UNIVERSITATEA "*POLITEHNICA*" TIMIŞOARA
FAC. AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

CONTRIBUŢII LA OPTIMIZAREA ARHITECTURILOR CU PARALELISM LA NIVELUL INSTRUCŢIUNILOR

TEZĂ DE DOCTORAT

Conducător ştiinţific:
Prof. dr. ing. CRIŞAN STRUGARU

Doctorand:
ş. l. ing. LUCIAN N. VINŢAN

TIMIŞOARA
1997

CUPRINS

1. INTRODUCERE. ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCȚIUNILOR

Această lucrare abordează probleme aparținând arhitecturilor de calcul care exploatează paralelismul existent la nivelul instrucțiunilor mașină într-un program.(*Instruction Level Parallelism*- ILP). Aceste arhitecturi sunt caracterizate de o procesare agresivă, neconvențională, a instrucțiunilor mașină pe baza paralelismului realizat atât la nivel temporal (pipeline) cât și la nivel "spațial" (unități funcționale multiple).

O remarcă importantă ar fi aceea că domeniul este deosebit de fecund și deci cu un caracter mai degrabă evoluționist, constructiv, decât revoluționar [Hen96]. Această afirmație este justificată și de succesele comerciale deosebite, implementate în special în cadrul microprocesoarelor avansate actuale, într-un timp extrem de scurt de la apariția acestor concepte la nivel de cercetare.

Din punct de vedere istoric, primele procesoare neconvenționale au fost cele din dotarea supercomputerelor CDC-6600 (1964), considerat a fi primul calculator pipeline-vectorial viabil, și respectiv IBM-360/91 (1966), considerat ca fiind precursorul calculatoarelor cu execuție multiplă a instrucțiunilor. Așadar, CDC-6600 era caracterizat de o execuție paralelă a instrucțiunilor la nivel temporal, iar IBM-360/91 de o procesare paralelă atât din punct de vedere temporal cât și spațial, prin exploatarea inovatoare, în paralel, a unor unități funcționale multiple [Pat82, Tom67].

În anul 1981 s-a anunțat la Universitatea Berkeley, S.U.A., realizarea primului microprocesor VLSI (*Very Large Scale Integration*), bazat pe procesarea pipeline a instrucțiunilor și cu set optimizat de instrucțiuni în vederea facilizării implementării limbajului C, microprocesor numit Berkeley I RISC [Pat82]. Acronimul **RISC** (*Reduced Instruction Set Computer)* a fost propus de către *David Patterson,* coordonatorul proiectului. După părerea autorului acestei lucrări, la ora actuală, termenul de RISC este inexact înțeles "add literam", mai potrivit din punct de vedere semantic ar fi cel de procesor scalar pipeline cu set optimizat de instrucțiuni.

Multe din ideile arhitecturii RISC se datorează, după cum chiar creatorii arată, pionierilor care au proiectat supercomputerul CDC-6600 (*Seymour Cray*). Totuși s-au adus și idei fundamental noi, precum cele legate de optimizarea setului de instrucțiuni în vederea implementării limbajelor de nivel înalt, simplificării unității de comandă, optimizării instrucțiunilor consumatoare de timp , etc. De altfel, aceste idei au readus în discuție fundamentele proiectării calculatoarelor numerice [Pat82], care prin prisma cercetărilor unor pionieri precum *Eckert, Mauchly, von Neumman* și *Wilkes*, păreau să fie satisfăcătoare.Ideea novatoare principală a constat în optimizarea structurii hardware bazat pe cerințele impuse prin aplicațiile utilizatorului.

Limitarea principială a performanței arhitecturilor RISC la o instrucțiune/ciclu mașină sau tact, a adus începând din anul 1987 în discuție un concept și mai îndrăzneț: acela de **mașină cu execuții multiple ale instrucțiunilor (MEM).** Aceste mașini au drept principal deziderat depășirea barierei de o instrucțiune/tact și atingerea unor rate de procesare de mai multe instrucțiuni/tact. Aceste procesoare extind paralelismul temporal de tip pipeline la unul spațial bazat pe aducerea și execuția simultană a mai multor instrucțiuni mașină. Evident că acest model presupune existența mai multor unități funcționale de procesare. Arhitecturile MEM sunt implementate la ora actuală în două variante distincte: procesoare **superscalare** și respectiv procesoare **VLIW** (Very Long Instruction Word).

În varianta superscalară cade în sarcina hardului să verifice dinamic independenţa instrucţiunilor aduse şi să le lanseze în execuţii multiple spre unităţile de execuţie pipeline-izate. Desigur că aceste arhitecturi au o complexitate hardware deosebită determinată de detecţia şi soluţionarea hazardurilor între instrucţiuni, precum şi de exploatarea paralelismului determinat de gradul de independenţă între aceste instrucţiuni. De ex., complexitatea logicii de detecţie a independenţei instrucţiunilor ce se doresc a fi lansate în execuţie simultan, creşte proporţional cu pătratul numărului de instrucţiuni din bufferul de prefetch. De asemenea, ele sunt limitate în posibilitatea de a exploata paralelismul la nivelul instrucţiunilor, de capacitatea limitată a acestui buffer de prefetch [Joh91, Sto93].

La procesoarele VLIW, mai rare decât cele superscalare, cade în sarcina compilatorului de a reorganiza programul original în scopul "împachetării" într-o singură instrucţiune multiplă a mai multor instrucţiuni RISC primitive şi independente, care vor fi alocate unităţilor de execuţie în conformitate strictă cu poziţia lor în instrucţiunea multiplă. Un procesor VLIW aduce mai multe instrucţiuni primitive simultan şi le lansează în execuţie simultan spre unităţile funcţionale. Spre deosebire de modelul superscalar, aici rutarea instrucţiunilor spre diferitele unităţi de execuţie, se face static, anterior execuţiei, de către o componentă a compilatorului zisă scheduler sau reorganizator. Şi acest model arhitectural are dezavantaje precum: incompatibilităţi soft între variante succesive de procesoare, necesităţi sporite de memorare ale programelor, dificultăţi tehnologice legate de numărul de terminale, etc. Avantajul principal faţă de modelul superscalar constă în simplitatea hardware, în special în ceea ce priveşte logica de detecţie a hazardurilor şi lansare în execuţie.

Aceste maşini MEM sunt încadrabile în clasa MIMD (*Multiple Instruction Multiple Data)*, [Flyn96], deşi în taxonomia iniţială a lui *M. Flynn*, erau incluse în clasa SISD (*Single Instruction Single Data*), însă trebuie arătat că ele nu se confundă cu sistemele multiprocesor care exploatează paralelismul diverselor task-uri (*coarse grain parallelism*) şi care sunt principalele componente ale acestei clase arhitecturale. Maşinile MEM. exploatează paralelismul la nivelul instrucţiunilor maşină dintr-un anumit program, după cum am arătat (*fine grain parallelism*). Spre deosebire de MIMD- urile propriu zise, ele deţin avantajul de a menţine practic nemodificată viziunea convenţională, secvenţială, a programatorului în limbaj evoluat. Această calitate le-a asigurat de altfel un enorm succes comercial acestor arhitecturi, altfel, neconvenţionale. O oarecare ironie face ca aceste modele evolutive să se fi dovedit mult mai eficiente şi generale în exploatarea paralelismului decât alte structuri considerate revoluţionare, precum masivele de procesare de ex., care - în ciuda unor cercetări asidue şi îndelungate- au produs un impact neglijabil asupra utilizatorului obişnuit. Desigur însă, limitări fundamentale specifice tuturor sistemelor paralele, precum legea lui *Amdahl*, acţionează defavorabil şi asupra acestor arhitecturi ILP

O problemă esenţială este deci următoarea: există oare suficient paralelism ILP într-un program uzual pentru ca aceste cercetări să se justifice în continuare? Mai este oare acest domeniu fecund ? Răspunsul la această problemă este unul pozitiv şi este dat în capitolul 7 al acestei lucrări.

Toate aceste procesoare se consideră a fi de tip OTA- *Operation Triggered Architectures.* Recent, la Universitatea din Delft (Olanda), s-a introdus conceptul de procesoare TTA- *Transport Triggered Architectures*, care se pare că exploatează mai bine paralelismul de tip "fine- grain parallelism" printr-o arhitectură ILP având o doar singură instrucţiune condiţionată (MOVE) [Nee96].

Caracteristic domeniului ILP este optimimizarea interfeţei hardware- software a procesorului. Se porneşte de la ideea ca maşina hardware trebuie construită astfel încât să

execute optimal aplicații reale, cât mai diverse și cât mai generale, scrise în limbaje evoluate. Orice investigare prin prisma acestei idei, duce în final la legături directe, altfel de neimaginat, între caracteristici ale limbajului sau aplicației de nivel înalt și parametrii unor structuri hardware prin excelență. Nu cred să existe un alt domeniu al științei și ingineriei calculatoarelor în care să se expliciteze mai bine legătura strânsă între aplicația utilizatorului potențial și respectiv un detaliu de proiectare hardware. Pentru aceasta, instrumentele de analiză și investigație care trebuie construite, sunt complexe, dificile și cel mai adesea laborioase.

Astfel într-o cercetare a acestui domeniu, sunt necesare instrumente precum: crosscompilatoare, schedulere pentru optimizarea codului obiect, simulatoare complexe, benchmark-uri reprezentative a căror execuție să fie simulată, etc. Toate aceste instrumente de investigare este recomandabil să fie dublate de metode analitice, teoretice, de descriere și modelare a structurilor cercetate. Dar poate mai importantă, este abilitatea cercetătorului de a întrezări soluții la dificilele provocări ale domeniului sau chiar de a investiga probleme noi, neabordate încă. Prin fecunditatea sa la nivelul realizărilor și sumelor cheltuite la ora actuală pe plan mondial, se apreciază că domeniul ILP se situează între primele 2-3 domenii de cercetare în știința calculatoarelor, în prezent.

Această lucrare constituie atât o cercetare bibliografică laborioasă și actualizată într-un domeniu abordat în premieră la noi după știința autorului, cât și o cercetare asupra unor probleme dificile, deschise, ale acestui domeniu pasionant, la care se încearcă soluții originale.

Relativ la termenii științifici și tehnici utilizați, aceștia au fost în general preluați direct din literatura de specialitate, considerând acest fapt un "rău mai mic" decât traducerile în românește, deseori inexacte, uneori "ciudate" sau alteori chiar nocive.

Pe scurt, lucrarea este structurată astfel:

În capitolul 2 se prezintă o sinteză originală asupra problematicii procesoarelor scalare pipeline cu set optimizat de instrucțiuni mașină (RISC). Se prezintă principalele caracteristici arhitecturale ale acestor structuri de calcul, problemele implicate de metodele neconvenționale de procesare promovate precum și soluțiile care se impun pentru eficientizarea lor. Se pune accent în prezentare pe dualitatea hardware- software necesară a fi optimizată în cadrul acestor arhitecturi pipeline.

În capitolul 3, ca o continuare firească, se abordează problematica mai generală a arhitecturilor de calcul de tip MEM. Similar, se definesc principiile execuțiilor multiple ale instrucțiunilor în cadrul structurilor superscalare și respectiv VLIW. Se prezintă apoi principalii algoritmi hardware și software propuși în vederea unei execuții optimale, "Out of Order", a instrucțiunilor din program.De asemenea sunt scoase în evidență problemele care apar, tehnicile și soluțiile cele mai valoroase prezentate în literatura de specialitate la ora actuală.

În capitolul 4 se prezintă contribuțiile originale ale autorului în analiza de performanță și respectiv determinarea unor parametri optimali de proiectare, pentru diferite probleme deschise, de mare interes, în arhitecturile MEM. În principiu, metodele propuse sunt de tip analitic, bazându-se în principal pe conceptele de vector de coliziune și respectiv automat finit de stări. Complexitatea modelelor, văzute ca o alternativă de investigare la simulările foarte laborioase, este incomparabilă cu aceea a simulărilor și determină soluții pe baza unor metode relativ simple dar laborioase, de analiză numerică.

Capitolul 5 constituie o cercetare și în mare măsură o verificare pe bază de simulare, a concluziilor extrase din metodele analitice dezvoltate în capitolul precedent. Mai precis, pe baza unui simulator special conceput, se abordează analiza unor arhitecturi cache integrate într-un procesor RISC superscalar, de tip *Harvard* și respectiv unificat.

Simulatorul este astfel conceput încât să genereze parametrii optimi de proiectare, relativ la problema abordată. Specific acestei investigaţii este caracterul ei cantitativ.

În capitolul 6 sunt prezentate cercetări ale autorului cu privire la dificila problemă a predicţiei branch-urilor în cadrul arhitecturilor superscalare de procesoare. Se prezintă o metodă analitică originală de evaluare a performanţei schemelor de predicţie de tip Branch Target Buffer, precum şi o analiză bazată pe simulare a unui predictor corelat pe 2 nivele, integrat într-o arhitectură superscalară.

Capitolul 7 prezintă investigaţiile autorului relative la problema schedulingului în arhitecturile MEM. Astfel, se prezintă rezultatele obţinute în optimizarea basic-block-urilor cu ajutorul unui scheduler şi simulator special concepute. De asemenea, se prezintă o cercetare, relativă la gradul teoretic de paralelism disponibil şi exploatabil şi respectiv la cuantificarea "barierelor" ce stau în calea atingerii acestui grad maximal de paralelism.

În ultimul capitol se prezintă principalele concluzii desprinse pe parcursul acestei lucrări şi totodată se scot în evidenţă contribuţiile originale ale autorului aşa cum apar ele pe parcursul prezentei lucrări.

Lucrarea se încheie cu lista bibliografică a lucrărilor utilizate pe parcursul cercetării.

În final, ţin să-i mulţumesc în mod deosebit conducătorului meu de doctorat, domnul Prof. dr. ing. CRIŞAN STRUGARU, un pionier al cercetării în domeniul microprogramării şi microprocesoarelor la noi în ţară, pentru îndrumarea sa competentă, discuţiile şi sfaturile profesionale mature, acordate cu bunăvoinţă încă din perioada premergătoare înscrierii oficiale în această activitate şi continuate pe parcursul examenelor şi referatelor susţinute, până în prezent. De asemenea îi mulţumesc pentru încrederea, răbdarea şi căldura pe care permanent le-a arătat faţă de persoana mea.

2.ARHITECTURI PIPELINE SCALARE CU SET OPTIMIZAT DE INSTRUCȚIUNI

2.1 MODELUL RISC. GENEZĂ ȘI CARACTERISTICI GENERALE

Microprocesoarele RISC (**Reduced Instruction Set Computer**) au apărut ca o replică la modelul convențional de procesor de tip CISC (**Complex Instruction Set Computer**). Multe dintre instrucțiunile mașină ale procesoarelor CISC sunt foarte rar folosite în soltul de bază, cel care implementează sistemele de operare, utilitarele, translatoarele, etc. Lipsa de eficiență a modelului convențional CISC a fost pusă în evidență prin anii '80 de arhitecturi precum INTEL 80x86, MOTOROLA 680x0, iar in domeniul (mini)sistemelor în special de către arhitecturile VAX-11/780 și IBM - 360,370, cele mai cunoscute la acea vreme.

Modelele CISC sunt caracterizate de un set foarte bogat de instrucțiuni, formate de instrucțiuni de lungime variabilă, numeroase moduri de adresare deosebit de sofisticate, etc. Evident că această complexitate arhitecturală are o repercursiune negativă asupra performanței mașinii.

Primele microprocesoare RISC s-au proiectat la Universitățile din Stanford (coordonator *John Hennessy*) și respectiv Berkeley (coordonator *David Patterson,* cel care a și propus denumirea RISC), din California, USA (1981) [Pat82]. Spre deosebire de CISC-uri, proiectarea sistemelor RISC are în vedere că înalta performanță a procesării se poate baza pe simplitatea și eficacitatea proiectului. Strategia de proiectare a unui microprocesor RISC trebuie să țină cont de analiza aplicațiilor posibile pentru a determina operațiile cele mai frecvent utilizate, precum și optimizarea structurii în vederea unei execuții cât mai rapide a instrucțiunilor. Dintre aplicațiile sistemelor RISC se amintesc : conducerea de procese în timp real, procesare de semnale (DSP), calcule științifice cu viteză ridicată, elemente de procesare în sisteme multiprocesor și alte sisteme cu prelucrare paralelă.

Caracteristicile de bază ale modelului RISC sunt următoarele [DEC91,Pat82,Hen96]:

-Timp de proiectare și erori de construcție mai reduse decât la variantele CISC.

-Unitate de comandă hardware în general cablată, cu firmware redus sau deloc, ceea ce mărește rata de execuție a instrucțiunilor.

-Utilizarea tehnicilor de procesare pipeline a instrucțiunilor, ceea ce implică o rată teoretică de execuție de o instr./ciclu.

-Memorie sistem de înaltă performanță, prin implementarea unor arhitecturi de memorie de tip cache și MMU (Memory Management Unit). De asemenea conțin mecanisme hardware de memorie virtuală bazate pe paginare.

-Set relativ redus de instrucţiuni simple, majoritatea fără referire la memorie şi cu puţine moduri de adresare. În general, doar instrucţiunile LOAD/STORE sunt cu referire la memorie (**arhitectura tip LOAD / STORE**). La implementările recente caracteristica de "set redus de instrucţiuni" nu trebuie înţeleasă add literam ci mai corect în sensul de **set optimizat de instrucţiuni** în vederea implementării aplicaţiilor propuse (în special implementării limbajelor de nivel înalt- HLL).
-Datorită unor particularităţi ale procesării pipeline (în special hazardurile pe care aceasta le implică), apare necesitatea unor compilatoare optimizate zise şi reorganizatoare sau schedulere, cu rolul de a reorganiza programul sursă pentru a putea fi procesat optimal din punct de vedere al timpului de execuţie.
-Format fix al instrucţiunilor, codificate în general pe un singur cuvânt de 32 biţi, mai recent pe 64 biţi (Alpha 21164, Power PC-620).
-Necesităţi de memorare a programelor mai mari decât în cazul microsistemelor convenţionale, datorită simplităţii instrucţiunilor cât şi reorganizatoarelor care pot acţiona defavorabil asupra "lungimii" programului sursă .
-Set de registre generale substanţial mai mare decât la CISC-uri, în vederea lucrului "în ferestre" (register windows), util în optimizarea instrucţiunilor CALL / RET. Numărul mare de registre generale este util pentru mărirea spaţiului intern de procesare, tratării optimizate a evenimentelor de excepţie, model ortogonal de programare, etc. Registrul R0 este cablat la zero în majoritatea implementărilor, pentru optimizarea modurilor de adresare şi a instrucţiunilor .

Microprocesoarele RISC reprezintă **modele cu adevărat evolutive** în istoria tehnicii de calcul. Primul articol despre acest model de procesare a apărut în anul 1981 (*David Patterson, Carlo Sequin*), iar peste numai 6-7 ani toate marile firme producătoare de hardware realizau microprocesoare RISC scalare în scopuri comerciale sau de cercetare. Performanţa acestor microprocesoare creşte cu cca. 75% în fiecare an.

2.2 SET DE INSTRUCŢIUNI, REGIŞTRI INTERNI LA MODELUL ARHITECTURAL RISC

În proiectarea setului de instrucţiuni aferent unui microprocesor RISC intervin o multitudine de consideraţii dintre care se amintesc [DEC91, Pat94, Hen96, Pat82]:

a). Compatibilitatea cu seturile de instrucţiuni ale altor procesoare pe care s-au dezvoltat produse software consacrate (compatibilitatea de sus în jos, valabilă de altfel şi la CISC-uri). Portabilitatea acestor produse pe noile procesoare este condiţionată de această cerinţă care vine în general în contradicţie cu cerinţele de performanţă ale sistemului.

b). În cazul microprocesoarelor, setul de instrucţiuni este în strânsă dependenţă

cu tehnologia folosită, care de obicei limitează sever performanţele (constrângeri legate de aria de integrare, numărul de pini, cerinţe restrictive ale tehnologiei, etc.).

c). Minimizarea complexităţii unităţii de comandă precum şi minimizarea fluxului de informaţie procesor-memorie.

d). În cazul multor microprocesoare RISC setul de instrucţiuni maşină e ales pentru a fi suport pentru implementarea unor limbaje de nivel înalt.

Setul de instrucţiuni al unui microprocesor RISC este caracterizat de simplitatea formatului precum şi de un număr limitat de moduri de adresare. De asemenea, se urmăreşte ortogonalizarea setului de instrucţiuni. Deosebit de semnificativ în acest sens, este chiar primul microprocesor RISC numit RISC I Berkeley [Pat82].

Acesta deţine un set de 31 instrucţiuni grupate în 4 categorii : aritmetico-logice, de acces la memorie, de salt / apel subrutine şi speciale. Microprocesorul deţine un set de 32 regiştri generali pe 32 biţi. Formatul instrucţiunilor de tip registru-registru şi respectiv de acces la memorie (LOAD / STORE) este dat in figura 2.1 :

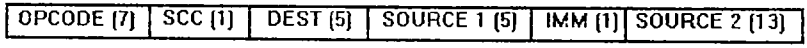


Fig. 2.1

Câmpul IMM = 0 arată că cei mai puţini semnificativi (cmps) 5 biţi ai câmpului SOURCE2 codifică al 2-lea registru operand.

Câmpul IMM = 1 arată că SOURCE2 semnifică o constantă pe 13 biţi cu extensie semn pe 32 biţi.

Câmpul SCC semnifică validare / invalidare a activării indicatorilor de condiţie, corespunzător operaţiilor aritmetico-logice executate.

În ciuda setului redus de instrucţiuni al procesorului RISC I, acesta poate sintetiza o multitudine de instrucţiuni aparent "inexistente" la acest model. Practic nu se "pierd" instrucţiuni ci doar "opcode"- uri, şi ca o consecinţă, în plus, se simplifică substanţial logica de decodificare şi unitatea de comandă [Pat91, Pat94, Vin96].

