt polimorfe.

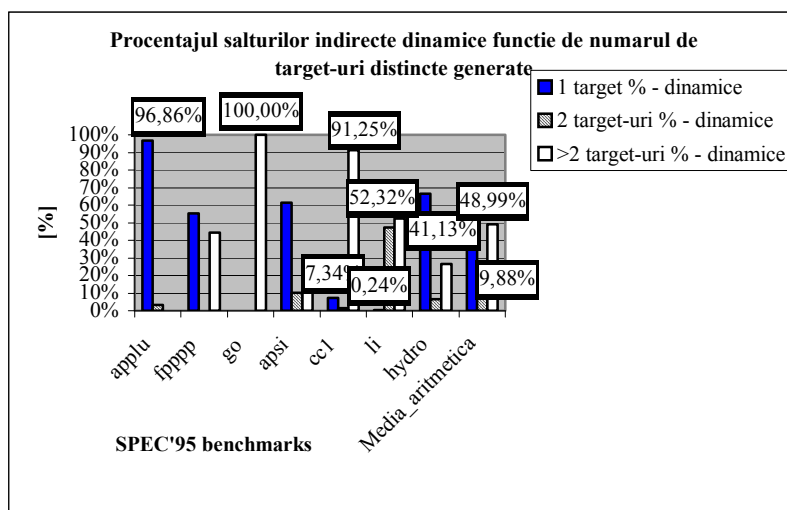


Figura 7.15. Aritatea salturilor indirecte – din punct de vedere **dinamic**

Observând graficul (figura 7.15), concluziile anterior enunțate pe parcursul acestui capitol de rezultate devin practic mult mai clare. Astfel:

- Pe benchmark-ul applu.ss acuratețea predicției de 92% obținută chiar și cu un predictor de tip LastValue (fără istorie) este justificată dacă ținem cont de faptul că din totalul salturilor indirecte dinamice 96% sunt monomorfe.
- Mediile aritmetice pe cele 4 programe de test bogate în salturi indirecte (apsi, cc1, li, hydro), în concordanță cu rezultatele obținute de Driesen, exprimă faptul că deși salturile monomorfe sunt prezente într-o proporție covârșitoare din punct de vedere static – 72.04% comparativ cu cele duomorfe – 9.27% și respectiv polimorfe – 18.69%, ele reprezintă doar 33.92% din totalul branch-urilor indirecte dinamice executate. Salturile duomorfe constituie 16.47% din totalul celor indirecte dinamice iar cele polimorfe, deși puține din punct de vedere static, dinamic constituie aproape jumătate din total (49.61%).
- Rezultatele slabe privind acuratețea predicției salturilor indirecte pe benchmark-ul cc1.ss (**maxim – 75.41%** pentru o tabelă Target Cache cu 128 intrări, 8-way asociativă, corelată cu ultimele 8 salturi condiționate, având implementat și mecanism de confidență pe 2 biți), superioare totuși celor raportate de alți cercetători [Dri98, Cha97], sunt justificate dacă ținem cont de faptul că din punct de vedere dinamic **salturile polimorfe** sunt prezente în proporție de **91.25%**, iar un singur salt indirect static poate avea chiar și 44 de adrese destinație – vezi subcapitolul 4.3.

- Tot procentajul substanțial de salturi polimorfe dinamice 52.32% și dispersia ridicată a target-urilor anumitor salturi – vezi subcapitolul 4.3, respectiv insignifiant – 0.24% monomorfe dinamice, stă la baza limitării acurateții predicției (în ciuda modificărilor arhitecturale aduse structurii Target Cache) pe benchmark-ul li.ss (**90.54%** - în aceleași condiții de simulare ca cele expuse anterior referitoare la benchmark-ul cc1.ss).

Pornind de la predictoarele cascade pe mai multe niveluri și respectiv hibride [Dri98b, Dri98c], am implementat software și simulat o structură hibridă de predicție, compusă dintr-un predictor de tip LastValue și cel mai bun predictor contextual determinat în urma simulărilor (vezi figurile 7.5 și 7.6), cu istorie și pattern fix, în două ipostaze: (istorie redusă – 32 și pattern de lungime 3 biți, sau istorie bogată – 256 și pattern de lungime 6 biți) selecția făcându-se pe bază de aritate, după cunoașterea în prealabil a informațiilor de profil aferente fiecărui salt indirect.

Predictorul hibrid – o singură structură face predicție la un moment dat - (LastValue + Contextual), **cu selecție bazată pe aritate îmbunătățește acuratețea predicției salturilor indirecte cu 3.03% comparativ cu un predictor contextual având o istorie de 32 și pattern de căutare 3, respectiv 2.44% față de un predictor contextual care beneficiază de o istorie mai bogată (256 de target-uri și pattern de căutare 6)**. Câștigul în acuratețe (vezi figurile 7.16 și 7.17) este mai pronunțat dacă comparația se face cu cea mai performantă structură de predicție de tip TargetCache (8 way asociativă, 128 seturi, HRgLength=8, Extend=1, bLRU=2, bConf=2, Threshold=0): 3.20% dacă predictorul hibrid reține o istorie redusă a target-urilor (ultimele 32) și respectiv 5.42% dacă predictorul hibrid este caracterizat de o istorie bogată.

Un fapt, doar în aparență “*surprinzător*”, îl reprezintă acuratețea de predicție superioară schemei hibride, obținută de structura Target Cache pe benchmark-ul li.ss. O primă explicație poate consta în procentajul extrem de scăzut (0.24%) de salturi indirecte dinamice monomorfe (vezi figura 7.15), întrucât, în cazul benchmark-ului cc1.ss caracterizat de 7.34% salturi dinamice monomorfe folosind structura hibridă de predicție rezultă o creștere a acurateții predicției de la 75.41% (cu Target Cache) la 91.86%. O altă explicație ar putea fi caracterul nerepetitiv al target-urilor și o dispersie de cel mult 14 target-uri distincte (vezi subcapitolul 4.3) aferente aceluiași salt indirect. Practic, un singur salt indirect static dintr-un total de 16 prezente în urma simulării primelor 5.000.000 de instrucțiuni dinamice, generează 14 target-uri distincte, care pot fi reținute cu succes de un Target Cache cu 128 intrări, 8 way asociativ folosind informație de corelație extinsă.

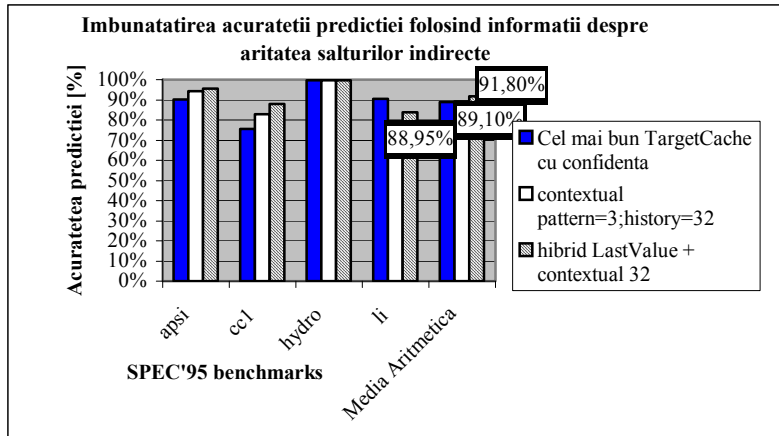


Figura 7.16. Acuratețea predicției salturilor indirecte folosind o structură hibridă și cunoscând informații de profil (istorie **redusă** a predictorului contextual)

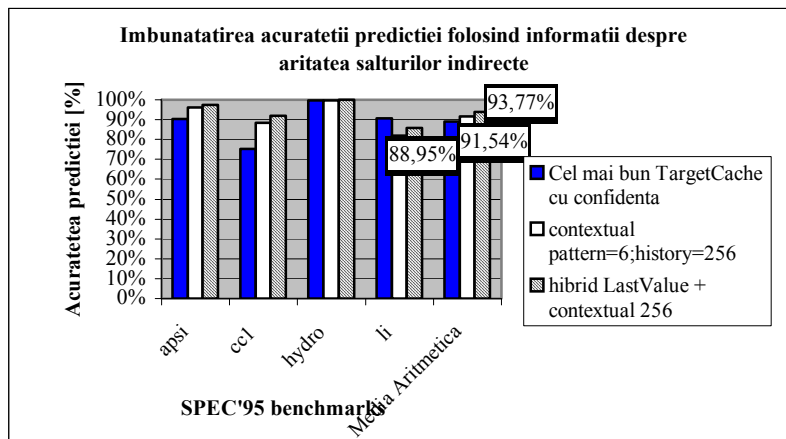


Figura 7.17. Acuratețea predicției salturilor indirecte folosind o structură hibridă și cunoscând informații de profil (istorie **bogată** a predictorului contextual)

7.1.2. SIMULAREA PROPRIILOR PROGRAME DE TEST. REZULTATE.

În acest subcapitol sunt prezentate rezultatele statistice obținute în urma simulării propriilor programe de test descrise în subcapitolul 4.2: Spre deosebire de benchmark-urile SPEC'95 simulate care sunt în întregime procedurale, scrise în limbajul C, dintre cele cinci surse propuse de autor două sunt obiectuale (*back_*, *Moștenire_simplă3*) scrise în C++.

	Instrucțiuni statice de salt indirect	Totalul de instrucțiuni statice de salt	% instrucțiuni statice de salt indirect	Instrucțiuni dinamice de salt indirect	Totalul de instrucțiuni dinamice de salt	% instrucțiuni dinamice de salt indirect
Moștenire simpla1	14	585	2.393	512	22226	2.303
Back_	22	639	3.443	145236	4636747	3.132
Hanoi	12	591	2.030	1291	211390	6.107
Moștenire simpla3	11	475	2.316	72	9307	0.774
Qsort	11	376	2.926	61	3966	1.538
Sort	14	632	2.215	1035	1272867	0.081

Tabelul 7.9. Contorizarea numărului de salturi indirecte statice / dinamice din totalul instrucțiunilor executate dinamic din propriile programe de test

Deși procentajul instrucțiunilor dinamice de salt indirect este redus (sub 7%) rezultatele exprimate în tabelul 7.9 sunt în concordanță cu cele obținute pe benchmark-urile SPEC'95. În timp ce numărul salturilor indirecte statice este fix fiind și justificat în subcapitolul 4.2, numărul instrucțiunilor dinamice este variabil și depinde de parametrii de intrare.

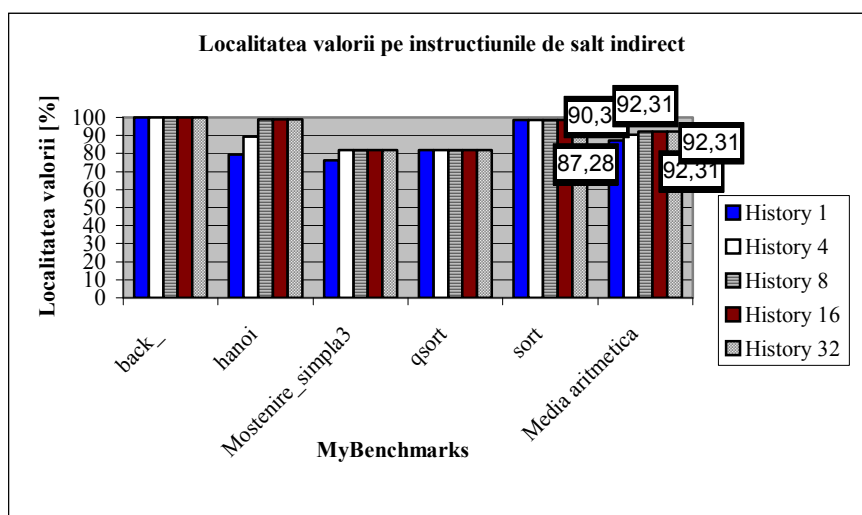


Figura 7.18. Localitatea valorilor pentru instrucțiuni de salt indirect ($j \leq x$, unde $x \neq 31$)

În medie aritmetică se observă o localitate extrem de ridicată pe testele proprii propuse (92%). Rezultatul surprinzător obținut pe testul **back_** se poate datora atât numărului relativ ridicat de instrucțiuni dinamice de salt

indirect cât și **numărului redus de obiecte (declarate individual și nu în cadrul unei structuri de date cu legături - tablouri sau liste) și metodelor aferente apelate recursiv**, mărturie în acest sens stând localitatea de doar 80% determinată pe benchmark-ul **Mostenire_simpla3**.

La o privire atentă a rezultatelor statistice (sursa de date a chart-ului din figura 7.18) se observă că doar 24 de instrucțiuni de salt indirect cauzează pierderea de localitate pentru **back_** respectiv 11 instrucțiuni pentru **qsort** practic *miss*-urile *de start rece* - oarecum normal existând un singur apel al funcției `qsort()`, însă ținând cont de numărul de instrucțiuni dinamice de salt indirect rezultă gradele de localitate respective.

Se observă că o istorie de 4 target-uri memorate poate fi considerată **practic optimă** (localitate de doar 90% și nu 92%) în vederea predicției. În acest sens am determinat acuratețea predicției folosind predictorul PPM complet în care *pattern*-ul de căutare variază între 1 și 4 într-un context de 32 target-uri memorate pentru fiecare instanță de salt indirect (vezi figura 7.19). Întrucât rezultatele simulărilor pentru un context mai bogat au evidențiat o aceeași concluzie (*pattern* optim de 4 valori) s-a optat pentru afișarea unui singur grafic.

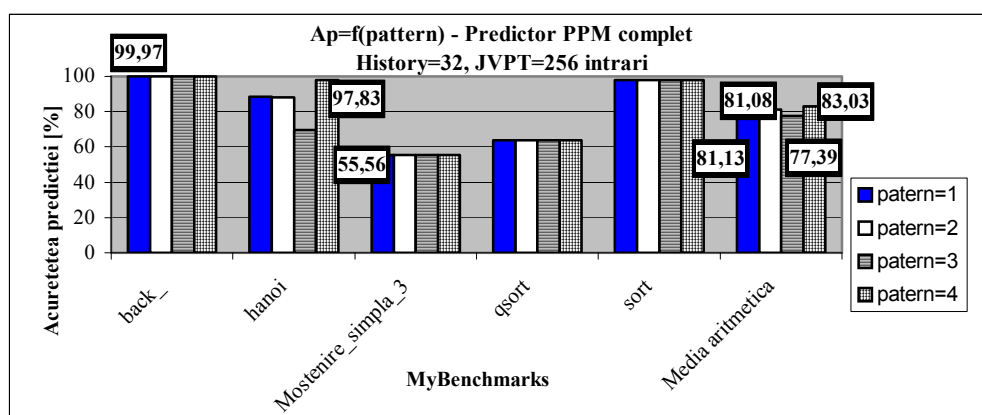


Figura 7.19. Acuratețea predicției aferentă instrucțiunilor de salt indirect găsite în propriile programe de test, utilizând un predictor de tip PPM complet, în funcție de lungimea *pattern*-ului.

Acuratețea predicției foarte scăzută pe benchmark-urile **mostenire_simpla3** și **qsort** (indiferent de lungimea *pattern*-ului) demonstrează **ineficiența predictorului PPM complet în cazul utilizării de masive eterogene** (de obiecte sau alte structuri de date). Este posibil ca un predictor incremental să se comporte mult mai bine în cazul celor două benchmark-uri. De asemenea, se observă că **recursivitatea**, întâlnită în

cadrul benchmark-urilor *back_* și *hanoi*, **poate fi exploatată cu succes cu ajutorul predictorului PPM complet.**

Rezultatul evidențiat de figura 7.19 arată un pattern optim de 4 valori și în același timp faptul că pe teste propuse acuratețea de predicție nu este influențată de *history* (contextul de target-uri memorate) poate și datorită numărului redus de instrucțiuni dinamice de salt indirect.

	History = 32, JVPT=256							
	Pattern 1		Pattern 2		Pattern 3		Pattern 4	
	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]	Pred [%]	Unpred [%]
back	99.99	0	99.99	0	99.99	0	99.99	0
hanoi	90.05	15.38	90.10	29.41	78.45	1.50	99.84	14.29
Moștenire simpla_3	100	38.09	100	38.09	100	28.57	100.00	28.57
qsort	100	0	100	0	100	0	100	0
sort	100	0	100	0	100	0	100	0
Media aritmetica	98.01	10.70	98.02	13.50	95.69	6.02	99.97	8.57

Tabelul 7.10. Procentajul de instrucțiuni de salt indirect predictibile identificate corect și nepredictibile prezise greșit de către automatul de clasificare aferent predictorului PPM complet.

În cazul predictoarelor de tip Target Cache *mapate direct* sau "2-way" asociative, creșterea tabelii de predicție (a numărului de seturi) determină îmbunătățirea acurateții de predicție (vezi figura 7.20). Întrucât cu o istorie de 4 target-uri reținute pentru fiecare instrucțiune dinamică de salt indirect se obține o localitate de peste 90%, sunt îndreptățite rezultatele **privind acuratețea predicției folosind un Target Cache "4-way" asociativ, creșterea fiind asimptotică odată cu creșterea numărului de seturi.**

Un ultim aspect menționat, este acela că pe teste propuse de autor, inevitabil mai simple decât niște benchmark-uri standardizate, acuratețea de predicție s-a dovedit mai mare în cazul schemei Target Cache decât în cazul predictorului PPM complet (în primul rând datorită celor două aplicații care lucrează cu tablouri). Cu toate acestea, deși ele au fost propuse pentru a evidenția caracteristici de program care generează instrucțiuni de salt indirect, în multe aspecte programele de test ale autorului s-au comportat similar cu benchmark-urile SPEC'95. O încercare de explicație ar fi următoarea: **Target Cache-ul implementat nu derivă din PPM.** El se bazează pe context, dar **nu contorizează frecvența acestuia** (ca PPM-ul).

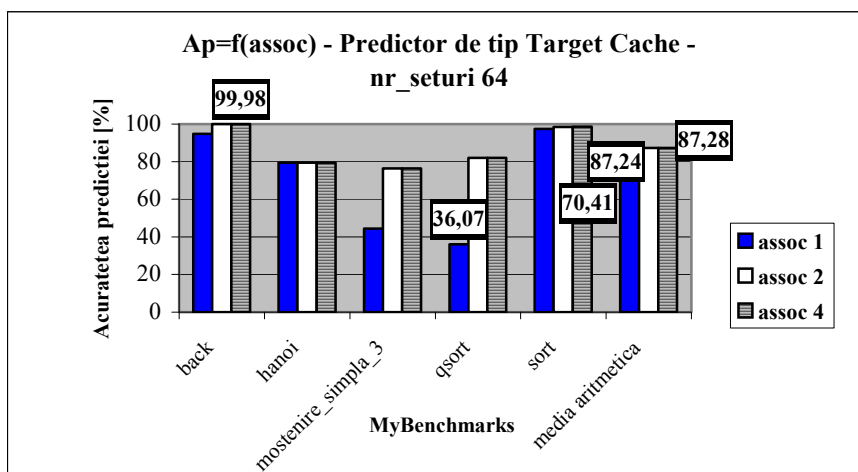


Figura 7.20. Acuratețea predicției aferentă instrucțiunilor de salt indirect găsite în testele proprii, utilizând un predictor de tip TargetCache, în funcție de gradul de asociativitate.

7.2. CERCETĂRI PRIVIND VECINĂTATEA ȘI PREDICȚIA VALORILOR INSTRUCȚIUNILOR

Rezultatele prezentate în această secțiune au fost determinate cu ajutorul simulatorului *Value Predictor* descris în subcapitolul 6.1. Pornind de la o arhitectură superscalară minimă, se va studia în ce măsură va fi afectată performanța simulatorului de către variația parametrilor acestuia. Toate simulările din acest subcapitol au fost făcute pentru 5.000.000 de instrucțiuni dinamice în cazul benchmark-urilor SPEC'95. Pentru început vrem să determinăm *câtă localitate a valorilor există în benchmark-urile SPEC?* Una din problemele importante pusă în proiectare constă în "*Cât de multă istorie să fie folosită în predicție?*" **Aceste întrebări sunt esențiale și corelate. Gradul de localitate măsurat pentru ultimele k valori produse ($History=k$) ne oferă o limită ultimativă a acurateții de predicție obținabile. În plus determinarea lui k optim implică faptul că acesta va fi folosit în implementarea predictorului contextual optim.**

Figura de mai jos evidențiază influența istoriei asupra nivelului de localitate a valorilor în cazul utilizării **adresei instrucțiunilor** de tip Load

(PC). De remarcat că ultima coloană (Average) constituie media aritmetică obținută pe benchmark-urile simulate.

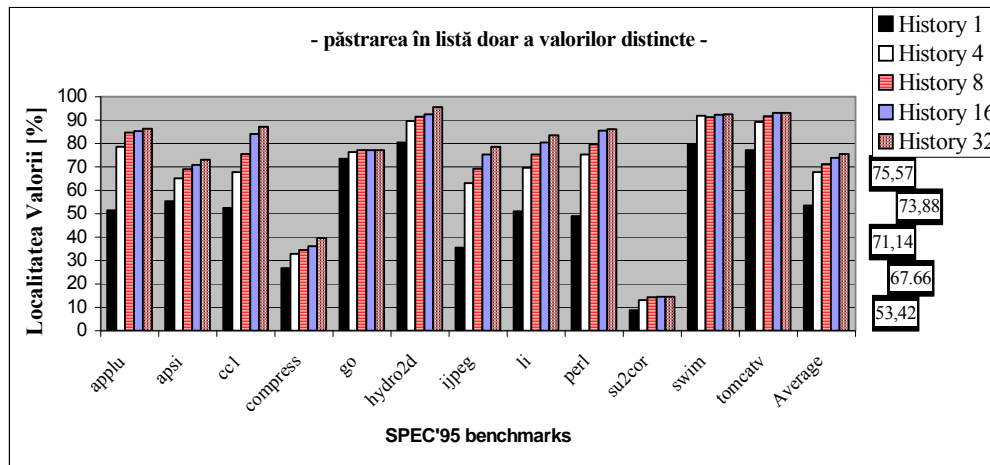


Figura 7.21. Localitatea valorilor utilizând adresa (PC) instrucțiunilor de tip Load

Exploatând corelația dintre adresele instrucțiunilor Load și valorile citite din memorie de către acestea, având o "adâncime" a istoriei de 1 (regăsirea aceleiași valori în resursa asignată ca și în cazul precedentului acces), programele de test exprimă o localitate a valorii de peste 53% în timp ce extinzând verificarea în spațiul ultimelor 8 accese la memorie se obține o localitate de peste 71%. Rezultatele subliniază că majoritatea instrucțiunilor Load statice aferente unui program exprimă o variație redusă a valorilor pe care le încarcă pe parcursul execuției. Maximul de localitate obținut cu o istorie de 32 de valori este de **75.57%**.

Din următoarea figură reiese influența istoriei asupra nivelului de localitate a valorilor în cazul în care s-a utilizat **adresa datei**. Tendința este similară doar că valorile sunt mai mari: *cu cât crește istoria (history=1 ⇒ localitatea=69%) cu atât crește localitatea valorii (history=32 ⇒ localitatea=85%)*.

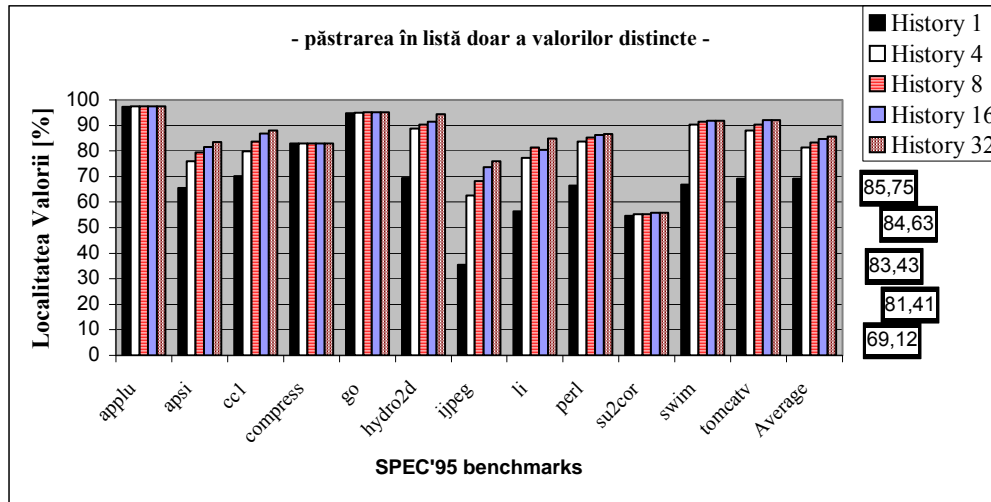


Figura 7.22. Localitatea valorilor utilizând adresa datei pentru instrucțiunile de tip Load

Din cele două figuri reiese că pentru instrucțiunile Load, cu o **istorie de 8 valori** se obține optimul de localitate. Pentru „adâncimi” mai mari ale istoriei nu se obțin creșteri semnificative ale gradului de vecinătate. Vom compara acum pentru o istorie de 8, rezultatele obținute din punct de vedere al localității prin utilizarea adresei instrucțiunii versus cele obținute prin utilizarea adresei datei.

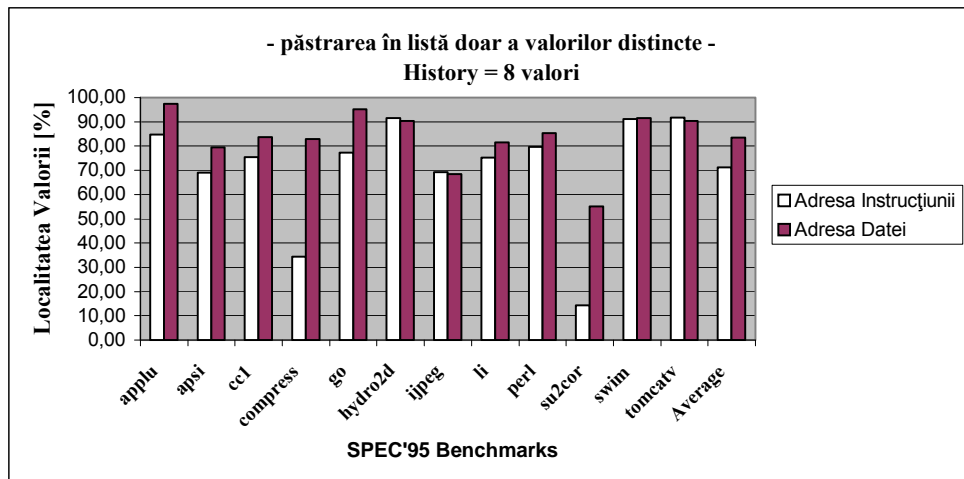


Figura 7.23. Localitatea valorilor cu adâncimea istoriei de 8 (PC vs. DataAddress).