Potenţial, programele RISC pot conţine mai multe CALL-uri decât cele convenţionale, în principal pentru că instrucţiunile complexe implementate în procesoarele CISC vor fi subrutine în cazul procesoarelor RISC. Din acest motiv procedura CALL / RET se impune a fi cât mai rapidă în procesarea RISC. Modelul utilizării registrelor în fereastre îndeplineşte acest deziderat prin reducerea traficului de date cu memoria sistem (Berkeley I, SPARC, etc.) Principiul de lucru constă în faptul că fiecare instrucţiune CALL alocă o nouă fereastră de registre pentru a fi utilizată de procedura apelată, în timp ce o instrucţiune RET va restaura vechea fereastră de registre. Se consideră spre exemplificare 3 procese soft, care se apelează imbricat ca în figura

următoare (Fig.2.2) :

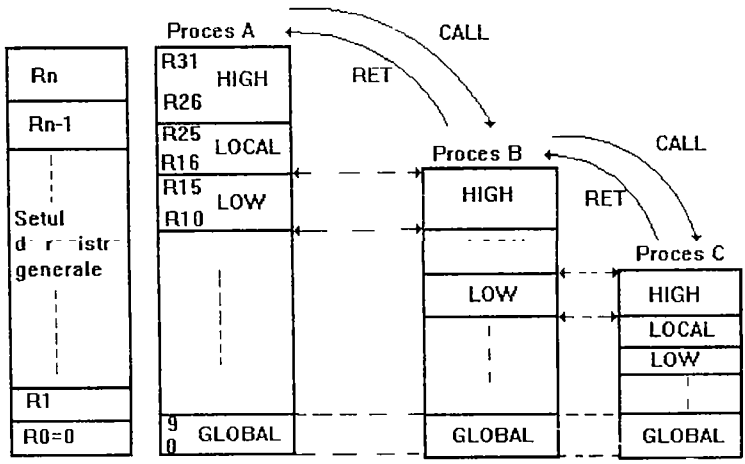


Fig. 2.2

Statistici deosebit de serioase arată că în cadrul implementărilor de limbaje HLL (High Level Languages) instrucțiunile CALL / RET sunt frecvent folosite și totodată cele mai mari consumatoare de timp (ocupă între 5-45% din referirile la memorie).

Fiecare proces are alocată o fereastră de regiștri constând în 3 seturi distincte : HIGH, LOCAL și LOW. Registrele globale R0 - R9 conțin parametrii globali ai proceselor soft. Acești regiștri sunt comuni tuturor proceselor și nu se salvează / restaurează niciodată.

Regiștrii locali R16 - R25 sunt utilizați pentru memorarea scalarilor locali ai procesului curent. O instrucțiune CALL determină ca registrele LOW din procesul master apelant să devină registre HIGH în procesul SLAVE apelat. Odată cu schimbarea ferestrelor, o instrucțiune CALL memorează registrul PC într-un anumit registru al noii ferestre.

O instrucțiune RET determină o acțiune reciprocă, adică regiștrii HIGH ai procesului curent vor deveni regiștrii LOW pentru procesul MASTER în care se revine.

În regiștrii LOW procesul MASTER transmite automat parametrii spre procesul SLAVE, respectiv din acești regiștri procesul MASTER curent preia rezultatele de la procesul SLAVE apelat.

În regiștrii HIGH procesul curent memorează rezultatele, respectiv din acești regiștri procesul curent preia parametrii transmiși de către procesul MASTER apelant. Se observă că aceste comutări de ferestre au drept principal scop eliminarea stivei și deci a timpului consumat cu accesarea acesteia.

Referitor la cele de mai sus apar 2 tipuri de excepţii ca fiind posibile :
-window overflow (WO) poate să apară după o instrucţiune CALL în care schimbarea setului de regiştrii LOW ai procesului curent în regiştrii HIGH ai procesului viitor devine imposibilă datorită numărului limitat de regiştri generali implementaţi;
-window underflow (WU) poate să apară după o instrucţiune RET care determină ca schimbarea regiştrilor HIGH ai procesului curent în regiştri LOW ai procesului viitor să devină imposibilă.

Dintre microprocesoarele RISC consacrate, cele din familia SPARC (sistemele SUN) au implementat un set de maxim 32 ferestre a câte 32 regiştri fiecare. Numărul mare de regiştri generali necesari pentru implementarea conceptului de register windows implică reducerea frecvenţei de tact a procesorului, fapt pentru care conceptul nu e implementat în toate microprocesoarele RISC comerciale.

Studii statistice realizate încă din perioada de pionierat de către Halbert şi Kessler, arată că dacă procesorul deţine 8 ferestre de regiştri, excepţiile WO şi WU vor apare în mai puţin de 1% din cazuri. Evident că supraplasarea trebuie evitată în cadrul rutinei de tratare a excepţiei prin trimiterea / recepţionarea parametrilor proceselor în / din buffere de memorie (stive).

De precizat că setul larg de regiştri generali este posibil de implementat la microprocesoarele RISC prin prisma tehnologiilor VLSI disponibile (MOS, ECL, GaAs), întrucât unitatea de comandă, datorită simplităţii ei, ocupă un spaţiu relativ restrâns (la Berkeley RISC I de ex. aproximativ 6% din aria de integrare). Restul spaţiului rămâne disponibil pentru implementarea unor suporturi de comunicaţie interprocesor (transputere), memorii cache, MMU, set regiştri generali extins, arii sistolice de calcul (DSP-uri), etc.

2.3 ARHITECTURA SISTEMULUI DE MEMORIE LA PROCESOARELE RISC

Principalele caracteristici ale sistemului de memorie aferent unui sistem RISC sunt următoarele:

Management intern de memorie (MMU- Memory Management Unit) în majoritatea implementărilor. MMU-ul are rolul de a translata adresa virtuală emisă de către microprocesor într-o aşa numită adresă fizică de acces la memoria principală şi respectiv de a asigura un mecanism de control şi protecţie-prin paginare sau/şi segmentare a memoriei- a accesului la memorie. În general, MMU paginează memoria principală în majoritatea implementărilor cunoscute, oferind şi resursele hardware necesare mecanismelor de memorie virtuală.

Memorii cache interne şi externe cu spaţii în general separate pentru instrucţiuni şi dat

Registre de tip WB (write buffer) cu rol de gestionare a scrierii efective a datei în memoria sistem. Pentru a asigura consistența datelor scrise în cache, acestea trebuie scrise și în memoria principală. În scopul eliberării microprocesorului de acest lucru, WB capturează data și adresa emise de către microprocesor și execută scrierea în memoria principală, permițând microprocesorului să-și continue activitatea fără să mai aștepte până când data a fost efectiv scrisă in memoria principală. Așadar, WB se comportă ca un mic procesor de ieșire lucrând în paralel cu microprocesorul. Arhitectura de principiu este caracterizată in figură:

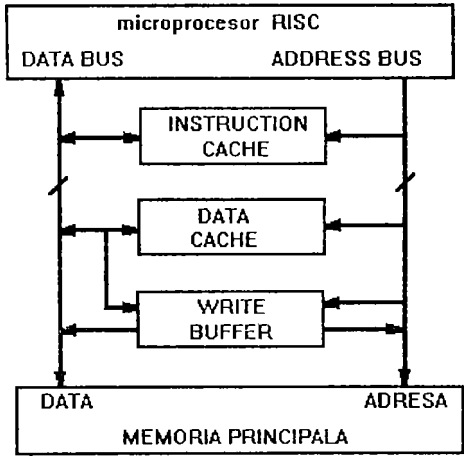


Fig. 2.3

În afara spațiilor separate de instrucțiuni și date **(arhitectură Harvard)** și a procesorului de ieșire (WB) nu există alte probleme majore specifice microprocesoarelor RISC în arhitectura sistemelor de memorie. În principiu, ierarhizarea sistemului de memorie este aceeași ca și la sistemele CISC (memorii cache, memorie virtuală). Nu intrăm aici în detalii legate de organizarea cache-urilor, a mecanismelor de memorie virtuală etc., întrucât aceste probleme sunt similare cu cele de la sistemele CISC, fiind deci foarte cunoscute și bine documentate bibliografic [Hen96, Sto93, Sta96].

Problema majoră a sistemelor de memorie pentru ambele variante constă în decalajul tot mai accentuat între performanța microprocesoarelor care crește anual cu 50-100% și respectiv performanța tehnologică a memoriilor (latența) care au o rată de creștere de doar 7% pe an. Prin urmare acest "semantic gap" microprocesor- memorie reprezintă o provocare pentru îmbunătățirea performanțelor arhitecturilor de memorie aferente arhitecturilor pipeline scalare și nu numai.

2.4. PROCESAREA PIPELINE A INSTRUCŢIUNILOR ÎN CADRUL PROCESOARELOR SCALARE

Cea mai importantă caracteristică arhitecturală a acestor microprocesoare o constituie **procesarea pipeline** a instrucţiunilor şi datelor. În fapt, toate celelalte caracteristici arhitecturale ale RISC-urilor au scopul de a adapta structura procesorului la procesarea pipeline. Acest concept a migrat de la sistemele mari de calcul la microprocesoare datorită progresului tehnologic. Toate supercalculatoarele au implementat acest concept. Primul calculator pipeline viabil a fost- la acea vreme - supersistemul CDC 6600 (firma Control Data Company, şef proiect *Seymour Cray*, 1964).

2.4.1 DEFINIREA CONCEPTULUI DE ARHITECTURĂ PIPELINE SCALARĂ

Tehnica de procesare pipeline reprezintă o tehnică de procesare paralelă a informaţiei prin care un proces secvenţial este divizat în subprocese fiecare subproces fiind executat într-un segment special dedicat care operează în paralel cu celelalte segmente. Fiecare segment execută o procesare parţială a informaţiei. Rezultatul obţinut în segmentul i este transmis în tactul următor spre procesare segmentului i+1. Rezultatul final este obţinut numai după ce informaţia a parcurs toate segmentele, la ieşirea ultimului segment. Denumirea de pipeline provine de la analogia cu o bandă industrială de asamblare. Este caracteristic acestor tehnici faptul că diversele procese se pot afla în diferite faze de prelucrare în cadrul diverselor segmente, simultan. Suprapunerea procesărilor e posibilă prin asocierea unui registru fiecărui segment din pipeline. Registrele produc o separare între segmente astfel încât fiecare segment să poată prelucra date separate (Fig.2.4).

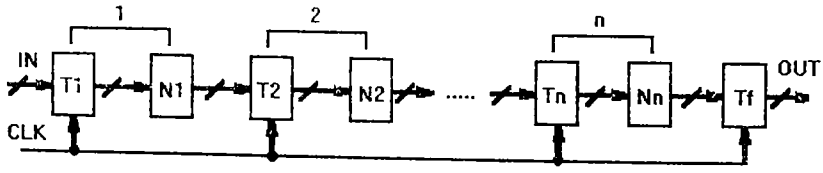


Fig. 2.4

Ti - regiştri tampon

Ni - nivele de prelucrare (combinaţionale sau secvenţiale)

Este evident că nivelul cel mai lent de prelucrare va stabili viteza de lucru a

benzii. Aşadar, se impune în acest caz partiţionarea unui eventual proces mai lent în subprocese cu timpi de procesare cvasiegali şi interdependenţe minime. Există 2 soluţii principiale, discutate în literatură, de a compensa întârzierile diferite pe diversele nivele de prelucrare [Pop92]:

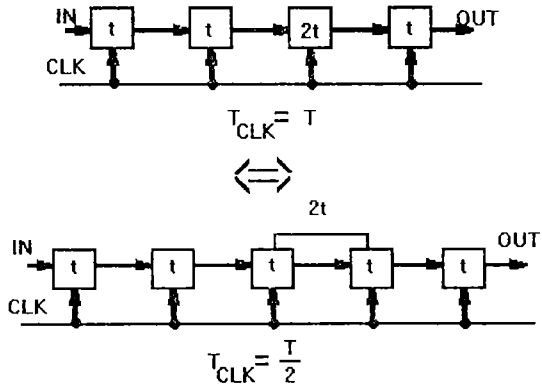


Fig. 2.5

Această soluţie de balansare a benzii necesită circuite suplimentare şi poate fi realizată doar cu condiţia ca sarcinile alocate respectivului nivel să poată fi descompuse.

O altă soluţie de balansare se bazează pe conectarea în paralel a unui alt nivel cu aceeaşi întârziere ca cel mai lent. Şi aici trebuiesc circuite suplimentare. În plus, apare problema sincronizării şi controlului nivelelor care lucrează în paralel. Soluţia e utilizată când sarcinile alocate nivelului mai lent nu pot fi descompuse (Fig.2.6).

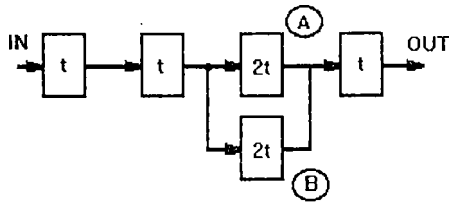


Fig. 2.6

În Tact (i) informaţia se trimite spre procesare nivelului următor pe calea A. În Tact (i+1) informaţia se trimite analog pe calea B, implementându-se deci o demultiplexare.

Se defineşte **rata de lucru** R a unei benzi de asamblare ca fiind numărul de

procese executate în unitatea de timp T (ciclu maşină sau tact). Considerând m posturi de lucru prin care trec n procese, rezultă că banda le va executa într-un interval de timp $\Delta t = (m + n - 1) * T$. Normal, pentru că $m * T$ este timpul de "setup" al benzii, adică timpul necesar terminării primului proces. Aşadar, rata de execuţie a unei benzi prin care trec n procese este:

$$R_n = \frac{n}{(m + n - 1) * T} \qquad (2.1.)$$

Rata ideală este de un proces per ciclu, întrucât:

$$R_\infty = \lim_{n \to \infty} R_n = \frac{1}{T} \qquad (2.2.)$$

După se va arăta, această rată teoretică nu se va putea atinge în practică, nu numai datorită faptului că se prelucrează un număr finit de procese şi datorită timpului de setup, ci mai ales datorită unor blocaje în funcţionarea normală a benzii numite hazarduri.

2.4.2. PRINCIPIUL DE PROCESARE ÎNTR-UN PROCESOR PIPELINE

Procesarea pipeline a instrucţiunilor reprezintă o tehnică de procesare prin intermediul căreia fazele aferente multiplelor instrucţiuni sunt suprapuse în timp. Arhitectura microprocesoarelor RISC este mai bine adaptată la pipeline decât cea a sistemelor convenţionale CISC, datorită instrucţiunilor de lungime fixă, a modurilor de adresare specifice, a structurii interne bazate pe registre generale, etc. Microprocesoarele RISC uzuale deţin o structură pipeline de instrucţiuni întregi pe 4 - 5 - 6 nivele. De exemplu, microprocesoarele MIPS au următoarele nivele tipice:

1. Nivelul IF (instruction fetch) - se calculează adresa instrucţiunii ce trebuie citită din cache-ul de instrucţiuni sau din memoria principală şi se aduce instrucţiunea;

2. Nivelul RD (ID) - se decodifică instrucţiunea adusă şi se citesc operanzii din setul de regiştri generali. În cazul instrucţiunilor de salt, pe parcursul acestei faze se calculează adresa de salt;

3. Nivelul ALU - se execută operaţia ALU asupra operanzilor selectaţi în cazul instrucţiunilor aritmetico-logice; se calculează adresa pentru instr. LOAD / STORE;

4. Nivelul MEM - se accesează memoria cache de date sau memoria principală, însă numai pentru instrucţiunile LOAD / STORE. Acest nivel pe funcţia de citire poate

pune probleme datorate neconcordanței între rata de procesare și timpul de acces la memoria principală. Rezultă deci că într-o structură pipeline cu N nivele, memoria trebuie să fie în principiu de N ori mai rapidă decât într-o structură de calcul convențională. Acest lucru se realizează prin implementarea de arhitecturi de memorie rapide (cache, memorii cu acces întrețesut, etc.). Desigur că un ciclu cu MISS în cache pe acest nivel (ca și pe nivelul IF de altfel), va determina stagnarea acceselor la memorie sau chiar a procesării interne.

La scriere, problema aceasta nu se pune datorită procesorului de ieșire specializat WB care lucrează în paralel cu procesorul central după cum deja am arătat.

5. Nivelul WB (write buffer) - se scrie rezultatul ALU sau data citită din memorie (în cazul unei instrucțiuni LOAD) în registrul destinație din setul de regiștri generali ai microprocesorului.

Prin urmare, printr-o astfel de procesare se urmărește o rată ideală de o instrucțiune per ciclu mașină ca în figură, deși după cum se observă, timpul de execuție pentru o instrucțiune dată nu se reduce.

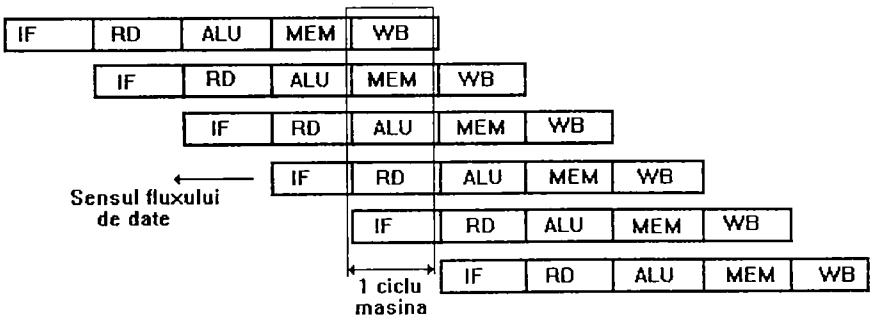


Fig. 2.7

Se observă imediat necesitatea suprapunerii a 2 nivele concurențiale : nivelul IF și respectiv nivelul MEM, ambele cu referire la memorie. În cazul microprocesoarelor RISC această situație se rezolvă deseori prin legături (busuri) separate între procesor și memoria de date respectiv de instrucțiuni (**arhitectură Harvard**).

Deși această împărțire pe 5 nivele este caracteristică multor microprocesoare RISC, ea are o deficiență importantă și anume că nu orice instrucțiune trece prin toate cele 5 nivele de procesare. Astfel, nivelul MEM este exploatat doar de către instrucțiunile LOAD / STORE, nivelul ALU de către instrucțiunile aritmetico-logice și LOAD / STORE, iar instrucțiunile de comparare sau memorare (STORE) nu folosesc nivelul WB. Probabil asemenea observații au determinat proiectanții anumitor microprocesore RISC (de ex. microprocesorul HARP proiectat la Universitatea din Hertfordshire, UK), să

comprime nivelele ALU şi MEM într-unul singur. In acest caz calculul adreselor de memorie se face în nivelul RD prin mecanisme care reduc acest timp de calcul [Ste91].

În literatură se citează un model de procesor numit superpipeline. Acesta este caracterizat printr-un număr relativ mare al nivelelor de procesare. Desigur că în acest caz detecţia şi corecţia hazardurilor este mai dificilă (vezi în continuare). Arhitecturile superpipeline se pretează la tehnologiile cu grade de împachetare reduse unde nu este posibilă multiplicarea resurselor hardware, în schimb caracterizate prin viteze de comutaţie ridicate (ECL, GaAs). O asemenea arhitectură caracterizează de ex. procesoarele din familia DEC Alpha. Avantajul principal al arhitecturilor superpipeline este că permit frecvenţe de tact deosebit de ridicate (200-500 MHz la nivelul tehnologiilor actuale),aspect normal având în vedere super- divizarea stagiilor de procesare.

2.4.3. STRUCTURA HARDWARE A UNUI PROCESOR RISC

În continuare se prezintă o structură hardware de principiu a unui procesor RISC compatibil cu modelul de programare al microprocesorului RISC Berkeley I anterior prezentat. De remarcat că structura permite procesarea pipeline a instrucţiunilor, adică atunci când o instrucţiune se află într-un anumit nivel de procesare, instrucţiunea următoare se află în nivelul anterior. Stagiile de interconectare ale nivelelor structurii (IF / ID, ID / EX, EX / MEM, MEM / WB) sunt implementate sub forma unor regiştri de încărcare, actualizaţi sincron cu fiecare nou ciclu de procesare. Memorarea anumitor informaţii de la un nivel la altul este absolut necesară, pentru ca informaţiile conţinute în formatul instrucţiunii curente să nu se piardă prin suprapunerea fazelor de procesare.

În acest sens, să ne imaginăm de exemplu ce s-ar întâmpla dacă câmpul DEST nu ar fi memorat succesiv din nivel în nivel şi rebuclat la intrarea file-ului de regiştri generali (vezi Fig.2.8). Utilitatea acestui câmp devine oportună abia în faza WB când şi data de înscris în setul de regiştri generali devine disponibilă (cazul instrucţiunilor aritmetico-logice sau STORE). Să remarcăm că setul de regiştri generali permite simultan două citiri şi o scriere.

Multiplexorul de la intrarea ALU are rolul de a genera registrul sursă 2, în cazul instrucţiunilor aritmetico-logice, respectiv indexul de calcul adresă (constantă pe 13 biţi cu extensie semn), în cazul instrucţiunilor LOAD / STORE.

Sumatorul SUM 2 calculează adresa de salt în cazul instrucţiunilor de branch după formula:

$$PCnext = PC + Ext.semn(IR18 - 0) \qquad (2.3.)$$

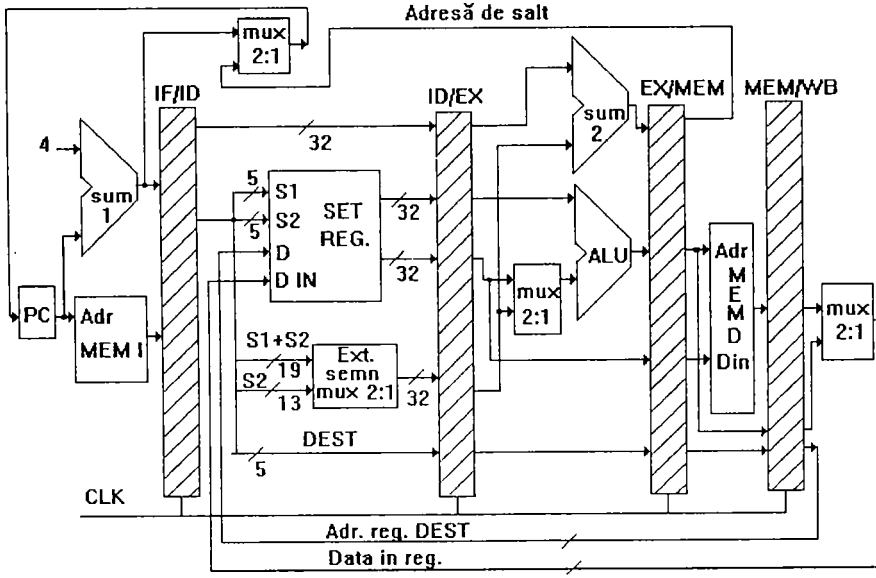


Fig. 2.8

Multiplexorul 2:1 de la ieşirea nivelului MEM / WB multiplexează rezultatul ALU în cazul unei instrucţiuni aritmetico-logice, respectiv data citită din memoria de date în cazul unei instrucţiuni LOAD. Ieşirea acestui multiplexor se va înscrie în registrul destinaţie codificat în instrucţiune.