Așa cum se poate observa, s-au obținut rezultate mai bune (cu **17.28%**) atunci când s-a folosit adresa datei. Dorim să exploatăm această localitate a valorilor prin utilizarea diferitelor tehnici de predicție prezentate în subcapitolele 2.2.2÷2.2.4. Trebuie precizat că **localitatea valorii exprimată în funcție de "adresa datei", deși mai mare, are dezavantajul că în procesul de predicție după adresa datei se va pierde timp întrucât predicția se va face în faza ID (decode) - pentru modul de adresare imediat, direct și indirect registru, respectiv ALU - pentru modul de adresare indexat, după calculul adresei. În consecință, deși probabil acuratețea predicției va fi mai mare, timpul de execuție va fi mai relaxat și performanța globală va avea de suferit față de schema clasică.**

Un alt deziderat al proiectului l-a constituit exploatarea gradului de localitate exprimat și de alte tipuri de instrucțiuni (cele aritmetico-logice – ALU, despre cele de salt indirect – Jindir s-a arătat în subcapitolul 7.1.1). Figura următoare prezintă localitatea valorii extrasă din suita SPEC'95 pentru instrucțiunile aritmetico-logice.

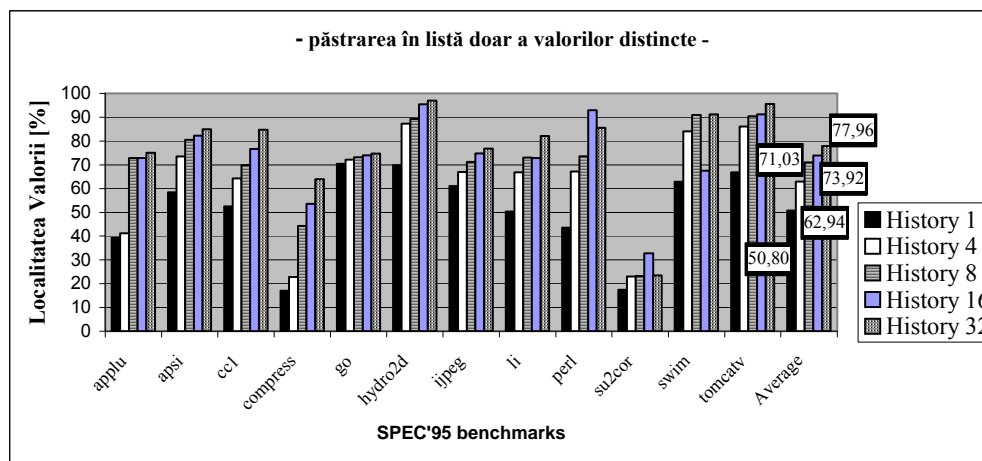


Figura 7.24. Localitatea valorilor utilizând adresa (PC) instrucțiunilor de tip ALU

Privind comparativ rezultatele grafice din figurile 7.21 și 7.24, un aspect foarte interesant de remarcat se referă la faptul că **pentru un grad de localizare redus (istorie ∈ {1,4}) instrucțiunile load exprimă o localitate mai bună decât cele aritmetico-logice, însă într-un context bogat (istorie ∈ {16,32}) situația se inversează.** Rezultatul nu surprinde însă deoarece este firească **variația mult mai rapidă a valorilor prin regiștri decât prin locațiile de memorie.** Cu toate acestea după cum se va vedea și

în figura 7.35 păstrând o plajă de 32 de valori pentru fiecare registru este suficient să captăm o localitate ridicată ($\approx 76\%$).

Gradul ridicat de vecinătate a valorilor exprimat de resursele anterior amintite (instrucțiuni Load, ALU, sau adrese ale datelor din memorie) conduce în mod natural la ideea predicției pe aceste resurse. Figurile următoare evidențiază în ce măsură este influențată acuratețea de predicție de dimensiunea tabelii de predicție și cât de eficient este automatul de clasificare implementat. În primă fază s-a considerat un predictor fără istorie de tip *last value*. Graficele ilustrează acuratețea predicției pentru tabelii de dimensiuni mai mari sau egale cu 64 de locații. Simulările au fost efectuate și pentru tabelii de dimensiuni de 8, 16 și 32 de locații însă acuratețea predicției, depășește extrem de rar 20% (doar pe 2 din cele 12 benchmark-uri și cu tabelă LastValuePredictionTable – LVPT asociativă). Deși statisticile au fost realizate atât pentru structuri de predicție asociative cât și mapate direct, datorită cantității uriașe de informație care rezultă și care ar îngreuna poate modul de înțelegere și interpretare, s-a optat pentru ilustrarea grafică doar a rezultatelor privitoare la o tabelă de predicție asociativă. Rezultatele simulărilor (vezi figurile 7.26 și 7.27) arată că o creștere a capacității tabelii de predicție conduce la îmbunătățirea semnificativă a acurateții de predicție, dimensiunea optimă a tabelii de predicție fiind de **512 de locații**.

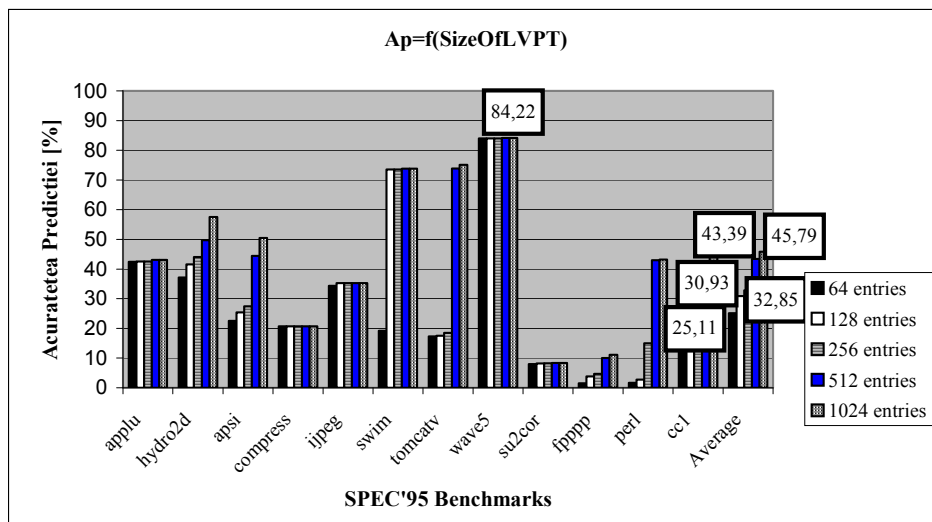


Figura 7.25. Acuratețea predicției utilizând adresa instrucțiunii (PC) și o tabelă asociativă în funcție de dimensiunea tabelii LVPT – instrucțiuni Load

Valorile foarte mici în medie armonică pentru acuratețea de predicție se datorează în primul rând celor trei benchmark-uri SPEC'95 (su2cor, fpppp, perl) cu rezultate foarte slabe din acest punct de vedere (< 10%).

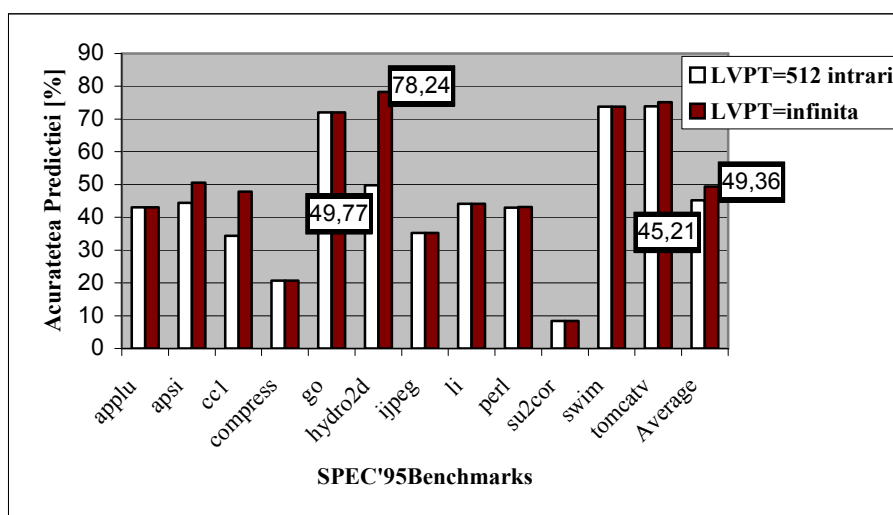


Figura 7.26. Acuratețea predicției utilizând adresa instrucțiunilor de tip Load și o tabelă asociativă: Studiu comparativ între o tabelă *optimă* din punct de vedere a capacității LVPT și una “*infinită*”

Rezultatele relativ apropiate (diferență de 9.18%) în ce privește acuratețea predicției pe o tabelă LVPT infinită și una cu 512 intrări constituie un motiv în favoarea implementării în hardware a schemei cu 512 locații. Diferențele cantitative dintre graficele (figura 7.25 vs. figura 7.26) se datorează faptului că doar 10 benchmark-uri folosite în simulare au fost identice: la primul programele de test distincte au fost *fpppp* și *wave5* (cu rezultate contradictorii din punct de vedere al acurateții de predicție – cele două extreme) iar la cel din urmă simulări distincte au fost făcute pe *li* și *go*.

SPEC'95 bench	64		128		256		512		1024	
	Pred	Unpr	Pred	Unpr	Pred	Unpr	Pred	Unpr	Pred	Unpr
Applu	87.56	83.52	87.56	83.53	87.57	83.31	87.68	83.55	87.68	83.55
Apsi	96.28	85.05	94.97	78.94	94.32	77.17	93.66	84.36	93.71	89.33
Cc1	94.13	73.83	93.93	74.78	93.52	77.71	93.64	84.18	94.35	89.34
Compre ss	89.07	93.17	89.07	93.17	89.07	93.16	89.07	93.15	89.07	93.15
Fpppp	99.41	78.23	99.17	76.29	98.3	67.12	96.01	87.06	94.96	71.1
Hydro	98.86	69.06	96.41	70.68	95.88	67.26	95.32	69.01	94.91	74.74
Ijpeg	86.87	98.65	87.15	99.5	87.15	99.5	87.15	99.5	87.15	99.5

Perl	83.16	79.62	87.50	77.03	92.11	84.23	95.22	89.34	95.17	89.44
Su2cor	97.49	99.6	97.34	99.59	97.3	99.57	97.2	99.56	97.19	99.57
Swim	95.78	42.02	98.55	76.62	98.55	76.41	97.28	75.56	97.28	75.56
Tomcat	95.48	63.88	95.51	61.88	95.49	40.31	97.78	88.35	97.8	91.24
Wave5	99.3	91.45	99.3	91.26	99.3	91.03	99.29	90.91	99.29	90.98
Medie Aritmetică	96.3	75.36	96.91	82.26	96.94	77.41	97.13	88.28	97.12	88.93

Tabelul 7.11. Procentajul de instrucțiuni Load predictibile și nepredictibile identificate de către Predictorul de tip "*Last Value*". LVPT - asociativă și indexată cu PC-ul instrucțiunii

În tabelul 7.11 coloanele *Pred* reprezintă procentajul acelor Load-uri care au fost clasificate predictibile și predicția a fost corectă, iar în coloanele *Unpr* se află procentajul Load-urilor care au fost clasificate de automat ca fiind nepredictibile și predicția a fost într-adevăr greșită. Automatul de clasificare reprezintă un numărator saturat pe 2 biți și este cel folosit și în cazul instrucțiunilor de salt indirect (vezi figura 5.28). Acesta este incrementat/decrementat cu fiecare predicție corectă/incorectă, respectiv în cazul unei predicții incorecte noua valoare aflată în resursă (registru, locație de memorie) este suprascrisă (istoria fiind unitară în tabela LVPT). Se poate observa, că automatul se comportă destul de bine (a identificat corect peste **96%** din Load-urile predictibile și peste **75%** din cele nepredictibile) dar urmează în viitor să fie analizate și alte automate pentru a găsi pe cel optim. Diferența de ($\approx 4\%$ în cazul load-urilor clasificate predictibile) față de o clasificare ideală poate fi datorată și modului de evacuare "*primitiv*" din tabela LVPT. Slăbiciunea automatului propus constă în faptul că este prea conservator, el clasificând ca nepredictibile multe instrucțiuni care sunt realmente predictibile.

Există însă și variante care schimbă strategia de modificare a valorii bazat pe "*histerezis*". Un exemplu de mecanism de histerezis constă într-un numărator saturat (pe i biți) asociat fiecărei intrări în tabela de predicție. Numărătorul este incrementat/decrementat atunci când predicția este corectă/incorectă, respectiv valoarea memorată în tabelă este evacuată numai când valoarea indicată de către număratorului asociat este sub un prag prestabilit (poate fi 2^{i-1}). Un alt mecanism de histerezis nu modifică valoarea predicționată în tabelă până când noua valoare nu a apărut repetitiv de un anumit număr de ori [se asignează un grad de încredere fiecărei valori existente la o anumită locație (adresă) în memorie (PC/data address)]. Dezavantajul că trebuie stocat în memorie (cache sau memoria centrală) gradul de încredere împreună cu valoarea respectivă [Flo03, Lip96].

Rezultatele unor simulări nereprezentate aici, care ilustrează acuratețea predicției utilizând adresa datei și o tabelă asociativă în funcție de

dimensiunea tabelii LVPT, exprimă următoarea concluzie. **Acuratețea de predicție în cazul predictorului LVPT cu tabela asociativă folosind adresa datei este mai mare față de cazul în care accesul în tabela LVPT se face cu adresa instrucțiunii**, aspect observat și la vecinătatea valorii instrucțiunilor de tip Load și este în concordanță cu rezultatele altor cercetători [Cal99b]. Din punct de vedere al *timing*-ului câștigul nu este la fel de pronunțat întrucât valoarea prezisă va fi înaintată instrucțiunilor dependente aflate în așteptare mai târziu (cu unul sau două nivele în structura pipeline) decât în cazul utilizării adresei instrucțiunii, care se cunoaște încă de la începutul fazei de aducere IF.

Următoarea figură (figura 7.27) arată cum este influențată acuratețea predicției de tipul tabelii, care poate fi mapată direct sau asociativă și de tipul adresei care indexează tabela de predicție (adresa instrucțiunii sau adresa datei).

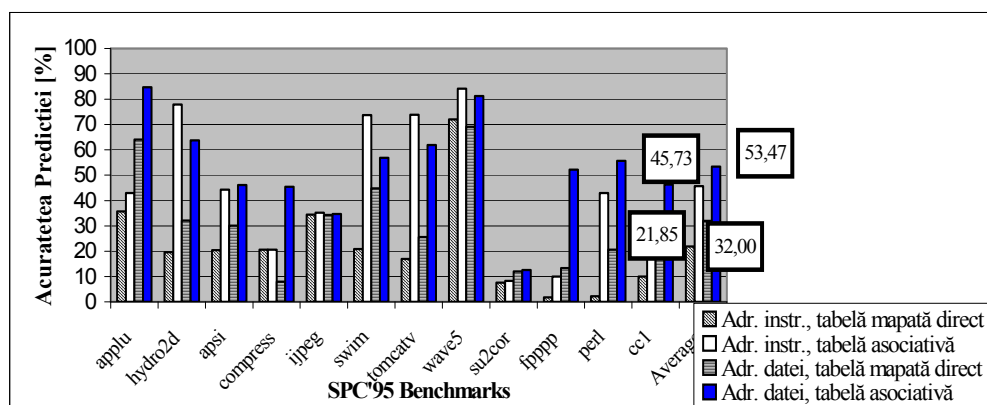


Figura 7.27. Acuratețea predicției utilizând o tabelă optimă din punct de vedere a capacității sale - 512 intrări în funcție de tipul tabelii LVPT – instrucțiuni Load

Se observă (în medie) superioritatea acurateții de predicție în cazul folosirii unei table de predicție (LVPT) asociativă față de una mapată direct (**45.73%** - indexare cu PC, respectiv **53.47%** - indexare cu adresa datei vs. **21.85%** - indexare cu PC, respectiv **32.00%** - indexare cu adresa datei). Acest fapt se datorează fenomenului de interferență care apare în cazul tabelilor mapate direct. Conform concluziei stabilite la studiul localității în funcție de strategia de indexare a tabelilor (PC vs. DataAddress) - vezi figura 7.23 - rezultatele sunt similare și din punct de vedere al acurateții de predicție (acuratețea de predicție prin indexarea LVPT cu PC mai mică decât cea obținută prin indexarea LVPT cu DataAddress). Rezultatele obținute sunt în concordanță cu cele obținute de Lipasti [Lip96] care

considerând că se memorează ultima valoare produsă de către o anumită instrucțiune a obținut o acuratețe de predicție (*medie aritmetică*) de 49%. De asemenea, din cercetările lui Lipasti rezultă că, memorând ultimele 4 valori produse de către o anumită instrucțiune și că abilitatea predictorului de a alege valoarea corectă este perfectă, acuratețea de predicție medie este de 61%. Întrucât localitatea medie obținută prin simulare, având o istorie de 4 valori este de 67.66% rezultă o diferență relativ mare între predicție și localitate. Este posibil ca această diferență să fie datorată și faptului că atât eu cât și Lipasti am *“forțat”* puțin în obținerea localității, prin **păstrarea în istoria de valori aferente unei instrucțiuni doar a valorilor distincte**, care pot apare nu neapărat la fiecare instanță a respectivei instrucțiuni (abordare propusă de Lipasti).

Astfel, în continuare am efectuat simulările privind localitatea dar în lista de valori rezultate pentru o anumită instrucțiune am păstrat ultimele *“history”* valori, indiferent dacă unele dintre acestea s-au repetat (**nedistincte**). **Localitatea valorilor obținută pentru ultimele „history” instanțe ale aceleiași instrucțiuni Load statice deși inferioară cazului în care s-au luat în calcul ultimele „history” valori distincte generate de aceeași instrucțiune Load statică, este mai realistă.**

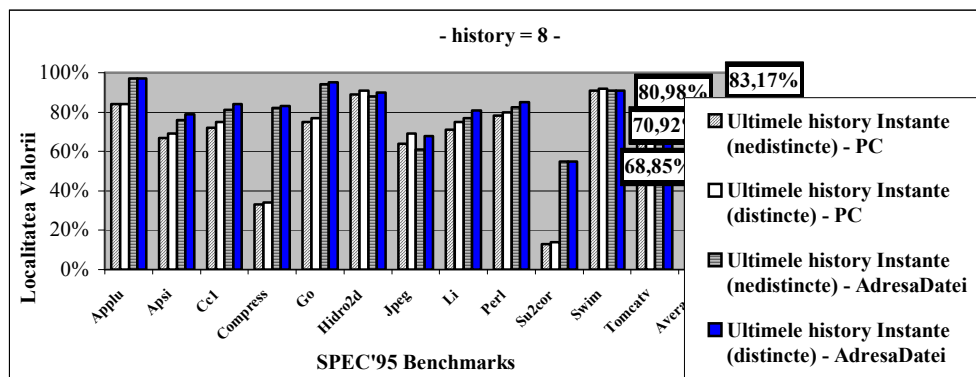


Figura 7.28. Studiu comparativ privind localitatea valorii pentru instrucțiuni de tip Load în condițiile unei istorii considerate optimă

Se poate stabili practic o corelație între acuratețea predicției valorilor obținută cu o schemă de tip „Last Value” (LVPT) și acest tip de localitate (figura 7.27 vs. figura 7.28). De fapt, structura LVPT (sau alta) va stabili o predicție pentru fiecare instrucțiune Load, indiferent dacă se va genera o valoare identică sau nu față de cele precedente. După cum se observă în figura 7.29, fără nici o excepție curba acurateții de predicție modelează curba localității valorilor (cazul ultimelor *history* instanțe).

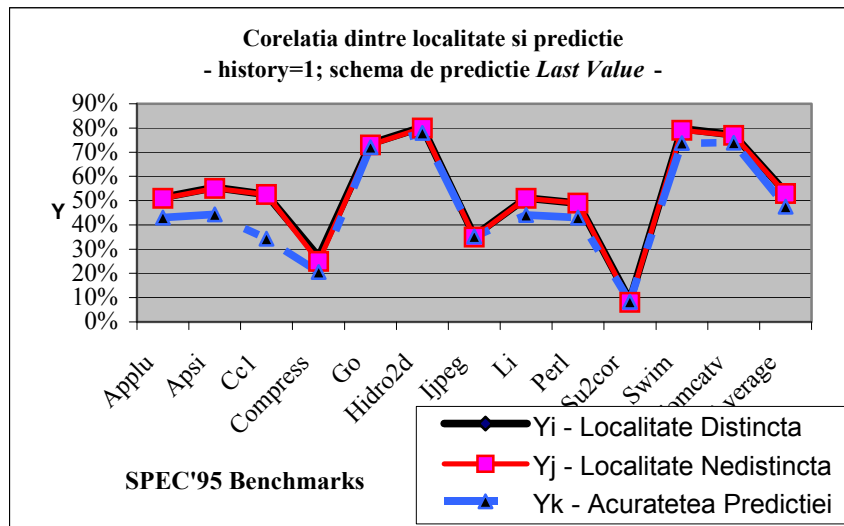


Figura 7.29. Determinarea gradului de corelație dintre localitatea valorii pentru instrucțiunile de tip Load și acuratețea predicției determinată cu o structură de tip LastValue infinită ca și capacitate

Pentru determinarea gradului de corelație existent între localitatea valorii aferentă instrucțiunilor Load și acuratețea predicției obținută cu o structură de tip *LastValue* infinită ca și capacitate au fost introduse două metrici ($M_1 = \sum(y_i - y_k)^2$ și $M_2 = \sum(y_j - y_k)^2$) conform celor două abordări prezentate (Lipasti vs. proprie), unde y_i , y_k și y_j au semnificația din figura 7.29. Acestea reprezintă distanța Hamming dintre gradul de localitate respectiv acuratețea de predicție obținute pe fiecare din benchmark-urile SPEC'95 (mai puțin *perl* care are un comportament ciudat). Deși diferențele dintre M_1 (0.0697) și M_2 (0.0650) sunt foarte mici, valoarea cea mai apropiată de zero este a lui M_2 , generând concluzia că, **curba acurateții de predicție este modelată mai bine de curba localității valorilor aferentă ultimelor instanțe (chiar și nedistincte) de instrucțiuni Load**. Diferența redusă între cele două metrici se datorează condițiilor problemei - history = 1 (pentru localitate) și predictor fără istorie (*LastValue*). O generalizare a acestei probleme de corelație (history>1) poate fi făcută doar în condițiile utilizării unui alt tip de predictor (predictoare cu istorie: incrementale, contextuale, PPM complet). După cum se observă în figura 7.28 cu cât parametrul *history* este mai mare este de așteptat ca diferența dintre M_1 și M_2 să se accentueze.

Experimentele anterioare le-am repetat și pe benchmark-urile Stanford. Deși geneza benchmark-urilor Stanford [Col93] a avut loc mult mai devreme decât cea a programelor de test SPEC'95, iar numărul

instrucțiunilor dinamice este foarte mic (într-un singur caz peste 1.000.000) și cu toate că aria de aplicabilitate a primelor nu se extinde și la aplicațiile grafice și multimedia, rutine critice ale sistemelor de operare, arhivatoare, compilatoare etc, din punct de vedere al localității valorilor procentajul obținut pe testele Stanford (prin măsurători centrate atât pe producător - instrucțiune cât și pe memorie) este similar cu cel al benchmark-urilor SPEC'95. Folosirea structurilor de date regulate (tablouri uni și bidimensionale) favorizează obținerea de grade de localitate de peste 80% în medie armonică atunci când se folosește adresa datei.

O altă investigație realizată a urmărit determinarea dimensiunii optime a tabeli de predicție LastValue în cazul instrucțiunilor aritmetico-logice (vezi figura 7.30).

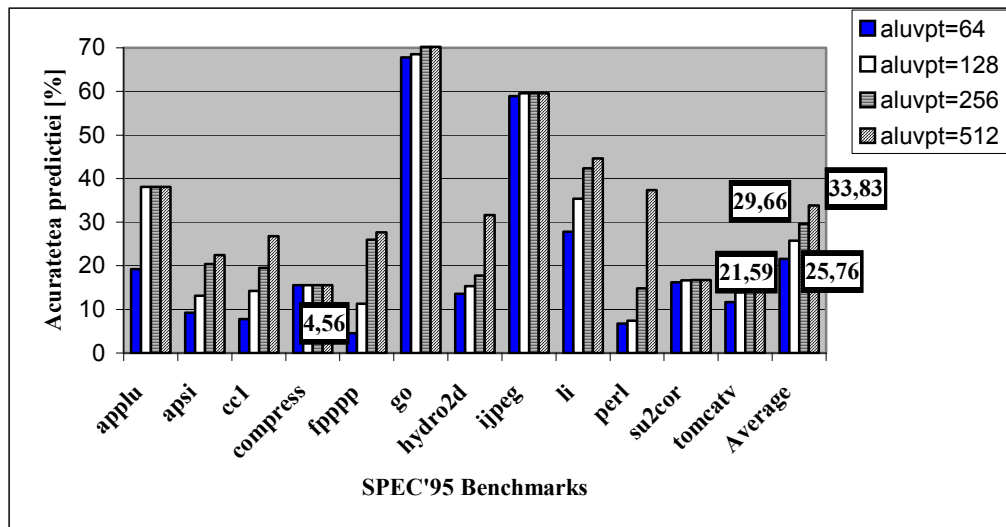


Figura 7.30. Acuratețea predicției instrucțiunilor de tip Aritmetico-Logic utilizând o tabelă asociativă în funcție de dimensiunea tabeli ALUPT

Rezultatele destul de modeste atât în cazul instrucțiunilor Load (45.73%) dar mai ales ALU (33.83%), impun implementarea unor predictoare care folosesc în predicție istoria valorilor. Investigațiile următoare se referă doar la instrucțiunile de tip Load. Figura 7.31 reprezintă acuratețea predicției în cazul folosirii predictorului incremental. Consecință imediată la gradul superior de localitate pe care adresele de date le au față de cele de instrucțiuni, predicția este mai bună în cazul unui predictor ce folosește ca index adresa datei. Din păcate și timpul de predicție va fi în acest caz mai mare.

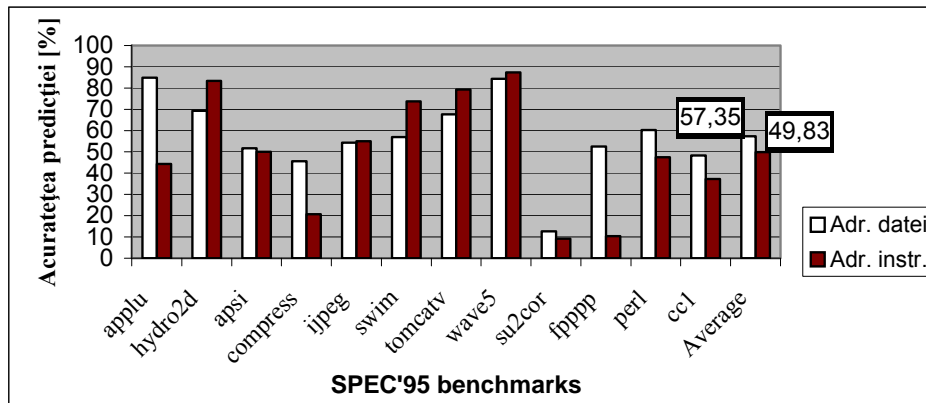


Figura 7.31. Acuratețea predicției pentru predictorul incremental

Se va analiza în cele ce urmează influența dimensiunii contextului asupra predicției în cazul predictorilor contextuale de tip PPM. În continuare se va folosi o istorie de 256 de valori.

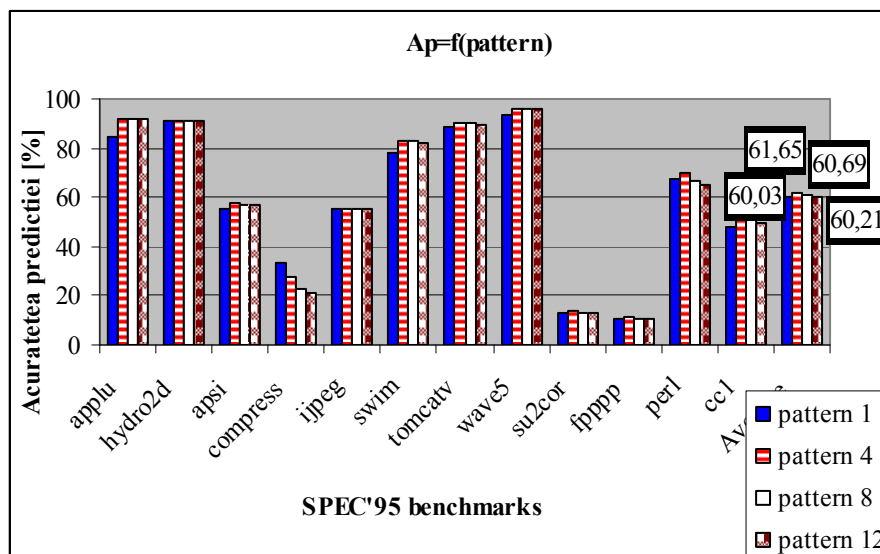


Figura 7.32. Acuratețea predicției utilizând predictorul contextual și adresa instrucțiunii

Deoarece predictorile PPM complete sunt dificil și costisitor de implementat în hardware, urmează să se studieze comportamentul unui predictor contextual de tip PPM simplificat, care să conțină doar două predictorare *Markov*: unul de ordinul (N) și altul de ordinul 0. Astfel dacă predictorul *Markov* de ordinul (N) nu produce o predicție (contextul nu este

găsit în secvența de valori) se activează predictorul *Markov* de ordinul (0), unde N reprezintă dimensiunea contextului.

Analog cu cazul salturilor indirecte, pe baza figurii 7.33 și în urma altor simulări efectuate care au vizat structuri de predicție PPM complet indexate cu adresa datei a rezultat că dimensiunea optimă a contextului (ferestrei de căutare în șirul celor 256 de valori memorate pt. fiecare Load din VHT) este 4. Deși după cum s-ar putea crede un context mai bogat poate conduce la o acuratețe mai ridicată a predicției, simulările arată că începând cu o anumită dimensiune a contextului, acesta se comportă ca “zgomot” și, acuratețea începe să scadă. Dimensiunea tabeli de predicție este de 512 locații, dovedită optimă pentru predictoarele LastValue.

Se va studia în continuare influența dimensiunii contextului asupra predicției în cazul predictoarelor hibride (vezi schema acestora din figura 2.17, subcapitolul 2.2.4).

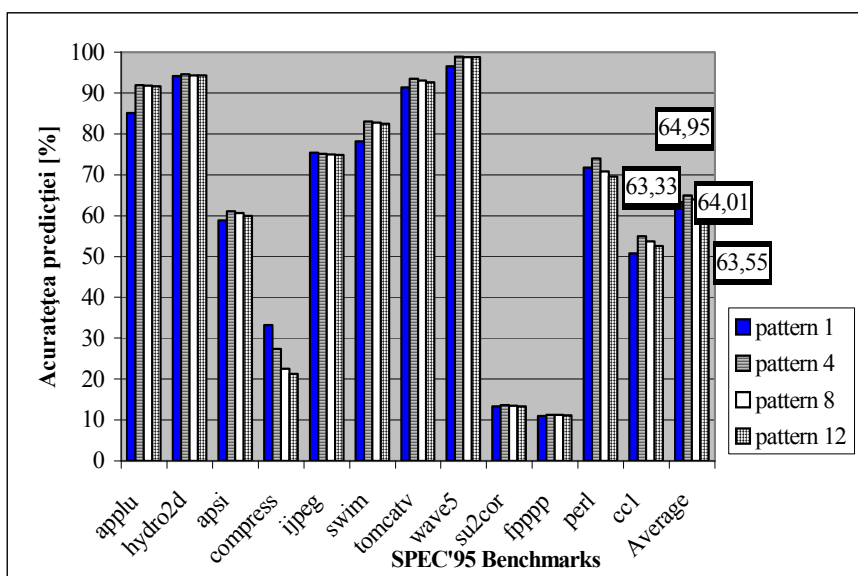


Figura 7.33. Acuratețea predicției utilizând predictorul hibrid și adresa instrucțiunii

La fel ca și în cazul predictorului contextual, din figura 7.33 reiese că dimensiunea optimă a ferestrei curente de căutare pentru predictorul hibrid este 4. În urma simulărilor realizate cu aplicația *ValuePredictor* (descrisă în subcapitolul 6.1) îmbunătățită cu modulul de calcul al *timing-ului* (vezi subcapitolul 6.1.2), un predictor hibrid cu 128 de locații, o istorie de 8 valori și un pattern de 4, generează o acuratețe medie de predicție de **53%** și un

câștig mediu de performanță de **28.87%**, față de o arhitectură standard de procesare care nu înglobează predicția valorilor.

În figura următoare (7.34) se compară cele patru tehnici de predicție utilizate: predicție “*last value*”, predicție incrementală, contextuală și respectiv hibridă.

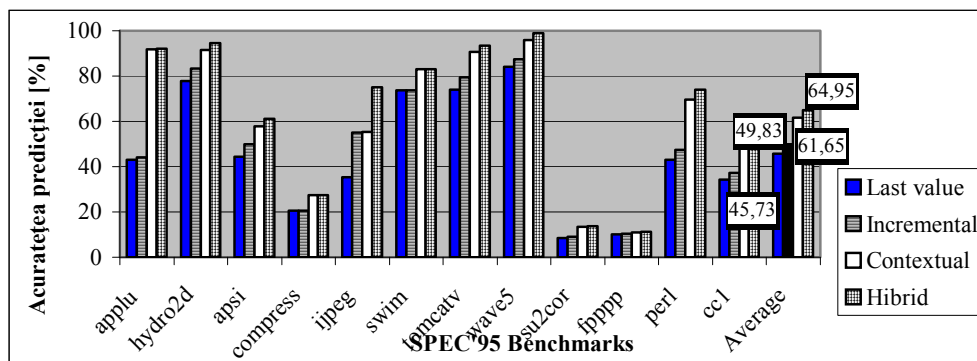


Figura 7.34. Compararea celor patru tehnici de predicție pentru adresa instrucțiunii

Se poate observa sinergismul predictorului hibrid, cu acesta obținându-se cele mai bune rezultate, procentajul mediu al valorilor prezise corect fiind de **65%** în cazul utilizării adresei instrucțiunii și respectiv de **69%** în cazul în care s-a folosit adresa datei (nu s-a mai prezentat graficul în extenso pentru cel de-al doilea caz).

În tabelul următor (7.12) se prezintă creșterea relativă a performanței obținută cu predictorul incremental, contextual și cel hibrid față de predictorul de tip “*last value*”.

Predictor	Predicție cu adresa instrucțiunii	Predicție cu adresa datei
Incremental	4.1 %	3.89 %
Contextual	15.92 %	12.36 %
Hibrid	19.21 %	15.61 %

Tabelul 7.12. Creșteri de performanță obținute cu cele trei **predictoare față de predictorul de tip “*last value*”**

Se poate observa, că adunând creșterea adusă de predictorul incremental cu creșterea adusă de predictorul contextual, se obține o valoare mai mare decât creșterea adusă de predictorul hibrid, indiferent de tipul adresei utilizate. Acest lucru se poate datora și faptului că predictorul hibrid implementat acordă întotdeauna prioritate predictorului contextual, cel incremental fiind folosit doar atunci când acesta nu poate genera o predicție. O posibilă soluție pentru eliminarea acestei *rigidități* în selecția

predictorului component ar fi să se implementeze un mecanism de metapredicție bazat pe confidență. În principiu, va avea prioritate predictorul al cărei confidență este mai mare, la un moment dat. În subcapitolul 7.3.2 este implementată o astfel de arhitectură referitoare la predicția valorilor centrată pe regiștrii procesorului. Toate aceste experimente au ca scop obținerea unei acurateți de predicție a valorii resurselor cât mai ridicată, și implicit o performanță globală de procesare mai mare.

7.3. PREDICȚIA VALORILOR REGIȘTRILOR CPU.

7.3.1. REZULTATE BAZATE PE PREDICTOARELE DE VALORI: INCREMENTAL, CONTEXTUAL ȘI HIBRID CU PRIORITIZARE STATICĂ.

O evaluare originală, prezentată inițial în [Flor02] pune în evidență conceptul de localitate a valorilor asociate regiștrilor generali aferenți procesorului (MIPS). Figurile 7.35 și respectiv 7.36 evidențiază că localitatea valorii pe anumiți regiștri speciali ai arhitecturii MIPS este remarcabilă (cca. 90%), conducând în mod evident la ideea predicției valorilor cel puțin pentru acești regiștri. La baza acestor afirmații stau caracteristicile setului de regiștri generali ai procesorului MIPS (vezi subcapitolul 6.2). Evaluările realizate se bazează pe rezultatele colectate în urma simulărilor a celor două versiuni de benchmark-uri SPEC: 5 programe de numere întregi (*li, go, perl, ijpeg, compress*) și 3 flotante (*swim, hydro, wave5*) din suita SPEC'95 și respectiv 7 programe de numere întregi (*gzip, b2zip, parser, crafty, gcc, twolf* and *mcf*) din suita SPEC2000. Au fost simulate benchmark-uri din cele două suite pentru a compara comportamentul acestora și pentru a stabili influența programelor de test asupra caracteristicilor microarhitecturale ale predictoarelor de valori.

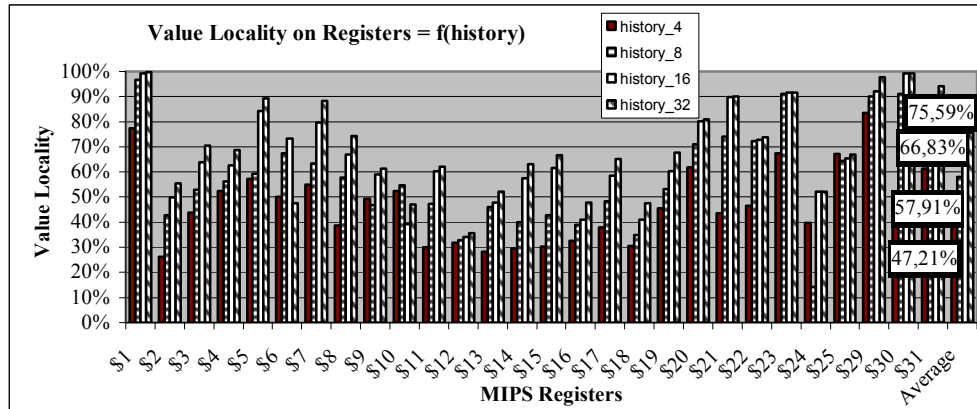


Figura 7.35. Localitatea valorilor pe regiștri de uz general (întregi) ai procesorului MIPS (rezultatele simulării a 500.000.000 instrucțiuni dinamice aferente benchmark-urilor **SPEC'95**)

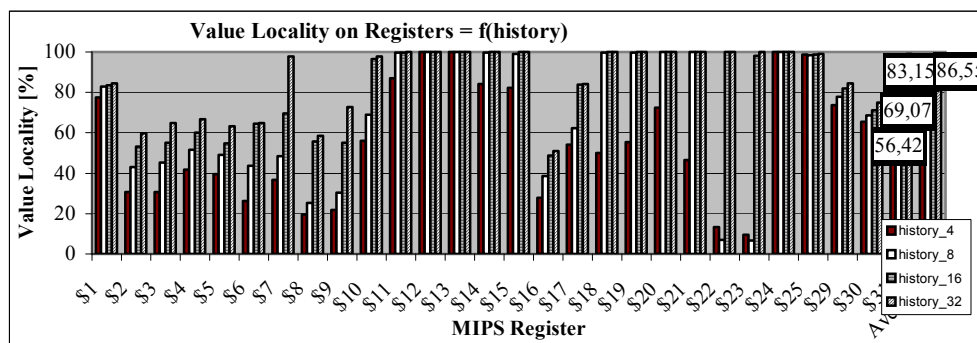


Figura 7.36. Localitatea valorilor pe regiștri de uz general (întregi) ai procesorului MIPS (rezultatele simulării a 500.000.000 instrucțiuni dinamice aferente benchmark-urilor **SPEC2000**)

Pornind de la rezultatele extrem de promițătoare privind localitatea valorii pe regiștrii procesorului MIPS, problema predicției valorilor la nivelul instrucțiunilor a fost extinsă și la alte tipuri de resurse [Vin05]. În investigația făcută am examinat o selecție a regiștrilor favorabili și predictoare diferite de valori pentru a capta anumite tipuri de predictibilitate existente în programele de calcul. Pentru selectarea regiștrilor care exprimă cele mai bune rezultate din punct de vedere al acurateții de predicție (peste 60%, respectiv peste 80%) s-a folosit un predictor dovedit optim (în cazul instrucțiunilor Load) și anume cel *hibrid* – alcătuit dintr-unul incremental și unul contextual (vezi subcapitolul 2.2.2.4), păstrând un *context* de 256

valori și un *pattern* de 4, atunci când în predicție se alege varianta contextuală.

Ca și metrică în acest caz, prin acuratețea de predicție se înțelege raportul dintre numărul de cazuri în care valoarea prezisă a registrului destinație a unei instrucțiuni, a fost identică cu cea rezultată în urma execuției, și totalul de situații în care respectivul registru a fost utilizat ca și destinație (vezi ecuația 6.2 din subcapitolul 6.2.1). Fiecare coloană din figurile 7.37 și 7.38 constituie media aritmetică pentru fiecare registru în parte a acurateții de predicție pe cele 8 benchmark-uri SPEC'95 amintite anterior și respectiv pe cele 7 benchmark-uri SPEC2000. Întrucât regiștrii \$0, \$26, \$27 au funcții dedicate, și analizând explicațiile din subcapitolul 6.2 referitoare la regiștrii procesorului MIPS este de înțeles faptul că acuratețea de predicție pe acești regiștri este 0.

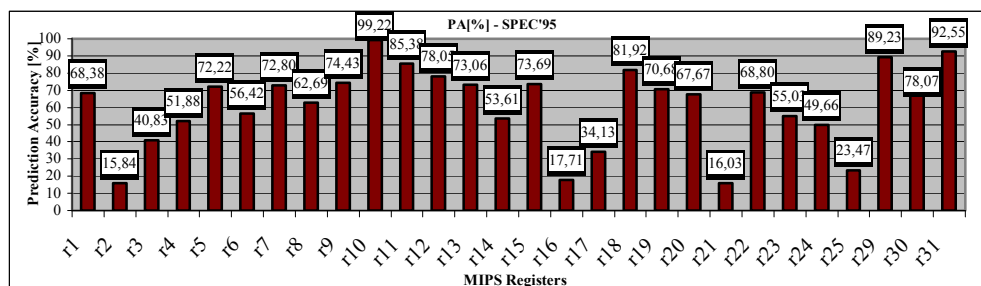


Figura 7.37. Acuratețea predicției valorilor pe regiștrii procesorului MIPS folosind un predictor hibrid (rezultatele simulării a 500.000.000 instrucțiuni dinamice aferente benchmark-urilor SPEC'95)

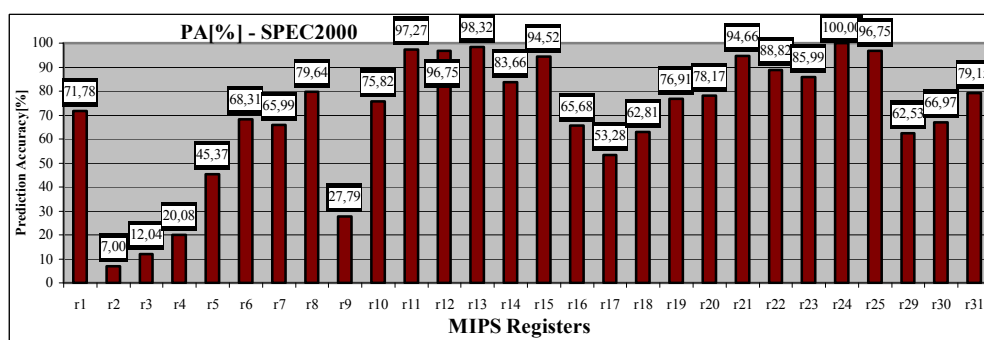


Figura 7.38. Acuratețea predicției valorilor pe regiștrii procesorului MIPS folosind un predictor hibrid (rezultatele simulării a 500.000.000 instrucțiuni dinamice aferente benchmark-urilor SPEC2000)

Următoarele investigații s-au concentrat asupra regiștrilor favorabili, având acuratețea predicției mai mare decât pragul impus de (60% respectiv 80%), conform statisticilor din figurile 7.37 și 7.38. După cum se poate observa regiștrii cu acuratețe mai mare de 60% sunt: 1, 5, 7÷13, 15, 18÷20, 22, 29÷31 pe suita SPEC'95, și respectiv, 1, 6÷8, 10÷16, 18÷25, 29÷31 pe testele SPEC2000. Gradul de utilizare ($nr_total_de_scrieri_în_reg_destinație / nr_total_de_instrucțiuni_simulate$) al celor mai semnificativi regiștri în număr de 17 pe benchmark-urile SPEC'95 este de 19.36%, rezultat ce arată că doar respectivul procentaj din instrucțiuni au ca destinație cei 17 regiștri. Rezultatul echivalent pe benchmark-urile SPEC2000 exprimă un grad de utilizare de 13.24% al celor 22 regiștrii de uz general selectați.

Rezultatele, extrem de interesante și în concordanță cu cele de la măsurarea localității, au determinat studierea comparativă a acurateții de predicție în funcție de schemele de predicție descrise în subcapitolul 6.2.1 (*LastValue*, *Incremental*, *Contextual*, *Hibrid*) pentru cei mai predictibili dintre regiștri (vezi figurile 7.39 și 7.40). Predictorul contextual și cel hibrid folosesc o istorie de 256 de valori și un pattern de căutare de 4 valori.

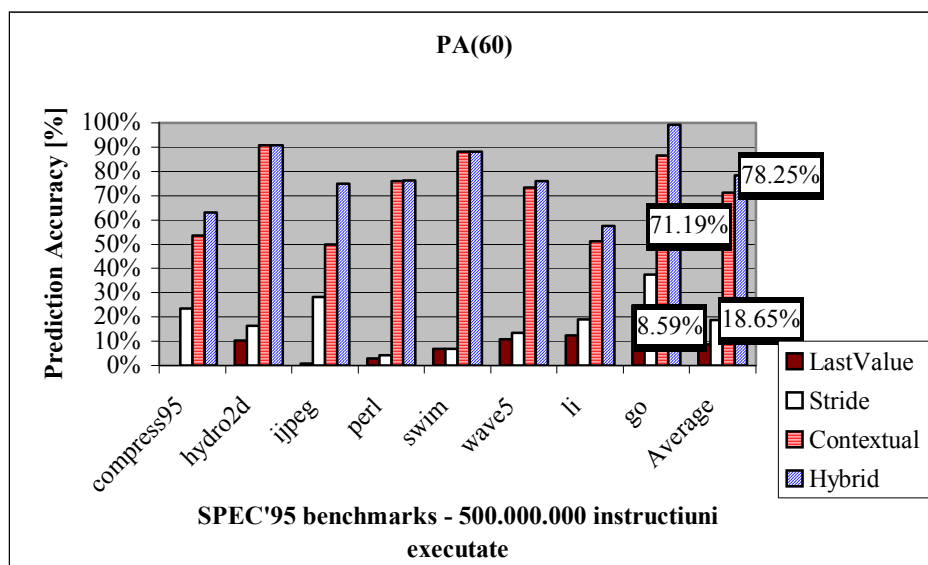


Figura 7.39. Compararea celor patru tehnici de predicție pentru cei mai predictibili (17) regiștri ai procesorului MIPS

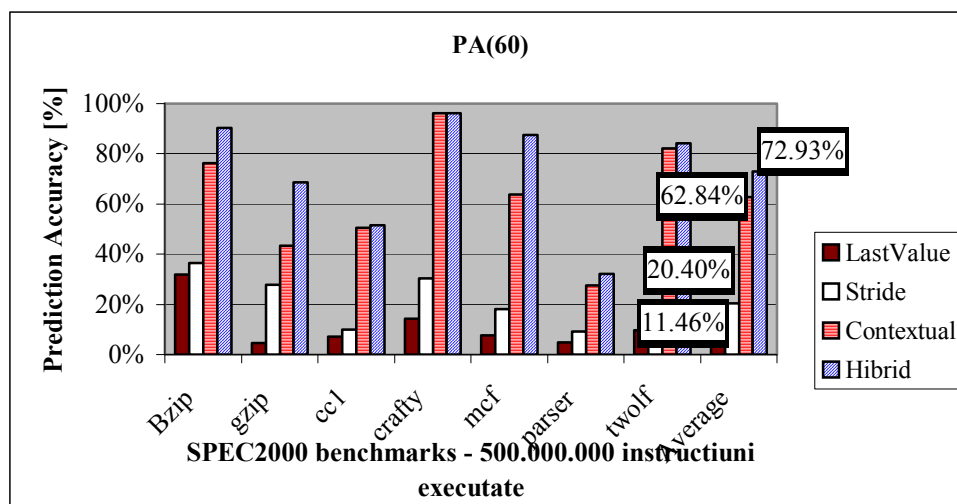


Figura 7.40. Compararea celor patru tehnici de predicție pentru cei mai predictibili (22) regiștri ai procesorului MIPS

Rezultatele bazate pe simulare întăresc convingerea că un predictor hibrid (contextual + incremental) generează cea mai mare acuratețe de predicție – în medie pe cei 17 regiștrii favorabili selectați în cazul testelor SPEC'95 de 78.25%, și echivalent pe cei 22 regiștri selectați în cazul benchmark-urilor SPEC2000 de 72.93%.

În continuare se încearcă o selecție mai „elitistă” considerând doar regiștrii care au dovedit o acuratețe de predicție mai mare de 80% (vezi figurile 7.41 și 7.42). Selecția se bazează din nou pe rezultatele grafice exprimate în figurile 7.37 și 7.38. Sunt 8 regiștri care respectă această condiție: 1, 10÷12, 18, 29÷31 pe suita SPEC'95 și respectiv 16 din benchmark-urile SPEC2000: 1, 8, 11÷15, 20÷25, 29÷31. Regiștrii 1, 29÷31 sunt incluși în această statistică chiar dacă nu satisfac condiția impusă deoarece ei îndeplinesc și unele funcții speciale [Flo03] iar gradul de localitate exprimat de aceștia este semnificativ (peste 80%) după cum poate fi observat în figura 7.36. Gradul de utilizare al acestor regiștri este de această dată de doar 10.58% pe benchmark-urile SPEC'95, și respectiv 9.01% pe SPEC2000.

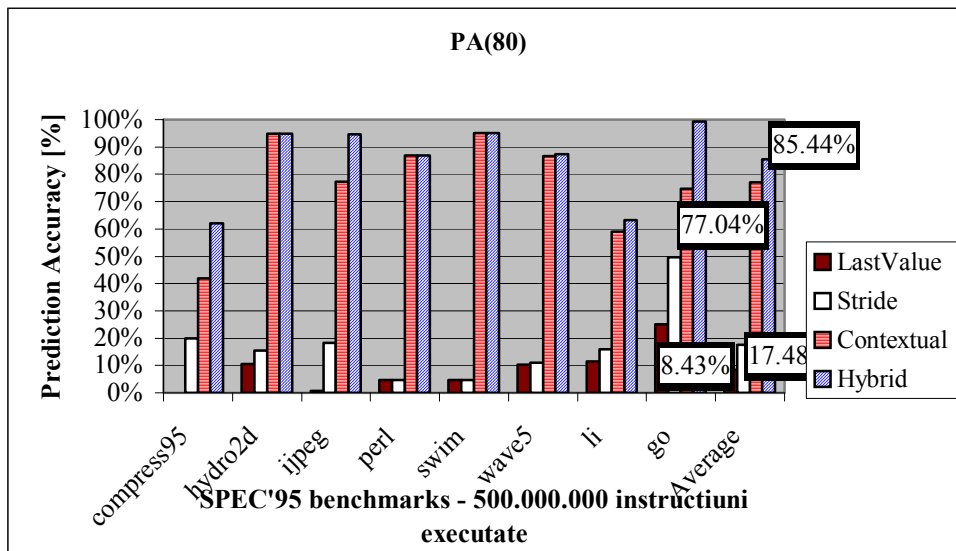


Figura 7.41. Acuratețea predicției folosind cei 8 regiștri favorabili selectați din testele SPEC '95

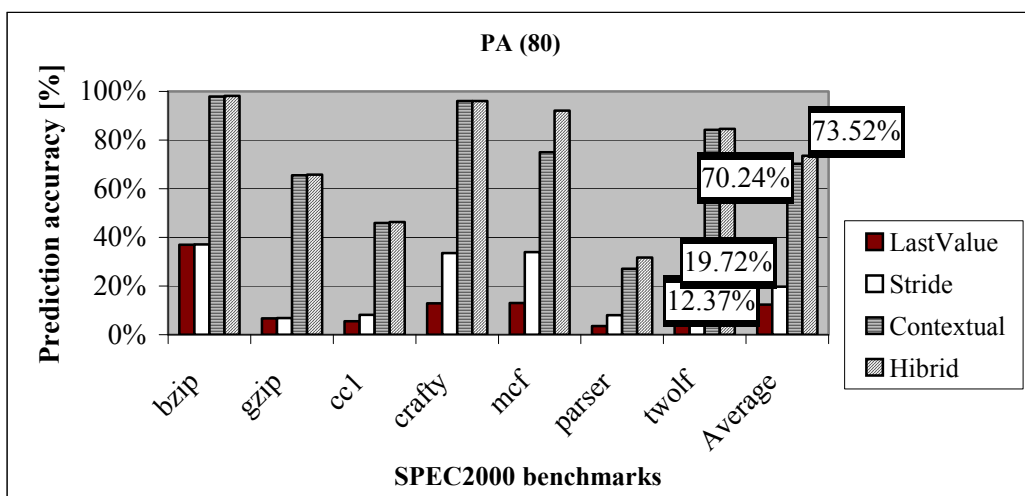


Figura 7.42. Acuratețea predicției folosind cei 16 regiștri favorabili selectați din testele SPEC 2000

În figurile 7.41 și 7.42 fiecare coloană reprezintă acuratețea predicției pe regiștrii favorabili dintr-un anumit benchmark, măsurată ca și raport

dintre numărul predicțiilor corecte pentru oricare din regiștrii favorabili și numărul situațiilor în care respectivii regiștri au fost folosiți ca destinație.