Evident că întreaga structură este comandată de către o unitate de control. Detalii despre proiectarea unei asemenea unităţi sunt date de exemplu în [Pat94], cap.6.

2.4.4 PROBLEMA HAZARDURILOR ÎN STRUCTURILE PIPELINE

Hazardurile constituie acele situaţii care pot să apară în procesarea pipeline şi care pot determina blocarea (stagnarea) procesării, având o influenţă negativă asupra ratei de execuţie a instrucţiunilor. Conform unei clasificări consacrate aceste hazarduri sunt de 3 categorii : **hazarduri structurale, de date şi de ramificaţie.**

2.4.4.1 HAZARDURI STRUCTURALE (HS) : PROBLEME IMPLICATE ŞI SOLUŢII

Sunt determinate de conflictele la resurse comune, adică atunci când mai multe

procese simultane aferente mai multor instrucţiuni în curs de procesare, accesează o resursă comună. Pentru a le elimina prin hardware, se impune multiplicarea acestor resurse. De exemplu, un procesor care are un set de regiştri generali de tip uniport şi în anumite situaţii există posibilitatea ca 2 procese să dorească să scrie în acest set simultan. Prezentăm mai jos (Fig.2.9) o structură ALU implementată la microprocesorul RISC superscalar HARP care permite 4 operaţii ALU simultane. Prin partiţionarea timpului afectat nivelului WB în două, în cadrul acestui nivel se pot face două scrieri (în prima jumătate a nivelului WB se înscrie în setul de regiştri conţinutul căii A, iar în a doua parte a nivelului WB se înscrie conţinutul căii B). În principiu, un astfel de procesor poate să execute 4 instrucţiuni simultan cu condiţia ca numai două dintre ele să aibă nivelul WB activ. Evident că cele 4 seturi de regiştri generali deţin conţinuturi identice în orice moment. Deci, prin multiplicarea resurselor hardware s-a creat posibilitatea execuţiei mai multor operaţii (instrucţiuni) fără a avea conflicte la resurse[Ste92,93,95].

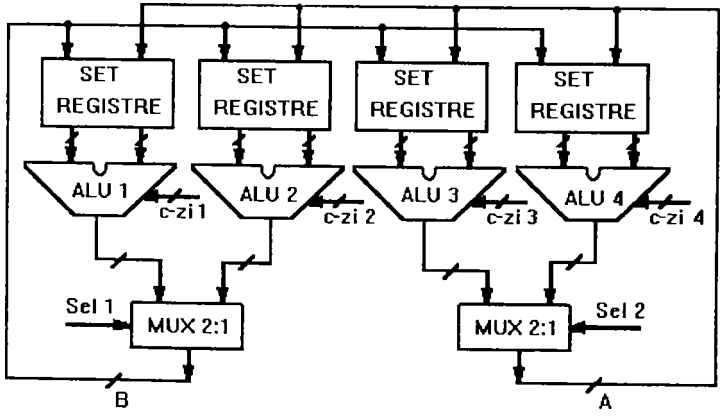


Fig. 2.9

O altă situaţie de acest fel poate consta în accesul simultan la memorie a 2 procese distincte unul de aducere a instrucţiunii (IF), iar celălalt de aducere a operandului (nivelul MEM). După cum am mai arătat, această situaţie se rezolvă în general printr-o arhitectură Harvard a busurilor. Totuşi, există microprocesoare care deţin busuri şi cache-uri unificate pe instrucţiuni şi date(Power PC 601, de exemplu). O cercetare detaliată a acestei probleme practic nedezbătută în literatura de specialitate, va constitui un obiectiv important al prezentei lucrări (Cap. 4,5).

2.4.4.1.1. MODELĂRI ALE HAZARDURILOR STRUCTURALE PE BAZA VECTORILOR DE COLIZIUNE

Să considerăm, fără a reduce din generalitate, o structură pipeline cu 5 nivele având timpul de setup de 7 cicli, descrisă în funcţionare de tabelul Tab.2.1.

ciclu nivel	T1	T2	T3	T4	T5	T6	T7
N1	X						
N2		X	X				
N3			X	X			
N4					X		X
N5						X	

Tab. 2.1

Un X în tabel semnifică faptul că în ciclul Ti nivelul Nj este activ adică procesează informaţii. Să mai considerăm că această structură pipeline corespunde unui anumit proces (de ex. procesarea unei instrucţiuni). Se observă că un alt proces de acest tip, nu poate starta în ciclul următor T2 datorită coliziunilor care ar putea să apară între cele două procese pe nivelul (N2, T3) şi respectiv (N3, T4). Mai mult, următorul proces n-ar putea starta nici măcar în T3 din motive similare de coliziune cu procesul anterior în (N4, T5). În schimb procesul următor ar putea starta în T4 fără a produce coliziuni (sau hazarduri structurale cum le-am denumit), deci la 2 cicli după startarea procesului curent.

Se defineşte **vectorul de coliziune** al unei structuri pipeline având timpul de setup de (N+1) cicli, un vector binar pe N biţi astfel: dacă bitul i, i∈ {0, 1, ..., N-1} 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 de ex. pentru structura pipeline anterioară că vectorul de coliziune este 110000, însemnând deci că procesul următor nu trebuie startat în T2 sau T3, 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 obţină o rată de procesare (procese / ciclu) maximă? Considerând o structură pipeline cu timpul de setup 8 şi având vectorul de coliziune 1001011 ar trebui procedat ca în figurile următoare:

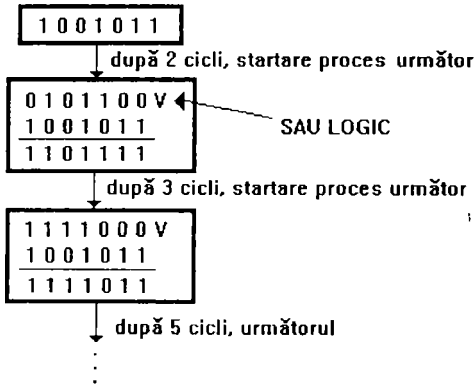


Fig. 2.10

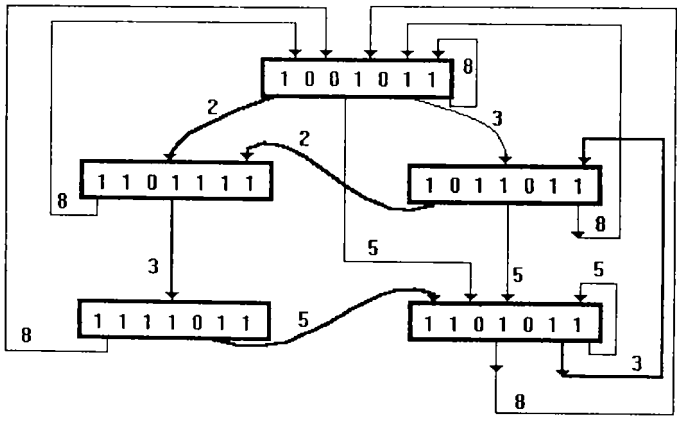


Fig. 2.11

Vectorul unei stări Sj, se obține după relația:

$$Vj = (VC)\ V\ (Vi^{*}(m)), \qquad (2.4.),\ \text{unde:}$$

V = SAU logic

VC = vector coliziune

$Vi^{*}(m)$ = vectorul Vi deplasat logic la stânga cu (m) poziții binare

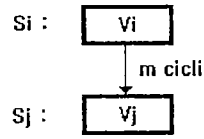


Fig. 2.12

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, avem 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 cicli, adică 0.31 procese / ciclu. De menționat că o structură convențională ar procesa 0.125 procese / ciclu.

Nu întotdeauna startarea procesului următor imediat ce acest lucru devine posibil (**"greedy strategy"**), duce la o performanță maximă. Un exemplu în acest sens ar fi o structură pipeline cu vectorul de coliziune asociat 01011. E adevărat însă că o asemenea strategie conduce la un algoritm mai simplu.

Este clar că 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.

În [Sto93] paginile 182 - 190 se arată pe baza unei tehnici atribuită lui Patel și Davidson (1976) că prin modificarea tabelei aferentă structurii pipeline în sensul introducerii unor întârzieri, se poate îmbunătăți performanța benzii.

2.4.4.1.2.CONTROLUL STRUCTURILOR PIPELINE MULTIFUNCȚIONALE

Să considerăm spre exemplu 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 prin VC(Pi, Pj) am notat vectorul de coliziune între procesul curent Pi și procesul următor Pj.

Odată determinați acești vectori în baza tabelelor de descriere aferente celor două procese controlul structurii s-ar putea face prin următoarea schemă de principiu:

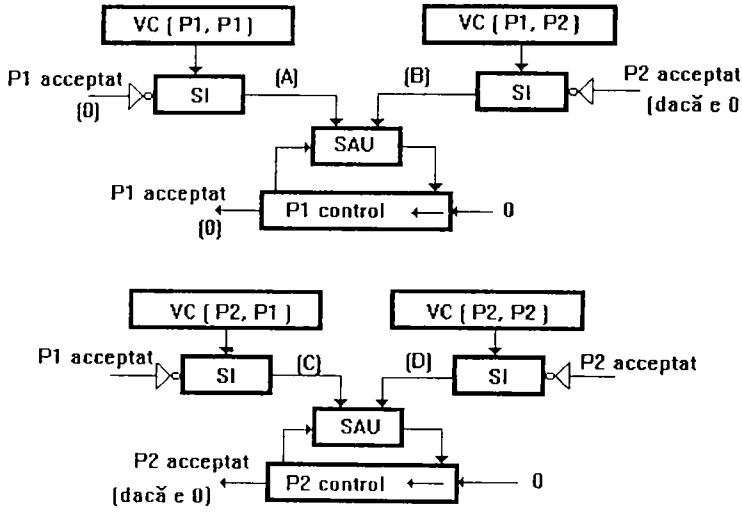


Fig. 2.13

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 c.m.s. 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) respectic 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".

2.4.4.2 HAZARDURI DE DATE: DEFINIRE, CLASIFICARE, SOLUȚII DE EVITARE A EFECTELOR DEFAVORABILE

Apar când o instrucțiune depinde de rezultatele unei instrucțiuni anterioare în bandă. Pot fi la rândul lor clasificate în 3 categorii, dependent de ordinea acceselor de citire, respectiv scriere, în cadrul instrucțiunilor.

2.4.4.2.1 HAZARDURI RAW: DEPENDENȚE REALE

Considerând instrucțiunile i și j succesive, **hazardul RAW** apare atunci când instrucțiunea j încearcă să citească o sursă înainte ca instrucțiunea i să scrie în aceasta. Apare deosebit de frecvent în implementările actuale de procesoare pipeline. Să considerăm secvența de instrucțiuni de mai jos procesată într-o structură pe 5 nivele, ca

în figura următoare. Se observă că data ce urmează a fi incărcată în R5 este disponibilă doar la finele nivelului MEM aferent instrucțiunii I1, prea târziu pentru procesarea corectă a instrucțiunii I2 care ar avea nevoie de această dată cel târziu la începutul nivelului său ALU. Aşadar, pentru o procesare corectă, I2 trebuie stagnată cu un ciclu maşină.

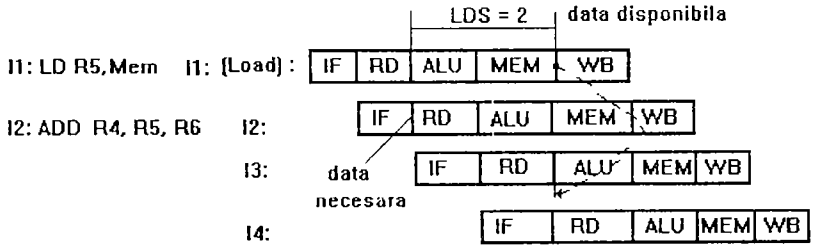
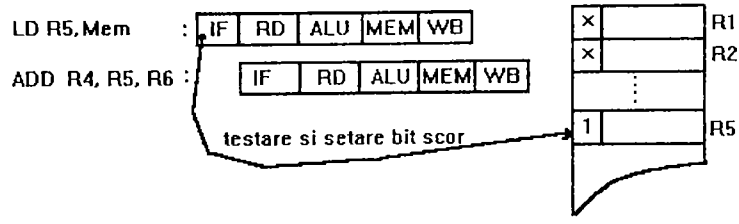


Fig. 2.14

Aceasta se poate realiza prin software, de exemplu prin inserarea unei instrucțiuni NOP între I1 şi I2 (umplerea "load delay slot-ului"- LDS) sau o altă instrucțiune utilă independentă de instrucțiunea I1. O altă modalitate de rezolvare constă în stagnarea hardware a instrucțiunii I2, stagnare determinată de detecția hazardului RAW de către unitatea de control. Realizarea întârzierii hardware se poate baza pe tehnica "scoreboarding" propusă pentru prima dată de către *Seymour Cray*(1964-CDC 6600). Se impune ca fiecare registru al procesorului din setul de regiştri să aibă un "bit de scor" asociat. Dacă bitul este zero, registrul respectiv e disponibil, dacă bitul este 1, registrul respectiv este ocupat. Dacă pe un anumit nivel al procesării este necesar accesul la un anumit registru având bitul de scor asociat pe 1, respectivul nivel va fi întârziat, permiţându-i-se accesul doar când bitul respectiv a fost şters de către procesul care l-a setat. De remarcat că ambele soluții bazate pe stagnarea fluxului (soft...re - NOP sau hardware - semafoare) au acelaşi efect defavorabil asupra performanței.



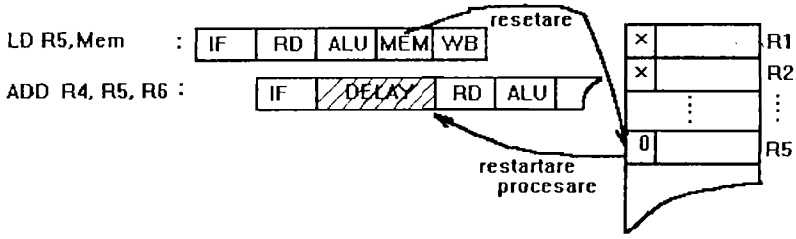
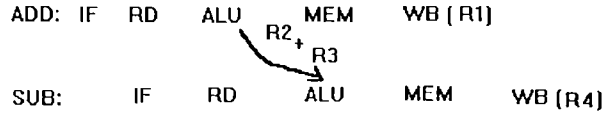


Fig. 2.15

Există situații în care hazardul RAW se rezolvă prin hardware fără să cauzeze stagnări ale fluxului de procesare ca în cazul anterior. Aceste tehnici de rezolvare se numesc **tehnici forwarding** (bypassing) bazate pe "pasarea" rezultatului instrucțiunii i, în mod anticipat, nivelului de procesare aferent instrucțiunii j care are nevoie de acest rezultat. De exemplu să considerăm secvența:

ADD R1, R2, R3 ; R1 <------ (R2) + (R3)
SUB R4, R1, R5 ; R4 <------ (R1) - (R5)



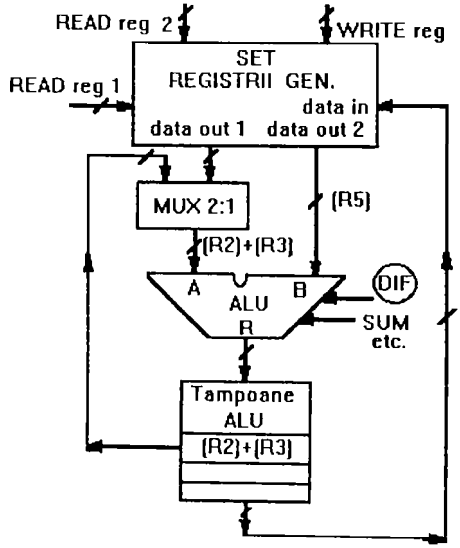


Fig. 2.16

Rezultatul ALU aferent primei instrucții (R2 + R3) e memorat în tampoanele ALU la finele fazei ALU a instrucțiunii ADD. Dacă unitatea de control va detecta hazardul RAW, va selecta pe parcursul fazei ALU aferente instrucțiunii SUB la intrarea A a ALU, tamponul care conține (R2) + (R3) (în urma fazei ALU a instrucțiunii ADD), evitând astfel hazardul RAW.

2.4.4.2.2. HAZARDURI WAR

Poate să apară atunci când instrucțiunea j scrie o destinație înainte ca aceasta să fie citită ca o sursă de către instrucțiunea i. Poate să apară când într-o structură pipeline există o fază de citire posterioară unei faze de scriere. De exemplu, modurile de adresare indirectă cu predecrementare pot introduce acest hazard, de aceea ele nici nu sunt implementate în arhitecturile de tip RISC.

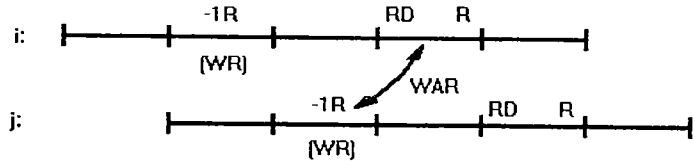


Fig. 2.17

De precizat că aceste **hazarduri WAR** (Write After Read), pot apare și datorită execuției instrucțiunilor în afara ordinii lor normale, din program (execuție Out of Order), după cum se va remarca în continuare. Această procesare Out of Order este impusă de creșterea performanței si se poate realiza atât prin mijloace hardware cât și software legat de optimizarea programelor pe arhitecturile pipeline.

2.4.4.2.3. HAZARDURI WAW. CONCLUZII LA HAZARDURI DE DATE

Hazardul WAW (Write After Write), apare atunci când instrucțiunea j scrie un operand înainte ca acesta să fie scris de către instrucțiunea i. Așadar, în acest caz scrierile s-ar face într-o ordine eronată. Hazardul WAW poate apărea în structurile care au mai multe nivele de scriere sau care permit unei instrucțiuni să fie procesată chiar dacă o instrucțiune anterioară este blocată. Modurile de adresare indirectă cu postincrementare pot introduce acest hazard.

De asemenea, acest hazard poate să apară în cazul execuției Out of Order a instrucțiunilor.

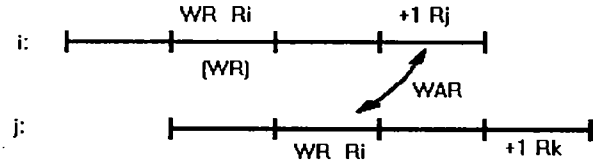


Fig. 2.18

Hazardurile de tip WAW sau WAR nu reprezintă hazarduri reale, ci mai degrabă conflicte de nume. Ele pot fi eliminate de către compilator prin redenumirea resurselor utilizate de program. De aceea se mai numesc dependențe de ieșire respectiv antidependențe.

Ex.: I1: MULF Ri, Rj, Rk ; Ri <--- (Rj) * (Rk)
I2: ADD Rj, Rp, Rm ; Rj <--- (Rp) + (Rm)

În acest caz poate să apară hazard WAR întrucât instrucțiunea I1 fiind o instrucțiune de coprocesor flotant se va încheia în execuție după I2 care este o instrucțiune de procesor cu operanzi întregi. Am presupus deci că numărul de nivele aferent structurii pipeline a coprocesorului este mai mare decât numărul de nivele aferent procesorului. Rezultă deci că instrucțiunile I1, I2 se termină "Out Of Order" (I2 înaintea lui I1). Secvența reorganizată prin software, care elimină hazardul WAR este :

MULF Ri, Rj, Rk
ADD Rx, Rp, Rm

MOV Rj, Rx Rj <- Rx (după Ri <- (Rj) x (Rk))

Prezentăm în continuare un exemplu de reorganizare a unui program în vederea eliminării hazardurilor de date și a procesării sale optimale:

I0 ADD R3, R1, R2
I1 LD R9, A(R7)
I2 ADD R4, R3, R2
I3 ADD R5, R4, R6
I4 LD R4, A(R6)
I5 LD R2, A(R4)

Graful dependențelor de date corespunzător acestei secvențe este următorul (Fig.2.19):

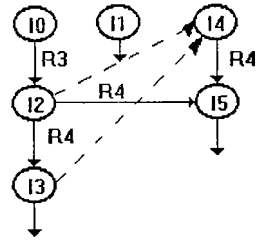


Fig. 2.19

Aparent, pe baza grafului de mai sus, secvența reorganizată ar starta astfel:

I0 ADD R3, R1, R2
I1 LD R5, A(R7)
I4 LD R4, A(R6)
I2 ADD R4, R3, R2
I3 ADD R5,R4,R6, etc.

Se observă în acest caz că execuția Out of Order a instrucțiunilor I4 și I2 determină o procesare eronată a programului prin hazardul de tip WAW prin registrul R4. De asemenea, între instrucțiunile I3 și I4 există hazard WAR prin același registru. Așadar detecția hazardului WAW între instrucțiunile I2 și I4 determină impunerea unei relații de precedență între aceste instrucțiuni, adică procesarea trebuie să se realizeze In Order.

În cazul nostru, este necesar deci ca I3, dependentă RAW de I2, să se execute înaintea instrucțiunii I4. Aceste restricții au fost evidențiate prin linie punctată în graful dependențelor de date, anterior prezentat.

Ținând cont de cele de mai sus, pe baza grafului dependențelor de date, secvența reorganizată corect va fi:

I0 ADD R3, R1, R2
I1 LD R9, A(R7)
I2 ADD R4, R3, R2
I3 ADD R5, R4, R6
I4 LD R4, A(R6)
NOP; Compensare LDS!
I5 LD R2, A(R4)

Rata de procesare în acest caz este de 6/7 instr./tact, performanță superioară programului inițial întrucât prin procesarea Out of Order, se evită la maximum dependențele de date între instrucțiuni. Hazardul WAR între I3 și I4 prin registrul R4 ar putea fi eliminat prin redenumirea registrului R4 în instrucțiunea I4.

De asemenea, relații suplimentare de precedență între instrucțiuni impun într-un

mod analog hazardurile de tip WAR (în ex. nostru între I3 şi I4). După cum am mai arătat, hazardurile WAR şi WAW nu reprezintă conflicte reale, ci doar aşa-zise **conflicte de nume**, ele nefiind considerate dependenţe adevărate. Considerând un procesor cu mai mulţi regiştri fizici decât cei logici, precedenţele impuse de aceste hazarduri pot fi eliminate uşor prin redenumirea regiştrilor logici cu cei fizici (**register renaming**). Principiul constă în existenţa unei liste a regiştrilor activi, adică folosiţi momentan, şi o alta a celor liberi (adică a celor nefolosiţi momentan). Fiecare schimbare a conţinutului unui registru logic prin program, se va face asupra unui registru fizic disponibil în lista regiştrilor liberi şi care va fi trecut în lista regiştrilor activi. În acest caz secvenţa în discuţie s-ar transforma în următoarea:

I0: ADD R3a, R1a, R2a
I1: LD R9a, A(R7a)
I2: ADD R4a, R3a, R2a
I3: ADD R5a, R4a, R6a
I4: LD R4b, A(R6a)
I5: LD R2b, A(R4b)

În acest caz în optimizare nu ar mai avea importanţă decât dependenţele RAW, celelalte fiind eliminate. În baza grafului dependenţelor de date procesarea optimală ar însemna în acest caz: I0, I1, I4, I2, I3, I5, etc.

Să observăm că deocamdată am prezentat exclusiv problema reorganizării aşa numitelor "basic-block"-uri sau unităţi secvenţiale de program.

Se numeşte **basic-block** o secvenţă de instrucţiuni cuprinsă între două instrucţiuni de ramificaţie, care nu conţine nici o instrucţiune de ramificaţie şi care nu conţine nici o instrucţiune destinaţie a unei instrucţiuni de ramificaţie.

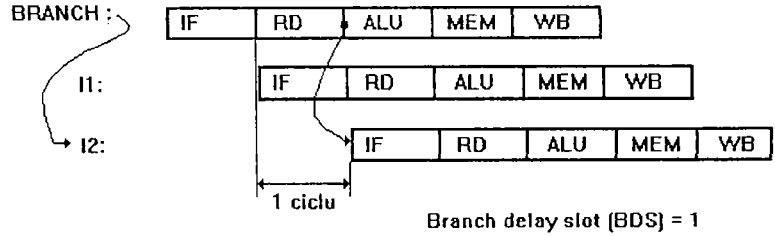
Optimizarea programelor in basic - block-uri e rezolvată. Din păcate, prin concatenarea mai multor basic- block-uri optimizate nu se obţine un program optimizat. Se pune deci problema unei **optimizări globale**, a întregului program (vom reveni în capitolul următor).

2.4.4.3 HAZARDURI DE RAMIFICAŢIE (HR): PROBLEME IMPLICATE ŞI SOLUŢII

Pot fi generate de către instrucţiunile de ramificaţie. Cauzează pierderi de perfomanţă în general mai importante decât hazardurile structurale şi de date. Efectele defavorabile ale instrucţiunilor de ramificaţie (branch) 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) şi calculează în avans noul PC (program counter). Diverse statistici arată că instrucţiunile de salt necondiţionat au o frecvenţă între

2 - 8% din instrucţiunile unui program, iar cele de salt condiţionat între 11 - 17%. S-a arătat că salturile condiţionate simple se fac cu o probabilitate de cca. 50%, loop-urile cu o probabilitate de cca. 90%, iar majoritatea salturilor orientate pe bit nu se fac [Mil89].

Considerând o structură pipeline pe 5 nivele şi că la finele nivelului RD adresa de salt este disponibilă, efectul defavorabil al unei instrucţiuni de salt este sugerat în figura 2.20.



Branch delay slot (BDS) = 1

Daca saltul se va face, atunci I1 se va executa in mod nedorit.

Fig. 2.20

Evident că în alte structuri BDS-ul poate fi mai mare ca 1.

O primă soluţie ar fi aceea de a dezvolta unitatea de control hardware în vederea detectării prezenţei saltului şi de a întârzia procesarea instrucţiunilor următoare cu un număr de cicli egal cu BDS-ul, până când adresa de salt e disponibilă. Soluţia implică reducerea performanţelor. Acelaşi efect l-ar avea şi "umplerea" BDS-ului cu instrucţiuni NOP.

2.4.4.3.1. STRATEGII SOFTWARE DE ELIMINARE A H.R. ÎN STRUCTURILE PIPELINE

Se bazează pe reorganizarea secvenţei de instrucţiuni maşină, astfel încât efectul defavorabil al salturilor să fie eliminat. În continuare vom prezenta o astfel de strategie atribuită lui Gross şi Hennessy. Mai întâi se defineşte o instrucţiune de salt situată la adresa b spre o instrucţiune situată la adresa l, ca un **salt întârziat de ordinul n,** dacă instrucţiunile situate la locaţiile b, b + 1, ..., b + n şi l sunt executate întotdeauna când saltul se face. O primă soluţie de optimizare ar fi aceea de mutare a instucţiunii de salt cu n instrucţiuni "în sus". Acest lucru e posibil doar atunci când nici una dintre precedentele n instrucţiuni nu afectează determinarea condiţiilor de salt. În general, se poate muta instrucţiunea de salt cu doar k instrucţiuni "în sus" unde k reprezintă numărul maxim de instrucţiuni anterioare care nu afectează condiţiile de salt, $k \in (0, n]$. Când acest lucru se

întâmplă , se poate aplica succesiv o altă tehnică ce constă în duplicarea primelor (n - k) instrucţiuni plasate începând de la adresa destinaţie a saltului, imediat după instrucţiunea de salt şi modificarea corespunzătoare a adresei de salt la acea adresă care urmează imediat după cele (n - k) instrucţiuni "originale".

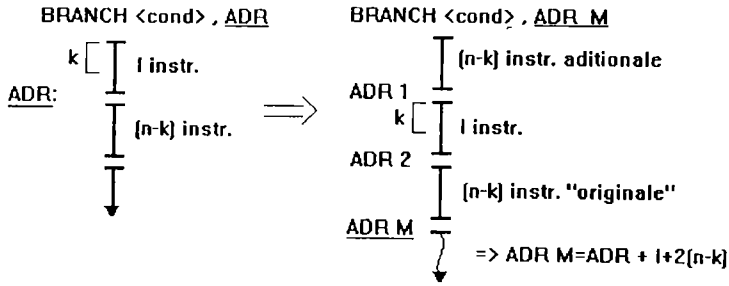


Fig. 2.21

În cazul în care saltul condiţionat nu se execută, e necesar ca nici una dintre cele (n - k) instrucţiuni adiţionale să afecteze execuţia următoarelor (l + n - k) instrucţiuni. Metoda se recomandă atunci cînd saltul se face predominant. În acest caz se realizează un câştig de performanţă în schimb se suplimentează spaţiul de memorare al programului. De remarcat că aceste microprocesoare nu au în general registru de flaguri şi deci salturile condiţionate nu se fac pe baza acestor flaguri ci pe baza unei condiţii specificate explicit în opcode-ul instrucţiunii de salt.

În continuare se prezintă un exemplu concret de reorganizare în vederea eliminării efectului BDS- ului, considerând însă că BDS- ul instrucţiunii de salt este doar de l tact.

a)

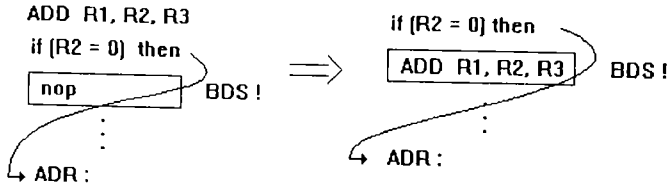


Fig. 2.22

Îmbunătăţeşte rata de execuţie intotdeauna. Dacă această reorganizare nu e posibilă, se încearcă una dintre următoarele 2 variante ((b) sau (c)).

b)

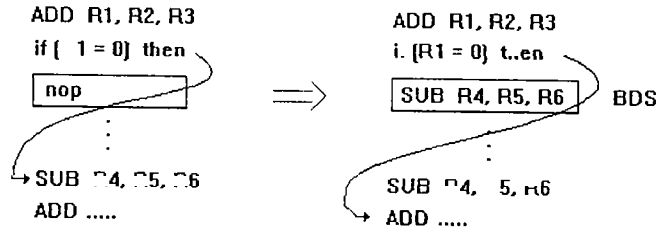


Fig. 2.23

Rata de execuţie creşte atunci cînd saltul condiţionat se face. Dacă saltul NU se face, trebuie ca instrucţiunea introdusă în BDS (SUB R4, R5, R6), să nu provoace execuţia eronată a ramurii de program respective. Din păcate, măreşte zona de cod oricum şi în plus necesită timp de execuţie sporit cu o instrucţiune adiţională în cazul în care saltul nu se face.

c)

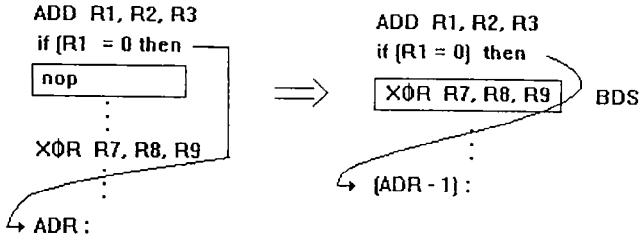


Fig. 2.24

Creşte performanţa atunci când saltul condiţionat NU se face. E necesar ca instrucţiunea introdusă în BDS (aici XOR), să NU provoace execuţia eronată a ramurii de program în cazul în care saltul se face.

În concluzie se urmăreşte "umplerea" BDS-ului cu instrucţiuni utile şi care să nu afecteze programul din punct de vedere logic.

În general microprocesoarele RISC, care deţin un BDS de una-două instrucţiuni, au posibilitatea - printr-un bit din codul instrucţiunii de salt condiţionat - să introducă stagnări hardware în cazul salturilor condiţionate sau să se bazeze pe umplerea BDS-ului cu instrucţiuni NOP sau cu alte instrucţiuni utile de către reorganizator (scheduler).

2.4.4.3.2 STRATEGII HARDWARE DE PREDICŢIE A BRANCH-URILOR ÎN PROCESOARELE PIPELINE SCALARE

Aceste strategii hardware de predicţie a branch-urilor au la bază **predicţia prin hardware** a ramurii de salt condiţionat precum şi determinarea în avans a noului PC.

Cercetări recente insistă pe această problemă, întrucât s-ar elimina necesitatea reorganizărilor soft ale programului sursă şi deci s-ar obţine o independenţă faţă de maşină.

O metodă consacrată în acest sens o constituie metoda **"branch prediction buffer" (BPB).** BPB-ul reprezintă o mică memorie adresată cu cei mai puţin semnificativi biţi ai PC-ului aferent unei instrucţiuni de salt condiţionat.

Cuvântul BPB este constituit în principiu dintr-un singur bit. Dacă e 1, atunci se prezice că saltul se va face, iar dacă e 0, se prezice că saltul nu se va face. Evident că nu se poate şti în avans dacă predicţia este corectă. Oricum, structura va considera că predicţia este corectă şi va declanşa aducerea instrucţiunii următoare de pe ramura prezisă. Dacă predicţia se dovedeşte a fi falsă structura pipeline se evacuează şi se va iniţia procesarea celeilale ramuri de program. Totodată, valoarea bitului de predicţie din BPB se inversează.

BPB cu un singur bit are un dezavantaj care se manifestă cu precădere în cazul buclelor de program ca cea din figură, în care saltul se va face (N - 1) ori şi odată, la ieşirea din buclă nu se face. Bazat pe tehnica BPB în acest caz vom avea uzual 2 predicţii false: una la intrarea în buclă (prima parcurgere) şi alta la ieşirea din buclă (ultima parcurgere a buclei).

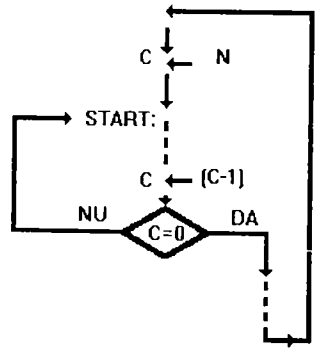


Fig. 2.25

Aşadar, acurateţea predicţiei va fi de (N - 2) * 100 / N %, iar saltul se face în proporţie de (N - 1) * 100 / N %. Pentru a elimina acest dezavantaj se utilizează 2 biţi de predicţie modificabili conform grafului de tranziţie de mai jos (numărător saturat). În acest caz acurateţea predicţiei unei bucle care se face de (N - 1) ori va fi (N - 1) * 100 / N %.

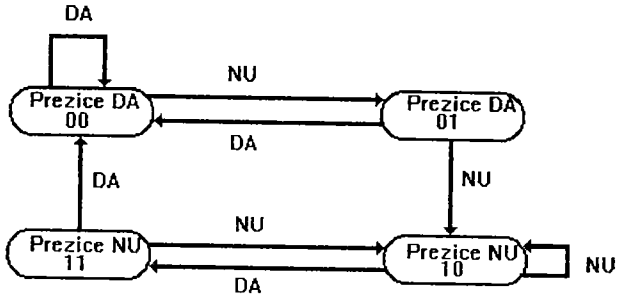


Fig. 2.26

Prin urmare, în cazul în care se prezice că branch-ul se va face, aducerea noii instrucțiuni se face de îndată ce conținutul noului PC e cunoscut. În caz contrar, se evacuează structura pipeline și se atacă cealaltă ramură a instrucțiunii de salt. Totodată, biții de predicție se modifică în conformitate cu graful din figură numit și numărător saturat. (Fig.2.26)

Probabilitatea p ca predicția să fie corectă pentru o instrucțiune de salt, e dată de relația:

$$p = p1 * p2 + (1-p2) * p3, \qquad (2.5.)$$ unde:

p1, p2 - probabilitatea ca predicția adresată în BPB să fie corectă și să se refere la respectiva instrucțiune de salt;

(1-p2)*p3 - probabilitatea ca predicția să fie corectă, deși nu se referă la instrucțiunea în curs de aducere. (Există .posibilitatea ca 2 instrucțiuni de branch distincte să aibă cei mai puțini semnificativi biți ai PC-ului identici).

Este evident că maximizarea probabilității P se obține prin maximizarea probabilităților p1, p2 (p3 nu depinde de caracteristicile BPB-ului).

O altă problemă delicată constă în faptul că deși predicția poate fi corectă, de multe ori adresa de salt (noul PC) nu este disponibilă în timp util, adică la finele fazei IF. Timpul necesar calculului noului PC generează HR și are deci un efect defavorabil asupra ratei de procesare. Soluția la această problemă este dată de metoda de predicție numită **"branch target buffer"** (BTB).

Un BTB este constituit dintr-un BPB care conține pe lângă biții de predicție, noul PC de după instrucțiunea de salt condiționat și eventual alte informații. De exemplu, un cuvânt din BTB ar putea conține și instrucțiunea țintă. Astfel ar crește performanța, nemaifiind necesar un ciclu de aducere a acestei instrucțiuni, dar în schimb ar crește costurile de implementare.Diferența esențială între memoriile BPB și BTB constă în faptul că prima e o memorie operativă iar a 2-a asociativă.

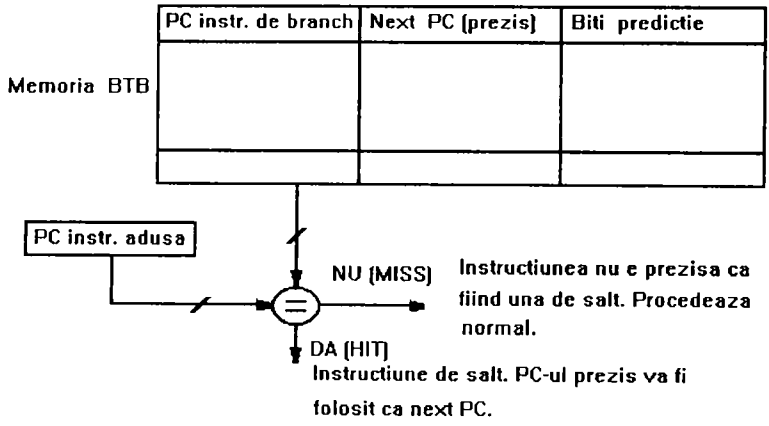


Fig. 2.27

La începutul fazei IF se declanşează o căutare asociativă în BTB după conţinutul PC-ului în curs de aducere. În cazul în care se obţine hit se obţine în avans PC-ul aferent instrucţiunii următoare. Mai precis, considerând o structură pipeline pe 3 faze (IF, RD, EX) algoritmul de lucru cu BTB-ul este următorul [Hen96]:

IF) Se trimite PC-ul instrucţiunii ce urmează a fi adusă spre memorie şi spre BTB. Dacă PC-ul trimis corespunde cu un PC din BTB se trece în pasul RD 2, altfel în pasul RD1.

RD1) Dacă instrucţiunea adusă e o instrucţiune de branch, se trece în pasul EX 1, altfel se continuă procesarea normală.

RD2) Se trimite PC-ul prezis din BTB spre memoria de instrucţiuni. În cazul în care condiţiile de salt sunt satisfăcute, se trece în pasul EX 3, altfel în pasul EX 2.

EX1) Se introduce PC-ul instrucţiunii de salt precum şi PC-ul prezis în BTB. De obicei această alocare se face în locaţia cea mai de demult neaccesată (last recently used) LRU.

EX2) Predicţia s-a dovedit eronată. Trebuie reluată faza IF de pe cealaltă ramură.

EX3) Predicţia a fost corectă, însă numai dacă PC-ul predicţionat este într-adevăr corect, adică neschimbat.În acest caz, se continuă execuţia normală.

În tabelul următor sunt rezumate avantajele şi dezavantajele tehnicii BTB.

Instr. în BTB ?	Predicţie	Realitate	Cicli penalizare
Da	Da	Da	0(Ctt)
Da	Da	Nu	Ctn
Da	Nu	Nu	0
Da	Nu	Da	Cnt
Nu	-	Da	Ct
Nu	-	Nu	0

Tab. 2.2

Rezultă că numărul de cicli de penalizare CP este dat de următoarea relație:

$$CP = P_{BTB} (P_{tn} * C_{tn} + P_{nt} * C_{nt}) + (1 - P_{BTB}) * P * C_t , \qquad (2.6.),$$ unde:

P_{BTB} - probabilitatea ca instrucțiunea de salt să se afle în BTB;
P_{tn} - probabilitatea ca saltul să fie prezis că se face și în realitate nu se face;
P_{nt} - probabilitatea ca saltul să fie prezis că nu se face și în realitate se va face;
P - probabilitatea ca respectiva instrucțiune de salt să se facă;

Această relație, justificată în [Hen96], necesită amendamente serioase pentru a fi utilă și exactă, după cum se va arăta într-o contribuție ulterioară a acestei lucrări. În acest caz, rata de procesare a instrucțiunilor ar fi dată de relația:

$$IR = 1/(1 + P_b * CP), \text{ [instr./tact]}, \qquad (2.7.)$$

unde P_b= probabilitatea ca instrucțiunea curentă sa fie una de ramificație.

Un model matematic simplu al acestei tehnici pentru un BTB cu N biți de memorare, se referă la maximizarea funcției [Per93] :

$$F = \sum_{i=1}^{s} P_{ex}(i) [P(i) * P_{tt}(i) * V(i) - (1 - P(i))* P_{tn}(i)*W(i)] \qquad (2.8.)$$

astfei încât $\sum_{i=1}^{s} n(i) \le N$, unde :

n(i) - numărul de biți din BTB alocat instrucțiunii de branch i;
N - numărul total de biți din BTB;
S - numărul de instrucțiuni branch procesate în cadrul programului;

Relativ la expresia funcției F avem următorii termeni:

$P_{ex}(i)$ - probabilitatea ca branch-ul i să se execute în cadrul programului ;
P(i) - probabilitatea ca branch-ul i să se facă;
$P_{tt}(i)$ - probabilitatea ca branch-ul i să fie prezis că se face și într-adevăr se va face;
V(i) - numărul de cicli economisiți în cazul unei predicții corecte a branch-ului i;
W(i) - numărul de cicli de penalizare în cazul unei predicții incorecte a branch-ului i;

Obs.1) $P_{tt}(i) = P_{tn}(i) = 0$, dacă branch-ul i nu se află în BTB.

Obs.2) S-a considerat că BTB nu îmbunătățește performanța pentru o predicție corectă de tipul "saltul nu se face" ($P_{nn}(i) = 0$), întrucât în acest caz structura se comportă la fel ca și o structură fără BTB. De asemenea, pentru o predicție incorectă a faptului că "saltul se face" , am considerat costul același cu cel pe care l-am avea fără BTB; din acest

motiv $P_{nt}(i)$ nu intră în expresia funcției.

Obs.3) Un branch trebuie introdus în BTB, **cu prima ocazie când el se va face**. Un salt care ar fi prezis că nu se va face nu trebuie introdus în BTB pentru că nu are potențialul de a îmbunătăți performanța (nu intră în expresia funcției F). Există strategii care atunci când trebuie evacuat un branch din BTB îl evacuează pe cel cu potențialul de performanță minim, care nu coincide neapărat cu cel mai puțin folosit (vezi [Dub91, Per93]). Astfel în [Per93] se construiește câte o variabilă MPP (Minimum Performance Potential), implementată în hardware , asociată fiecărui cuvânt din BTB. Evacuarea din BTB se face pe baza MPP-ului minim. Acesta se calculează ca un produs între probabilitatea ca un branch din BTB să fie accesat și respectiv probabilitatea ca saltul să se facă. Minimizarea ambilor factori duce la minimizarea MPP-ului și deci la evacuarea respectivului branch din BTB, pe motiv că potențialul său de performanță este minim.

În literatură se arată că prin astfel de scheme se ajunge la predicții corecte în (80-90)% din cazuri. Există implementări de mare performanță în care biții de predicție sunt gestionați și funcție de "istoria" respectivei instrucțiuni de salt, pe baze statistice (INTEL NEX GEN, TRON, etc). Prin asemenea implementări crește probabilitatea de predicție corectă a branch-ului.

În literatură [Hen96, Dub91, Per93], bazat pe testări laborioase, se arată că se obțin predicții corecte în 88% din cazuri folosind un bit de predicție și respectiv în 93% din cazuri folosind 16 biți de predicție. Acuratețea predicțiilor crește asimptotic cu numărul biților de predicție utilizați, adică practic cu "istoria predicției". Se arată că pentru a obține performanțe satisfăcătoare sunt necesare predicții corecte în peste 97% din cazuri [Yeh92].

Schema de predicție pe 4 stări din figura 3.27 poate fi generalizată ușor la $N = 2^k$ stări. Se poate arăta că există $N^{2N} * 2^N$ (stări x ieșiri) automate distincte de predicție cu N stări, deși multe dintre acestea sunt triviale din punct de vedere al predictiilor salturilor.

În [Nai95] se arată într-un mod elegant, pe bază teoretică și de simulare, că schema din fig. 2.26 este cvasioptimală în mulțimea acestor automate de predicție. După cum vom arăta, prin scheme de predicție corelată a salturilor se pot obține performanțe superioare.