Rezultatele schemei incrementale sunt mai mult decât modeste și ușor mai bune decât cele oferite de predictorul *last value*. Se observă eficiența schemei hibride de predicție (**85.44%** - în medie aritmetică, ajungând în unele cazuri particulare la acurateți remarcabile de peste **96%**). Superioritatea față de celelalte trei scheme este explicabilă dacă ținem cont că predictorul hibrid implementat acordă întotdeauna prioritate predictorului contextual, cel incremental fiind folosit doar atunci când acesta nu poate genera o predicție. Evident, soluția este lipsită de flexibilitate. O posibilă soluție mai bună pentru rezolvarea acestei probleme ar fi să se implementeze două automate în fiecare locație a tabelului de predicție, unul pentru predictorul incremental și încă unul pentru cel contextual. Va avea prioritate predictorul al cărui numărător asociat (grad de încredere) are valoarea mai mare, la un moment dat. În cazul în care cele două valori sunt egale, predictorului contextual i se acordă prioritate. Rezultate privitoare la soluții adaptive de prioritizare în scheme hibride de predicția valorilor centrată pe contextul CPU se regăsesc în subcapitolul 7.3.2.

Următoarele două figuri (7.43 și 7.44) evidențiază *speed-up-ul* obținut de o microarhitectură speculativă care înglobează predictoarele de valori centrate pe contextul CPU descrise în subcapitolul 6.2.1.comparativ cu un procesor superscalar generic. Modelul de timing utilizat în simulare este cel prezentat în paragraful 6.1.2. Cantitativ acest câștig este de **17.30%** pe suta SPEC'95, și respectiv de **13.58%** pe SPEC2000.

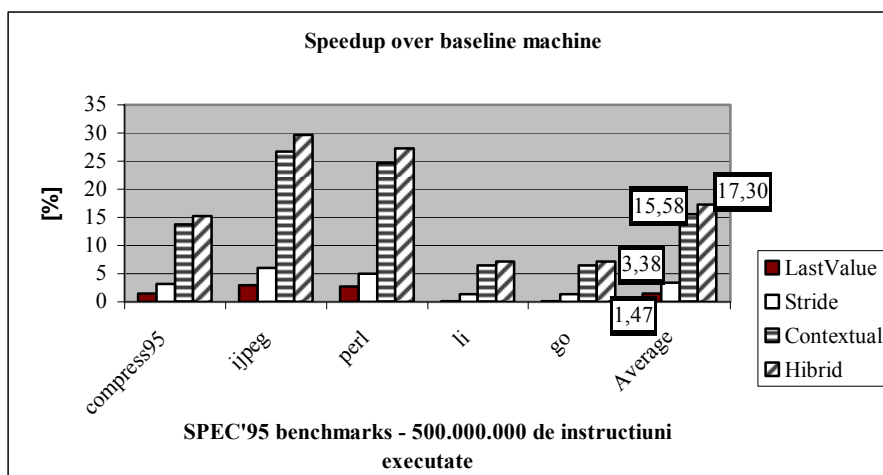


Figura 7.43. Speedup-ul obținut față de o arhitectură generică folosind predictoare de valori pe cei 8 regiștri favorabili (SPEC'95)

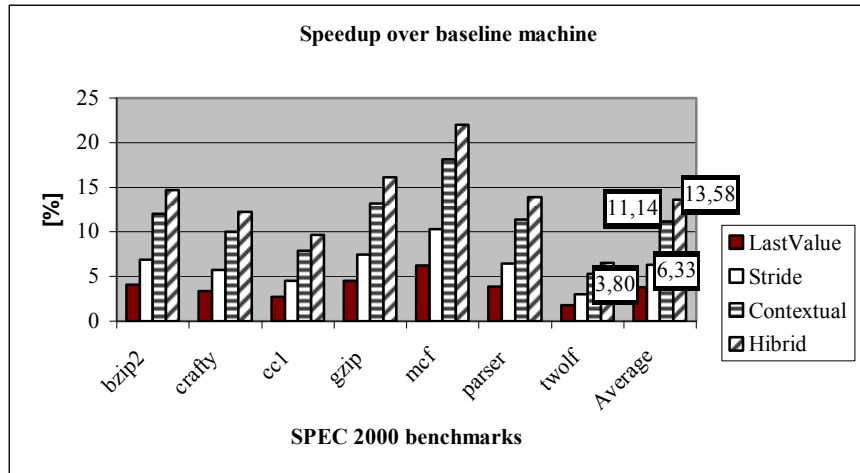


Figura 7.44. Speedup-ul obținut față de o arhitectură generică folosind predictoare de valori pe cei 16 regiștri favorabili (SPEC2000)

Interesant de remarcat corelând figura 7.42 cu figura 7.44, o diferență de doar 3% între acuratețea predictorului hibrid față de cel contextual conduce la o creștere a *speedup*-ului cu încă 2.44%.

7.3.2. EVALUAREA PREDICȚIEI NEURONALE APLICATĂ REGIȘTRILOR PROCESORULUI.

În această subsecțiune toate simulările au fost efectuate pe 5.000.000 de instrucțiuni dinamice aferente benchmark-urilor SPEC'95. În experimentele următoare pe lângă acuratețea predicției s-a mai considerat o metrică numită *confidență* (sau acuratețe de predicție locală) care reprezintă numărul de valori corect predicționate aferente unui anumit registru (R_i) când predictorul său atașat se află într-o stare predictibilă.

Pentru început a fost evaluată acuratețea predicției metapredictorului non-adaptiv empiric (vezi figura 6.8) în funcție de pragul impus (numărul minim de comportamente favorabile întâlnite într-o istorie dată). S-a considerat lungimea vectorului de istorie $k=3$.

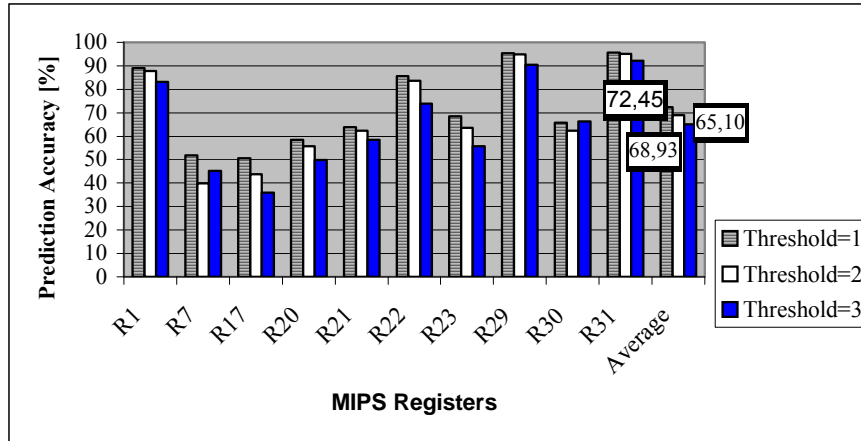


Figura 7.45. Metapredictor non-adaptiv empiric: **acuratețea predicției** pentru diferite praguri

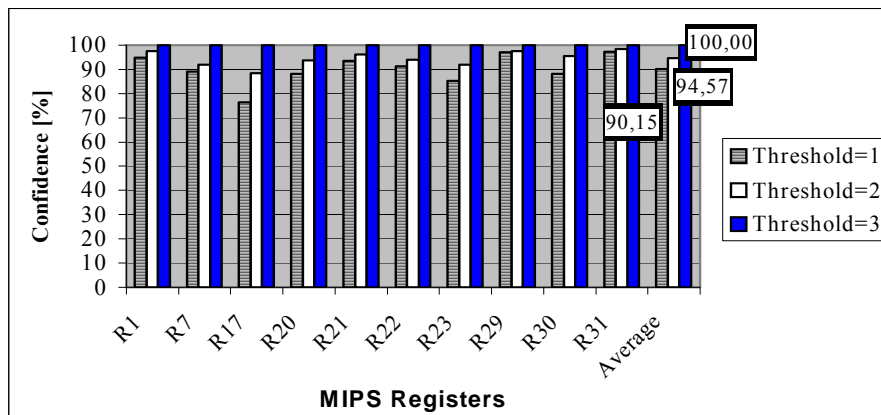


Figura 7.46. Metapredictor non-adaptiv empiric: **confidența** pentru diferite praguri

Statisticile următoare se referă la metapredictorul static, non-adaptiv bazat pe automate de confidență cu 4 stări.

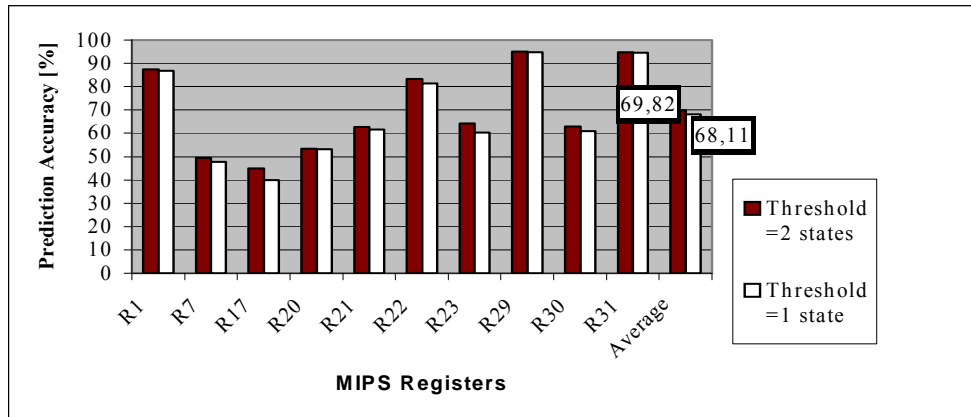


Figura 7.47. Acuratețea predicției măsurată pentru diferite praguri (predicția într-una sau 2 stări predictibile) folosind un metapredictor bazat pe automate

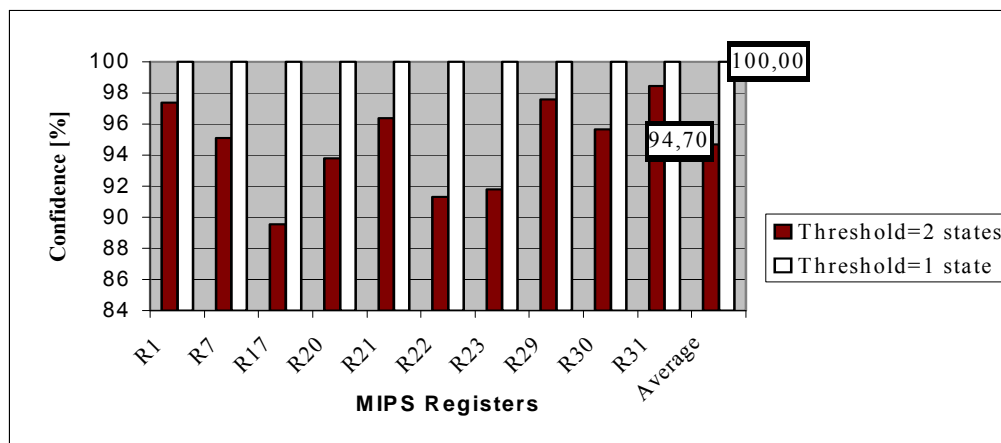


Figura 7.48. Confidența măsurată pentru diferite praguri (predicția într-una sau 2 stări predictibile) folosind un metapredictor bazat pe automate

Ultima structură de metapredicție este adaptivă, bazată pe o rețea neurală de tip feed-forward (*MultiLayerPerceptron*) cu algoritm de învățare *back-propagation*. S-a considerat o istorie a comportamentelor fiecărui predictor component pe $k=3$ biți, fapt ce conduce la un nivel de intrare în rețeaua neurală de $3 \cdot k=9$ biți.

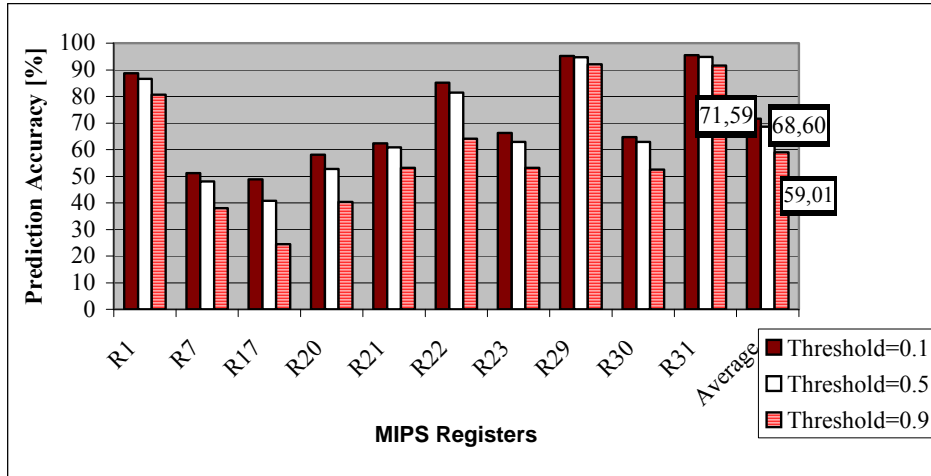


Figura 7.49. Acuratețea predicției generată de un metapredictor neural pentru praguri diferite

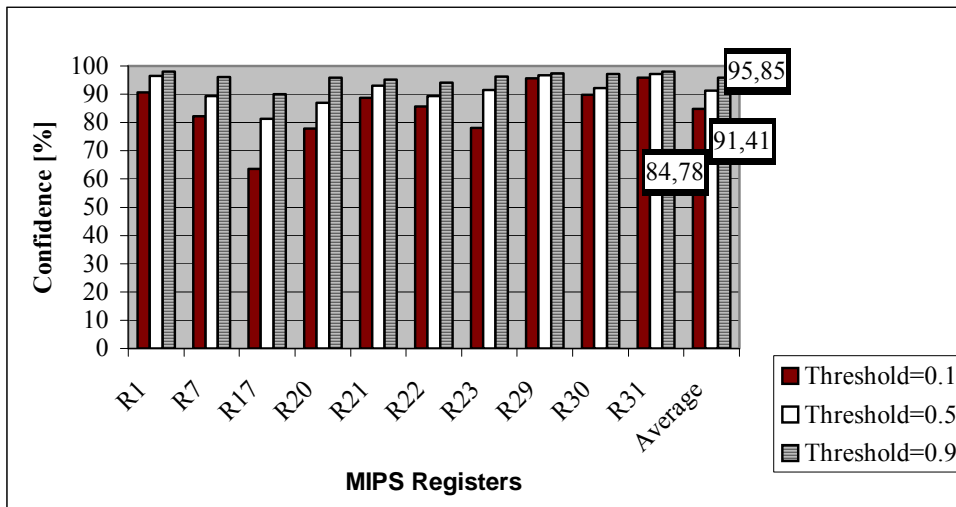


Figura 7.50. Confidența asigurată de metapredictorul neural pentru diferite praguri

Din statisticile anterior exprimate (figurile 7.45 ÷ 7.50) se observă că, pentru fiecare metapredictor, cu cât pragul crește (procesul de selecție devine mai elitist) acuratețea globală de predicție scade iar confidența crește spre 100%. Practic, probabilitatea ca predicția locală generată de o stare ridicată de confidență să fie corectă crește semnificativ prin reducerea cazurilor când structura efectuează o predicție. Judecând din punct de vedere strict al confidenței, se observă că metapredictorul non-adaptiv bazat

pe automate, care prezice doar în starea "*puternic predictibilă*" este optim. **Acuratețea globală de predicție** este de **68.11%** cu o **încredere absolută de 100%**.

Un ultim experiment realizat a urmărit determinarea istoriei optime care trebuie considerată în procesul de metapredicție. Pentru un proces de predicție mai puțin selectiv (threshold = 0.1) se observă că prin reținerea ultimelor trei comportamente aferente fiecărui predictor component se obțin rezultate optime ($A_p(k=1)=71.83\%$ vs. $A_p(k=3)=71.59\%$, respectiv $Confidența(k=1)=83.49\%$ vs. $Confidența(k=3)=84.78\%$).

Ținând cont de rezultatele experimentale anterioare, tabelul următor exprimă următoarea concluzie interesantă: **în ciuda unei acurateți globale de predicție destul de reduse (72.45% în medie aritmetică pe toți regiștrii favorabili) există anumiți regiștri (R_1 , R_{22} , R_{29} și R_{31}), cu acurateți de predicție de peste 90%. Câștigul în acuratețe obținut pe acești regiștri față de predictorul hibrid cu prioritizare fixă, dovedit optim în subcapitolul 7.3.1, este de 2.27%.**

Acuratețea predicției pe anumiți regiștri ai procesorului				
Numărul regiștrului	Metapredictor bazat pe automat (threshold=1 stare)	Metapredictor non-adaptiv empiric (threshold = 1)	Metapredictor neural (threshold=0)	Predictor hibrid cu prioritizare fixă
1	89.034	89.028	88.999	86.81
22	85.477	85.647	84.146	81.43
29	95.289	95.287	95.284	94.74
31	95.642	95.643	95.599	94.48
Average	91.3605	91.4012	91.007	89.365
<i>Câștigul de acuratețe obținut vis a vis de predictorul hibrid cu prioritizare fixă [%]</i>	2.23	2.27	1.83	

Tabelul 7.13. Acuratețea predicției pe cei mai predictibili 4 regiștri ai procesorului

8. CONCLUZII ȘI CONTRIBUȚII ORIGINALE. DEZVOLTĂRI ULTERIOARE

În acest capitol sunt trecute în revistă contribuțiile științifice ale acestei lucrări, este evidențiat câștigul cantitativ al fiecărei tehnici introduse și sunt realizate comparații între aceste tehnici.

Performanța ridicată a microprocesoarelor moderne se datorează execuției în paralel a cât mai multor instrucțiuni respectiv microfîre de execuție independente. Întrucât există o varietate de factori care limitează paralelismul la nivelul instrucțiunilor (ILP), în ultima perioadă au fost propuse și testate o serie de tehnici ne-speculative (reutilizarea dinamică a instrucțiunilor) respectiv speculative (predicția dinamică a ramificațiilor de program și predicția valorilor centrată pe diverse tipuri de resurse) pentru depășirea acestor limitări. În cadrul acestei lucrări s-a propus o abordare integratoare *hardware-software* cu scopul creșterii performanței procesoarelor cu paralelism la nivelul instrucțiunilor, urmată în mod natural de o serie de tehnici predictive și speculative referitoare la predicția salturilor / apelurilor indirecte și predicția valorilor resurselor (instrucțiuni Load, ALU, adrese de memorie aferente instrucțiunilor Load și regiștrii procesorului).

O primă contribuție originală încearcă să evidențieze „falsa discrepanță” – modul diferit de abordare al celor două "emisfere", hardware și software, în care își desfășoară activitatea cercetătorii din știința calculatoarelor, pentru creșterea performanței procesoarelor moderne. Ideea că arhitectura procesoarelor interacționează "accidental" cu domeniul software este complet greșită, între hardware și software existând în realitate o simbioză și interdependență puternică, încă neexplorate corespunzător. Procesoarele se proiectează odată cu compilatoarele care le folosesc iar relația dintre ele este foarte strânsă: **benchmark-urile sunt compilate pentru arhitectura respectivă iar compilatorul trebuie să genereze cod care să exploateze caracteristicile arhitecturale, altfel codul generat va fi ineficient.** Datorită tendințelor manifestate în ultimii ani, de trecere la medii vizuale și obiectuale de programare bazate pe concepte avansate (moștenire, polimorfism), **aplicațiile obiectuale au devenit o mare provocare atât pentru comunitatea compilatoriștilor cât și pentru arhitecții de microprocesoare,** mai ales că există diferențe semnificative între caracteristicile programelor procedurale și cele ale

programelor obiectuale, cu implicații și asupra performanțelor acestor programe (viteză de execuție, consum de memorie).

Prin intermediul unor programe de test, relativ simple, procedurale și obiectuale, am arătat că **cele două "emisfere" software și hardware sunt doar în aparență „disjuncte”**. Cele 2 programe obiectuale C++ și 3 procedurale C propuse evidențiază **corpuri și construcții de program procedurale și obiectuale** care generează la nivelul codului obiect **salturi / apeluri indirecte**. Pe baza rezultatelor obținute se desprind câteva concluzii clare:

- Instrucțiunile de salt indirect apar mult mai frecvent în programele obiectuale decât în cele procedurale.
- Prezența salturilor indirecte în **programele procedurale** se datorează în *principal* următoarelor două aspecte:
 - **Apelurilor indirecte de funcții prin pointeri.**
 - **Construcțiilor de tip switch/case** având anumite caracteristici.
- Un alt motiv care favorizează prezența salturilor indirecte în programe îl reprezintă **prezența funcțiilor de bibliotecă** (vezi cazul *qsort* precum și alte biblioteci legate dinamic din cadrul aplicațiilor desktop – *DLL*).
- **Legarea dinamică (*late binding*) realizată prin polimorfism** (care se bazează la rândul său pe conceptul de **moștenire**) generează apeluri indirecte de funcții în cazul **programelor obiectuale** (C++, Java, Smalltalk).

În mod natural, cercetările proprii au continuat cu realizarea de statistici privind **frecvența salturilor / apelurilor indirecte** urmate de **implementarea unor scheme de predicție** (o parte din ele originale) dedicate acestui tip de instrucțiuni. Pentru raportarea în concordanță cu majoritatea cercetătorilor în arhitecturi de calcul, investigațiile cantitative privind gradele de localitate a valorii și determinarea acurateții predicției s-au bazat pe setul standardizat de instrumente SimpleScalar 3.0b, unanim acceptat de cercetătorii în arhitectura calculatoarelor. Evaluările arhitecturilor propuse le-am efectuat folosind o serie de simulatoare execution-driven dezvoltate din setul de instrumente SimpleScalar. Simularea a fost realizată pe cele două versiuni de benchmark-uri SPEC ('95 și 2000), precum și pe programele de test proprii.

O primă concluzie reliefează **aportul crescut al salturilor / apelurilor indirecte asupra costului global generat de predicțiile greșite** și respectiv **necesitatea dezvoltării de noi mecanisme de predicție (structuri hibride, predictoare cascade pe mai multe niveluri sau adaptate din predicția valorilor) pentru predicția cu acuratețe ridicată a acestora**. Practic „*lupta este extrem de aprigă, la nivel de procent*” în

încercarea de a predicționa cu acuratețe salturile indirecte, în sensul că fiecare câștig potențial (introdus de o nouă tehnică sau îmbunătățiri aduse structurii de predicție), oricât de mic (1, 2%) trebuie exploatat. Performanța globală a arhitecturilor este extrem de sensibilă la predicția salturilor indirecte, întrucât, în ciuda **frecvenței relativ reduse a acestora în programele procedurale** testate, **o singură instrucțiune statică de salt indirect poate genera în momentul execuției peste 6200 de instanțe dinamice**. Dificultatea predicției se datorează **dispersiei extrem de ridicate a target-urilor unora dintre aceste instrucțiuni**.

O concluzie desprinsă din analiza rezultatelor obținute subliniază **asemănarea existentă între problema predicției valorilor și problema predicției adreselor destinație aferente instrucțiunilor de salt indirect**. Pentru început a fost determinat **gradul de localitate exprimat de instrucțiunile de salt indirect** în programele de uz general, ca **o limită ultimativă a acurateții de predicție obținabile**. Gradul ridicat de localitate exprimat în medie de instrucțiunile de revenire din proceduri / funcții sunt în corelație cu gradul ridicat de localitate (**90%**) observat pe registrul **\$31** (return address register) al procesorului MIPS. De asemenea, **localitatea valorii** obținută pe cele 5 programe proprii de test, **extrem de ridicată (92% - medie aritmetică) se datorează numărului redus de obiecte** (declarate individual și nu în cadrul unei structuri de date cu legături - tablouri sau liste) **și metodelor aferente apelate recursiv**.

Exploatarea gradului ridicat de localitate aferent instrucțiunilor de salt indirect s-a realizat prin intermediul predictorului contextual de tip PPM complet. Cu ajutorul acestuia s-a încercat determinarea pattern-ului optim de căutare, stabilirea corelației existente între salturi în funcție de context. Acuratețea de predicție maximă (**91.58% – superior rezultatelor raportate de Chang [Cha97] pe schema TargetCache**) se obține pentru un pattern de căutare de **4**, îndreptățind și afirmațiile altor cercetători. Oarecum firesc, **creșterea istoriei valorilor reținute implică o acuratețe de predicție mai mare prin creșterea pattern-ului de căutare (se distinge o corelație între salt-uri mai îndepărtate dacă contextul ar permite acest lucru)**. Rezultatele întăresc afirmația cercetătorilor [Tho03] care susțin păstrarea și utilizarea unei istorii cât mai bogate în procesul de predicție aferent instrucțiunilor de salt întrucât unele salturi corelate pot apărea la o distanță considerabilă în șirul de instrucțiuni dinamice. **Procentajul mediu al instrucțiunilor de salt indirect clasificate predictibile și predicționate corect de către automatul de clasificare este destul de ridicat (90÷94%)**. Se observă însă procentaje reduse (**Unpred ≤ 27%**) pentru salturile clasificate nepredictibile și într-adevăr predicționabile greșit. Practic,

slăbiciunea automatului propus constă în faptul că este prea conservator, el clasificând ca nepredictibile multe instrucțiuni care sunt în realitate predictibile.

Dificultatea implementării în hardware a predictorilor PPM complete impun însă, ca etapă viitoare de cercetare fie găsirea unor structuri mai simple (vezi structura Target Cache modificată) dar cu rezultate echivalente din punct de vedere al acurateții predicției, fie analiza comportamentului unui predictor contextual de tip PPM simplificat, care să conțină doar două predictoare *Markov*: unul de ordinul (N) și altul de ordinul 0. Din simulările realizate pe programele proprii de test a rezultat **ineficiența predictorului PPM complet în cazul utilizării de masive eterogene** (de obiecte sau alte structuri de date), situație în care un predictor incremental se comportă mult mai bine. În schimb **recursivitatea poate fi exploatată cu succes cu ajutorul predictorului PPM complet**.