În acord cu literatura de specialitate, mărirea numărului N de stări al automatului de predicție pe k biți nu conduce la creșteri semnificative ale performanței.

2.4.4.3.2.1. PREDICȚIA CORELATĂ A RAMIFICAȚIILOR

Schemele de predicție anterior prezentate se bazau pe comportarea recentă a unei instrucțiuni de salt, de aici predicționându-se comportarea viitoare a acelei instrucțiuni de

salt. Este posibilă îmbunătățirea acurateței 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** și au fost introduse pentru prima dată în 1992 de către Yeh și Patt [Hen96, Yeh92].

Să considerăm pentru o primă exemplificare a acestei idei o secvență de program C extrasă din banchmark-ul Eqntott din cadrul grupului SPEC 92:

```
(b1)   if (x = = 2)
           x = 0;
(b2)   if (y = = 2)
           y = 0;
(b3)   if (x ! = y) {
```

Se observă imediat că în acest caz dacă salturile b1 și b2 nu se vor face atunci saltul b3 se va face sigur (x = y = 0). Așadar saltul b3 nu depinde de comportamentul său anterior ci de comportamentul anterior al salturilor b1 și b2, fiind deci corelat cu acestea. Evident că în acest caz schemele de predicție anterior prezentate nu vor da randament.

Să considerăm acum pentru analiză o secvență de program C simplificată împreună cu secvența obținută în urma compilării (s-a presupus că ...iabila x este asignată registrului R1).

```
if     (x = = 0)          (b1)   BNEZ  R1, L1
       x = 1;                    ADD   R1, R0, #1
if     (x = = 1)          L1:    SUB   R3, R1, #1
                          (b2)   BNEZ  R3, L2
```

Se poate observa că dacă saltul condiționat b1 nu se va face, atunci nici b2 nu se va face, cele 2 salturi fiind deci corelate.

Vom particulariza secvența anterioară, considerând iterații succesive ale acesteia pe parcursul cărora x variază de exemplu între 0 și 5. Un BPB clasic, inițializat pe predicție NU, având un singur bit de predicție, s-ar comporta ca în tabelul 2.3.

x	Predictie b1	Actiune b1	Predictie noua b1	Predictie b2	Actiune b2	Predictie noua b2
5	NU	DA	DA	NU	DA	DA
0	DA	NU	NU	DA	NU	NU
5	NU	DA	DA	NU	DA	DA
0	DA	NU	NU	DA	NU	NU

Tab. 2.3

Aşadar o astfel de schemă ar predicţiona în acest caz, întotdeauna greşit!

Să analizăm acum comportarea unui predictor corelat având un singur bit de corelaţie (se corelează deci cu instrucţiunea de salt anterior executată) şi un bit de predicţie. Acesta se mai numeşte şi predictor corelat (1, 1). Acest predictor va avea 2 biţi de predicţie pentru fiecare instrucţiune de salt: primul bit predicţionează dacă instrucţiunea de salt actuală se face sau nu, în cazul în care instrucţiunea anterior executată nu s-a făcut iar al doilea analog, în cazul în care instrucţiunea de salt anterior executată s-a făcut. Există deci următoarele 4 posibilităţi.

Biţi predicţie	Predicţie dacă precedentul salt nu s-a făcut	Predicţie dacă precedentul salt s-a făcut
NU / NU	NU	NU
NU / DA	NU	DA
DA / NU	DA	NU
DA / DA	DA	DA

Tab. 2.4

Ca şi în cazul BPB-ului clasic cu un bit, în cazul unei predicţii care se dovedeşte a fi eronată bitul de predicţie indicat se va complementa.

Comportarea predictorului (1,1) pe secvenţa anterioară de program este prezentată în continuare (s-a considerat că biţii de predicţie asociaţi salturilor b1 şi b2 sunt iniţializaţi pe NU / NU).

x	Predictie b1	Actiune b1	Predictie noua b1	Predictie b2	Actiune b2	Predictie noua b2
5	**NU**/NU	DA	DA/NU	NU/**NU**	DA	NU/DA
0	DA/**NU**	NU	DA/NU	**NU**/DA	NU	NU/DA
5	**DA**/NU	DA	DA/NU	NU/**DA**	DA	NU/DA
0	DA/**NU**	NU	DA/NU	**NU**/DA	NU	NU/DA

Tab. 2.5

După cum se observă în tab.2.5., singurele două predicţii incorecte sunt când x = 5 în prima iteraţie. În rest, predicţiile vor fi întotdeauna corecte, schema comportându-se deci foarte bine spre deosebire de schema BPB clasică.

În cazul general, un predictor corelat de tip (m,n) utilizeaza comportarea precedentelor m instrucţiuni de salt executate, alegând deci o anumită predicţie de tip Da sau Nu din 2^m iar n reprezintă numărul biţilor utilizaţi în predicţia fiecărui salt.

Un alt avantaj al acestor scheme este dat de simplitatea implementării hardware,

cu puţin mai complexă decât cea a unui BPB clasic. Aceasta se bazează pe simpla observaţie că "istoria" celor mai recent executate m salturi din program, poate fi memorată într-un registru binar de deplasare pe m ranguri (registru de predicţie). Aşadar adresarea cuvântului de predicţie format din n biţi şi situat într-o tabelă de predicţii, se poate face foarte simplu prin concatenarea c.m.p.s. biţi ai PC-ului instrucţiunii de salt curente cu acest registru de deplasare în adresarea BPB-ului de predicţie. Ca şi în cazul BPB-ului clasic, un anumit cuvânt de predicţie poate corespunde la mai multe salturi. Există în implementare 2 nivele deci: un **registru de predicţie** al cărui conţinut concatenat cu PC- ul c.m.p.s. al instrucţiunii de salt, pointează la un cuvânt din **tabela de predicţii** (aceasta conţine biţii de predicţie, adresa destinaţie, etc.). În [Yeh92], nu se face concatenarea PC- registru de predicţie şi în consecinţă se obţin rezultate nesatisfăcătoare datorită **interferenţei diverselor salturi** la aceeaşi locaţie din tabela de predicţii, lucru constatat şi eliminat de noi pri simulări proprii.

De remarcat că un BPB clasic reprezintă un predictor de tip (0,n), unde n este numărul biţilor de predicţie utilizaţi.

Numărul total de biţi utilizaţi în implementarea unui predictor corelat de tip (m,n) este:

$$N = 2^m * n * NI, \qquad (2.9.)$$

unde NI reprezintă numărul de intrări al BPB-ului utilizat.

O comparare echitabilă între schemele de predicţie clasice şi cele corelate trebuie să impună acelaşi număr de biţi utilizaţi în implementarea celor 2 scheme de comparat. Aşa de exemplu în [Hen96] se compară un predictor (0,2) de capacitate 4k cu predictor (2,2) de capacitate 1k. Acurateţea predicţiilor schemei corelate este clar mai bună. Simulările s-au făcut pe procesorul DLX bazat pe 10 banchmark-uri SPEC 92. Schema corelată a obţinut predicţii corecte în 82%-100% din cazuri. Mai mult, predictorul (2,2) obţine rezultate superioare în comparaţie cu un BPB având un număr infinit de locaţii.

O altă soluţie constă în aducerea instrucţiunilor din cadrul ambelor ramuri ale branch-ului în structuri pipeline paralele (**multiple instructions streams**). Când condiţia de salt e determinată, una din ramuri se va abandona. Desigur că în acest caz sunt necesare redundanţe ale resurselor hard precum şi complicaţii în logica de control. Dacă pe o ramură a programului există de ex. o instrucţiune de tip STORE, procesarea acestei ramuri trebuie oprită întrucât există posibilitatea unei alterări ireparabile a unei locaţii de memorie. Această soluţie implică creşteri serioase ale costurilor, dar, se pare că ar fi singura capabilă să se apropie oricât de mult faţă de idealul predicţiei absolut corecte. În cazul microprocesoarelor, aceste mecanisme de prefetch a ambelor ramuri, nu se aplică în prezent decât în cazuri rare, în principal datorită lărgimii de bandă limitate între

microprocesor şi memorie. Tehnica se întâlneşte frecvent în cazul supercomputerelor.

Aceste tehnici de predicţie hardware a branch-urilor, datorită complexităţii lor, nu sunt implementate în mod uzual în microprocesoarele RISC scalare, întrucât se preferă tehnicile software de "umplere" a BDS-ului (limitat în general la o instrucţiune) cu instrucţiuni utile, anterioare celei de salt. În schimb, predicţia hardware este implementată în cazul unor procesoare superscalare, unde datorită BDS-ului de câteva instrucţiuni, umplerea lui cu instrucţiuni anterioare independente devine practic imposibilă.

2.4.5 PROBLEMA EXCEPŢIILOR ÎN ARHITECTURILE PIPELINE DE TIP RISC

La sesizarea unui eveniment de excepţie se vor inhiba toate procesele de scriere, atât pentru instrucţiunea care a provocat excepţia, cât şi pentru următoarele aflate în bandă. Aceasta previne orice modificare a contextului procesorului care ar putea fi cauzată de continuarea procesării acestor instrucţiuni. În principiu, după terminarea instrucţiunii anterioare celei care a provocat excepţia, se intră în protocolul de tratare, în cadrul căruia se salvează intern sau extern PC-ul instrucţiunii care a provocat excepţia, precum şi contextul procesorului.

În particular, în cazul în care instrucţiunea care a provocat excepţia se află într-un BDS de ordinul n şi saltul se face, atunci trebuie reluate cele n instrucţiuni BDS, precum şi instrucţiunea la care se face saltul. În acest caz trebuie salvate (n + 1) PC-uri pentru că în general adresele instrucţiunilor din BDS şi respectiv adresa instrucţiunii la care se face saltul nu sunt contigue.

Dacă în cazul unei excepţii structura poate fi oprită astfel încât instrucţiunile anterioare celei care a provocat excepţia să poată fi complet executate şi respectiva instrucţiune împreună cu cele ulterioare ei să poată fi reluate în condiţii deterministe, se zice că avem o **excepţie precisă**. În caz contrar excepţia se zice **imprecisă**. Exemplu de excepţie imprecisă :

```
DIVF  F0, F2, F4
ADDF  F6, F6, F8
SUBF  F10, F10, F14
```

În acest caz instrucţiunile se vor termina Out of Order, adică ADDF şi SUBF se vor termina înaintea instrucţiunii DIVF. Să presupunem că instrucţiunea DIVF a determinat o derută aritmetică într-un moment în care ADDF şi SUBF s-au încheiat. Această situaţie implică o excepţie imprecisă, întrucât reluarea instrucţiunii DIVF se va face cu conţinutul regiştrilor F6 şi F10 alterat. Aceste situaţii sunt evident nedorite, iar dacă apar trebuie eliminate. Relativ la excepţiile imprecise, în literatură se precizează următoarele posibilităţi de soluţionare :

a) Contextul CPU să fie dublat printr-un aşa-numit **"history-file"**, care să păstreze toate resursele modelului de programare. În acest history-file se înscriu noile rezultate la finele terminării "normale" (pur secvenţiale) a instrucţiunilor. În cazul apariţiei unei excepţii imprecise contextul procesorului se va încărca din acest context de rezervă (CYBER 180 / 990). Există şi alte variaţiuni pe această idee.

b) Prin această soluţie nu se permite terminarea unei instrucţiuni în bandă, până când toate instrucţiunile anterioare nu se vor fi terminat fără să cauzeze o excepţie. Astfel se garantează că dacă a apărut o excepţie în cadrul unei instrucţiuni, nici o instrucţiune ulterioară acesteia nu s-a încheiat şi totodată instrucţiunile anterioare ei s-au încheiat normal. Soluţia implică întârzieri ale procesării (MIPS R 2000 / 3000).

O altă problemă o constituie **excepţiile simultane**. Dacă luăm în considerare o procesare pe 5 nivele, în cadrul fiecărui nivel pot apare următoarele excepţii :

IF - derută accesare pagină memorie, acces la un cuvânt nealiniat, etc.

RD - cod ilegal de instrucţiune

EX - diverse derute aritmetice (overflow)

MEM - ca şi la IF

WB - acces la resurse privilegiate în modul de lucru user.

Rezultă imediat posibilitatea apariţiei simultane a 2 sau mai multe evenimente de excepţie.

Să considerăm secvenţa de instrucţiuni din figură, în cadrul căreia apar simultan două excepţii:

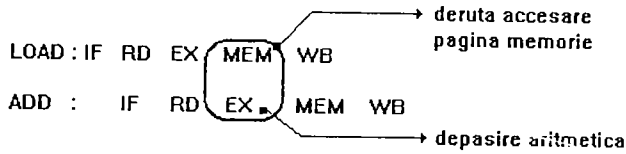


Fig. 2.28

Soluţia ar consta în tratarea prioritară a derutei instrucţiunii LOAD, după care se va relua această instrucţiune. Apoi va apare deruta de depăşire aferentă instrucţiunii ADD care va fi şi ea tratată.

Un caz mai dificil este acela în care excepţiile apar Out of Order ca în exemplul de mai jos :

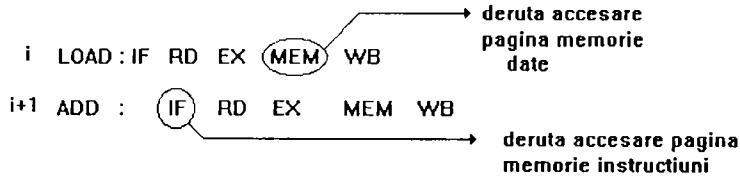


Fig. 2.29

În acest caz ar fi posibile 2 soluţii de principiu :

1) Să existe un flag de stare excepţie aferent fiecărei instrucţiuni şi care să fie testat la intrarea în nivelul WB. Dacă există setată vreo excepţie, se va trece în protocolul de tratare. Astfel se garantează că toate excepţiile din cadrul unei anumite instrucţiuni vor fi văzute înaintea excepţiilor apărute pe parcursul unei instrucţiuni următoare.

2) Se bazează pe tratarea excepţiei de îndată ce aceasta a apărut

La sesizarea derutei din cadrul instrucţiunii (i + 1) se vor inhiba instrucţiunile (i - 2), (i - 1), i, (i + 1) şi prin protocolul de tratare se va relua instrucţiunea (i - 2). Apoi se va sesiza deruta din cadrul instrucţiunii i urmând ca după tratarea ei instrucţiunea i să se reia. Evident că deruta aferentă nivelului IF din cadrul instrucţiunii (i + 1) a fost anterior eliminată şi deci nu va mai apare. Menţionăm că majoritatea microprocesoarelor RISC deţin suficiente resurse hardware interne care să le permită în cazul apariţiei unei excepţii salvarea internă a contextului CPU. Evident că limitarea resurselor interne nu implică limitarea posibilităţii de imbricare a excepţiilor.

Ca şi procesoarele CISC, procesoarele RISC deţin regiştri de stare excepţie, regiştri care conţin descrierea evenimentului de excepţie curent, regiştri care memorează adresa virtuală care a cauzat o excepţie, etc..

2.4.6 AMBIGUITATEA REFERINŢELOR LA MEMORIE (ANALIZA ANTIALIAS)

Dependenţele cauzate de variabilele aflate în memorie reprezintă o frână în calea obţinerii performanţei. Pentru exemplificare să considerăm secvenţa de program:

ST 4 (Ri), R1

LD R2, 8 (Rj)

După cum deja am arătat, există motive ca instrucţiunea LD să se execute înaintea instrucţiunii ST din motive de eficienţă a execuţiei. Acest lucru este posibil numai dacă cele 2 adrese de memorie sunt întotdeauna diferite. Este evident că dacă la un anumit moment ele sunt identice, semantica secvenţei se modifică inacceptabil.

În general această problemă se rezolvă static, de către compilator, atunci când acest lucru e posibil. O componentă a acestuia ("disambiguating routine") compară cele 2 adrese de memorie şi returnează una dintre următoarele 3 posibilităţi:

a) adrese întotdeauna distincte;

b) adrese întotdeauna identice;

c) cel puțin 2 adrese identice sau nu se poate determina.

Așadar, doar în primul caz putem fi siguri că execuția anterioară a instrucțiunii LD față de instrucțiunea ST (sau simultană în cazul unui procesor MEM- Mașină cu Execuție Multiplă , vezi capitolul 3) îmbunătățește performanța fără a cauza alterarea semantică a programului. Din păcate, nu se poate decide întotdeauna acest lucru în momentul compilării.

Dezambiguizarea statică dă rezultate bune în cazul unor adresări liniare și predictibile ale memoriei (de ex. accesări de tablouri, matrici, etc.). Ea presupune rezolvarea unor ecuații diofantice mai mult sau mai puțin complexe, similare cu cele necesare vectorizării buclelor de program [Sto93].

Prin urmare un reorganizator de program bazat pe dezambiguizarea statică va fi deosebit de conservativ în acțiunile sale.

Dacă această comparare a adreselor de memorie se face pe parcursul procesarii programului prin hardware, se zice că avem o dezambiguizare dinamică. Aceasta este mai performantă decât cea statică dar necesită resurse hardware suplimentare și deci costuri sporite[Ste96, Nic89, Hua94]].

Pentru a pune în evidență performanța superioară a variantei dinamice, să considerăm secvența:

```
for i = 1 to 100 do
        a[ 2i ]=....
        y =  f(..., a[i+4], ...)
end
```

Într-un singur caz din 100 (i = 4), cele 2 referințe la memorie a[2i] respectiv a[i + 4] sunt identice. Așadar, o dezambiguizare statică va fi conservativă, nepermițând optimizarea buclei deși doar in 99% din cazuri acest lucru este posibil. O variantă dinamică însă, va putea exploata mai eficient acest fapt. Pe un procesor superscalar sau VLIW acest lucru este și mai avantajos întrucât cele 2 operații din buclă se vor putea realiza simultan.

Se consideră că progresele în această problemă pot duce la creșteri semnificative de performanță în domeniul paralelismului la nivelul instrucțiunilor, după cum vom demonstra și noi într-o contribuție ulterioară.

2.4.7 EXECUȚIA CONDIȚIONATĂ ȘI SPECULATIVĂ

Execuția condiționată se referă la implementarea unor așa numite instrucțiuni condiționate. O instrucțiune condiționă se va executa dacă o variabilă de condiție inclusă

în corpul instrucțiunii îndeplinește condiția dorită. În caz contrar, instrucțiunea respectivă nu va avea nici un efect (NOP).

Variabila de condiție poate fi memorată într-un registru general al procesorului sau în regiștri special dedicați acestui scop numiți regiștri booleeni.

Astfel de exemplu, instrucțiunea CMOVZ R1, R2, R3 mută (R2) în R1 dacă (R3) = 0. Instrucțiunea TB5 FB3 ADD R1, R2, R3 execută adunarea numai dacă variabilele booleene B5 și B3 sunt '1' respectiv '0'. În caz contrar, instrucțiunea este inefectivă. Desigur că variabilele booleene necesită biți suplimentari în corpul instrucțiunii.

Execuția condiționată a instrucțiunilor este deosebit de utilă în eliminarea salturilor condiționate dintr-un program, simplificând programul și transformând deci hazardurile de ramificație în hazarduri de date. Să considerăm spre exemplificare o construcție **if-then-else** ca mai jos:

```
if      (R8<1)              LT   B6, R8, #1;    if R8<1, B6<---1
        R1 = R2 + R3,       BF   B6, Adr1;      Dacă B6=0 salt la Adr1
else                        ADD  R1, R2, R3
        R1 = R5 - R7;       BRA  Adr2 ; salt la Adr2
R10 = R1 + R11;             Adr1: SUB  R1, R5, R7
                            Adr2: ADD  R10, R1, R11
```

Prin rescrierea acestei secvențe utilizând instrucțiuni condiționate se elimină cele 2 instrucțiuni de ramificație obținându-se următoarea secvență de program:

```
        LT   B6, R8, #1
TB6     ADD  R1, R2, R3
FB6     SUB  R1, R5, R7
        ADD  R10, R1, R11
```

Este clar că timpul de execuție pentru această secvență este mai mic decât cel aferent secvenței anterioare. Se arată că astfel de transformări reduc cu cca. 25-30% instrucțiunile de salt condiționat dintr-un program.[Col95, Ste96].

Această execuție condiționată a instrucțiunilor facilitează execuția speculativă. Codul situat după un salt condiționat în program și executat inainte de stabilirea condiției și adresei de salt se numește **cod cu execuție speculativă.** Acest mod de execuție a instrucțiunilor poate fi deosebit de util în optimizarea execuției unui program.

Prezentăm în continuare o secvență de cod inițială și care apoi e transformată de către scheduler în vederea optimizării execuției prin speculația unei instrucțiuni.

```
SUB   R1, R2, R3          SUB   R1, R2, R3
LT    B8, R1, #10         LT    B8, R1, #10
BT    B8, Adr             FB8   ADD  R7, R8, R1;  speculativă
ADD   R7, R8, R1          BT    B8, Adr
```

SUB R10, R7, R4 SUB R10, R7, R4

Execuția speculativă a instrucțiunii ADD putea fi realizată și în lipsa variabilelor de gardă booleene dar atunci putea fi necesară redenumirea registrului R7 (dacă acesta ar fi în viață pe ramura pe care saltul se face).

Orice instrucțiune - cu excepția celor de tip STORE - poate fi executată speculativ. O posibilă strategie de a permite instrucțiuni STORE speculative constă în introducerea unui Write Buffer. Memorarea se va face întâi aici și abia când condiția de salt este cunoscută se va înscrie în memorie [Col95].

Pe lângă avantajele legate de eliminarea salturilor, facilizarea execuției speculative, etc., execuția condiționată are și câteva dezavantaje dintre care amintim:

- instrucțiunile condiționate anulate (NOP) necesită totuși un timp de execuție. În cazul speculației, în aceste condiții performanța în execuție scade.
- dacă variabila de condiție e evaluată târziu, utilitatea instrucțiunii condiționate va fi micșorată.
- promovarea unei instrucțiuni peste mai multe ramificații condiționate în vederea execuției speculative necesită gardări multiple.
- instrucțiunile condiționate pot determina scăderea frecvenței de tact a microprocesorului.

Având în vedere cele de mai sus, utilitatea execuției condiționate este încă discutată. MIPS, POWER-PC, SPARC, ALPHA dețin doar o instrucțiune de tip MOVE condiționată, în timp ce alte microarhitecturi precum HEWLET PACKARD PA, HARP, HSA, etc., permit execuția condiționată a majorității instrucțiunilor mașină.

La ora actuală există încă puține evaluări cantitative care să stabilească avantajele/dezavantajele acestei idei într-un mod clar.