Referitor la predictorul de tip **TargetCache** original propus de Chang [Cha97], acuratețea maximă de predicție obținută (**79.82%**), a fost inferioară rezultatelor generate de predictorul PPM complet, fapt ce a condus la realizarea unor investigații proprii pornind de la structura Target Cache. **O nouă contribuție originală** a acestei lucrări **a avut în vedere îmbunătățirea acurateții predicției aferente instrucțiunilor de salt indirect prin modificări aduse structurii Target Cache**. Pentru început a fost studiată **influența istoriei globale a salturilor condiționate asupra predicției**. Pentru benchmark-urile cu o dispersie ridicată a target-urilor, **utilizarea istoriei globale a salturilor condiționate în indexarea Target Cache-ului joacă un rol important în creșterea acurateții predicției salturilor indirecte** (în medie cu până la **16.93%** iar în cazuri particulare chiar și cu **45%**). În medie aritmetică pe 7 benchmark-uri SPEC'95, acuratețea optimă de predicție se obține prin reținerea comportamentului global al ultimelor 4 salturi condiționate. O critică generală ce poate fi adusă **schemelor de predicție adaptive corelate pe două niveluri** constă în faptul că **folosesc insuficientă informație de corelație pentru identificarea cu precizie a contextului de apariție a saltului de prezis**. Astfel, am dezvoltat un predictor *bazat pe calea până la saltul indirect* care folosește drept informație de predicție pe lângă PC-ul saltului indirect și istoria globală a salturilor condiționate și PC-urile corespondente acestor salturi, în vederea reducerii coliziunilor unor contexte diferite în tabela Target Cache. Pentru benchmark-urile caracterizate de un procentaj ridicat de salturi indirecte **extinderea informației de corelație, la costuri identice de implementare determină creșterea acurateții predicției** acestora (cu **8.64%** pentru o istorie de 4 PC-uri reținute respectiv cu **15.16%** când istoria este 8). Acuratețea predicției instrucțiunilor de salt indirect (**88.21%**) este

totuși inferioară celei obținute cu un predictor PPM complet (**89.33%**) cu toate că există și rezultate extraordinare, care evidențiază performanțe egale obținute de un Target Cache și un predictor PPM complet.

O altă contribuție originală se referă la **îmbunătățirea acurateții de predicție** generată de o structură de tip TargetCache **printr-o ignorare selectivă a efectuării unor predicții**. Astfel, Target Cache-ul a fost îmbogățit cu un *grad de confidență* aferent fiecărei locații din structură, adăugându-se și un mecanism de inserare / evacuare în / din set bazat pe **LRU**, **confidență** respectiv pe superpoziția celor două (**MPP** – minim de performanță potențial). **Probabilitatea ca predicția generată de o stare ridicată de încredere să fie corectă crește substanțial** prin restrângerea cazurilor în care se face predicție (cu **3.57% până la 11.45% în funcție de prag** – statistici realizate pe benchmark-urile bogate în instrucțiuni dinamice de salt indirect) dar, evident, procentajul cazurilor în care se face predicție scade dramatic. Tehnica de extindere a informației de corelație își dovedește și în acest caz superioritatea (creșterea cu **5.62%** a acurateții de predicție). Prin adăugarea unui automat de confidență, performanța globală a predictorului se îmbunătățește **când acesta este mai puțin selectiv**. Acuratețea de predicție a Target Cache-ului cu confidență (**88.97%**) este încă inferioară celei obținute cu cel mai performant predictor PPM complet (**89.33%**) dar se apropie semnificativ (diferența **sub 0.41%**), permițând **înlocuirea unei scheme extrem de complexe (PPM complet) cu una fezabilă hardware (Target Cache)**.

Am realizat o statistică privind aritatea salturilor indirecte (numărul target-urilor generate dinamic de un salt / apel indirect). Astfel, salturile indirecte cu un singur target sunt într-o proporție de 33% în programele de calcul în timp ce cele cu trei sau mai multe target-uri constituie aproape jumătate din total (**49.61%**). **O altă contribuție originală** vizează implementarea unui **predictor hibrid (LastValue+Contextual) cu selecție bazată pe aritate, care îmbunătățește acuratețea predicției salturilor indirecte** cu procentaje cuprinse între 2.44% și 5.42% în funcție de structura de predicție cu care se face comparația **ajungându-se în medie aritmetică la acurateți maxime de predicție de 93.77%**, apropiate de valorile maxime raportate în literatura de specialitate (94.8% cu predictor cascadat pe 3 niveluri). Procentajul substanțial de salturi polimorfe dinamice și dispersia ridicată a target-urilor anumitor salturi stă la baza limitării acurateții predicției. Rezultatele bune obținute obligă **la introducerea și exploatarea predictoarelor hibride și cascadeate și în alte domenii de cercetare, în vederea creșterii paralelismului la nivelul instrucțiunilor: predicția salturilor condiționate și predicția valorilor instrucțiunilor**. Una din intențiile de viitor se referă la **înlocuirea metapredicției bazată pe**

informații de aritate cu o rețea neuronală care, bazat pe istorie, să selecteze între diferitele predictoare componente, structura ce va prezice saltul indirect curent. De asemenea, o idee interesantă ce va fi abordată o reprezintă **studiul fezabilității unui predictor de salturi indirecte, corelat pe bază de arbori de decizie** și senzitiv la informația de corelație cu adevărat utilă.

În continuare au fost studiate probleme legate de vecinătatea valorilor și predicția valorilor (VP) instrucțiunilor cu consecința execuției speculative a instrucțiunilor și cu influențe benefice asupra timpului de procesare. Concluzia de bază este că există grade optimiste de acuratețe a valorilor (chiar dacă se reține doar ultima valoare), atât din punct de vedere al rezultatelor centrate pe producător (*Instruction Centred* - 54%) cât și din punctul de vedere al celor centrate pe adresele memoriei de date (*Memory Centred* - 70%). Aceste localități cresc, în mod evident, odată cu creșterea numărului de valori istorice memorate (76% respectiv 86%, pentru o istorie de 32 de valori). Rezultatele sunt utile pentru că ele arată dacă și în ce condiții predicția valorilor este fezabilă. Astfel, istoria memorată poate fi un foarte util indicator pentru proiectarea predictorului atașat. Compromisurile actuale care se fac în VP sunt – *o istorie redusă* – reprezentând o acuratețe de predicție joasă dar cost scăzut sau – *o istorie bogată de predicție* – acuratețe ridicată de predicție dar costuri hardware ridicate. O altă concluzie reflectă superioritatea tabelelor de predicție asociative. Dimensiunea optimă a acestora este de 512 locații, diferența față de o tabelă infinită (ca și număr de locații) fiind nesemnificativă – sub 6% (indexarea se face cu PC-ul instrucțiunii) respectiv sub 14% (indexarea se face cu adresa datei). Simulările efectuate au demonstrat un grad ridicat de localitate și pe instrucțiunile de tip aritmetico-logic (78% - medie aritmetică). Pentru viitor cercetarea trebuie continuată cu evaluarea arhitecturilor hibride de predicție, implementarea predictoarelor de valori neurale de tip *perceptron* sau *MultiLayerPerceptron*, utilizarea a mai mult de două valori distincte în cadrul unui predictor perceptron.

O contribuție originală pune în evidență conceptul de vecinătate a valorilor asociate regiștrilor generali aferenți procesorului. Deși la o primă analiză a ideii am putea fi descurajați de gradul ridicat de interferențe care pot apare, se poate observa că **localitatea valorii pe anumiți regiștri speciali ai arhitecturii MIPS este remarcabilă (cca. 90%)**, ceea ce conduce în mod natural la ideea predicției valorilor, cel puțin pentru acești regiștri favorabili. Ideea asocierii câte unui predictor de valori pentru anumiți regiștri – **predictoare centrate pe regiștri și nu pe instrucțiuni**, ar putea **implica tehnici arhitecturale novatoare – structuri de predicție mult mai simple** (32-128 de locații), și, în consecință, **performanțe**

îmbunătățite, complexitate și costuri mai reduse ale microarhitecturilor speculative bazate pe acest concept.

În investigația efectuată am examinat o selecție a regiștrilor favorabili și predictoare diferite de valori pentru a capta anumite tipuri de predictibilitate existente în programele de calcul. Bazat pe simulări laborioase, s-a arătat că, beneficiind de o istorie suficient de bogată (ultimele 256 de valori), **un predictor hibrid (contextual + incremental) elimină problema interferențelor și poate atinge o acuratețe medie de predicție de 85.44%, existând și cazuri particulare în care acuratețea ajunge la 96%**. Funcționarea acestor tipuri de predictoare se bazează însă pe o **prioritizare statică, fixă, în alegerea tipului de predictor** ce urmează a fi folosit în procesul de predicție, ceea ce conduce însă la o **soluție neoptimală**. Pentru rezolvarea acestei probleme **am implementat câteva structuri de metapredicție, care selectează dinamic, bazat pe diverse grade de încredere, structura care să fie utilizată la un moment dat pentru predicție**. Au fost propuse două tipuri de metapredictoare: ne-adaptive, prin atașarea unui automat de confidență sau registru binar de deplasare fiecărui predictor component, și respectiv adaptive, care utilizează o rețea neurală de tip feedforward MultiLayerPerceptron cu algoritm de învățare *backpropagation*. Se observă că, pentru fiecare metapredictor, cu cât procesul de predicție devine mai selectiv probabilitatea ca predicția locală generată de o stare ridicată de confidență să fie corectă crește semnificativ prin reducerea cazurilor când structura efectuează o predicție. Pe baza simulărilor se observă că **acuratețea medie de predicție pe regiștrii R₁, R₂₂, R₂₉ și R₃₁, este de 91.40%. Câștigul în acuratețe obținut pe acești regiștri față de predictorul hibrid cu prioritizare fixă, este de 2.27%**. De asemenea, s-a arătat că **predicția centrată pe regiștri conduce la o creștere a performanței globale cu cca. 15%**.

O continuare a cercetării ar putea să aibă în vedere implementarea în hardware a unui metapredictor dinamic neural eficient (bazat pe perceptroane simple). De asemenea ar putea fi testată și fezabilitatea unui predictor de valori centrat pe contextul CPU bazat pe arbori dinamici de decizie și senzitiv la informația de context cu adevărat relevantă pentru predicție.



BIBLIOGRAFIE

- [Aco02] **Acostăchioaie D.** – *Programare C și C++ pentru Linux*, Editura Polirom, Iași, 2002.
- [Aig96] **Aigner G., Hoelzle U.** - *Eliminating Virtual Function Calls in C++Programms*. In Proc. 10th European Conference on Object Oriented Programming, June 1996.
- [Ball93] **Ball T., Larus J.R.** – *Branch prediction for free* in Proceedings of the ACM SIGPLAN’93 Conference on Programming LanguageDesign and Implementation, June 1993, pp. 300–313.
- [Bow98] **Bowers K. R., Kaeli D.** – *Characterizing the SPEC JVM98 benchmarks on the Java virtual machine*. Technical report, Northeastern University, Dept. of ECE, Computer Architecture Group, 1998.
- [Bre02] **Brekelbaum E., Rupley J., Wilkerson C., Black B.** – *Hierarchical scheduling windows*. In Proceedings of the 34th International Symposium on Microarchitecture, Istanbul, Turkey, December 2002.
- [Brea02] **Breazu M.** – *Programare orientată pe obiecte. Concepte*, Editura Universității “Lucian Blaga” Sibiu, 2002.
- [Bur97] **Burger D., Austin T.** – *The SimpleScalar Tool Set, Version 2.0*, University of Wisconsin Madison, USA, Computer Science Department, Technical Report #1342, June, 1997.
- [Cal94] **Calder B., Grunwald D., Zorn B.** – *Quantifying Behavioral Differences Between C and C++ Programs*, Journal of Programming Languages, pages 323-351, Vol. 2, Num. 4, 1994.
- [Cal94b] **Calder B., Grunwald D.** – *Reducing Indirect Function Call Overhead in C++ Programs*, In 1994 ACM Symposium on Principles of Programming Languages, pages 397-408, January 1994.
- [Cal95] **Calder B., Grunwald D., Linsay D.** – *Corpus-based static branch prediction*, SIGPLAN Notices, June 1995, pp.79-92.

- [Cal97] Calder B., Grunwald D., Jones M., Lindsay D., Martin J., Mozer M., Zorn B. – *Evidence-based static branch prediction using machine learning*, ACM Transactions on Programming Languages and Systems, vol. 19, no. 1, pp. 188–222, Jan. 1997.
- [Cal99] Calder B., Reinman G., Tullsen D. – *Selective Value Prediction*, In Proceedings of the 26th International Symposium on Computer Architecture, pg. 64-74, May 1999.
- [Cal99b] Calder B., Feller P., Eustace A. – *Value Profiling and Optimization*, Journal of Instruction-Level Parallelism 1, SUA, 1999.
- [Cha97] Chang P.Y., Hao E., Patt Y.N. - *Target Prediction for Indirect Jumps*, ISCA '97 (<http://www.eecs.umich.edu/HPS>).
- [Chen96] Chen I.K., Coffey J.T., Mudge T. – *Analysis of Branch Prediction via Data Compression*, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Cambridge, MA, USA, October 1996, pp. 128-137.
- [Cle84] Cleary, J. G., Witten, I. H. – *Data compression using adaptive coding and partial string matching*. IEEE Transactions on Communications, Vol. 32, No. 4, April 1984.
- [Cme93] Cmelik R. F., Keppel D. – *Shade: A Fast Instruction-Set Simulator For Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993. Also published as Technical Report CSE-TR 93-06-06, University of Washington, 1993.
- [Col93] Collins, R. – *Developing A Simulator for the Hatfield Superscalar Processor*, Division of Computer Science, Technical Report No. 172, University of Hertfordshire, December 1993.
- [Con95] Conte T., Menezes K., Mills P., Patel B. – *Optimization of Instruction Fetch Mechanism for High Issue Rates*, Proceedings of the 22nd International Symposium on Computer Architecture, June 1995.
- [Con99] Connors D., Hwu W.M. – *Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results*, IEEE 1072-4451, 1999.
- [Con00] Connors D.A., Hunter H.C., Cheng B., Hwu W.W. – *Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse*, In Proceedings of the 9th International Conference on Architectural Support

for Programming Languages and Operating Systems (ASPLOS IX), Cambridge, MA, USA, November 2000.

[Cor90] Cormen, T. H., Leiserson, C. E., Rivest, R. L. – *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[Dav92] Davidson J.W., Holler A.M. – *Subprogram Inlining: A study of its effects on program execution time*, IEEE Transaction on Software Engineering, 18(2): 89-102, February, 1992.

[Des02] Deswet V., Goeman B., Bosschere K. – *Independent hashing as confidence mechanism for value predictors in microprocessors*, International Conf. EuroPar, Augsburg, Germany, 2002.

[Des04] Desmet V., De Bosschere K. – *Decision Trees for Improving Heuristic-Based Static Branch Prediction*, Fifth FTW PhD Symposium, 2004.

[Des04a] Desmet V., Eeckhout L., De Bosschere K. – *Evaluation of the Gini-index for Studying Branch Prediction Features*, Proceedings of the 6th International Conference on Computing Anticipatory Systems (CASYS). American Institute of Physics. AIP Conference Proceedings. Vol. 718. 2004. pp. 376-384.

[Die98] Dieffendorff K. – *K7 challenges Intel*. Microprocessor Report, 12(14), October 1998.

[Ding04] Ding Y, Li Z. – *A Compiler Scheme for Reusing Intermediate Computation Result*, Proceedings of IEEE/ACM 2004, International Symposium on Code Generation and Optimization (CGO 2004), March 2004, Palo Alto, California.

[Dri98a] Driesen K., Holzle U. – *Accurate Indirect Branch Prediction*. In Proceedings of the International Symposium on Computer Architecture, pages 167-178, Barcelona, Spain, June 1998.

[Dri98b] Driesen K., Holzle U. – *Improving Indirect Branch Prediction With Source- and Aritybased Classification and Cascaded Prediction*. Technical Report TRCS98-07, Computer Science Department, University of California, Santa Barbara, 15 March 1998.

[Dri98c] Driesen K., Holzle U. – *The Cascaded Predictor: Economical and Adaptive Branch Target Prediction*. Micro'98 Conference Proceedings, Dallas, Texas, December 1998.

- [Dri99] **Driesen K., Holzle U.** – *Multi-stage Cascaded Prediction*. EuroPar'99 Conference Proceedings, Toulouse, France, September 1999.
- [Ega03] **Egan, C., Steven, G., Quick, P., Anguera, R., Steven, F. and Vintan, L.** – *Two-level branch prediction using neural networks*, Journal of Systems Architecture 49(12), Elsevier, December, 2003.
- [Eve96] **Evers M., Chang P.Y., Patt Y.N.** – *Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches*, ISCA '96.
- [Fal04] **Falcon A., Stark J., Ramirez A., Lai K., Valero M.** – *Prophet / Critic Hybrid Branch Prediction*, In Proceedings of the International Symposium on Computer Architecture, Munchen, Germany, June 2004.
- [Fern03] **Fern A., Givan R., Falsafi B., Vijaykumar T.N.** – *Dynamic Feature Selection for Hardware Prediction*, Technical Report Purdue University, USA, 2003.
- [Fis92] **Fisher J.A., Freudenberger S.M.** – *Predicting conditional branch directions from previous runs of a program*, in Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 85-95, Boston, Mass, October 1992, ACM.
- [Flo00] **Florea, A. Egan C.** – *Reducing the Technological Gap between an Advanced Processor and the Memory Hierarchy System*, Transactions on Automatic Control and Computer Science, vol 45, no. 4, Timisoara, Romania, 2000.
- [Flo02] **Florea A., Vintan L., Sima D.** – *Understanding Value Prediction through Complex Simulations*, Proceedings of the 5th International Conference on Technical Informatics, University “Politehnica” of Timisoara, Romania, October, 2002.
- [Flo03] **Florea A., Vințan L.** – *Simularea și optimizarea arhitecturilor de calcul în aplicații practice*, Editura MatrixRom, București, 2003, ISBN 973-685-605-4.
- [Flo03a] **Florea A.** – *Experimente privind simularea unor structuri paralele de procesare a informației*, Referat de doctorat nr. 2, Universitatea „Politehnica” București, Iulie, 2003.

- [Flo04] **Florea A., Vintan L., Miha I. Z.** – *Understanding and Predicting Indirect Branch Behavior*, Studies in Informatics and Control Journal: With Emphasis on Useful Applications of Advanced Technology, March 2004, Vol.13, No. 1, Bucharest.
- [Flo04a] **Florea A.** – *Predicția valorilor și reutilizarea dinamică a instrucțiunilor în arhitectura superscalară parametrizabilă*, Referat de doctorat nr. 3, Universitatea „Politehnica” București, Aprilie, 2004.
- [Fra93] **Franklin, M.** – *Multiscalar Processors*, Ph. D Thesis, University of Wisconsin, 1993.
- [Gal93] **Gallant S.I.** - *Neural Networks and Expert Systems*, MIT Press. 1993.
- [Gab98] **Gabbay F., Mendelsohn A.** – *The Effect of Instruction Fetch Bandwidth on Value Prediction*, In Proceedings of the 25th International Symposium on Computer Architecture, June, 1998.
- [Gab98a] **Gabbay F., Mendelsohn A.** – *Using Value Prediction To Increase The Power Of Speculative Execution Hardware*, ACM Transactions on Computer Systems, vol 16, nr. 3, August, 1998.
- [Gon99] **Gonzalez A., Tubella J., Molina C.** – *Trace-Level Reuse*, International Conference on Parallel Processing, September 21 – 24, 1999, Japan.
- [Henn00] **Henning J.** - *SPEC CPU2000: Measuring CPU Performance in the New Millennium*, Computing Practices, 0018-9162/00 © 2000 IEEE.
- [Henn03] **Hennessy J., Patterson D.** – *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3rd Edition, 2003.
- [Hin01] **Hinton G., Carmean D. et al** – *The microarchitecture of the Pentium 4 processor*. Intel Technology Journal Q1, 2001.
- [Hoa62] **Hoare, C. A. R.** -. Quicksort. *Computer Journal*, 5(1):10-15, 1962.
- [Hua99] **Huang J., Lilja D.** – *Exploiting Basic Block Value Locality with Block Reuse*, Proceedings of the The Fifth International Symposium on High Performance Computer Architecture, p.106, January 09-12, 1999.
- [Intel97] **Intel press release.** *The Next Generation of Microprocessor Architecture: A 64-bit Instruction Set Architecture (ISA) Based on EPIC Technology*. Intel Corporation October 1997.

[Intel02] http://radified.com/CPU/Intel_northwood_pentium_4.htm, January 2002.

[Intel03] http://www.lostcircuits.com/cpu/intel_p4ee/, October 2003.

[Jac99] **Jacobson Q., Smith J.E.** – *Instruction Pre-Processing in Trace Processors*, Proceedings of the 5th International Symposium on High Performance Computer Architecture, 1999, SUA.

[JILP04] *The Journal of Instruction-Level Parallelism: the 1st Branch Prediction Championship*, <http://www.jilp.org/cbp>.

[Jim02] **Jimenez D., Lin C.** – *Neural methods for dynamic branch prediction*. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[Jim02a] **Jimenez D.** – *Fast Path-Based Neural Branch Prediction*, in the Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), December 2003.

[Jim02b] **Jimenez D. A.** – *Delay-Sensitive Branch Predictors for Future Technologies*, PhD Thesis, Technical Report TR-02-2, Department of Computer Sciences, The University of Texas at Austin, USA, May, 2002.

[Juan98] **Juan, T., Sanjeevan, S., Navarro, J.** – *Dynamic History-Length Fitting: A third level of adaptivity for branch prediction*, In Proceedings of the 25th Annual International Symposium on Computer Architecture, Barcelona, Spain, June 1998.

[Jou90] **Jouppi N.** – *Improving Direct-Mapped Cache Performance by the addition of a Small Fully Associative Cache and Prefetch Buffers*, Proceedings of the 17th International Symposium On Computer Architecture, 1990.

[Kae97] **Kaeli D. R., Emma P.** – *Improving the Accuracy of History-Based Branch Prediction*. *IEEE Transactions on Computer Architecture*, 46(4):469-472, April 1997.

[Kal98] **Kalamatianos J., Kaeli D.R.** – *Predicting Indirect Branches via Data Compression*. In Proc. of 31st International Symposium on Microarchitecture, pages 272-281, Dec. 1998.

[Kavi03] Kavi K., Chen P. – *Dynamic Function Result Reuse*, Proceedings of the 11th International Conference on Advanced Computing (ADCOM-2003), Combatore, India, 17-20 December, 2003.

[Kes99] Kessler R. E. – *The Alpha 21264 microprocessor*. IEEE Micro 19, 2 (March/April), 24–36.

[Knu71] Knuth D.E. – *An empirical study of FORTRAN programs*, Software, Practice and Experience, 1:105-133, 1971.

[Lam79] Lamport L. – *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*. IEEE Transactions on Computers, C-28(9):690-691, Sept. 1979.

[Lar95] Larus J., Schnarr E. – *EEL: Machine-Independent Executable Editing*, In PLDI'95 Conference Proceedings. Pp. 291-300, La Jolla, California, June 1995.

[Lee97] Lee C., Potkonjak M. and Mangione-Smith W. - *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*, 1997.

[Lee98] Lee D., Crowley P., Baer J.L., Anderson T. – *Execution Characteristics of Desktop Applications on Windows NT*, 25th Annual International Symposium on Computer Architecture, Barcelona, Spain, June 1998.

[Lee99] Lee S., Wang Y., Yew P. – *Decoupling Value Prediction on Trace Processors*, In Proceedings of the 6th International Symposium on High Performance Computer Architecture, 1999.

[Lee02] Lee S., Yew P. – *On Augmenting Trace Cache for High-Bandwidth Value Prediction*, IEEE Transactions on Computers, Vol. 51, No. 9, September 2002.

[Lep00] Lepak K., Lipasti M. – *On The Value Locality of Store Instructions*, International Symposia on Computer Architecture, Vancouver, Canada, 2000.

[Lep00a] Lepak K. M., Lipasti M. – *Silent Stores for Free*, Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO33), California, USA, 2000.

[Lip96] Lipasti M., Wilkerson C., Shen P. – *Value Locality and Load Value Prediction*, 17th ASPLOS International Conference VII, pg. 138-147, MA, SUA, October 1996.

[Lip96b] Lipasti M., Shen J.P. – *Exceeding The Dataflow Limit Via Value Prediction*, Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture, December, 1996.

[Lip01] Lippasti M., Martin M., Sorin D., Cain H., Hill M. – *Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing*, Proceedings of the 34th Annual ACM / IEEE International Symposium on Microarchitecture, Austin, Texas, December, 2001.

[Mar99] Marcuello P., Tubella J., Gonzales A. – *Value Prediction for Speculative Multithreaded Architectures*, 1072-4451/IEEE, 1999.

[Mar00] Marcuello P., Gonzales A. – *A quantitative assessment of threadlevel speculation techniques*. In Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS '00), pages 595-604, Cancún (Mexico), May 2000.

[McFar93] McFarling S. – *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[Min88] Minsky M., Papert S. – *Perceptrons*. In Neurocomputing: Foundations of Research, pages 157–169. MIT Press, 1988.

[Moff90] Moffat, A. - *Implementing the PPM data compression scheme*. IEEE Transactions on Communications, Vol. 38, No. 11, November 1990.

[Mud96] Mudge T. Chen, I., Coffey, J. – *Limits to Branch Prediction*, Technical Report, University of Michigan, January 1996.

[Nai95] Nair, R. – *Optimal 2-Bit Branch Predictors*, IEEE Transaction on Computers, No. 5, 1995.

[Nak99] Nakra, T., Gupta, R., Soffa, M. L. – *Global context-based value prediction*, in 5th International Symposium on High Performance Computer Architecture, 1999.

[Oni66] Onicescu O. – *Theorie de l'information. Energie informationelle*. C. R., Academie Science, Ser. A-B, Tome 263:841–842, 1966.

[Pan92] Pan S.T., So K., Rahmeh J.T. – *Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation*, ASPLOS V Conference, Boston, October, 1992.

[Pat97] Patel S.J., Friendly D. H., Patt Y. N. – Critical Issues Regarding The Trace Catch Fetch Mechanism. Technical report, University of Michigan, 1997.

[Pat98] Patel S., Evers M., Patt Y. – *Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing*, Proc. 25th Int'l Symp. Computer Architecture, pp. 262-271, June 1998.

[Per93] Perleberg C., Smith A. J. – *Branch Target Buffer Design and Optimisation*, IEEE Trans. Computers, No. 4, 1993.

[Pett90] Pettis K., Hansen R.C. – *Profile guided code positioning*. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pages 16-27, June 1990.

[Pop00] Popescu Th. – *Serii de timp*, Editura Tehnică, București, 2000.

[Pos00] Postiff M., Greene D., Lefurgy C., Helder D., Mudge T. – *The MIRV SimpleScalar/PISA Compiler*, University of Michigan EECS Department Tech. Report CSE-TR-421-00. April 2000.

[Qui93] Quinlan J.R. – *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

[Ros62] Rosenblatt F. – *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, New York.

[Rot97] Rotenberg E., Jacobson Q., Sazeides Y., Smith J.E. – *Trace Processors*, in Proceedings of the 30th International Symposium on Microarchitecture, pp. 138-148, 1997.

[Roth99] Roth A., Moshovos A., Sohi G.S. – *Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation*, Proceedings of the 13th International Conference on Supercomputing, Rhodes, Greece, 1999.

[Rych98] Rychlik B., Faistl J., Krug B., Kurland A., Jung J., Velez M., Shen J. - *Efficient and Accurate Value Prediction Using Dynamic Classification*, Technical Report of Microarchitecture Research Team,

Department of Electrical and Computer Engineering, Carnegie Mellon Univ., 1998.

[Sas00] **Sastry S.S., Bodik R., Smith J.E.** – *Characterizing Coarse-Grained Reuse of Computation*, 3rd ACM Workshop on Feedback Directed and Dynamic Optimization in conjunction with MICRO 33, December 2000.

[Saz97] **Sazeides Y., Smith J.E.** – *The Predictability of Data Values*, Proceedings of The 30th Annual International Symposium on Microarchitecture, December, 1997.

[Saz99] **Sazeides Y.** – *An analysis of value predictability and its application to a superscalar processor*, PhD Thesis, University of Wisconsin-Madison, 1999.

[Sed92] **Sedgewick, R.** – *Algorithms in C++*. Reading, Massachusetts: Addison-Wesley, 1992.

[Seng04] **Seng J., Hamerly G.** – *Exploring Perceptron-Based Register Value Prediction*, The 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2) in conjunction with the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 11), October 2004, Boston, SUA.

[Sez02] **Seznec A., Felix S., Krishnan V., Sazeides Y.** – *Design tradeoffs for the Alpha EV8 conditional branch predictor*. In Proceedings of the 29th International Symposium on Computer Architecture, May 2002.

[Sez04] **Seznec A.** – *Revisiting the Perceptron Predictor*, IRISA research reports, IRISA Editeur, May, 2004.

[Sias04] **Sias J.W., Ueng S., Kent G., Steiner I., Nystrom E., Hwu W.** – *Field-testing IMPACT EPIC research results in Itanium 2*, The 31th Annual International Symposium on Computer Architecture, Munchen, Germany, June 2004.

[Sil99] **Silc J., Robic B., Ungerer T.** – *Processor Architecture, from Dataflow to Superscalar and beyond*, Springer-Verlag, 1999.

[Smi84] **Smith A., Lee J.** – *Branch Prediction Strategies and Branch Target Buffer Design*, Computer 17:1, January 1984.

[SPEC] – *The SPEC benchmark programs*, <http://www.spec.org>.

- [Spra02] Sprangle E., Carmean D. – *Increasing processor performance by implementing deeper pipelines*. In Proceedings of the 29th International Symposium on Computer Architecture, Anchorage, Alaska, May 25 - 29, 2002.
- [Sod97] Sodani A., Sohi G. – *Dynamic Instruction Reuse*, Proceedings of the 24th International Symposium on Computer Architecture, pp. 194-205, June 1997.
- [Sod98] Sodani A., Sohi G. – *An Empirical Analysis of Instruction Repetition*, Int'l ASPLOS Conference, 0-8186-8609-X/IEEE, 1998.
- [Sod00] Sodani A. – *Dynamic Instruction Reuse*, PhD Thesis, University of Wisconsin – Madison, USA, 2000.
- [Sta98] Stark J., Evers M., Patt Y. – *Variable Length Path Branch Prediction*, In Proceedings of the 8th Int'l Conference of Architectural Support for Programming Languages and Operating Systems, pages 170-179, 1998.
- [Ste99] Steven G. – *Exploiting Instruction-Level Parallelism in High Performance Processors*, Proceedings of International Conference “Beyond 2000: Engineering Research Strategies”, 25-26 Nov. '99, Vol. XXXVIII, ISSN 1221-4949, Editura Universităţii "L. Blaga", Sibiu, (Romania), 1999.
- [Sti94] Stiliadis, D., Varma, A. – *Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches*, TR UCSC-CRL-93-41, University of California, 1994 (republished in a shorter version in IEEE Trans. on Computers, May 1997).
- [Tar04] Tarjan D., Skadron K. – *Revisiting the Perceptron Prediction Again*, Technical Report CS-2004-28, University of Virginia, USA, September 2004.
- [Tat00] Tate D., Steven G., Steven F. – *Static Scheduling for Out-of-order Instruction Issue Processors*, Proceedings of 6th Australasian Computer Architecture Conference [ACAC2000](#): 90-96.
- [Tho03] Thomas R., Franklin M., Wilkerson C., Stark J. – *Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History*, The 30th Annual International Symposium on Computer Architecture, San Diego, California, June 09 - 11, 2003.

[Tho04] **Thomas A., Kaeli D.** – *Value Prediction with Perceptrons*, The 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2) in conjunction with the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 11), October 2004, Boston, SUA.

[Tull99] **Tullsen D. M., Seng J. S.** – *Storageless Value Prediction using Prior Register Values*, Proceedings of the 26th International Symposium on Computer Architecture, May 1999.

[Vin99] **Vințan L., Egan, C.** – *Extending Correlation in Branch Prediction Schemes*, International Euromicro'99 Conference, Milano, Italy, September 1999.

[Vin99a] **Vințan L., Florea A.** - *Investigating New Branch Prediction Through Quantitative Approach* – Beyond 2000: Engineering Research Strategies, November 25-27, Sibiu, 1999.

[Vin99b] **Vintan L., Iridon M.** – *Towards a High Performance Neural Branch Predictor*, International Joint Conference on Neural Networks (IJCNN CD-ROM, ISBN 0-7803-5532-6), Washington DC, USA, 10-16 July, 1999.

[Vin00] **Vințan L.** - *Arhitecturi de procesoare cu paralelism la nivelul instrucțiunilor*, Editura Academiei Române, București, ISBN 973-27-0734-8, 2000 (264 pg.).

[Vin00a] **Vințan N. L., Florea A.** – *Microarhitecturi de procesare a informației*, Editura Tehnică, București, ISBN 973-31-1551-7, 2000 (312 pg.).

[Vin00b] **Vintan L. Căndea C., Staicu M.** – *Automatic Synthesis of Branch Prediction Schemes through Genetic Programming*, Transactions on Automatic Control and Computer Science, Special Issue Dedicated to Fourth International Conf. on Technical Informatics (CONTI'2000), Volume 45 (59), No 4, ISSN 1224-600X, University "Politehnica" of Timisoara, Romania, 2000.

[Vin00c] **Vintan L.** – *Towards a Powerful Dynamic Branch Predictor*, ROMJIST, nr.3, Academia Română, 2000.

[Vin01] **Vintan L., Florea A.** – *Cross-Fertilisation between Computer Architecture and other Computer Science Fields*, Proceedings of the

Conference on Computer Science and Control Systems (CSCS-13), Buurești, România, 01-03 June, 2001.

[Vin02] **Vințan L.** – *Predicție și speculație în microprocesoarele avansate*, Editura MatrixRom, București, ISBN 973-685-497-3, 2002.

[Vin04] **Vințan L., Gellert A., Florea A.** – *Register Value Prediction using Metapredictors*, Proceedings of the 8th International Symposium on Automatic Control and Computer Science, Iasi, October 2004.

[Vin05] **Vințan L., Florea A., Gellert A.** – *Focalizing Dynamic Value Prediction to CPU's Context*, IEE Proc. Computers & Digital Techniques, ISSN: 1350-2387, United Kingdom, 2005.

[Wal99] **Wallace S., Calder B., Tullsen D.** - *Threaded Multiple Path Execution*, 25th Int'l Symp. on Computer Architecture (ISCA), Barcelona, June, 1999.

[Wang97] **Wang K., Franklin M.** – *Highly Accurate Data Value Prediction using Hybrid Predictors*, Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1997.

[Watt01] **Watterson S., Debray S.** – *Goal-Directed Value Profiling*, Proc. Compiler Construction 2001 (CC 2001), April 2001.

[Yeh92] **Yeh, T., Patt, Y.** - *Alternative Implementations of Two-Level Adaptive Branch Prediction*, Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 124-134, 1992.

[Yeh93] **Yeh, T., Marr D.T., Patt, Y.** – *Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache*, Proceedings of the 7th International Conference on Supercomputing, pp. 67-76, 1993.

[Zha00] **Zhao Q., Lilja D.J.** – *Compiler-Directed Static Clasification of Value Locality Behavior* Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 00-07, July, 2000.

[Zhou03] **Zhou, H., Flanagan, J., Conte, Th.** – *Detecting Global Stride Locality in Value Streams*, Proceedings of the 30th International Symposium on Computer Architecture, June 2003, San Diego, California.

ANEXA 1

EXEMPLE JUSTIFICATIVE PRIVIND IMPACTUL LA NIVEL MICROARHITECTURAL AL UNOR TEHNICI DE ÎMBUNĂTĂȚIRE A PERFORMANȚEI PROCESOARELOR PRIN PREDICȚIA SALTURILOR INDIRECTE

Anexa de față cuprinde două aplicații .cpp, simple dar sugestive și care reliefează două aspecte importante la nivel microarhitectural. Primul exemplu urmărește să evidențieze situații în care extinderea informației de corelație pentru instrucțiunile de salt indirect are sens, contribuind la creșterea acurateții predicției acestora. A doua aplicație vine însă să dovedească limitările tehnicii de extindere a informației de corelație și respectiv ineficiența uneori a creșterii gradului de asociativitate pentru o structură Target Cache în vederea obținerii dezideratului de performanță ridicată prin predicția cu acuratețe a salturilor indirecte. Compilarea surselor s-a făcut cu GNU C++ 2.6.3. și opțiunea de optimizare `-O3` pe un procesor Intel80x86 și sistem de operare Linux Red Hat 7.3. Opțiunea `-O3` pe lângă reducerea dimensiunii codului și a timpului de execuție permite inlining-ul aplicat funcțiilor și aplicarea *delay slot*-ului pentru instrucțiunile de salt (*f_inline_functions*, *f_delayed_branch*).

Comenzile cu ajutorul cărora obținem codul asamblare sunt:

■ pentru sursa de program C:

```
./xgcc nume_fisier.c -S
```

■ pentru programele obiectuale CPP:

```
./cc1plus nume_fisier.cpp -s
```

Codul obiect pentru arhitectura SimpleScalar se obține cu ajutorul comenzilor:

■ pentru ambele tipuri de programe (atât C cât și C++):

```
./xgcc nume_fisier_sursa.c -o nume_fisier_destinatie.ss
```

Se reamintește că opțiunea de compilare `-S` realizează preprocesarea și compilarea codului sursă, iar opțiunea `-s` generează doar codul asamblare fără faza de preprocesare.

```

#include <stdio.h>
void main(){
char c; int i, x, z;
int v=random(5);
scanf("%d %d",&x,&z);
if(x == 3){
  if(z == 6)
    v = 3;
  else
    for(i=0;i<2;i++){
      v++;
      v = v%5;
    }
  switch(v){
    case 0:c='a'; break;
    case 1:c='b'; break;
    case 2:c='c'; break;
    case 3:c='d'; break;
    case 4:c='e'; break;
  }
}
}

file 1 "switch_cpp"
# GNU C++ 2.6.3 SimpleScalar
running sstrip compiled by GNU C
# Cc1 defaults:
# -mgas -mfpOPT
# Cc1 arguments (-G value = 8, Cpu
= default, ISA = 1):
# -quiet -dumpbase -o
gcc2_compiled:
__gnu_compiled_cplusplus:
.sdata
.align 2
$L31:
$L32:
$L33:
$L34:
$L35:
$L36:
$L37:
$L38:
$L39:
$L40:
$L41:
$L42:
$L43:
$L44:
$L45:
$L46:
$L47:
$L48:
$L49:
$L50:
$L51:
$L52:
$L53:
$L54:
$L55:
$L56:
$L57:
$L58:
$L59:
$L60:
$L61:
$L62:
$L63:
$L64:
$L65:
$L66:
$L67:
$L68:
$L69:
$L70:
$L71:
$L72:
$L73:
$L74:
$L75:
$L76:
$L77:
$L78:
$L79:
$L80:
$L81:
$L82:
$L83:
$L84:
$L85:
$L86:
$L87:
$L88:
$L89:
$L90:
$L91:
$L92:
$L93:
$L94:
$L95:
$L96:
$L97:
$L98:
$L99:
$L100:
$L101:
$L102:
$L103:
$L104:
$L105:
$L106:
$L107:
$L108:
$L109:
$L110:
$L111:
$L112:
$L113:
$L114:
$L115:
$L116:
$L117:
$L118:
$L119:
$L120:
$L121:
$L122:
$L123:
$L124:
$L125:
$L126:
$L127:
$L128:
$L129:
$L130:
$L131:
$L132:
$L133:
$L134:
$L135:
$L136:
$L137:
$L138:
$L139:
$L140:
$L141:
$L142:
$L143:
$L144:
$L145:
$L146:
$L147:
$L148:
$L149:
$L150:
$L151:
$L152:
$L153:
$L154:
$L155:
$L156:
$L157:
$L158:
$L159:
$L160:
$L161:
$L162:
$L163:
$L164:
$L165:
$L166:
$L167:
$L168:
$L169:
$L170:
$L171:
$L172:
$L173:
$L174:
$L175:
$L176:
$L177:
$L178:
$L179:
$L180:
$L181:
$L182:
$L183:
$L184:
$L185:
$L186:
$L187:
$L188:
$L189:
$L190:
$L191:
$L192:
$L193:
$L194:
$L195:
$L196:
$L197:
$L198:
$L199:
$L200:
$L201:
$L202:
$L203:
$L204:
$L205:
$L206:
$L207:
$L208:
$L209:
$L210:
$L211:
$L212:
$L213:
$L214:
$L215:
$L216:
$L217:
$L218:
$L219:
$L220:
$L221:
$L222:
$L223:
$L224:
$L225:
$L226:
$L227:
$L228:
$L229:
$L230:
$L231:
$L232:
$L233:
$L234:
$L235:
$L236:
$L237:
$L238:
$L239:
$L240:
$L241:
$L242:
$L243:
$L244:
$L245:
$L246:
$L247:
$L248:
$L249:
$L250:
$L251:
$L252:
$L253:
$L254:
$L255:
$L256:
$L257:
$L258:
$L259:
$L260:
$L261:
$L262:
$L263:
$L264:
$L265:
$L266:
$L267:
$L268:
$L269:
$L270:
$L271:
$L272:
$L273:
$L274:
$L275:
$L276:
$L277:
$L278:
$L279:
$L280:
$L281:
$L282:
$L283:
$L284:
$L285:
$L286:
$L287:
$L288:
$L289:
$L290:
$L291:
$L292:
$L293:
$L294:
$L295:
$L296:
$L297:
$L298:
$L299:
$L300:
$L301:
$L302:
$L303:
$L304:
$L305:
$L306:
$L307:
$L308:
$L309:
$L310:
$L311:
$L312:
$L313:
$L314:
$L315:
$L316:
$L317:
$L318:
$L319:
$L320:
$L321:
$L322:
$L323:
$L324:
$L325:
$L326:
$L327:
$L328:
$L329:
$L330:
$L331:
$L332:
$L333:
$L334:
$L335:
$L336:
$L337:
$L338:
$L339:
$L340:
$L341:
$L342:
$L343:
$L344:
$L345:
$L346:
$L347:
$L348:
$L349:
$L350:
$L351:
$L352:
$L353:
$L354:
$L355:
$L356:
$L357:
$L358:
$L359:
$L360:
$L361:
$L362:
$L363:
$L364:
$L365:
$L366:
$L367:
$L368:
$L369:
$L370:
$L371:
$L372:
$L373:
$L374:
$L375:
$L376:
$L377:
$L378:
$L379:
$L380:
$L381:
$L382:
$L383:
$L384:
$L385:
$L386:
$L387:
$L388:
$L389:
$L390:
$L391:
$L392:
$L393:
$L394:
$L395:
$L396:
$L397:
$L398:
$L399:
$L400:
$L401:
$L402:
$L403:
$L404:
$L405:
$L406:
$L407:
$L408:
$L409:
$L410:
$L411:
$L412:
$L413:
$L414:
$L415:
$L416:
$L417:
$L418:
$L419:
$L420:
$L421:
$L422:
$L423:
$L424:
$L425:
$L426:
$L427:
$L428:
$L429:
$L430:
$L431:
$L432:
$L433:
$L434:
$L435:
$L436:
$L437:
$L438:
$L439:
$L440:
$L441:
$L442:
$L443:
$L444:
$L445:
$L446:
$L447:
$L448:
$L449:
$L450:
$L451:
$L452:
$L453:
$L454:
$L455:
$L456:
$L457:
$L458:
$L459:
$L460:
$L461:
$L462:
$L463:
$L464:
$L465:
$L466:
$L467:
$L468:
$L469:
$L470:
$L471:
$L472:
$L473:
$L474:
$L475:
$L476:
$L477:
$L478:
$L479:
$L480:
$L481:
$L482:
$L483:
$L484:
$L485:
$L486:
$L487:
$L488:
$L489:
$L490:
$L491:
$L492:
$L493:
$L494:
$L495:
$L496:
$L497:
$L498:
$L499:
$L500:
$L501:
$L502:
$L503:
$L504:
$L505:
$L506:
$L507:
$L508:
$L509:
$L510:
$L511:
$L512:
$L513:
$L514:
$L515:
$L516:
$L517:
$L518:
$L519:
$L520:
$L521:
$L522:
$L523:
$L524:
$L525:
$L526:
$L527:
$L528:
$L529:
$L530:
$L531:
$L532:
$L533:
$L534:
$L535:
$L536:
$L537:
$L538:
$L539:
$L540:
$L541:
$L542:
$L543:
$L544:
$L545:
$L546:
$L547:
$L548:
$L549:
$L550:
$L551:
$L552:
$L553:
$L554:
$L555:
$L556:
$L557:
$L558:
$L559:
$L560:
$L561:
$L562:
$L563:
$L564:
$L565:
$L566:
$L567:
$L568:
$L569:
$L570:
$L571:
$L572:
$L573:
$L574:
$L575:
$L576:
$L577:
$L578:
$L579:
$L580:
$L581:
$L582:
$L583:
$L584:
$L585:
$L586:
$L587:
$L588:
$L589:
$L590:
$L591:
$L592:
$L593:
$L594:
$L595:
$L596:
$L597:
$L598:
$L599:
$L600:
$L601:
$L602:
$L603:
$L604:
$L605:
$L606:
$L607:
$L608:
$L609:
$L610:
$L611:
$L612:
$L613:
$L614:
$L615:
$L616:
$L617:
$L618:
$L619:
$L620:
$L621:
$L622:
$L623:
$L624:
$L625:
$L626:
$L627:
$L628:
$L629:
$L630:
$L631:
$L632:
$L633:
$L634:
$L635:
$L636:
$L637:
$L638:
$L639:
$L640:
$L641:
$L642:
$L643:
$L644:
$L645:
$L646:
$L647:
$L648:
$L649:
$L650:
$L651:
$L652:
$L653:
$L654:
$L655:
$L656:
$L657:
$L658:
$L659:
$L660:
$L661:
$L662:
$L663:
$L664:
$L665:
$L666:
$L667:
$L668:
$L669:
$L670:
$L671:
$L672:
$L673:
$L674:
$L675:
$L676:
$L677:
$L678:
$L679:
$L680:
$L681:
$L682:
$L683:
$L684:
$L685:
$L686:
$L687:
$L688:
$L689:
$L690:
$L691:
$L692:
$L693:
$L694:
$L695:
$L696:
$L697:
$L698:
$L699:
$L700:
$L701:
$L702:
$L703:
$L704:
$L705:
$L706:
$L707:
$L708:
$L709:
$L710:
$L711:
$L712:
$L713:
$L714:
$L715:
$L716:
$L717:
$L718:
$L719:
$L720:
$L721:
$L722:
$L723:
$L724:
$L725:
$L726:
$L727:
$L728:
$L729:
$L730:
$L731:
$L732:
$L733:
$L734:
$L735:
$L736:
$L737:
$L738:
$L739:
$L740:
$L741:
$L742:
$L743:
$L744:
$L745:
$L746:
$L747:
$L748:
$L749:
$L750:
$L751:
$L752:
$L753:
$L754:
$L755:
$L756:
$L757:
$L758:
$L759:
$L760:
$L761:
$L762:
$L763:
$L764:
$L765:
$L766:
$L767:
$L768:
$L769:
$L770:
$L771:
$L772:
$L773:
$L774:
$L775:
$L776:
$L777:
$L778:
$L779:
$L780:
$L781:
$L782:
$L783:
$L784:
$L785:
$L786:
$L787:
$L788:
$L789:
$L790:
$L791:
$L792:
$L793:
$L794:
$L795:
$L796:
$L797:
$L798:
$L799:
$L800:
$L801:
$L802:
$L803:
$L804:
$L805:
$L806:
$L807:
$L808:
$L809:
$L810:
$L811:
$L812:
$L813:
$L814:
$L815:
$L816:
$L817:
$L818:
$L819:
$L820:
$L821:
$L822:
$L823:
$L824:
$L825:
$L826:
$L827:
$L828:
$L829:
$L830:
$L831:
$L832:
$L833:
$L834:
$L835:
$L836:
$L837:
$L838:
$L839:
$L840:
$L841:
$L842:
$L843:
$L844:
$L845:
$L846:
$L847:
$L848:
$L849:
$L850:
$L851:
$L852:
$L853:
$L854:
$L855:
$L856:
$L857:
$L858:
$L859:
$L860:
$L861:
$L862:
$L863:
$L864:
$L865:
$L866:
$L867:
$L868:
$L869:
$L870:
$L871:
$L872:
$L873:
$L874:
$L875:
$L876:
$L877:
$L878:
$L879:
$L880:
$L881:
$L882:
$L883:
$L884:
$L885:
$L886:
$L887:
$L888:
$L889:
$L890:
$L891:
$L892:
$L893:
$L894:
$L895:
$L896:
$L897:
$L898:
$L899:
$L900:
$L901:
$L902:
$L903:
$L904:
$L905:
$L906:
$L907:
$L908:
$L909:
$L910:
$L911:
$L912:
$L913:
$L914:
$L915:
$L916:
$L917:
$L918:
$L919:
$L920:
$L921:
$L922:
$L923:
$L924:
$L925:
$L926:
$L927:
$L928:
$L929:
$L930:
$L931:
$L932:
$L933:
$L934:
$L935:
$L936:
$L937:
$L938:
$L939:
$L940:
$L941:
$L942:
$L943:
$L944:
$L945:
$L946:
$L947:
$L948:
$L949:
$L950:
$L951:
$L952:
$L953:
$L954:
$L955:
$L956:
$L957:
$L958:
$L959:
$L960:
$L961:
$L962:
$L963:
$L964:
$L965:
$L966:
$L967:
$L968:
$L969:
$L970:
$L971:
$L972:
$L973:
$L974:
$L975:
$L976:
$L977:
$L978:
$L979:
$L980:
$L981:
$L982:
$L983:
$L984:
$L985:
$L986:
$L987:
$L988:
$L989:
$L990:
$L991:
$L992:
$L993:
$L994:
$L995:
$L996:
$L997:
$L998:
$L999:
$L1000:

```

Figura A1.1. Exemplu ce justifică necesitatea extinderii informației de corelație în predicția salturilor indirecte

Prima aplicație, simplă la nivel *high-level*, dar mai complexă și dificil de urmărit la nivel asamblare urmărește să evidențieze situații în care extinderea informației de corelație pentru instrucțiunile de salt indirect are sens, contribuind la creșterea acurateții predicției acestora. În acest scop se vor evidenția două pattern-uri diferite de salturi condiționate dar cu comportament identic (pentru HRgLength=2) care conduc la aceeași instrucțiune de salt indirect. Aplicația conține două instrucțiuni de selecție simplă (*if*), una multiplă (*switch*) respectiv o instrucțiune repetitivă (*for*), a căror execuție este controlată prin intermediul unor variabile citite de la tastatură. Prin compilarea codului sursă *high-level* rezultă în principal patru instrucțiuni de salt condiționat, una de salt indirect și numeroase instrucțiuni de salt necondiționat (vezi figura A1.1).

Comportamentul saltului condiționat **B1** (**beq \$3, \$0, \$L27**) depinde de valoarea care va fi atribuită lui x după citirea de la tastatură (sw \$2, 28(\$fp)). Datorită corelației dintre saltul B1 și celelalte salturi condiționate (**B2**, **B3**, **B4**) se poate ajunge la situația în care dacă $x \neq 3$ B1 să se facă iar la celelalte trei salturi condiționate să nu se mai ajungă, însă nu aceasta s-a urmărit în cadrul acestui exemplu. Prin urmare, se consideră că x va lua valoarea 3 iar saltul **B1** nu se va face (NT). Salturile **B2** (**beq \$3, \$0, \$L28**) și **B3** (**bne \$3, \$0, \$L33**) depind de valoarea pe care o ia variabila z (sw \$2, 32(\$fp)), comportamentul lor fiind mutual exclusiv. Astfel, dacă $z=6$ rezultă că saltul **B2** nu se va face (NT), variabila v (sw \$2, 20(\$fp)) luând valoarea 3 și se va ajunge la saltul condiționat **B4** (**beq \$3, \$0, \$L41**) premergător saltului indirect (**j \$2**). Întrucât $v=3$ (parametru de control al construcției *switch/case*) rezultă că **B4** nu se va face (NT). În concluzie, o cale pe care se ajunge la saltul indirect **j \$2** este **B1, B2, B4** cu comportamentul NT, NT, NT. Dacă însă $z \neq 6$ atunci **B2** se va face (T) iar saltul **B3** se va executa repetat de două ori astfel (T, T) conform variabilei de control a buclei $i < 2$ – (sw \$3, 24(\$fp)). La a treia iterație a saltului condiționat **B3** acesta nu se va mai face (NT). Datorită instrucțiunilor din corpul instrucțiunii repetitive *for* rezultă că variabila de control pentru instrucțiunea de selecție *switch/case* va fi tot timpul sub valoarea 5 (numărul de cazuri posibile), ceea ce înseamnă că saltul **B4** nu se va face (NT). Așadar, o nouă cale dinamică prin care se ajunge la saltul indirect este următoarea: **B1, B2, B3, B3, B3, B4** comportamentul acestora fiind NT, T, T, T, NT, NT. Privind doar din prisma comportamentului ultimelor două salturi condiționate rezultă că indiferent pe ce cale ajung la saltul indirect se va accesa același set din Target Cache, sporind numărul de conflicte. Dacă se extinde informația de corelație păstrând pe lângă cei HRgLength biți ai globalHR și PC-urile corespondente se poate identifica mai clar target-ul spre care se „merge” de

fiecare dată. Astfel, în condițiile în care secvența de instrucțiuni supusă atenției s-ar relua din punct de vedere dinamic (apel de funcție, buclă de program, recursivitate), s-ar putea ca pentru calea B2, B4 – NT, NT, să fie prezis fără greșeală target-ul întrucât de fiecare dată variabila de selecție aferentă construcției switch/case ia aceeași valoare. Pentru calea B3, B4 – NT, NT se va accesa alt set în structura Target Cache, reducându-se din interferențe și crescând acuratețea de predicție a salturilor indirecte, chiar dacă pe această cale target-ul poate diferi de la o instanță la alta a saltului indirect, dependent de rezultatul returnat de funcția *random*.