3. ARHITECTURI CU EXECUȚII MULTIPLE ȘI PIPELINE-IZATE ALE INSTRUCȚIUNILOR

3.1 CONSIDERAȚII GENERALE. PROCESOARE SUPERSCALARE ȘI VLIW

Un deziderat ambițios este acela de se atinge viteze de procesare de mai multe instrucțiuni per tact. Procesoarele care inițiază execuția mai multor operații simultan într-un ciclu (sau tact) se numesc **procesoare cu execuții multiple** ale instrucțiunilor. Un astfel de procesor aduce din cache-ul de instrucțiuni una sau mai multe instrucțiuni simultan și le distribuie spre execuție în mod dinamic sau static (de către reorganizator), multiplelor unități de execuție.

Principiul acestor procesoare superscalare numite și **"mașini cu execuție multiplă" (MEM)** constă în existența mai multor unități de execuție paralele, care pot avea latențe diferite.

Pentru a facilita procesarea acestor instrucțiuni, ele sunt codificate pe un singur cuvânt de 32 sau 64 de biți uzual.

Dacă decodificarea instrucțiunilor, detecția dependențelor și lansarea în execuție se fac prin hardware, aceste procesoare MEM se mai numesc și **superscalare.**

Pot exista mai multe unități funcționale distincte, dedicate de exemplu diverselor tipuri de instrucțiuni flotante. Așadar execuțiile instrucțiunilor întregi, se suprapun cu execuțiile instrucțiunilor flotante (FP-Flotant Point). În cazul procesoarelor MEM, **paralelismul temporal** determinat de procesarea pipeline se suprapune cu un **paralelism spațial** determinat de existența mai multor unități de execuție. În general structura pipeline a coprocesorului are mai multe nivele decât structura pipeline a procesorului ceea ce implică probleme de sincronizare mai dificile decât în cazul procesoarelor pipeline scalare. Caracteristic procesoarelor superscalare este faptul că dependențele de date între instrucțiuni se rezolvă prin hardware, în momentul decodificării instrucțiunilor.

Modelul ideal de procesare superscalară, în cazul unui procesor care poate aduce și decodifica 2 instrucțiuni simultan este prezentat în figura 3.1.

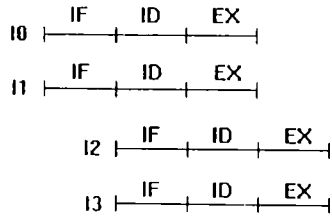


Fig. 3.1

Este evident că în cazul superscalar complexitatea logicii de control este mult mai ridicată decât în cazul pipeline scalar, întrucât detecția și sincronizările între structurile pipeline de execuție cu latențe diferite și care lucrează în paralel devin mult mai dificile. De exemplu un procesor superscalar având posibilitatea aducerii și execuției a "n" instrucțiuni mașină simultan, necesită n(n-1)/2 unități de detecție a hazardurilor de date între aceste instrucțiuni, ceea ce conduce la o complexitate ridicată a logicii de control.

S-ar putea deci considera aceste procesoare MEM ca fiind arhitecturi de tip **MIMD** (Multiple Instructions Multiple Data) în taxonomia lui Michael Flynn. De remarcat că în această categorie sunt introduse cu precădere sistemele multiprocesor care exploatează paralelismul la nivelul mai multor aplicații(**coarse grain parallelism**), arhitecturile RISC ca și cele de tip MEM exploatând paralelismul instrucțiunilor la nivelul aceleiași aplicații(**fine grain parallelism**). Desigur că arhitecturile pipeline scalare(RISC), sunt încadrabile în clasa **SISD** (Single Instruction Single Data), fiind deci incluse în aceeași categorie cu procesoarele cele mai convenționale(secvențiale) în această clasificare, ceea ce implică încă o slăbiciune a acestei sumare taxonomii.

Se prezintă în continuare o structură tipică de superscalar care deține practic 2 module ce lucrează în paralel: un procesor universal și un procesor destinat operațiilor în virgulă mobilă. Ambele module dețin unități de execuție proprii având latențe diferite.

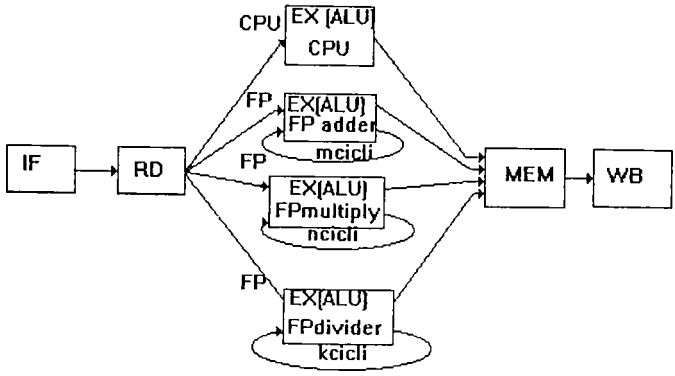


Fig. 3.2.

La anumite microprocesoare superscalare regiștrii CPU sunt diferiți de regiștrii FP, pentru a se reduce hazardurile structurale (în schimb creșteri serioase ale costurilor și dificultăți tehnologice) iar la altele (de ex. MOTOROLA 88100), regiștrii CPU sunt identici cu cei ai coprocesorului [Mot91, Mic90]. De ex., pentru eliminarea hazardurilor structurale, multe dintre aceste microprocesoare nu dețin "clasicul" registru al indicatorilor de condiție. Salturile condiționate se realizează prin compararea pe o anumită condiție, a 2 dintre registrele codificate în instrucțiune. HS la resursele hardware

interne se elimină prin multiplicarea acestora şi sincronizarea adecvată a proceselor.

De remarcat că procesoarele superscalare, determină apropierea ratei de execuţie la una sau, în cazul în care se pot aduce mai multe instrucţiuni simultan, la mai multe instrucţiuni per ciclu. Dificultăţile de sincronizare sporite, se rezolvă prin utilizarea unor tehnici hardware bazate pe "scoreboarding" deosebit de sofisticate. Majoritatea microprocesoarelor RISC actuale sunt de tip superscalar (conţin coprocesor integrat în chip). Un procesor superscalar care aduce din cache-ul de instrucţiuni mai multe instrucţiuni primitive simultan, poate mări rata de procesare la 2-3 instr./ciclu măsurat pe o mare diversitate de benchmark-uri, la nivelul realizărilor practice între anii 90 - 96.

Exemple remarcabile de microprocesoare superscalare comerciale de tip RISC, sunt: INTEL 960 CA, SUN SuperSPARC, MPC 601, 603, 620 (POWER PC), etc.

Procesoarele VLIW (Very Large Instruction Word) reprezintă procesoare care se bazează pe aducerea în cadrul unei instrucţiuni multiple a mai multor instrucţiuni independente pe care le distribuie spre procesare unităţilor de execuţie independente. Aşadar, rata de execuţie ideală la acest model, este de n instrucţiuni/ciclu. Pentru a face acest model viabil, sunt necesare instrumente soft de exploatare a paralelismului programului, bazate pe gruparea instrucţiunilor simple independente şi deci executabile în paralel, în instrucţiuni multiple. Arhitecturile VLIW sunt tot de tip MEM.

Principiul VLIW este sugerat în fig. 3.3:

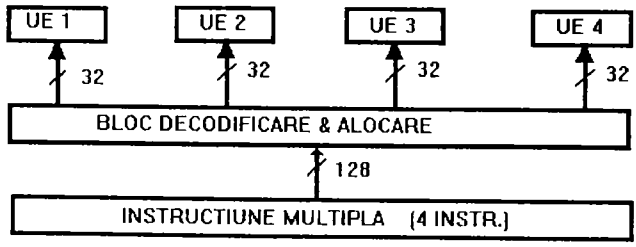


Fig. 3.3

Se încearcă, prin transformări ale programului, ca instrucţiunile primitive din cadrul unei instrucţiuni multiple să fie independente şi deci să se evite hazardurile, a căror gestionare ar fi deosebit de dificilă în acest caz. Performanţa procesoarelor VLIW este esenţial determinată de programele de compilare şi reorganizare care trebuie să fie deosebit de "inteligente".

Prin urmare, în cazul modelului de procesor VLIW, compilatorul trebuie să înglobeze mai multe instrucţiuni primitive independente în cadrul unei instrucţiuni multiple, în timp ce în cazul modelului superscalar, rezolvarea dependenţelor între instrucţiuni se face prin hardware, începând cu momentul decodificării acestor instrucţiuni. De asemenea, poziţia instrucţiunilor primitive într-o instrucţiune multiplă

determină alocarea acestor instrucțiuni primitive la unitățile de execuție în conformitate cu această poziție ocupată de ele, spre deosebire de modelul superscalar unde alocarea se face dinamic prin control hardware. Acest model de procesor nu mai necesită sincronizări și comunicații de date suplimentare între instrucțiunile primitive după momentul decodificării lor.

Un model sugestiv al principiului de procesare VLIW este prezentat în Fig.3.4.

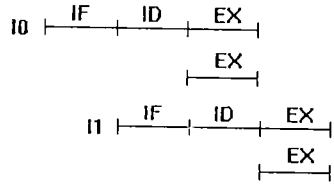


Fig. 3.4.

Pentru exemplificarea principiului de procesare MEM, să considerăm secvența de program de mai jos :

```
LOOP:   LD    F0,0(R1)
        ADD   F4,F0,F2
        SD    0(R1),F4
        SUB   R1,R1,#8
        BNEZ  R1, LOOP
```

Se va analiza în continuare cum ar trebui reorganizată și procesată secvența de program anterioară pentru a fi executată pe un procesor VLIW care poate aduce maxim 5 instrucțiuni primitive simultan și deține 5 unități de execuție distincte și anume: 2 unități LOAD / STORE (MEM1, MEM2), 2 unități de coprocesor flotant (FPP1, FPP2) și o unitate de procesare a instrucțiunilor întregi și branch-urilor.

	MEM 1	MEM 2	FPP 1	FPP 2	CPU / BRANCH
1	Loop: LD F0, 0(R1)	LD F6, -8(R1)			
2	LD F10, -16(R1)	LD F14, -24(R1)			
3	LD F18, -32(R1)	LD F22, -40(R1)	ADD F4, F0, F2	ADD F8, F6, F2	
4	LD F26, -48(R1)		ADD F12, F10, F2	ADD F16,F14, F2	
5			ADD F20, F18, F2	ADD F24,F22, F2	
6	SD 0(R1), F4	SD -8(R1), F8	ADD F28, F26, F2		
7	SD -16(R1), F12	SD -24(R1), F16			
8	SD -32(R1), F20	SD -40(R1), F24			SUB R1, R1, #48
9	SD 0(R1), F28				BNEZ R1, Loop
10					NOP

Tab. 3.1

De remarcat în acest caz o rată de procesare de 2.4 instrucțiuni / ciclu. Altfel spus, bucla de program anterioară se execută în doar 1.42 cicli (10 cicli / 7 bucle). De remarcat printre altele, o redenumire a regiștrilor absolut necesară acestei procesări agresive.

Posibilităţile hard/soft aferente unei asemenea procesări vor fi prezentate succint în continuare.

Este clar că performanţa procesoarelor MEM este esenţial determinată de programele de compilare şi reorganizare care trebuie să fie deosebit de "inteligente". Cercetări realizate în comun la Universitatea Stanford, USA şi firma DEC (Digital Equipment Corporation) pe procesoare VLIW cu 4 instrucţiuni simultane, au arătat că în aplicaţii reale se ajunge la execuţia a max 2 - 3 instrucţiuni / ciclu, prin compilatoare optimizate.

Deşi rare, există realizări comerciale de computere VLIW cu software de optimizare de succes pe piaţă : IBM RS / 6000 (4 instrucţiuni / ciclu , teoretic), INTEL 860 (maxim 2 instrucţiuni / ciclu), APOLLO DN 10000. Aceste realizări sunt disponibile comercial începând cu anul 1991, deşi cercetările au fost iniţiate începând din 1983.

Dificultăţile principale ale modelului VLIW sunt :

- Paralelismul limitat al aplicaţiei, ceea ce determină ca unităţile de execuţie să nu fie ocupate permanent.

- Incompatibilitate software cu modele succesive şi compatibile de procesoare scalare care nu pot avea în general un model VLIW identic datorită faptului că paralelismul la nivelul instrucţiunilor depinde de latenţele operaţiilor procesorului scalar şi de alte caracteristici hardware ale acestuia.

- Dificultăţi deosebite în reorganizarea aplicaţiei în vederea determinării unor instrucţiuni primitive independente sau cu un grad scăzut de dependenţe.

- Creşterea complexităţii hardware şi a costurilor ca urmare a resurselor multiplicate, căilor de informaţie "lăţite", etc.

- Creşterea necesităţilor de memorare ale programelor datorită reorganizărilor soft şi "împachetării" instrucţiunilor primitive care necesită introducerea unor instrucţiuni NOP (atunci când nu există instrucţiuni de un anumit tip disponibile spre a fi asamblate într-o instrucţiune multiplă).

În esenţă, prin aceste modele se încearcă exploatarea paralelismului din programe secvenţiale prin excelenţă, de unde şi limitarea principală a acestui domeniu de "low level parallelism".

Actualmente, datorită faptului că aceste procesoare sunt mult mai ieftine decât procesoarele vectoriale (superprocesoare), şi totodată foarte performante, se pune problema determinării unor clase largi de aplicaţii în care modele superscalar, superpipeline şi VLIW să se comporte mai bine sau comparabil cu modelul vectorial [Hen96, Sto93].

Performanţa procesoarelor pipeline, procesoarelor superscalare şi VLIW este în strânsă legătură cu progresele compilatoarelor specifice acestor structuri, compilatoare care trebuie să extragă cât mai mult din paralelismul existent la nivelul instrucţiunilor

programului. Din punct de vedere teoretic, performanța modelului MEM este aceeași cu a unuia superpipeline echivalent, problema fiind însă mai complicată și încă deschisă.

De remarcat că modelele superscalar și VLIW nu sunt exclusive, în implementările reale se întâlnesc adesea **procesoare hibride**, în încercarea de a se optimiza raportul performanță preț. După cum se va vedea, spre exemplu tehnicile soft de optimizare sunt comune ambelor variante de procesoare.

3.2 MODELE DE PROCESARE ÎN ARHITECTURILE SUPERSCALARE

În cazul procesoarelor superscalare sunt citate 3 modalități distincte de procesare și anume: In Order Issue In Order Completion **(IN - IN)**, In Order Issue Out of Order Completion **(IN - OUT)** și respectiv Out of Order Issue Out of Order Completion **(OUT -OUT).**

Pentru exemplificarea afirmației de mai sus, să considerăm o secvență de instrucțiuni I1 - I6 cu următoarele particularități: I1 necesită 2 cicli pentru execuție, I3 și I4 sunt în conflict structural, între I4 și I5 există hazard RAW iar I5 și I6 sunt de asemenea în conflict structural. În aceste condiții și considerând un procesor superscalar care poate aduce și decodifica 2 instrucțiuni simultan și care deține 2 unități de execuție, avem:

a) Modelul IN - IN

Este caracterizat prin faptul că procesorul nu decodifică următoarea pereche de instrucțiuni, decât în momentul în care perechea anterioară se execută. Așadar atât execuția cât și înscrierea rezultatelor se face în ordinea din program.

DECODIFICARE	EXECUȚIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
				3
		I3	I1, I2	4
I5, I6		I4		5
	I5		I3, I4	6
	I6			7
			I5, I6	8

Tab. 3.2

b) Modelul IN - OUT

Este caracterizat de faptul că execuția propriu-zisă se face în ordine, în schimb înscrierea rezultatelor se face de îndată ce o instrucțiune s-a terminat de executat.

Modelul este mai eficient decât cel precedent însă poate crea probleme de genul întreruperilor imprecise care trebuiesc evitate prin tehnici deja amintite în cap.2.

DECODIFICARE	EXECUŢIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
		I3	I2	3
I5, I6		I4	I1, I3	4
	I5		I4	5
	I6		I5	6
			I6	7

Tab. 3.3

c) Modelul OUT - OUT

Cel mai agresiv şi performant model de procesare a instrucţiunilor într-un procesor superscalar. Instrucţiunile sunt aduse şi decodificate sincron, presupunând deci existenţa unui buffer între nivelul de decodificare şi execuţie (instructions window). Astfel creşte capacitatea de anticipare a instrucţiunilor independente dintr-un program. Modelul permite o exploatare mai bună a paralelismului instrucţiunilor la nivelul unui program dat, prin creşterea probabilităţii de determinare a unor instrucţiuni independente, stocate în buffer.

DECODIFICARE	EXECUŢIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
I5, I6		I3	I2	3
	I6	I4	I1, I3	4
	I5		I4, I6	5
			I5	6

Tab. 3.4

Desigur că execuţia Out of Order este posibilă numai atunci când dependenţele de date între instrucţiuni o permit. Cade în sarcina hardului eliminarea dependenţelor şi alocarea instrucţiunilor din buffer la diversele unităţi de execuţie.

3.3 ARHITECTURA LUI TOMASULO

A fost proiectată şi implementată pentru prima dată în cadrul unităţii de calcul în

virgulă mobilă din cadrul sistemului IBM - 360 / 91 şi este atribuită lui ***Roberto Tomasulo*** [Tom67, Hen96, Vin96]. Arhitectura este una de tip superscalar având deci mai multe unităţi de execuţie, iar algoritmul de control al acestei structuri stabileşte relativ la o instrucţiune adusă, momentul în care aceasta poate fi lansată în execuţie şi respectiv unitatea de execuţie care va procesa instrucţiunea. Arhitectura permite execuţia multiplă şi Out of Order a instrucţiunilor şi constituie modelul de referinţă în reorganizarea dinamică a instrucţiunilor într-un procesor superscalar. De asemenea, algoritmul de gestiune aferent arhitecturii permite anularea hazardurilor WAR şi WAW printr-un mecanism hardware de redenumire a regiştrilor, fiind deci posibilă execuţia Out of Order a instrucţiunilor şi în aceste cazuri. Aşadar, singurele hazarduri care impun execuţia In Order sunt cele de tip RAW.

În cadrul acestei arhitecturi, detecţia hazardurilor şi controlul execuţiei instrucţiunilor sunt distribuite iar rezultatele instrucţiunilor sunt "pasate" direct unităţilor de execuţie prin intermediul unei magistrale comune numită **CDB (Common Data Bus).** Arhitectura de principiu este prezentată în fig.3.5. Ea a fost implementată prima dată în unitatea de virgulă mobila FPP a calculatorului IBM 360/91, pe baza căruia se va prezenta.

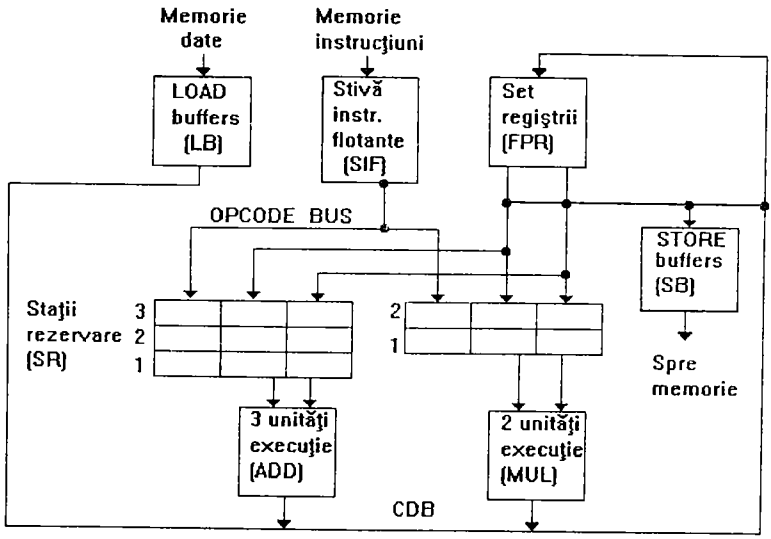


Fig. 3.5

Staţiile de rezervare (SR) memorează din SIF instrucţiunea ce urmează a fi lansată spre execuţie. Execuţia unei instrucţiuni începe dacă există o unitate de execuţie neocupată momentan şi dacă operanzii aferenţi sunt disponibili în SR. Fiecare unitate de

execuţie (ADD, MUL) are asociată o SR proprie. Precizăm că unităţile ADD execută operaţii de adunare/scădere iar unitaţile MUL operaţii de înmulţire/împărţire. Modulele LB şi SB memorează datele încărcate din memoria de date, respectiv datele care urmează a fi memorate. Toate rezultatele provenite de la unităţile de execuţie şi de la bufferul LB sunt trimise pe magistrala CDB. Bufferele LB şi SB precum şi SR deţin câmpuri de TAG necesare în controlul hazardurilor de date între instrucţiuni.

Există în cadrul acestei unităţi de calcul în virgulă mobilă 3 stagii de procesare a instrucţiunilor şi anume:

1) ***Startare*** - aducerea unei instrucţiuni din SIF într-o staţie de rezervare. Aducerea se va face dacă există o SR disponibilă. Dacă operanzii aferenţi se află în FPR vor fi aduşi în SR aferentă. Dacă instrucţiunea este de tip LOAD / STORE, va fi încărcată într-o SR numai dacă există un buffer (LB sau SB) disponibil. Dacă nu există disponibilă o SR sau un buffer, rezultă că avem un hazard structural şi instrucţiunea va aştepta până când aceste resurse se eliberează.

2) ***Execuţie*** - dacă un operand nu este disponibil, prin monitorizarea magistralei CDB de către SR se aşteaptă respectivul operand. În această fază se testează existenţa hazardurilor de tip RAW. Când ambii operanzi sunt disponibili, se execută instrucţiunea în unitatea de execuţie corespunzătoare.

3) ***Scriere rezultat (WB)*** - când rezultatul este disponibil se înscrie pe CDB şi de aici în FPR sau într-o SR care aşteaptă acest rezultat ("forwarding").

De observat că nu există pe parcursul acestor faze testări pentru hazarduri de tip WAR sau WAW, acestea fiind eliminate prin însăşi natura algoritmului de comandă după cum se va vedea. De asemenea, operanzii sursă vor fi preluaţi de către SR direct de pe CDB prin "forwarding" când acest lucru este posibil. Evident că ei pot fi preluaţi şi din FPR în anumite cazuri.

O SR deţine 6 câmpuri cu următoarea semnificaţie:

OP - codul operaţiei (opcode) instrucţiunii din SR.

Qj, Qk - codifică pe un număr de biţi unitatea de execuţie (ADD, MUL) sau numărul bufferului LB, care urmează să genereze operandul sursă aferent instrucţiunii din SR. Dacă acest câmp este zero, rezultă că operandul sursă este deja disponibil într-un câmp Vi sau Vj al SR sau pur şi simplu nu este necesar. Câmpurile Qj, Qk sunt pe post de TAG, adică atunci când o unitate de execuţie sau un buffer LB "pasează" rezultatul pe CDB, acest rezultat se înscrie în câmpul Vi sau Vj al acelei SR al cărei TAG coincide cu unitatea de execuţie sau bufferul LB care a generat rezultatul.

Vj, Vk - conţin valorile operanzilor sursă aferenţi instrucţiunii din SR. Remarcăm că doar unul dintre câmpurile Q respectiv V sunt valide pentru un anumit operand.

BUSY - indică atunci când este setat că SR şi unitatea de execuţie aferentă sunt ocupate momentan.

Regiştrii generali FPR şi bufferele SB deţin fiecare câte un câmp Qi, care codifică numărul unităţii de execuţie care va genera data ce va fi încărcată în respectivul registru general sau care va fi stocată în memoria de date. De asemenea, deţin câte un bit de BUSY. Bufferele SB deţin în plus un câmp care conţine adresa de acces precum şi un câmp care conţine data de înscris. Bufferele LB conţin un bit BUSY şi un câmp de adresă.

Spre a exemplifica funcţionarea algoritmului să considerăm o secvenţă simplă de program maşină:

	Start	Execuţie	WB
1. LF F6, 27(R1)	x	x	x
2. LF F2, 45(R2)	x	x	
3. MULTF F0, F2, F4	x		
4. SUBF F8, F6, F2	x		
5. DIVF F10, F0, F6	x		
6. ADDF F6, F8, F2	x		

În continuare prezentăm starea SR şi a FPR în momentul definit mai sus, adică prima instrucţiune încheiată, a 2-a în faza de execuţie iar celelalte aflate în faza de startare.

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	DA	SUB	(LOAD1)			LOAD2
ADD2	DA	ADD			ADD1	LOAD2
ADD3	NU					
MUL1	DA	MUL		(F4)	LOAD2	
MUL2	DA	DIV		(LOAD1)	MUL1	

Tab. 3.5

Regiştrii generali FPR

CÂMP	F0	F2	F4	F6	F8	F10
Qi	MUL1	LOAD2		ADD2	ADD1	MUL1
BUSY	DA	DA	NU	DA	DA	DA

Tab. 3.6

Din aceste structuri de date implementate în hardware, rezultă de exemplu că SR ADD1 urmează să lanseze în execuţie instrucţiunea SUBF F8, F6, F2. Valoarea primului operand (F6) se află deja în câmpul Vj unde a fost memorată de pe magistrala CDB ca urmare a terminării execuţiei primei instrucţiuni. Evident că rezultatul acestei instrucţiuni a fost preluat de pe CDB în registrul F6 dar şi în bufferul LB1. Al 2-lea operand al

instrucţiunii SUBF nu este încă disponibil. Câmpul de TAG Qk arată că acest operand va fi generat pe CDB cu "adresa" LOAD2 (LB2) şi deci această SR va prelua operandul în câmpul Vk de îndată ce acest lucru devine posibil. Preluarea acestui operand se va face de către toate SR care au un câmp de TAG identic cu LOAD2 (LB2).

Să considerăm de exemplu că latenţa unităţilor ADD este de 2 impulsuri de tact, latenţa unităţilor MUL este de 10 impulsuri de tact pentru o înmulţire şi respectiv 40 impulsuri de tact pentru o operaţie de împărţire. "Starea" secvenţei anterioare în tactul premergător celui în care instrucţiunea MULTF va intra în faza WB va fi:

	Start	Execuţie	WB
1. LF F6, 27(R1)	x	x	x
2. LF F2, 45(R2)	x	x	x
3. MULTF F0, F2, F4	x	x	
4. SUBF F8, F6, F2	x	x	x
5. DIVF F10, F0, F6	x		
6. ADDF F6, F8, F2	x	x	x

În acest moment, starea staţiilor de rezervare şi a setului de regiştri generali va fi următoarea:

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	NU					
ADD2	NU					
ADD3	NU					
MUL1	DA	MUL	(LOAD2)	(F4)		
MUL2	DA	DIV		(LOAD1)	MUL1	

Tab. 3.7

Regiştrii generali FPR

CÂMP	F0	F2	F4	F6	F8	F10
Qi	MUL1					MUL2
BUSY	DA	NU	NU	NU	NU	DA

Tab. 3.8

De remarcat că algoritmul a eliminat hazardul WAR prin registrul F6 între instrucţiunile DIVF şi ADDF şi a permis execuţia Out of Order a acestor instrucţiuni, în vederea creşterii ratei de procesare. Cum prima instrucţiune s-a încheiat, câmpul Vk aferent SR MUL2 va conţine valoarea operandului instrucţiunii DIVF, permiţând deci ca instrucţiunea ADDF să se încheie înaintea instrucţiunii DIVF. Chiar dacă prima instrucţiune nu s-ar fi încheiat, cîmpul Qk aferent SR MUL2 ar fi pointat la LOAD1 şi

deci instrucţiunea DIVF ar fi fost independentă de ADDF. Aşadar, algoritmul prin "pasarea" rezultatelor în SR de îndată ce acestea sunt disponibile, evită hazardurile WAR.

Pentru a pune în evidenţă întreaga "forţă" a algoritmului în eliminarea hazardurilor WAR şi WAW prin redenumire dinamică a resurselor, să considerăm bucla următoare:

```
LOOP:   LF F0, 0 (R1)
        MULTF F4, F0, F4
        SD 0 (R1), F4
        SUB R1, R1, #4
        BNEZ R1, LOOP
```

Considerând o unitate de predicţie a branchurilor de tip "branch-taken", 2 iteraţii succesive ale buclei se vor procesa ca mai jos:

	Start	Execuţie	WB
LF F0, 0 (R1)	x	x	
MULTF F4, F0, F2	x		
SD 0 (R1), F4	x		
LF F0, 0 (R1)	x	x	
MULTF F4, F0, F2	x		
SD 0 (R1), F4	x		

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	NU					
ADD2	NU					
ADD3	NU					
MUL1	DA	MUL		(F2)	LOAD1	
MUL2	DA	MUL		(F2)	LOAD2	

Buffere SB

CÂMP	SB 1	SB 2	SB 3
Qi	MUL1	MUL2	
BUSY	DA	DA	NU
ADR	(R1)	(R1)-8	
V			

Buffere LB

CÂMP	LB 1	LB 2	LB 3
ADR	(R1)	(R1)-8	
BUSY	DA	DA	NU
V			

Tab. 3.9

Se observă o procesare de tip "loop unrolling" ("netezirea buclei") prin hardware. Instrucţiunea LOAD din a 2-a iteraţie se poate executa înaintea instrucţiunii STORE din prima iteraţie întrucât adresele de acces sunt diferite în câmpurile din buffere. Ulterior şi

instrucțiunile MULTF se vor putea suprapune în execuție. De remarcat deci hazardul de tip WAW prin F0 între instrucțiunile de LOAD s-a eliminat cu ajutorul SR și a bufferelor SB și LB.

Arhitectura Tomasulo are deci avantajele de a avea logica de detecție a hazardurilor distribuită și prin renaming dinamic **elimină hazardurile WAW și WAR.** În schimb, arhitectura este complexă, necesitând deci costuri ridicate. Trebuie deci o logică de control complexă, capabilă să execute căutări / memorări asociative cu viteză ridicată. Având în vedere progresele mari ale tehnologiilor VLSI, variante ale acestei arhitecturi se aplică în multe procesoare superscalare actuale.

Puternicul mecanism de "forwarding" implementat prin această arhitectură o face, dintr-un punct de vedere analoagă cu structurile paralele numite cu flux de date (data-flow machines), care sunt structuri asincrone ce procesează imediat ce operanzii aferenți unui nod devin disponibili.

Acest mecanism de forwarding din arhitectura lui Tomasulo, are meritul de a reduce semnificativ din **presiunea la "citire" asupra setului general de regiștri logici**, speculând dependențele RAW între instrucțiuni.

Dezvoltări interesante ale arhitecturii Tomasulo sunt prezentate în [Pat85].

3.4 O ARHITECTURĂ REPREZENTATIVĂ DE PROCESOR SUPERSCALAR

Având în vedere ideile de implementare a execuțiilor multiple din arhitectura lui Tomasulo, o arhitectură superscalară reprezentativă este prezentată în figura 3.6:

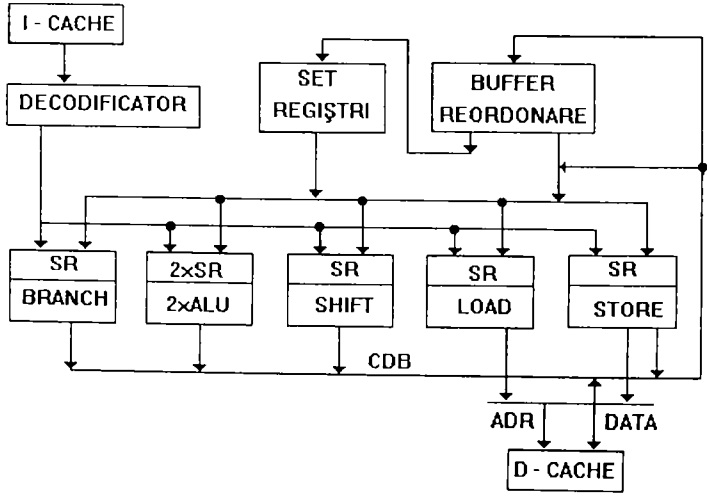


Fig. 3.6

Prin SR am notat stațiile de rezervare aferente unităților de execuție ale procesorului. Acestea implementează printre altele bufferul "instruction window" necesar procesoarelor superscalare cu execuție Out of Order. Numărul de locații al fiecărei SR se determină pe bază de simulare.

Deși performanța maximă a unei asemenea arhitecturi ar fi de 6 instrucțiuni/ciclu, în realitate, bazat pe simulări ample, s-a stabilit că rata medie de execuție este situată între 2-3 instrucțiuni / ciclu [Joh91]. În sub 1% din cazuri, măsurat pe benchmark-uri nenumerice, există un potențial de paralelism mai mare de 6 instrucțiuni / ciclu în cazul unei arhitecturi superscalare "pure". Aceasta se datorează în primul rând capacității limitate a bufferului de prefetch care constituie o limitare principială a oricărui procesor. În tehnologia actuală acesta poate memora între 8 - 64 instrucțiuni. Prezentăm pe scurt rolul modulelor componente din schemă.

Decodificatorul plasează instrucțiunile multiple în SR corespunzătoare. O unitate funcțională poate starta execuția unei instrucțiuni din SR imediat după decodificare dacă instrucțiunea nu implică dependențe și dacă unitatea de execuție este liberă. În caz contrar, instrucțiunea așteaptă în SR până când aceste condiții vor fi îndeplinite. Dacă mai multe instrucțiuni dintr-o SR sunt simultan disponibile spre a fi executate, procesorul o va selecta pe prima din secvența de instrucțiuni.

Desigur că este necesar un mecanism de arbitrare în vederea accesării CDB de către diversele unități de execuție (UE). În vederea creșterii eficienței, deseori magistralele interne sunt multiplicate. Prezentăm în fig. 3.7 circulația informației într-o structură superscalară, similară cu cea implementată la microprocesorul Motorola MC 88110.

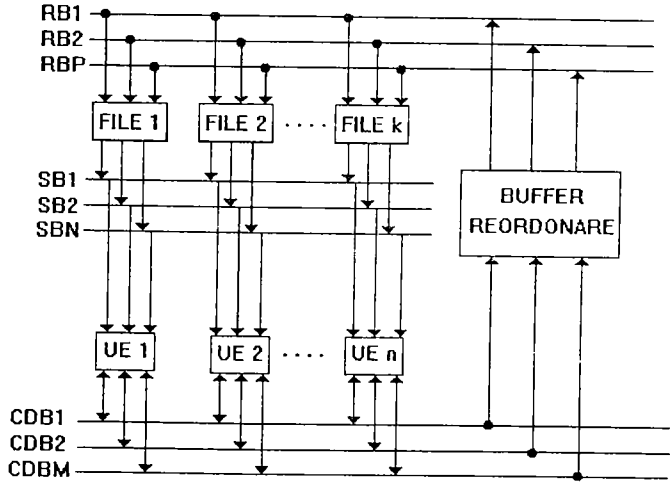


Fig. 3.7

Setul de regiştri generali (FILE) este multiplicat fizic. Conţinutul acestor seturi fizice este identic însă în orice moment. Am considerat că UE conţin şi staţiile de rezervare aferente. Din acest motiv, având în vedere mecanismul de "forwarding" implementat, comunicaţia între UE şi CDB s-a considerat bidirecţională.

Există 3 categorii de busuri comune şi anume: busuri rezultat (RB), busuri sursă (SB) si busuri destinaţie (CDB). N corespunde numărului maxim de instrucţiuni care pot fi lansate simultan în execuţie. Min (M, P) reprezintă numărul maxim de instrucţiuni care pot fi terminate simultan. Uzual se alege M = P.

Există implementate mecanisme de arbitrare distribuite în vederea rezolvării tuturor hazardurilor structurale posibile pe parcursul procesărilor.

Pe bază de simulare în [Jou94] se încearcă stabilirea unei arhitecturi optimale. Astfel se arată că pentru o rată de fetch şi execuţie de 4 instrucţiuni, procesarea optimă din punct de vedere performanţă/cost impune 7 busuri destinaţie, 4 unităţi de execuţie întregi şi 8 staţii de rezervare pentru unităţile LOAD / STORE. Pentru o asemenea arhitectură s-ar obţine o rată de procesare de 2.88 instrucţiuni / tact, măsurat însă pe benchmark-uri cu un puternic caracter numeric (Livermore Loops).

Ideea de bază este însă că hazardurile structurale se elimină şi aici prin multiplicarea resurselor hardware. Gradul de multiplicare trebuie însă stabilit prin simulări ample ori prin metode teoretice.

Bufferul de reordonare

Bufferul de reordonare (RB) este în legătură cu mecanismul de redenumire dinamică a regiştrilor în vederea execuţiei Out of Order precum şi cu necesitatea implementării unui mecanism precis de tratare a evenimentelor de excepţie(derute, devieri, întreruperi hard-soft, etc.). Acesta conţine un număr de locaţii care sunt alocate în mod dinamic rezultatelor instrucţiunilor.

În urma decodificării unei instrucţiuni, rezultatul acesteia este asignat unei locaţii din RB, iar numărul registrului destinaţie este asociat acestei locaţii. În acest mod, registrul destinaţie este practic redenumit printr-o locaţie din RB. În urma decodificării se crează prin hard un tag care reprezintă numele unităţii de execuţie care va procesa rezultatul instrucţiunii respective. Acest tag va fi scris în aceeaşi locaţie din RB. Din acest moment, când o instrucţiune următoare face referire la respectivul registru pe post de operand sursă, ea va apela în locul acestuia valoarea înscrisă în RB sau, dacă valoarea nu a fost încă procesată, tagul aferent locaţiei. Dacă mai multe locaţii din RB conţin acelaşi număr de registru (mai multe instrucţiuni în curs, au avut acelaşi registru destinaţie), se va genera locaţia cea mai recent înscrisă (tag sau valoare).

Este evident că RB se implementează sub forma unei memorii asociative, căutarea făcându-se după numărul registrului destinaţie la scriere, respectiv sursă la citire.

Dacă accesarea RB se soldează cu MISS, atunci operandul sursă va fi citit din setul de regiştri. În caz de HIT, valoarea sau tagul citite din RB sunt memorate în SR corespunzătoare.

Când o unitate de execuţie generează un rezultat, acesta se va înscrie în SR şi în locaţia din RB care au tagul identic cu cel emis de către respectiva unitate. Rezultatul înscris într-o SR poate debloca anumite instrucţiuni aflate în aşteptare.

După ce rezultatul a fost scris în RB, instrucţiunile următoare vor continua să-l citească din RB ca operand sursă până când va fi evacuat şi scris în setul de regiştri. Evacuarea (faza WB) se va face în ordinea secvenţei originale de instrucţiuni pentru a se putea evita excepţiile imprecise. Aşadar, redenumirea unui registru cu o locaţie din RB se termină în momentul evacuării acestei locaţii.

Bufferul RB poate fi gestionat ca o memorie FIFO. În momentul decodificării unei instrucţiuni, rezultatul acesteia este alocat în coada RB. Rezultatul instrucţiunii este înscris în momentul în care unitatea de execuţie corespunzătoare îl generează. Când acest rezultat ajunge în prima poziţie a RB, dacă nu au apărut excepţii, este înscris în setul de regiştri. Dacă instrucţiunea nu s-a încheiat atunci când locaţia alocată în RB a ajuns prima, bufferul RB nu va mai avansa până când această instrucţiune nu se va încheia. Decodificarea instrucţiunilor poate însă continua atât timp cât mai există locaţii disponibile în RB.

Dacă apare o excepţie, bufferul RB este golit, procesorul bazându-se pe contextul memorat In Order în setul general de regiştri. Astfel, deşi procesează Out of Order, procesorul superscalar implementează un mecanism de excepţii precise. Mecanismul este similar cu cel numit "history buffer" şi prezentat în cadrul procesoarelor pipeline scalare.

În general, capacitatea RB se stabileşte pe baza simulării unei arhitecturi superscalare, pe diverse programe de test.

Se apreciază bazat pe măsurări şi simulări laborioase pe o multitudine de benchmark-uri că performanţa unui procesor superscalar "pur" nu poate depăşi în medie 2-3 instr. / tact [Joh91].

3.5 PROBLEME SPECIFICE INSTRUCŢIUNILOR DE RAMIFICAŢIE ÎN ARHITECTURILE MEM. LIMITĂRI ALE ARHITECTURILOR MEM

Să considerăm o secvenţă de program care se execută astfel:

PC=743644 : I1
I2
I3
I4

	I5	(branch condiţionat)
PC=342234 :	I6	
	I7	
	I8	
	I9	(branch condiţionat)

Dacă am presupune că BDS-ul este de 2 cicli, procesarea secvenţei de mai sus pe un procesor superscalar (VLIW) cu procesare In Order care poate aduce şi executa maxim 4 instrucţiuni/ciclu, s-ar desfăşura ca în tabelul de mai jos:

	I1	I2	I3	1
I4	I5			2
				3
				4
	I6	I7	I8	5
I9				6

Tab. 3.10

Se observă că pentru a compensa BDS-ul de 10 instrucţiuni, ar trebui introduse în acesta 10 instrucţiuni anterioare instrucţiunii I5 şi care să nu o afecteze. Acest lucru este practic imposibil, de unde rezultă că asemenea metode sunt inefective pentru procesoarele superscalare. Din acest motiv predicţia hardware a branch-urilor pe baza unui BTB este implementată deseori în aceste procesoare. Pentru ca metoda de predicţie prezentată în Cap. 2 să funcţioneze şi în acest caz, sunt necesare câteva completări datorate aducerii şi execuţiei multiple a instrucţiunilor [Joh91].

Se va considera că o locaţie a memoriei cache de instrucţiuni conţine 4 câmpuri. Fiecare câmp la rândul său va fi format din: codul instrucţiunii respective, tagul format din biţii de adresă cei mai semnificativi, indexul de succesor (IS) şi indexul branch-ului în locaţie (IBL).

Subcâmpul IS indică următoarea locaţie din I-cache predicţionată a fi adusă şi respectiv prima instrucţiune ce trebuie executată din cadrul acestei locaţii.

Subcâmpul IBL indică dacă există sau nu o instrucţiune de salt în locaţia din cache şi dacă da, locul acesteia în cadrul locaţiei.

Pentru secvenţa anterioară de program, intrările în memoria I-cache se prezintă ca în figura următoare (IBL s-a exprimat în binar):

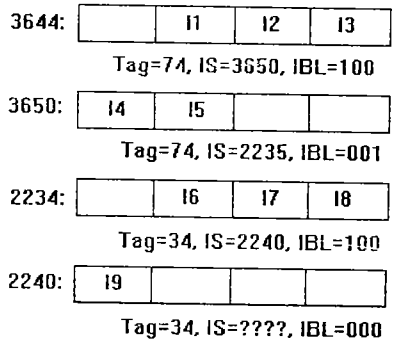


Fig. 4.8

Aşadar subcâmpul IS pointează spre prima instrucţiune care trebuie executată în cadrul unei locaţii din cache, iar subcâmpul IBL spre o eventuală instrucţiune de salt din cadrul aceleiaşi locaţii, predicţionată că se va face. Adresa completă a instrucţiunii la care se face saltul este conţinută în memoria BTB. În cazul salturilor predicţionate că nu se vor face, informaţia de predicţie lipseşte.

Există deja opinii care arată că arhitecturile superscalare şi VLIW conţin **limitări fundamentale** şi care ar trebui eliminate. Dintre aceste limitări amintim doar conflictele la resurse, datorate în principal centralizării acestora. O idee interesantă bazată pe descentralizarea resurselor se prezintă în [Fra92] şi are în vedere implementarea mai multor aşa numite "Instruction Windows" (IW)- un fel de buffere de prefetch, în locul unuia singur şi pe conceptul de **multithreading.** Lansarea în execuţie a instrucţiunilor se face pe baza determinării celor independente din fiecare IW. Desigur că trebuie determinate şi dependenţele inter- IW- uri.

Ideea principală constă în execuţia paralelă a mai multor secvenţe de program aflate în IW- uri diferite, bazat pe mai multe unităţi funcţionale (multithreading). Astfel de ex., 2 iteraţii 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** şi **scalabilitate**, deosebit de utile în dezvoltarea viitoare a arhitecturii.

O idee foarte recentă, o reprezintă **arhitecturile TTA- Transport Triggered Architectures**, dezvoltate la Universitatea din Delft, Olanda [Nee96]. Acestea oferă spre exploatare o granularitate mai mică a paralelismului Singura instrucţiune a unei asemenea arhitecturi este cea de MOVE, cu posibilitatea de execuţie condiţionată prin gardarea cu variabile booleene. O unitate funcţională TTA deţine 3 regiştri: sursă, trigger şi rezultat. Înscrierea unui operand în registrul trigger va determina automat activarea unităţii funcţionale aferente. Asfel de ex., codul TTA pentru adunarea a 2 numere

conține 3 MOVE- uri iar cel al unui branch condiționat 4 MOVE- uri, ultimul (cel de actualizare condiționată a PC-ului) fiind condiționat de o gardă booleană.