Cea de-a doua aplicație conține o instrucțiune repetitivă cu un parametru de control variat între 0 și 5 care include în corpul său o construcție switch/case, având același parametru de control pe post de variabilă de selecție (vezi figura A1.2). Prin compilarea codului sursă *high-level* rezultă în principal două instrucțiuni de salt condiționat, una de salt indirect și numeroase instrucțiuni de salt necondiționat. Pentru îmbunătățirea acurateții de predicție a saltului indirect (j \$2) se încearcă extinderea informației de corelație prin reținerea comportamentului ultimelor două salturi condiționate (HrgLength=2 – bne \$3, \$0, \$L30 și beq \$3, \$0, \$L38). Întrucât se folosește aceeași variabilă de control – *i* – stocată în memoria de date la adresa 20(\$fp), (vezi în figura A1.2 secvența cu *modificarea și stocarea variabilei de control i*), se observă foarte clar că cele două salturi condiționate sunt corelate. Astfel, dacă primul salt condiționat se face (T) atunci cel de-al doilea nu se va face (NT). Dacă variabila contor ia pe rând valorile 0, 1, 2, 3, și respectiv 4, atunci de fiecare dată când se ajunge la instrucțiunea de salt indirect registrul de istorie globală (globalHR – pe 2 biți) va avea tot valoarea (binară) 10₂ (vezi comportamentul celor două salturi condiționate din figura A1.2 – la fiecare iterație este reprezentat cu altă nuanță). Cu toate acestea, în fiecare din cele 5 cazuri target-ul instrucțiunii de salt indirect diferă. După cum se poate observa, în această situație nu ajută la nimic cunoașterea PC-urilor celor două salturi condiționate pentru îmbunătățirea acurateții predicției saltului indirect (la fiecare iterație fiind aceleași PC-uri ca și la anterioara execuție a saltului indirect, generându-se practic același *identificator de set* și respectiv *TAG* de acces în structura Target Cache, nereducându-se din miss-urile de conflict). Rezultă astfel o **limitare a avantajului introdus de tehnica de extindere a informației de corelație pentru pattern-uri de salturi condiționate de istorie redusă. Nici utilizarea unei structuri Target Cache cu grad ridicat de asociativitate nu constituie neapărat o soluție** în acest caz întrucât Tag-urile din set sunt toate distincte și instrucțiunile de salt condiționat și indirect generează același Tag de fiecare dată. Pentru aplicații *high-level* complexe care folosesc secvențe de instrucțiuni de genul

celor ilustrate mai sus (*for...*, *switch...*), este necesară păstrarea unei istorii „mai bogate” a comportamentului salturilor condiționate pentru o acuratețe ridicată a salturilor indirecte (concluzii reflectate de altfel și în subcapitolul – 7.1, cu rezultate ale simulărilor pe benchmark-urile SPEC’95).

```

#include <stdio.h>
void main(){
char c;
int i;
for(i=0;j<5;j++)
  switch (i){
    case 0:c='a'; break;
    case 1:c='b'; break;
    case 2:c='c'; break;
    case 3:c='d'; break;
    case 4:c='e'; break;
  }
}

file 1 "contra.cpp"
#GNU C++ 2.6.3 SimpleScalar running sstrich compiled by
GNU C
#Ccl defaults: -mgas -mgoPT
#Ccl arguments (-G value = 8, Cpu = default, ISA = 1):
# -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_cplusplus:
    .text
    .align 2
    .globl main
    .text
    .loc 1 3
    .ent main

main:
    .frame $fp,32,$31#vars=8,regs=20,args=16
    .mask 0xc0000000,-4
    .fmask 0x00000000,0
    subu $sp,$sp,32
    sw $31,28($sp)
    sw $fp,24($sp)
    move $fp,$sp
    jal __main
    sw $0,20($fp)
$L27:
    lw $2,20($fp)
    slt $3,$2,5
    move $2,$3
    andi $3,$2,0x00ff
    bne $3,$0,$L30      T | T | T | T | T | NT
    j $L28

$L30:
    lw $2,20($fp)
    sltu $3,$2,5
    beq $3,$0,$L38     NT | NT | NT | NT | NT | T
    lw $2,20($fp)
    move $3,$2
    sll $2,$3,2
    la $3,$L37

    addu $2,$2,$3
    move $3,$2
    lw $2,0($3)
    j $2
    rdata
    align 3
    align 2
$L37:
    .word $L32
    .word $L33
    .word $L34
    .word $L35
    .word $L36
    .text
$L32:
    li $2,0x00000061 #97
    sb $2,16($fp)
    j $L31
$L33:
    li $2,0x00000062 #98
    sb $2,16($fp)
    j $L31
$L34:
    li $2,0x00000063 #99
    sb $2,16($fp)
    j $L31
$L35:
    li $2,0x00000064 #100
    sb $2,16($fp)
    j $L31
$L36:
    li $2,0x00000065 #101
    sb $2,16($fp)
    j $L31
$L38:
$L31:
$L29:
    lw $3,20($fp)      # modificarea și
    addu $2,$3,1      # stocarea
    move $3,$2        # parametrului
    sw $3,20($fp)     # de control - i
    j $L27
$L28:
    move $2,$0
    j $L26
    j $L26
$L26:
    move $sp,$fp # sp not trusted here
    lw $31,28($sp)
    lw $fp,24($sp)
    addu $sp,$sp,32
    j $31
    .end main

```

Figura A1.2. Situație concretă ce dovedește ineficiența extinderii informației de corelație în predicția salturilor indirecte


```

PC = 56      bnez $t0, nu_sare
PC = 60      li $v0, 11
PC = 64      li $a0, 0x0a      // afișare pe rând nou
PC = 68      syscall

```

nu_sare:

```

PC = 72      lw $a0, ($sp)      //refacere număr prim
PC = 76      add $sp, $sp, 4    //refacere stivă
PC = 80      bnez $a1, RESTART

```

Cu toate că secvența aleasă nu este poate cel mai sugestiv exemplu și cu toate că instrucțiunile cu stiva puteau fi înlocuite de altele de transfer, s-a urmărit înțelegerea gradului ridicat de localitate existent pe regiștrii procesorului MIPS, și cum un predictor simplu de tip "*Last Value*" modificat (care reține 2 sau 4 valori ale fiecărui registru – vezi *Perceptronul Last-2 Value*) și cu mecanism "*perfect*" de selecție a valorii prezise poate exploata conceptul de localitate.

Metodologia de determinare a gradului de localitate pe regiștri constă în verificarea în fiecare fază *Write Back* dacă valoarea scrisă în respectivul registru se regăsește printre ultimele k valori anterior scrise. Predicția se realizează abia în faza ID (*decode*) după citirea valorii operanzilor din setul de regiștri generali.

Din exemplul prezentat se observă că în timpul execuției programului registrul v_0 ia doar două valori - coduri de apel sistem (1, respectiv 11), registrul t_0 păstrează valoarea constantă 25, sp ia una din valorile $0x7fffc000$ respectiv $0x7fffbffc$. Ceilalți regiștri își modifică valorile datorită instrucțiunilor incrementale/decrementale (reprezintă indecși de adresă, elemente distincte stocate în memorie - numere prime). Pe lângă valorile variabile pe care le ia din memorie registrul a_0 reține și două valori (32, respectiv 10) - parametri ai apelului sistem de afișare caracter.

ANEXA 3

LIMITAREA PREDICTIBILITĂȚII SALTURILOR PE ALGORITMI DE SORTARE RAPIDĂ

Anexa de față precum și următoarele două (anexele 4 și 5) reprezintă exemple concrete ce demonstrează necesitatea unor abordări integratoare de gen *hardware-software*, *tehnologie-arhitectură*, *algoritmi*, *concepte*, *metode*.

În ultimii ani, procesul de proiectare al procesoarelor s-a modificat radical. Astăzi, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsă legătură cu aplicațiile potențiale. Se pornește de la o arhitectură de bază, care este modificată și îmbunătățită dinamic, prin simulări laborioase pe *benchmark*-uri reprezentative (Stanford, SPEC).

Problematika *branch-urilor* în procesoarele pipeline superscalare reprezintă o provocare fundamentală în evoluția domeniului arhitecturii calculatoarelor. Instrucțiunile de ramificație acționează la nivelul control-flow generând pierderi de performanță prin necunoașterea la timp (în momentul fazei de aducere a instrucțiunilor) a direcției și adresei saltului. Reducerea efectelor defavorabile se poate face prin metode software bazate pe reorganizarea programului sursă (*scheduling*) sau prin metode hardware (predicția ramificațiilor de program care favorizează astfel execuția speculativă).

Unul din cele 8 *benchmark*-uri Stanford [Col93] îl constituie programul de sortare rapidă *Quicksort*. Într-unul din articolele despre predicția salturilor scris de Trevor Mudge în 1996 [Mud96], autorul spune și demonstrează, exemplificând pe *benchmark*-ul mai sus amintit, că predictibilitatea salturilor în unele programe poate fi analizată exact, furnizându-se o limită superioară a predictibilității. Pentru programe mai complexe, măsurarea limitei de predictibilitate se poate face utilizând algoritmul universal de compresie / predicție PPM (*predicție prin potrivire parțială*) [Cle84, Moff90]. Programele au o limită inerentă de predictibilitate datorată aleatorismului datelor de intrare. Această limită variază de la un set de date de intrare la altul.

Analiza efectuată se bazează pe algoritmul de sortare rapidă [Sed92], având ca obiect de interes instrucțiunile de salt condiționat. Algoritmul de sortare rapidă (Quicksort), primește ca date de intrare un șir de n elemente (presupuse distincte), și le ordonează crescător. Timpul de execuție în cazul cel mai defavorabil este $\theta(n^2)$ [Cor90]. Algoritmul este deseori cea mai bună soluție practică, deoarece are o comportare medie remarcabilă: timpul mediu de execuție este $\theta(n \lg n)$. *Sortarea se face pe loc* (în spațiul alocat șirului de intrare) și lucrează foarte bine chiar și într-un mediu de memorie virtuală. Se bazează pe paradigma “*divide și stăpânește*” [Hoa62, Cor90]. Pentru un subșir $A[p..r]$ rezultă operațiile:

⇒ **Divide:** Șirul $A[p..r]$ este împărțit în două subșiruri nevide $A[p..q]$ și $A[q+1..r]$, astfel încât fiecare element al subșirului $A[p..q]$ să fie mai mic sau egal cu orice element al subșirului $A[q+1..r]$. Indicele q este calculat de procedura de *partiționare*, de importanță primordială, pe baza unui element ales ca *pivot*.

⇒ **Stăpânește:** Cele două subșiruri $A[p..q]$ și $A[q+1..r]$ sunt sortate prin apeluri recursive ale algoritmului de sortare rapidă.

Din întreg algoritmul două instrucțiuni de salt fac obiectul analizei, și anume cele două bucle “*while*” în interiorul cărora se realizează comparația dintre elementele tabloului de sortat și valoarea pivotului.

```
while((array[++left_pointer]<pivot) && (left_pointer<=right));
```

```
while((array[--right_pointer]>pivot) && (right_pointer>=0));
```

Cele două salturi formează nucleul algoritmului și sunt dificil de prezis: rezultatul lor depinde substanțial de distribuția datelor de intrare. Celelalte branch-uri din program sunt 100% predictibile dacă sunt cunoscute date suficiente despre istoria anterioară a salturilor, sau există timp suficient de calcul.

Se consideră că cele n numere de sortat sunt distincte, deci ordinea inițială posibilă a elementelor este egal probabilă. Predictibilitatea căutată pentru un subșir variază în concordanță cu numărul de elemente al subșirului. La fiecare salt, predicția este următoarea: “*programul se va ramifica sau nu, în funcție de rezultatul comparației dintre noul element examinat și pivot*”. Astfel, dacă noul element examinat este mai probabil să fie mai mare / mai mic decât pivotul predicția este că saltul *se face / nu se face*.

Algoritmul optim de predicție păstrează (urmărește) - la fiecare pas - un număr variabil de rapoarte de elemente anterior examinate și compară această cantitate cu 1/2 pentru a decide cum va prezice noul salt. Justificarea

comparației cu $1/2$ este următoarea: dacă din totalul de elemente anterior examinate mai mult de jumătate ($1/2$) au fost mai mari decât pivotul (*s-a făcut saltul*), rezultă predicția va fi că saltul se va face (*și noul element va fi mai mare decât pivotul*). În plus, o partiționare optimă a tabloului de elemente se face exact la jumătate, ceea ce ar duce la un timp optim de execuție $O(n \lg n)$. Relația de recurență: $T(n) = 2 \cdot T(n/2) + \theta(n)$.

Această schemă de predicție este optimală, rata sa tinzând asimptotic crescător spre 75%, pentru un n foarte mare de elemente de intrare.

Demonstrație:

Se consideră $p = (\text{rangul pivotului}) / n$. **Rang pivot** (fie k) = câte elemente sunt mai mici decât pivotul. Rezultă p - uniform distribuit pe intervalul $(0, 1)$. Rezultă rata de succes a predicției se obține ca fiind $f(p) = \max(p, 1-p)$. Cel mult k elemente mai mici decât pivotul, rezultă rata maximă de predicție corectă în prima buclă "while" este $p = k/n$. Similar, în a doua buclă "while", există cel mult $n-k$ numere mai mari decât pivotul, rezultă predicția maximă corectă este: $1-p = (n-k)/n$. În concluzie, rata medie

de predicție este $A_p = \frac{1}{1-0} \int_0^1 f(p) dp = \int_0^{\frac{1}{2}} (1-p) dp + \int_{\frac{1}{2}}^1 p dp = 0.75$ (din teorema de

medie - integrarea funcțiilor continue).

Dacă se încearcă compresia istoriei salturilor pentru algoritmul Quicksort, fiecare simbol $(1/0)$ va ataca intrarea unui codificator aritmetic, care codifică în concordanță cu cea mai bună estimare a probabilității următorului simbol. Dacă $H(p) = -p \log_2(1/p) - (1-p) \log_2(1/(1-p))$ este funcția entropie binară, atunci compresia întregii istorii se face cu o rată egală cu

entropia medie, și anume $\int_0^1 H(p) dp = \frac{1}{2 \cdot \ln 2}$ biți per decizie (*prin compresie*

practic 1 bit nu mai presupune o decizie ci $2 \cdot \ln 2$ decizii). Pentru calculul entropiei medii sunt necesare noțiuni de analiză matematică elementară: limite de funcții în condiții de nedeterminare și integrale definite rezolvate cu metoda integrării prin părți. Cunoscând că, numărul total de posibilități de aranjare a celor n elemente este $n!$ (egal probabile) rezultă numărul total de biți necesari compresiei este $\log_2(n!) \approx n \cdot \log_2 n$. În concluzie, algoritmul de sortare rapidă Quicksort necesită aproape sigur $(2 \cdot \ln 2) \cdot n \cdot \log_2 n$ decizii în medie, ceea ce este în concordanță cu estimările de performanță anterior cunoscute [Hoa62, Sed92].

ANEXA 4

REDUCEREA COMPLEXITĂȚII UNOR STRUCTURI MICROARHITECTURALE PRIN DISPERSIA ADRESELOR

Termenul de "**dispersie**" evocă imaginea unei fărâmițări și amestecări aleatoare. Prin definiție *tabela de dispersie* este o structură eficientă de date pentru implementarea *dicționarelor*. În cazul cel mai defavorabil căutarea unui element într-o tabelă de dispersie poate necesita la fel de mult timp ca și căutarea unui element într-o listă înlănțuită - $O(n)$. În anumite ipoteze rezonabile, timpul necesar căutării unui element într-o tabelă de dispersie este $O(1)$. *Tabela de dispersie* reprezintă o alternativă eficientă la adresarea directă într-un tablou când numărul cheilor memorate efectiv este relativ mic față de numărul total de chei posibile.

Dificultatea în adresarea directă, dacă universul U este mare, constă în memorarea tabelului T , dată fiind limitarea resurselor a unui calculator uzual. Mai mult mulțimea K a cheilor efectiv memorate poate fi atât de mică relativ la U , încât majoritatea spațiului alocat pentru T ar fi irosit. Prin dispersie, un element având cheia k este memorat în locația $h(k)$. " h " se numește **funcție de dispersie** și este folosită pentru a calcula locația pe baza cheii k ; h transformă universul U al cheilor în locații ale unei table de dispersie $T[0..m-1]$. Funcția h trebuie să fie deterministă (intrarea dată k trebuie să producă întotdeauna aceeași ieșire $h(k)$). O funcție de dispersie bună satisface ipoteza dispersiei uniforme simple. Se disting trei tipuri de funcții: *dispersia prin diviziune*, *dispersia prin înmulțire* și *dispersia universală*.

Dezavantajul tabelelor de dispersie constă în faptul că două chei se pot dispersa în aceeași locație (*coliziune*). Există însă tehnici eficiente pentru rezolvarea conflictelor - evitarea sau minimizarea lor - prin alegerea potrivită a funcției de dispersie h .

O primă metodă o constituie **rezolvarea coliziunii prin înlănțuire**. Toate elementele ce se dispersează în aceeași locație sunt stocate într-o listă înlănțuită. Într-o tabelă de dispersie în care coliziunile sunt rezolvate prin

înlănțuire, o căutare *cu și fără succes* necesită, în medie, un timp $O(1+\alpha)$, în ipoteza **dispersiei uniforme simple** (dispersarea elementelor în oricare din cele m locații cu aceeași probabilitate, independent de locul în care s-au dispersat celelalte elemente), unde α este factorul de încărcare al tabelii, *supraunitar*.

O altă metodă o constituie **rezolvarea coliziunii prin adresare deschisă**. Prin adresare deschisă, toate elementele sunt memorate în interiorul tabelii de dispersie. Nu există liste sau elemente memorate în afara tabelii, așa cum se întâmplă în cazul înlănțuirii. **Avantajul** adresării deschise constă în evitarea folosirii pointerilor. Spațiul de memorie suplimentar, eliberat prin neutilizarea pointerilor, oferă tabelii de dispersie un număr mai mare de locații pentru același spațiu de memorie, putând rezulta coliziuni mai puține și acces mai rapid. Secvența de locații care se examinează nu se determină folosind pointerii, ci se calculează. *Inserarea* într-o tabelă de dispersie cu adresare deschisă, se face verificând tabelă până când se găsește o locație liberă în care se poate memora cheia. În loc să fie fixat în ordinea 0, 1, ..., $m-1$ (care necesită un timp de căutare $O(n)$), șirul de poziții examinate depinde de cheia ce se inserează.

Funcția de dispersie produce nu doar un singur număr, ci o întreagă secvență de verificare; dispersia uniformă reală este dificil de implementat și în practică sunt folosite aproximări convenabile (dispersia dublă). Se disting trei tehnici pentru calcularea secvențelor de verificare necesare adresării deschise: *verificarea liniară*, *verificarea pătratică* și *dispersia dublă*. Nici una dintre aceste tehnici nu satisface condiția dispersiei uniforme, deoarece nici una dintre ele nu e capabilă să genereze mai mult de m^2 secvențe de verificare diferite (în loc de $m!$ cât cere dispersia uniformă). Dispersia dublă are cel mai mare număr de secvențe de verificare și dă cele mai bune rezultate.

Prin adresare deschisă, factorul de încărcare al tabelii (α) este *subunitar*. Într-o astfel de tabelă de dispersie, în ipoteza dispersiei uniforme și presupunând că fiecare cheie din tabelă este căutată cu aceeași probabilitate, numărul mediu de verificări într-o căutare cu succes este cel

$$\text{mult: } \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}.$$

În continuare se prezintă două aplicații practice ale tabelilor de dispersie în domeniul arhitecturii calculatoarelor. Pentru o mai bună înțelegere se realizează și un scurt memento teoretic al celor două subdomenii alese: **organizarea sistemului ierarhic de memorie** în vederea reducerii decalajului tehnologic dintre acesta și procesoarele avansate și **creșterea acurateții predicției ramificațiilor de program**.

Accentuarea decalajului dintre viteza procesorului și cea a nivelului inferior de memorie se datorează următorilor factori: (i) perioada de tact al procesorului a scăzut mult mai rapid decât timpul de acces la memorie (DRAM) și (ii) tehnicile arhitecturale de proiectare pentru procesoarele pipeline superscalare au cauzat o reducere dramatică a numărului mediu de cicluri per instrucțiune (CPI). Reducerea acestui decalaj se poate realiza prin organizarea memoriei sistem într-o ierarhie, cu un cache foarte rapid de tip SRAM apropiat de CPU și o memorie principală de tip DRAM, aflată pe un nivel inferior.

Pentru a reduce rata de miss a cache-urilor mapate direct (fără să se afeceteze însă timpul de hit sau penalitatea în caz de miss), cercetătorul Norman Jouppi a propus conceptul de **victim cache** - o memorie mică complet asociativă, plasată între primul nivel de cache mapat direct și memoria principală. Blocurile înlocuite din cache-ul principal datorită unui miss sunt temporar memorate în victim cache. Dacă sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mică decât cea a memoriei principale [Jou90].

Schema este însă mai puțin eficace pentru blocuri mari, de capacități tipice cache-urilor microprocesoarelor curente. Pentru a minimiza numărul de interschimbări dintre cache-ul principal, mapat direct, și victim cache, Stiliadis și Varma au introdus un nou concept numit **selective victim cache (SVC)** [Sti94]. Mecanismul de predicție al SVC, bazat pe istoria folosirii blocurilor, stabilește dacă blocurile aduse din memoria principală sunt plasate în cache-ul principal sau în selective victim cache. Ideea principală a algoritmului de predicție este de a plasa blocurile, care sunt mai puțin probabil să fie accesate în viitor, în SVC iar cele mai probabil de a fi referite în cache-ul principal. Predicția este folosită și în cazul unui miss în cache-ul principal pentru a determina dacă este necesară o interschimbare a blocurilor conflictuale. Selective victim cache reduce rata de miss cu până la 50%, în funcție de dimensiunea blocului din cache. De asemenea, numărul de interschimbări este redus cu 50% sau mai mult față de folosirea unui victim cache simplu [Sti94, Flo00].

Algoritmul de predicție folosește doi biți de stare asociați fiecărui bloc, numiți **hit bit** și **sticky bit**. Bitul hit este asociat logic cu fiecare bloc din memoria principală și conține informații despre istoria blocurilor din cache. Într-o implementare perfectă, biții de hit trebuie memorați în nivelul L2 de cache sau în memoria principală și aduși în nivelul L1 de cache cu blocul corespondent. Această abordare este impracticabilă în majoritatea cazurilor. Bitul este setat doar dacă ultima dată când s-a aflat în cache-ul principal a fost accesat. În cazul în care blocul a fost în cache dar nu a fost niciodată accesat atunci bitul *hit* este 0. Dacă blocul este înlocuit din cache-

ul principal (L1 cache), starea bitului de hit trebuie actualizată în L2 cache sau în memoria centrală.

Bitul sticky este asociat logic cu blocul din cache-ul principal. De aceea este normal să se memoreze acest bit în cache-ul mapat direct ca parte a fiecărui bloc. Este setat când un bloc este adus pentru prima oară în cache-ul principal. La referirea unui bloc conflictual, dacă algoritmul de predicție decide ca blocul să nu fie înlocuit din cache-ul principal atunci bitul sticky este resetat. Dacă un acces ulterior în cache-ul principal intră în conflict cu blocul care are bitul sticky resetat, atunci blocul va fi înlocuit din cache-ul principal.

Pentru o mai bună reflectare a istoriei blocurilor în [Flo00, Vin00a] se generalizează algoritmul propus de Stiliadis, prin considerarea unor vectori binari în locul biților de stare *hit* și *sticky* și modificarea corespunzătoare a algoritmului de predicție. Astfel, vectorul *hit* va indica mai gradual rata de utilizare a blocului din memoria centrală la ultima sa apariție în cache-ul principal, în timp ce vectorul *sticky* exprimă gradul de conflictualitate al unui bloc din cache-ul principal, obținându-se performanțe superioare ale conceptului de SVC. Astfel s-a arătat că 2 biți de "hit" respectiv "sticky", conduc la rezultate optime în condiții fezabile ale implementării hardware.

Dacă ierarhia de memorie include un la doilea nivel de cache, este posibil să se memoreze biții de hit în cadrul blocurilor din acest nivel. Când un bloc este adus pe nivelul L1 de cache din nivelul L2, o copie locală a bitului de hit este memorată în blocul de pe nivelul L1. Aceasta elimină nevoia de acces a nivelului L2 de cache de fiecare dată când bitul hit este actualizat de către algoritmul de predicție. Când blocul este înlocuit din nivelul L1 de cache, bitul hit corespondent este copiat în nivelul L2. O problemă ar fi însă aceea că, multiple locații din memoria principală sunt forțate să împartă același bit de pe nivelul L2. Astfel, când un bloc este înlocuit de pe nivelul L2 de cache, toate informațiile lui de stare se pierd, reducând eficacitatea algoritmului de predicție. De fiecare dată când un bloc este adus pe nivelul L2 de cache din memoria principală, bitul hit al său trebuie setat la o valoare inițială. Pentru o secvență specifică de acces, valori inițiale diferite pot produce rezultate diferite. Cu cache-urile de pe nivelul L2 de dimensiuni mari, efectul valorilor inițiale este probabil mai mic.

O tratare alternativă este de a menține biții de hit în interiorul CPU, în cadrul nivelului L1 de cache. În [Sti94] se propun anumite implementări "aproximative", mai realiste, în sensul reducerii necesităților de memorare pentru biții de hit. Astfel de exemplu, se propune implementarea unui "*hit array*", ca parte a memoriei cache principale (L1) și care permite printr-un mecanism tip "mapare directă", mai multor biți de hit corespunzători blocurilor memoriei principale să fie mapați în aceeași locație a acestui "hit

array". Lungimea șirului este inevitabil mai mică decât numărul maxim de blocuri care pot fi adresate. Deci, mai mult de un bloc va fi mapat aceluiași bit de hit, cauzând datorită interferențelor un aleatorism ce trebuie introdus în procesul de predicție. Cu alte cuvinte se realizează o mapare a biților de hit teoretic rezidenți în memoria principală, într-un cache dedicat, cu rezultate obținute prin simulări, foarte apropiate de cele generate printr-o implementare ideală. Utilizarea unei tabele de dispersie cu rezolvarea coliziunilor prin înlănțuire sau adresare deschisă cu dispersie dublă conduce la obținerea unei performanțe similare cu situația ideală (câte un bit de hit pentru fiecare bloc din memoria principală). Tabelul A4.1 cuprinde rezultatul unor astfel de simulări [Flo00].

Implementare	ICache Usage	IVictim Usage	ICache interchg	IHIT %	DCache Usage	DVictim Usage	Dcache interchg	DHIT %	IR
Ideal	100%	95.31%	2928	94.79	73.14%	80.86%	1265	90.26	0.85
Hashing	100%	95.31%	2876	94.78	73.14%	80.86%	1374	90.12	0.85

Tabelul A4.1. Studiu comparativ privind o implementare ideală vs. o tabelă de dispersie a biților hit

Implementarea nivelului L1 de cache sistem este prezentată în figura A4.1. Bitul *sticky* este menținut cu fiecare bloc în cache-ul principal. Nici un bit de stare nu este necesar în *victim cache*. Biții de hit sunt păstrați în *hit array*, adresați de o parte a adresei de memorie. Dimensiunea șirului de biți de hit este aleasă ca un multiplu al numărului de blocuri din cache-ul principal. Astfel, avem relația:

$$\text{Dimensiunea șirului hit array} = \text{Număr de blocuri în cache-ul principal} \times H$$

unde H determină gradul de partajare a biților de hit de către blocurile memoriei principale. Se presupune că H este o putere a lui 2, $H=2^h$. *Hit array* poate fi adresat de adresa de bloc concatenată cu cei mai puțin semnificativi biți *h*, din partea de tag a adresei. O valoare mare pentru H implică mai puține interferențe între blocurile conflictuale la biții de hit. Dacă H este ales ca raport dintre dimensiunea cache-ului de pe nivelul L2 și cea a cache-ului de pe nivelul L1 (principal), atunci efectul este similar cu menținerea biților de hit în nivelul L2 de cache.

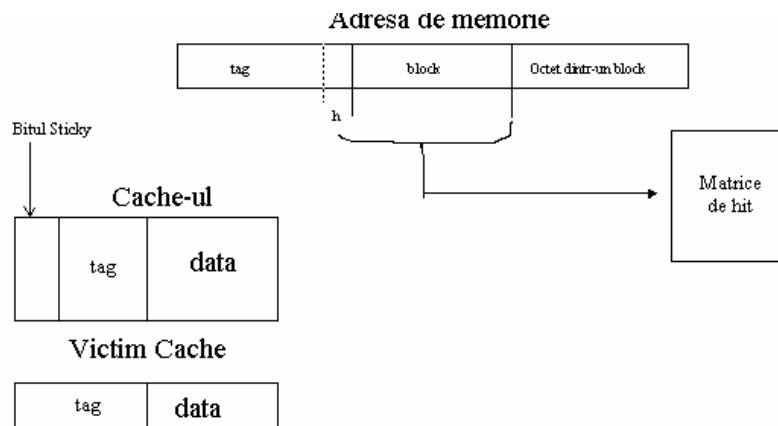


Figura A4.1. Implementarea schemei de memorare a biților de hit

Al doilea exemplu de aplicabilitate al tabelor de dispersie pe care îl prezint în această anexă îl constituie schemele de predicție hardware. S-a demonstrat că istoria salturilor este insuficientă pentru o mai bună corelație și în consecință pentru o acuratețe ridicată [Vin99]. În condițiile dezvoltării unor arhitecturi de procesoare pipeline și cu paralelism pronunțat la nivelul instrucțiunii, necesitatea unui predictor hardware de salturi eficient devine esențială. Pentru a nu se "simți" efectul defavorabil al ramificațiilor de program asupra performanței procesoarelor avansate acuratețea de predicție trebuie să fie foarte apropiată de 100%. În acest scop, Vințan [Vin99] a propus o nouă schemă de predicție adaptivă pe două nivele cu rezultate mai bune decât schema clasică GAP, în aceleași condiții de cost și complexitate hardware.

Este posibilă îmbunătățirea acurateții predicției dacă aceasta se va baza pe comportarea recentă a altor instrucțiuni de salt, întrucât frecvent aceste instrucțiuni pot avea o comportare corelată în cadrul programului. Schemele bazate pe această observație se numesc **scheme de predicție corelată** [Yeh92] (vezi figura A4.2). Există în implementare 2 niveluri: un **registru de predicție (HRg)** pe k biți (cuprinde "istoria" celor mai recent executate k salturi din program sau comportamentul ultimelor k apariții ale aceleiași instrucțiuni de salt) al cărui conținut concatenat cu cei mai puțini semnificativi biți ai PC-ului instrucțiunii de salt pointează la un cuvânt din **tabela de predicții** (aceasta conține automatul de predicție - de regulă un numărător saturat, adresa destinație, etc).

Schemele corelate de predicție adaptivă pe două niveluri determină acurateți medii de 97.7%, măsurate pe 9 din 10 benchmark-uri SPEC'95! Însă ținând cont că, de la o acuratețe de predicție de 100% la una de 97.7%, pentru un procesor superscalar care trimite simultan spre execuție 4

instrucțiuni, rata de procesare scade cu 40% de la $IR=4$ la $IR=4/1.4=2.8$ instr./ciclu, se poate spune că o acuratețe de 97.7% nu este suficientă.

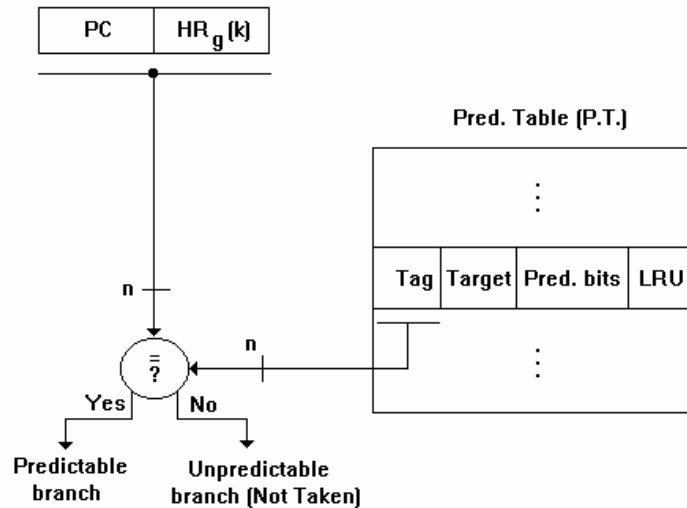


Figure A4.2. Schema GAP complet asociativă

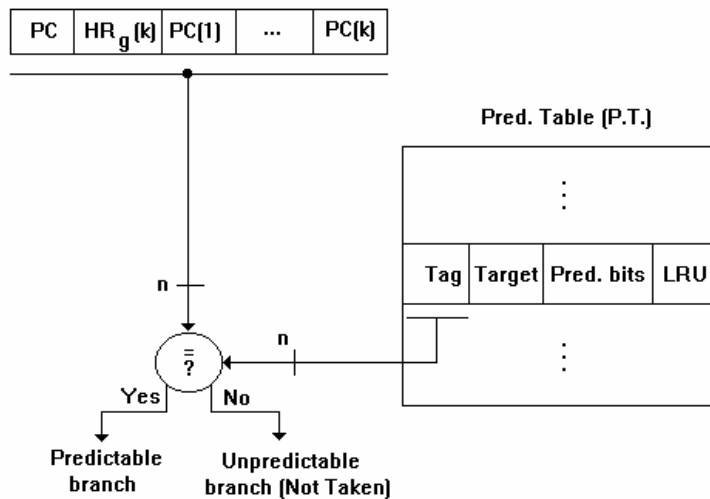


Figure A4.3. Schema GAP complet asociativă modificată (MGAp)

Se cunoaște că este dificil de predicționat corect un salt care are un comportament aleator în același context de predicție (HR_g , HR_l). Dacă însă, fiecare bit din cei k aparținând HR_g este asociat în procesul de predicție cu PC-ul corespondent, informația de corelație ar fi mai completă iar acuratețea

de predicție mai ridicată. Astfel, se va ști nu numai dacă ultimele k salturi s-au făcut sau nu ci și care anume, conform adreselor PC_1, PC_2, \dots, PC_k . În locul utilizării doar a HRg, contextul de predicție devine mai complet și mai complex, cuprinzând pe lângă HRg și etichetele salturilor PC_1, PC_2, \dots, PC_k (vezi figura A4.3). Astfel, considerând HRg pe k biți, și lungimea PC pe minim 8 biți (suficientă pentru benchmark-urile Stanford [Col93]), atunci lungimea corespondentă a tag-ului în tabela PT este $n=9 \cdot k$. PT este complet asociativă cu algoritmul de înlocuire MPP (potențialul de performanță minim) - produsul dintre LRU (probabilitatea ca respectivul salt să fie referit din nou) și probabilitatea ca saltul să fie *taken* (derivat din automatul de predicție).

Întrucât dimensiunea tabelului PT fiind redusă comparativ cu contextul bogat de predicție rezultă un număr semnificativ de înlocuiri, cu o influență negativă asupra acurateții de predicție (vezi tabelul A4.2).

<u>Bench</u>	<u>HRg</u>	<u>Br.no.</u>	<u>Prediction Accuracv</u>	<u>Incorrect predictions</u>	<u>Bad target</u>	<u>NT branches</u>	<u>No replaces</u>
sort	9	12601	74.92%	23.24%	1.83%	35.03%	0
sort	18	12601	71.67%	26.39%	1.94%	35.03%	862
sort	27	12601	68.17%	30.55%	1.29%	35.03%	2493
sort	36	12601	65.34%	33.94%	0.71%	35.03%	3380
sort	45	12601	62.97%	36.66%	0.37%	35.03%	3989
queens	9	38462	79.33%	20.62%	0.05%	49.87%	0
queens	18	38462	80.79%	19.16%	0.05%	49.87%	0
queens	27	38462	80.79%	19.16%	0.05%	49.87%	158
queens	36	38462	75.56%	24.42%	0.03%	49.87%	4612
queens	45	38462	68.95%	31.02%	0.03%	49.87%	8709
tree	9	32887	85.51%	10.67%	3.82%	26.52%	0
tree	18	32887	85.71%	10.57%	3.72%	26.52%	0
tree	27	32887	85.80%	10.79%	3.42%	26.52%	37
tree	36	32887	84.90%	11.79%	3.31%	26.52%	548
tree	45	32887	79.11%	17.78%	3.11%	26.52%	3309
puzzle	9	204527	94.65%	5.35%	0.00%	9.08%	13
puzzle	18	204527	94.45%	5.55%	0.00%	9.08%	3322
puzzle	27	204527	92.92%	7.08%	0.00%	9.08%	7349
puzzle	36	204527	91.63%	8.37%	0.00%	9.08%	10576
puzzle	45	204527	90.59%	9.41%	0.00%	9.08%	13074

Tabelul A4.2. Schema MGA_p - 100 intrări (4 din 8 benchmark-uri Stanford)

Rezultatele sunt contradictorii. În primul rând, un context "bogat" de predicție (k - valoare mare) implică o îmbunătățire a performanței (acurateții de predicție) deoarece fiecare context are asociat în PT propriul automat de

predicție. Pe de altă parte, un context "bogat" implică o creștere a numărului de înlocuiri, acuratețea de predicție diminuându-se datorită interferențelor. În acest sens, se impune realizarea unui compromis între cele două aspecte, materializat prin utilizarea unei tabele de dispersie cu rezolvarea coliziunilor prin înlănțuire sau adresare deschisă, care să comprime prin *hashing* o parte din informația de predicție minimizând astfel efectele defavorabile ale interferențelor. De exemplu se poate înlocui tabela de predicție (PT) asociativă cu una mapată direct. Fiecare locație din PT (la fiecare index) reprezintă o listă simplu înlănțuită de structuri de genul <**Tag, Target, Automat Predicție**>. În cazul unui conflict nu eliminăm ci inserăm în lista respectivă noua informație - context, predicție, adresă. La *hit*, se actualizează doar automatul de predicție. O soluție alternativă ar putea consta în comprimarea contextului de predicție printr-o funcție de tipul SAU EXCLUSIV. Astfel indexarea structurii PT s-ar face cu următoarea informație: $PC \text{ xor } PC_1 \text{ xor } PC_2 \text{ xor } \dots \text{ xor } PC_k$ concatenată sau comprimată cu registrul de istorie globală ($HR_g(k)$).

ANEXA 5

OPTIMIZAREA COLIZIUNILOR ÎN STRUCTURILE PIPELINE

Algoritmii *greedy* sunt aplicați în rezolvarea problemelor de optimizare; sunt compuși dintr-o secvență de pași, la fiecare pas existând mai multe alegeri posibile; pot fi priviți ca o particularizare a tehnicii *backtracking* în care se renunță la mecanismul de întoarcere. Algoritmii greedy aleg la fiecare moment de timp soluția ce pare a fi cea mai bună la momentul respectiv: o alegere optimă, făcută local, cu speranța că va conduce la un optim global. Cu toate acestea nu întotdeauna conduc la soluția optimă. Timpul de calcul este polinomial (de cele mai multe ori este necesară o sortare descrescătoare a datelor de intrare în funcție de prioritatea sau ponderea acestora la soluția globală - optimă). Metoda greedy este destul de puternică și se aplică cu succes unui spectru larg de probleme de optimizare [Cor90]:

- ⇒ Problema simplă dar netrivială a *selectării activităților*.
- ⇒ *Proiectarea unor coduri pentru compactarea datelor* - codurile Huffman.
- ⇒ Algoritmii de determinare a *arborelui parțial de cost minim*.
- ⇒ *Algoritmul lui Dijkstra pentru determinarea celor mai scurte drumuri pornind dintr-un vârf*.
- ⇒ Algoritmii greedy produc întotdeauna soluție optimă pentru "*matroizi*" (structuri combinatoriale).
- ⇒ Matroizii sunt utilizați în rezolvarea problemei *planificării sarcinilor de timp unitar*, sarcini având anumiți termeni limită de realizare și penalizări în caz de neîndeplinire.

Elemente ale strategiei *greedy*

În general, nu există o modalitate de stabilire a faptului că un algoritm greedy poate rezolva o anumită problemă particulară, dar există două caracteristici pe care le au majoritatea problemelor care se rezolvă prin tehnici greedy:

⇒ **Proprietatea de alegere greedy** (alegerile optime local conduc într-adevăr la o soluție optimă global).

- Realizează diferența dintre algoritmi greedy și programarea dinamică. La rezolvarea problemelor prin metoda programării dinamice, alegerea făcută la fiecare pas al algoritmului depinde de soluțiile subproblemelor. Într-un algoritm greedy alegerea făcută reprezintă soluția cea mai bună la momentul respectiv, subproblema rezultată fiind rezolvată după ce alegerea este făcută. Alegerea făcută de un algoritm greedy poate depinde de alegerile făcute anterior, dar nu poate depinde de alegerile viitoare sau soluțiile subproblemelor.
- Programarea dinamică rezolvă subproblemele în manieră "*bottom - up*" iar strategia greedy progresaază în maniera "*top - down*", realizând alegeri greedy succesive, reducând iterativ dimensiunea problemei inițiale.
- Proprietatea de alegere greedy se demonstrează astfel:
 - *Se examinează o soluție optimă local.*
 - *Se arată că soluția poate fi modificată astfel încât la fiecare pas este realizată o alegere greedy, iar această alegere reduce problema la una similară dar de dimensiuni mai reduse.*
 - *Se aplică principiul inducției matematice pentru a arăta că o alegere greedy poate fi utilizată la fiecare pas. Întrucât o alegere greedy conduce la o problemă de dimensiuni mai mici reduce demonstrația corectitudinii la demonstrarea faptului că o soluție optimă trebuie să evedențieze o substructură optimă.*

⇒ **Substructura optimă** (dacă o soluție optimă a problemei conține soluții optime ale subproblemelor).

- Proprietatea este exploatată atât de metoda greedy cât și de cea a programării dinamice.

Întrucât algoritmi greedy sunt aplicați în probleme de optimizare, o conexiune cu domeniul arhitecturii calculatoarelor o constituie problema gestionării unei structuri pipeline date, caracterizată de un vector de coliziune astfel încât să se obțină o rată de procesare maximă [Vin00]. Problema apare destul de des în realitate și se datorează hazardurilor structurale, generate de insuficiența resurselor hardware.

Cea mai importantă caracteristică arhitecturală a microprocesoarelor RISC scalare o constituie **procesarea pipeline** a instrucțiunilor și datelor. Aproape toate celelalte caracteristici arhitecturale ale procesoarelor RISC au scopul de a adapta structura acestora la procesarea pipeline. Este binecunoscut faptul că **tehnica de procesare pipeline** reprezintă o tehnică de procesare paralelă a informației prin care un proces secvențial este

divizat în subproces, fiecare subproces fiind executat într-un segment special dedicat și care operează în paralel cu celelalte segmente. Fiecare segment execută o procesare parțială a informației. Rezultatul obținut în segmentul i este transmis în tactul următor spre procesare segmentului $(i+1)$. Rezultatul final este obținut numai după ce informația a parcurs toate segmentele, la ieșirea ultimului segment. Diversele procese se pot afla în diferite faze de prelucrare în cadrul diverselor segmente, simultan. Suprapunerea procesărilor este posibilă prin asocierea unui registru de încărcare (latch) fiecărui segment din pipeline. Registrele produc o separare între segmente astfel încât fiecare segment să poată prelucra date separate.

Există însă și evenimente nedorite (**hazarduri**), care pot apare în procesarea pipeline și care duc la stagnarea procesării, având o influență negativă asupra ratei de execuție a instrucțiunilor. Conform unei clasificări consacrate aceste hazarduri sunt de 3 categorii : **hazarduri structurale, de date și de ramificație**.

Hazarduri structurale (HS) sunt determinate de conflictele la resurse comune (atunci când mai multe procese simultane aferente mai multor instrucțiuni în curs de procesare, accesează o resursă comună). Pentru a le elimina prin hardware, se impune de obicei multiplicarea acestor resurse.

Exemple:

- ✓ Un procesor care are un set de regiștri generali de tip uniport și în anumite situații există posibilitatea ca 2 procese să dorească să scrie în acest set simultan.
- ✓ O situație similară poate apare dacă se încearcă accesul simultan la memorie a 2 procese distincte: unul de aducere a instrucțiunii (IF), iar celălalt de aducere a operandului sau scriere a rezultatului în cazul unei instrucțiuni LOAD / STORE (nivelul MEM). Această situație se rezolvă în general printr-o arhitectură Harvard a busurilor și cache-urilor (busuri și spații de memorie separate pentru instrucțiuni și date). Se consideră în continuare, o structură pipeline cu 5 nivele având timpul de "setup" de 7 cicli, a cărei funcționare este descrisă în tabelul A5.1.

Ciclu/ Nivel	T1	T2	T3	T4	T5	T6	T7
N1	X						
N2		X	X				
N3			X	X			
N4					X	X	
N5							X

Tabelul A5.1. Descrierea funcționării unei structuri pipeline cu 5 nivele

Un X în tabel semnifică faptul că în ciclul T_i nivelul N_j este activ adică procesează informații. Se consideră că această structură pipeline corespunde unui anumit proces (procesarea unei instrucțiuni). Se observă că un alt proces de acest tip, nu poate starta în ciclul următor (T_2) datorită coliziunilor care ar putea să apară între cele două procese pe nivelul (N_2 , T_3) și respectiv (N_3 , T_4). Mai mult, următorul proces n-ar putea starta nici măcar în T_3 din motive similare de coliziune cu procesul anterior în nivelul (N_4 , T_6). În schimb procesul următor ar putea starta în T_4 fără a produce coliziuni sau hazarduri structurale cum le-am denumit, deci la 2 cicli după startarea procesului curent.

Se definește **vectorul de coliziune** al unei structuri pipeline având timpul de setup de $(N+1)$ cicli, un vector binar pe N biți astfel: dacă bitul i , $i \in \{1, \dots, N\}$ e 1 logic atunci procesul următor nu poate fi startat după i cicli de la startarea procesului curent, iar dacă bitul i este 0 logic, atunci procesul următor poate fi startat după i cicli fără a produce coliziuni cu procesul curent. Se observă pentru structura pipeline anterioară că vectorul de coliziune este 110000, însemnând deci că procesul următor nu trebuie startat în momentele T_2 sau T_3 , în schimb poate fi startat fără probleme oricând după aceea.

Se pune problema: *cum trebuie gestionată o structură pipeline dată, caracterizată printr-un anumit vector de coliziune, astfel încât să se obțină o rată de procesare (proces / ciclu) maximă?* Întrucât este o problemă de optim, este posibilă o soluție alegând o **strategie greedy**.

Considerând o structură pipeline cu timpul de "setup" de 8 cicli și având vectorul de coliziune 1001011 ar trebui procedat ca în figurile următoare (vezi figurile A5.1 și A5.2):

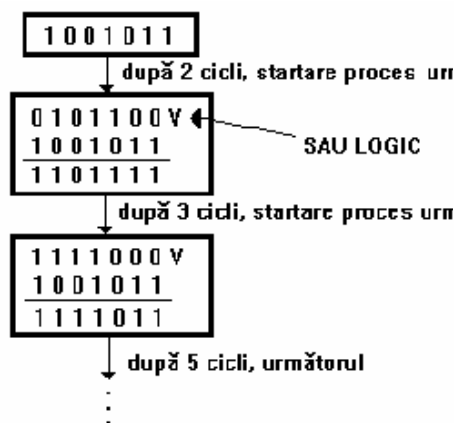


Figura A5.1 Principiul de lansare procese într-o structură pipeline cu hazarduri structurale

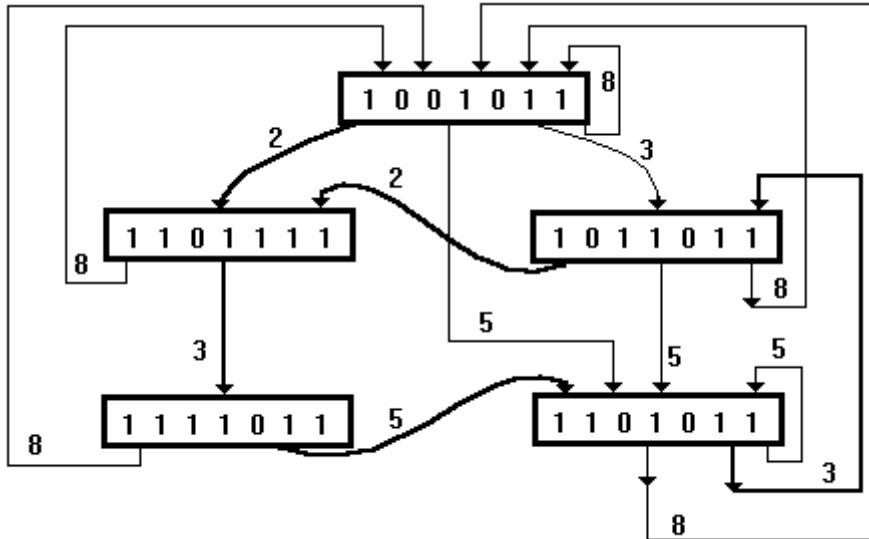


Figura A5.2. Graful de optimizare într-o structură pipeline

Vectorul unei stări S_j , se obține după relația:

$$V_j = (VC) \vee (V_i^*(m)) ,$$

V = SAU logic

VC = vector coliziune

$V_i^*(m)$ = vectorul V_i deplasat logic la stânga cu (m) poziții binare

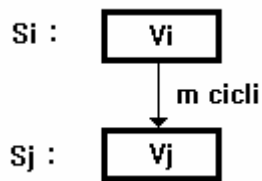


Figura A5.3. Trecerea dintr-o stare în alta

Pentru **maximizarea performanței**, se pune problema ca pornind din starea inițială a grafului, să se determine un drum închis în graf cu proprietatea ca NS / L să fie maxim, unde NS = numărul de stări al drumului iar L = lungimea drumului. În cazul anterior considerat, $L = 3 + 5 + 3 + 2 = 13$, iar $NS = 4$. Printr-o astfel de gestionare a structurii se evită coliziunile și se obține o performanță optimă de 4 procese în 13 cicluri, adică 0.31 procese / ciclu. De menționat că o structură convențională ar procesa

doar 0.125 procese / ciclu. Nu întotdeauna startarea procesului următor imediat ce acest lucru devine posibil ("**greedy strategy**"), conduce la o performanță maximă. Un exemplu în acest sens ar fi o structură pipeline cu vectorul de coliziune asociat 01011 (vezi figura A5.4). E adevărat însă că o asemenea strategie conduce la dezvoltarea unui algoritm mai simplu. Performanța maximă a unei structuri pipeline se obține numai în ipoteza alimentării ritmice cu date de intrare. În caz contrar, gestiunea structurii se va face pe un drum diferit de cel optim în graful vectorilor de coliziune.

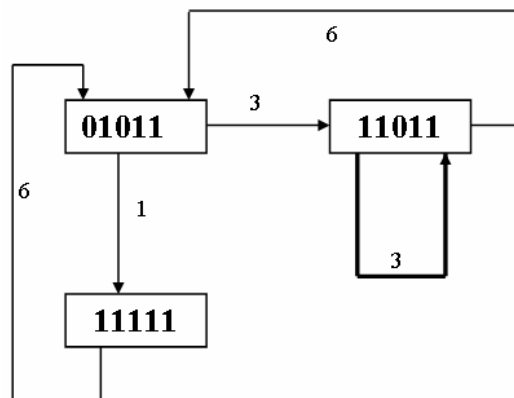


Figura A5.4. Exemplu ce arată că nu întotdeauna strategia *greedy* determină performanța maximă

După cum se observă într-o strategie *greedy* performanța obținută este de $2/7=0.286$ procese / ciclu în timp ce (vezi arcu îngroșat din figura A5.4) o strategie *non-greedy* conduce la o performanță mai bună (0.333 procese / ciclu).

Se consideră în continuare o structură pipeline bifuncțională capabilă să execute 2 tipuri de procese: P1 și respectiv P2. Aceste procese sunt descrise prin tabele adecvate, în care se arată ce nivele solicită procesul în fiecare ciclu. Este clar că aceste procese vor fi mai dificil de controlat. Pentru controlul acestora prin structură, este necesar a fi determinați mai întâi vectorii de coliziune și anume: $VC(P1, P1)$, $VC(P1, P2)$, $VC(P2, P1)$ și $VC(P2, P2)$, unde $VC(P_i, P_j)$ reprezintă vectorul de coliziune între procesul curent P_i și procesul următor P_j . Odată determinați acești vectori în baza tabelor de descriere aferente celor două procese controlul structurii s-ar putea face prin schema de principiu din figura A5.5.

Inițial registrul "P1 control" conține $VC(P1, P1)$, iar registrul "P2 control" conține $VC(P2, P2)$. Procesul următor care se dorește a fi startat în structură va face o cerere de startare către unitatea de control. Cererea va fi

acceptată sau respinsă după cum bitul cel mai semnificativ al registrului de control este 0 sau 1 respectiv. După fiecare iterație, registrul de control se va deplasa logic cu o poziție la stânga după care se execută un SAU logic între conținutul registrului de control, căile (A) și (B) respectiv căile de date (C) și (D), cu înscrierea rezultatului în registrul de control. Se observă în acest caz că *gestionarea proceselor se face după o strategie de tip "greedy"*.

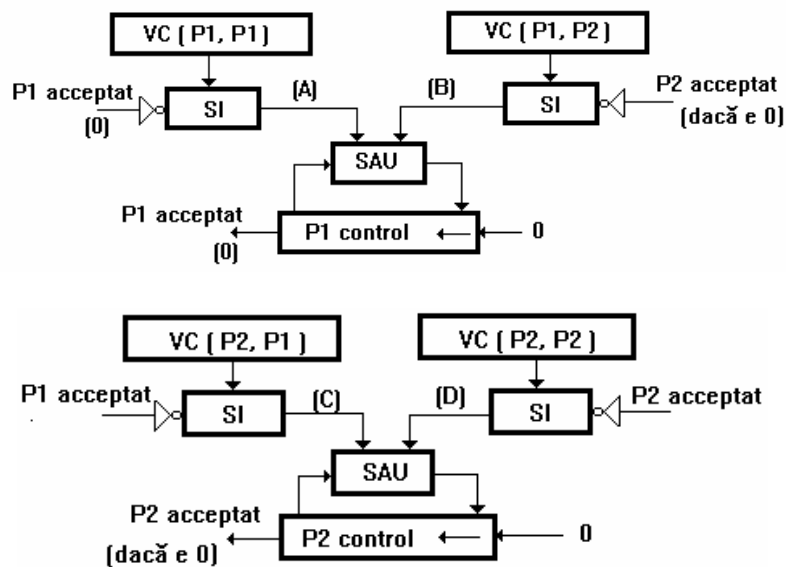


Figura A5.5. Controlul unei structuri pipeline bifuncționale