Pe lângă multiple avantaje legate de simplificarea arhitecturii hardware, se facilitează exploatarea mai agresivă a paralelismului fin prin metode de scheduling static. De asemenea utilizarea resurselor pare a fi mai bună iar setul de regiștri generali , în mare măsură, nu mai constituie o sursă de blocaj. Testele actuale arată că un procesor TTA este cu 25-50% mai rapid decât un procesor superscalar (OTA) echivalent. Nu putem să nu remarcăm că aceste idei noi provin din unele mai vechi precum microprogramarea dinamică în acest caz. Doar viitorul va putea da un răspuns ferm relativ la valoarea lor reală.

În continuare se vor prezenta câteva tehnici software utilizate în procesoarele superscalare și VLIW. Aceste alternative pot simplifica mult complexitatea hardware a procesorului. După cum se va vedea, utilizarea unor optimizări software elimină necesitatea execuției Out of Order a instrucțiunilor, bufferului "instruction window", redenumirii dinamice a regiștrilor, etc.

3.6 OPTIMIZAREA BASIC-BLOCK-URILOR ÎN ARHITECTURILE MEM

Ca și în cazul procesoarelor scalare, reorganizarea (scheduling) reprezintă procesul de aranjare a instrucțiunilor din cadrul unui program obiect astfel încât acesta să se execute într-un mod cvasioptimal din punct de vedere al timpului de procesare.

Procesul de reorganizare a instrucțiunilor determină creșterea probabilității ca procesorul să aducă simultan din cache-ul de instrucțiuni mai multe instrucțiuni independente. De asemenea asigură procesarea eficientă a operațiilor critice din punct de vedere temporal în sensul reducerii latențelor specifice acestor operații.

Se va aborda mai întâi problema optimizării " basic block"-urilor. De remarcat că schedulingul în procesoarele superscalare poate determina o simplificare substanțială a arhitecturii hardware aferentă acestora [Col95].

Se va analiza acum următoarea secvență de program:

```
I1:  ADD  R1, R11, R12
I2:  ADD  R1, R1, R13
I3:  SLL  R2, R3, #4;   R2<--R3 deplasat logic la stânga cu 4 poziții binare
I4:  AND  R2, R1, R2
I5:  ADD  R1, R14, R15
I6:  ADD  R1, R1, R16
I7:  ADD  R1, R1, #6
I8:  LD   R1, (R1)
```

I9: LD R4, (R4)

I10: ADD R1, R4, R1

I11: OR R1, R1, R2

I12: ST R1, (R4)

Considerând un procesor superscalar care decodifică simultan 4 instrucțiuni și deține 4 unități de execuție (2 unități ALU, o unitate LOAD / STORE și o unitate pentru deplasări / rotiri), procesul de execuție al secvenței anterioare s-ar desfășura ca în tabelul 3.11 (am presupus că doar instrucțiunile LOAD au latență și anume de 2 cicli mașină):

DECODIFICARE				EXECUȚIE				WRITEBACK		CICLU
I0	I1	I2	I4	ALU1	ALU2	SHF	LS	R1	R2	
1:add	2:add	3:sll	4:and							1
5:add	6:add	7:add	8:ld	I1		I3				2
9:ld	10:and	11:or	12:st		I2			I1	I3	3
				I4	I5			I2		4
				I6				I4	I5	5
					I7			I6		6
							I8	I7		7
							I9			8
								I8		9
				I10				I9		10
					I11			I10		11
							I12	I11		12

Tab. 3.11

De remarcat că rata de procesare a acestei secvențe este de 12 instrucțiuni per ciclu, adică 1,1 instrucțiuni / ciclu (s-a considerat procesare In Order).

Se observă că paralelismul potențial al secvențelor de instrucțiuni I1 - I4 respectiv I5 - I12 nu este exploatat.

În continuare se va prezenta un algoritm de reorganizare în "basic block"-uri (unități secvențiale de program) în vederea unei execuții cvasioptimale pe un procesor superscalar sau VLIW.

3.6.1 PARTIȚIONAREA UNUI PROGRAM ÎN "BASIC-BLOCK"-URI

Se are în vedere construirea **grafului de control** al unui program dat. Algoritmul de partiționare constă în următorii 2 pași:

1) Determinarea setului de lideri în cadrul programului. Se numește lider prima

instrucțiune dintr-un program, instrucțiunea destinație a oricărei instrucțiuni de branch sau orice instrucțiune următoare unei instrucțiuni de branch.

2) Partiționarea programului în unități secvențiale și construirea grafului de control. Fiecare unitate secvențială conține un singur lider și toate instrucțiunile de la acest lider până la următorul exclusiv.

Se determină predecesorii imediați față de o unitate secvențială de program. Poate fi un predecesor imediat al unei unități secvențiale date orice unitate secvențială care se poate executa înaintea unității date.

Se determină succesorii unei unități secvențiale de program. Se numește succesor al unei unități secvențiale de program orice unitate secvențială de program care poate să se execute după execuția celei curente.

În figura de mai jos se dă un exemplu de partiționare a unui program dat în "basic block"-uri și graful de control al programului.

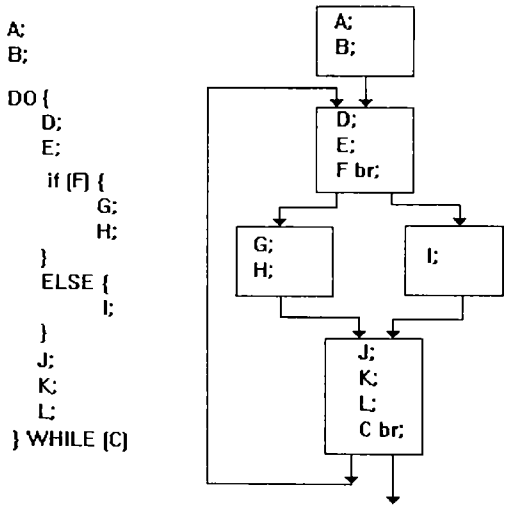


Fig. 3.9

3.6.2 CONSTRUCȚIA GRAFULUI DEPENDENȚELOR DE DATE ASOCIAT UNUI PROGRAM

După cum se va vedea în stabilirea unei secvențe reorganizate de program în vederea unei procesări cvasioptimale pe un procesor superscalar sau VLIW, graful dependențelor de date aferent unei unități secvențiale de program, se va dovedi deosebit de util. Un arc în acest graf semnifică o dependență RAW între cele 2 stări. Instrucțiunile care utilizează date din afara unității secvențiale de program se vor plasa în vârful grafului

astfel încât în ele nu va intra nici un arc. Pentru o instrucțiune dată se caută în jos proxima dependență RAW.

Cu aceste reguli simple, graful dependențelor de date corespunzător secvenței de program anterioare este prezentat mai jos:

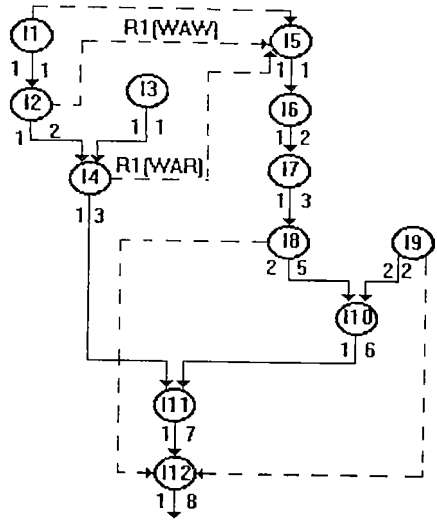


Fig. 3.10

În stânga arcului este scrisă latența operației respective. În dreapta arcului este scrisă latența maximă a drumului măsurată dintr-un vârf al arcului până în starea respectivă. Graful dependențelor specifică deci relații de ordine absolut necesare execuției corecte a programului dat.

Graful precedențelor

Se obține pe baza grafului dependențelor de date în baza faptului că există cazuri în care acesta poate să nu cuprindă toate precedențele necesare unei corecte reorganizări. Altfel spus, acest graf nu pune în evidență relațiile de precedență strictă în lansarea în execuție impuse de către dependențele de tip WAR respectiv WAW între instrucțiuni.

De exemplu, între instrucțiunile I2 și I5 există o dependență de tip WAW, iar între I4 și I5 una de tip WAR. Aceste dependențe ar obliga schedulerul să proceseze I2 și I4 înaintea instrucțiunii I5. Și totuși aceste dependențe între secvențele I1 - I4 și respectiv I5-I10 pot fi eliminate prin redenumirea regiștrilor care determină dependențele WAR și WAW (în cazul nostru registrul R1). Astfel, de ex., dacă în cadrul secvenței de instrucțiuni I1-I4 se redenumește registrul R1 cu un alt registru disponibil în acel moment (de ex. cu R5), atunci secvențele I1 - I4 și respectiv I5 - I10 devin complet independente,

permițând o procesare paralelă mai accentuată.

Redenumirea regiștrilor, ca și în cazul procesoarelor scalare, se poate face static prin software îm momentul compilării, sau dinamic, prin hardware, în momentul procesării. Trebuie deci redenumiți acei regiștri care determină dependențele WAR și WAW între ramuri independente ale unității secvențiale de program. Se arată că redenumirea regiștrilor crește numărul regiștrilor utilizați și timpul de viață al unui registru. Prin **timp de viață** al unui registru se înțelege numărul instrucțiunilor cuprinse între prima instrucțiune care actualizează respectivul registru și respectiv ultima instrucțiune care-l citește. Așadar, redenumirea regiștrilor crează dificultăți alocării regiștrilor. Redenumirea se poate face pe durata timpului de viață al registrului.

Important este însă faptul că prin redenumire, graful precedențelor devine inefectiv, singurul care impune precedențe reale fiind deci graful dependențelor de date prin dependențele RAW intre instrucțiuni.

Precedențele de mai sus au fost puse în evidență prin linii întrerupte în figura anterioară. (Fig.3.10)

De asemenea, trebuie respectată ordinea instrucțiunilor LOAD / STORE, așadar, instrucțiunile I8 și I9 trebuie să preceadă instrucțiunea I12. Această ultimă constrângere însă, nu introduce precedențe suplimentare în graful dependențelor de date în acest caz particular. Această problemă a fost detaliată în capitolul 2.

3.6.3 CONCEPTUL CĂII CRITICE

Calea critică a grafului precedent reprezintă drumul cu latență maximă, fiind deci reprezentată de secvența de instrucțiuni I5, I6, I7, I8, I10, I11 și I12. Latența acestei căi este de 8 cicli. Conceptul căii critice este important deoarece el indică faptul că după scheduling, profitând la maxim de paralelismul între instrucțiuni, programul se va putea executa în 8 cicli, adică într-un timp egal cu latența căii critice. Prin strategia sa, schedulerul va trebui ca în fiecare ciclu, pe cât posibil, să execute câte o instrucțiune din calea critică încercând simultan să suprapună peste această instrucțiune și alte instrucțiuni independente din program.

Într-un procesor ipotetic având resurse infinite, schedulerul optim ar trebui pur și simplu să urmeze calea critică, suprapunând peste operațiile de aici operații din alte căi. În cazul apariției unui hazard WAW sau WAR între instrucțiuni trebuiesc redenumite registrele implicate. De ex. în cazul anterior prezentat, simultan cu instrucțiunea I5 s-ar putea executa instrucțiunile I1, I3 și I9 în condițiile în care procesorul ar deține suficiente resurse hardware (2 unități ALU, o unitate de shiftare și o unitate LOAD / STORE). De asemenea, datorită hazardului WAW dintre I1 și I5, în instrucțiunea I1 ar trebui redenumit registrul R1 cu un alt registru disponibil din setul de regiștri. Cum în practică

un procesor nu deține totdeauna suficiente resurse în vederea executării celor de mai sus, rezultă că **timpul de execuție al programului reorganizat este mai mare sau cel mult egal cu latența căii critice.**

O reorganizare optimă ar însemna să se simuleze execuția tuturor variantelor posibile de programe reorganizate și să se măsoare ratele de procesare aferente, alegându-se varianta de program cu rata cea mai mare. Pentru programe mari acest deziderat ar implica uneori săptămâni sau chiar ani de procesare devenind deci prohibit [Joh91]. În practică se preferă algoritmi euristici bazați pe graful dependențelor, care dau rezultate apropiate de cele optimale, în schimb necesită timpi de execuție acceptabili. Așadar problema optimalității teoretice a scheduling-ului, nu se pune din probleme de timp. În plus algoritmii euristici utilizați în practică dau rezultate bune.

3.6.4 ALGORITMUL "LIST SCHEDULING" (LS)

Este unul dintre cei mai reprezentativi algoritmi în acest sens. Timpul de execuție este rezonabil întrucât algoritmul se execută într-o singură trecere prin graful dependențelor, generând în majoritatea cazurilor reorganizări optimale[Joh91,Cho95].

Algoritmul LS parcurge graful dependențelor asociat unității secvențiale de program de jos în sus. În fiecare pas se încearcă lansarea în execuție a instrucțiunilor disponibile. După ce aceste instrucțiuni au fost puse în execuție, instrucțiunile precedente devin disponibile spre a fi lansate în pasul următor. Fiecărei instrucțiuni i se atașează un grad de prioritate egal cu latența căii instrucțiunii. Dacă apare un conflict la resurse hardware comune între 2 sau mai multe instrucțiuni, are prioritate instrucțiunea cu un grad de prioritate mai mare.

Precizând că inițial se setează un contor de cicli la o valoare maximă, **pașii algoritmului** sunt următorii:

1) Instrucțiunea cea mai prioritară dintre instrucțiunile disponibile în setul curent este lansată în execuție dacă nu necesită o resursă ocupată în acest ciclu.

2) Dacă o instrucțiune a fost pusă în execuție în pasul 1, resursele utilizate de aceasta vor fi setate ca fiind ocupate pentru un număr de cicli egali cu latența instrucțiunii. Pentru exemplul nostru se va considera latența instrucțiunilor LOAD de 2 cicli, iar latența celorlalte instrucțiuni de un ciclu.

3) Dacă instrucțiunea a fost lansată în execuție în pasul 1 ea va fi ștearsă din lista instrucțiunilor disponibile în acel ciclu. Dacă instrucțiunea nu a fost lansată în execuție datorită unui conflict, reorganizarea va continua cu o altă instrucțiune disponibilă.

4) Se repetă pașii 1-3 până când nu mai există nici o instrucțiune disponibilă în acest ciclu.

5) Se decrementează contorul de cicli.

6) Se determină următorul set de instrucțiuni disponibile. Precizăm că o instrucțiune este disponibilă dacă diferența între numărul ciclului în care a fost lansată în execuție instrucțiunea succesoare și numărul ciclului curent este egală cu latența instrucțiunii.
7) Dacă setul determinat la pasul 6 este consistent se trece la pasul 1. În caz contrar, reorganizarea este completă.

Aplicarea algoritmului pe graful din exemplul considerat generează următoarea ordine de execuție a instrucțiunilor.

CICLU	NOD / PRIORITATE
8	12 / 8
7	11 / 7
6	10 / 6, 4 / 3
5	2 / 2, 3 / 1
4	8 / 5, 9 / 2, 1 / 1
3	7 / 3
2	6 / 2
1	5 / 1

Tab. 3.12

Execuția instrucțiunilor pe un procesor superscalar In Order Issue cu 4 u....ăți de execuție se va face ca în tabelul de mai jos (tab.3.13):

ALU 1	ALU 2	SHF	LD / ST	
I5				1
	I6			2
I7			I9	3
	I1		I8	4
I2		I3		5
I10	I4			6
	I11			7
			I12	8

Tab. 3.13

De remarcat că în ciclul 4 a existat un conflict structural între instrucțiunile I8 și I9. S-a dat prioritate instrucțiunii I8 pentru că are un grad de prioritate superior. Invers n-ar fi putut fi pentru că I8 și I7 sunt dependente RAW, de unde rezultă importanța prioritizării după latența nodurilor respective.

Pe această secvență de program se obține o rată de procesare de 12 / 8 = 1.5 instr./ciclu față de doar 1.1 instr. / ciclu cât era rata de procesare a variantei de program

nereorganizate, executată pe același procesor superscalar.

O altă observație foarte importantă este aceea că schedulingul poate îmbunătăți semnificativ **gradul de utilizare** al resurselor hardware.

O variantă de algoritm similar care însă ar parcurge graful de sus în jos, ar genera următoarea execuție:

ALU 1	ALU 2	SHF	LD / ST	
I5	I1	I3	I9	1
I6	I2			2
I7	I4			3
			I8	4
				5
I10				6
	I11			7
			I12	8

Tab. 3.14

Latența instrucțiunii I8 și dependența RAW între I8 și I10, au impus ca în ciclul 4 să nu se execute nici o operație. Performanța este însă și în acest caz de 1.5 instr. / ciclu.

Este evident că performanța unui procesor superscalar de tip In Order Issue care procesează un basic-block reorganizat optimal, este mai bună decât performanța unui procesor superscalar Out of Order care procesează programul neoptimizat. În al doilea caz paralelismul între instrucțiuni este limitat de capacitatea bufferului de prefetch (instruction window).

În literatură sunt citate 2 variante principiale de realizare a reorganizărilor software, între care există un compromis fundamental [Cha95, Joh91]. Prima este metoda postscheduling, care implică după compilare mai întâi alocarea regiștrilor, iar apoi reorganizarea propriu-zisă. În acest caz reorganizatorul este constrâns la niște false dependențe datorită faptului că alocatorul de regiștri utilizează un registru cât mai mult posibil, ceea ce duce la false dependențe de date care se rezolvă de către scheduler prin redenumirea regiștrilor (în exemplul nostru a fost necesară redenumirea registrului R1). A doua metodă se numește prescheduling și presupune mai întâi realizarea reorganizării codului obiect, iar apoi alocarea regiștrilor. În acest caz este posibil ca alocatorul de regiștri să nu poată păstra toate variabilele în regiștri, deoarece schedulerul prin redenumire mărește timpul de viață al regiștrilor utilizați.

Algoritmul LS pune la dispoziția structurii hard paralelismul la nivel de instrucțiuni dintr-un program, pe care structura respectivă îl exploatează la maxim.

În continuare vom aborda problema optimizării globale a programelor pentru procesoarele cu execuție multiplă a instrucțiunilor.

3.7 PROBLEMA OPTIMIZĂRII GLOBALE ÎN CADRUL PROCESOARELOR MEM

În continuare vom prezenta câteva tehnici software legate de optimizarea programelor pentru procesoarele superscalare şi VLIW. În paragraful precedent s-au prezentat câteva tehnici în vederea optimizării "basic-block"urilor (**optimizare locală**). Optimizarea "basic-block"urilor aferente unui program nu implică în mod necesar optimizarea întregului program datorită problemelor legate de instrucţiunile de ramificaţie.

Reorganizarea programelor care conţin branch-uri este mai dificilă întrucât aici "mutările" de instrucţiuni pot cauza incorectitudini ale programului reorganizat care ar trebui corectate. Această optimizare se mai numeşte şi **optimizare globală**.

Problema optimizării globale este una de mare actualitate şi interes, întrucât paralelismul la nivelul basic-block-urilor este relativ scăzut (2-3 instrucţiuni). Deoarece majoritatea programelor HLL sunt scrise în limbaje imperative şi pentru maşini secvenţiale cu un număr limitat de registre în vederea stocării temporare a variabilelor, este de aşteptat ca gradul de dependenţe între instrucţiunile adiacente să fie ridicat [Fra92]. Aşadar pentru mărirea nivelului de paralelism este necesară suprapunerea execuţiei unor instrucţiuni situate în basic-block-uri diferite.

Numeroase studii [Cho95,Cha95,Col95,Pot96] au arătat că paralelismul programelor de uz general poate atinge în variante idealizate (resurse hardware nelimitate, renaming perfect, analiză antialias perfectă, etc.) în medie 50-60 instrucţiuni simultane. De remarcat că schedulerele actuale, cele mai performante, raportează performanţe cuprinse între 3-7 instrucţiuni simultane. Se apreciază ca realiste obţinerea în viitorul apropiat a unor performanţe de 10-15 instrucţiuni simultane bazat pe îmbunătăţirea tehnicilor de optimizare globală.

Problema optimizării globale este una deschisă la ora actuală. Se prezintă în continuare în acest sens doar tehnica numită "Trace Scheduling", datorită faptului că este oarecum mai simplă şi mai clar documentată. Consider că marea majoritate a tehnicilor de optimizare globală nu au încă o justificare teoretică foarte solidă, bazându-se deseori pe o euristică dependentă de caracteristicile arhitecturii[Ebci86, Col95]. Ele au drept scop execuţia unei instrucţiuni cât mai repede posibil. Constrângerile sunt date în principal de resursele hardware limitate.

Apoi se vor prezenta pe scurt două tehnici de optimizare a buclelor de program pentru arhitecturile MEM.

Pentru claritate, se va considera în continuare că instrucţiunile de salt nu conţin BDS-uri.

3.7.1 TEHNICA "TRACE SCHEDULING" (TS)

Este atribuită cercetătorului american ***Joshua Fischer*** [Fis81, Joh91, Hen96] de la Universitatea din Illinois, care a implementat-o în cadrul calculatorului VLIW numit BULLDOG(1986). O implementare mai veche a acestei tehnici a fost legată de optimizarea microprogramelor în arhitecturi microprogramate orizontal.

Se defineşte o cale (**"Trace"**) într-un program ce conţine salturi condiţionate, o ramură particulară a acelui program legată de o asumare dată a adreselor acestor salturi. Rezultă deci că un program care conţine n salturi condiţionate va avea 2^n posibile căi. Aşadar, o cale a unui program va traversa mai multe unităţi secvenţiale din acel program.

În figura următoare se prezintă un program compus din 2 căi distincte, şi anume TRACE1 şi TRACE2.

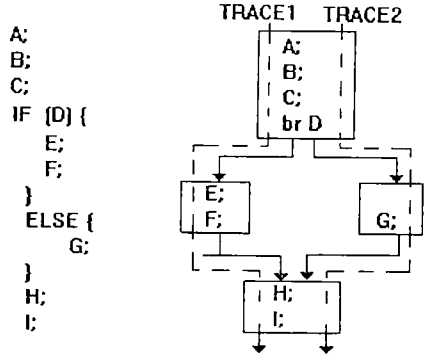


Fig. 3.11

Tehnica TS este similară cu tehnicile de reorganizare în "basic block"-uri, cu deosebirea că aici se va reorganiza o întreagă cale şi nu doar un basic block. În esenţă, ea se bazează pe optimizarea celor mai probabile căi în a fi executate. Spre exemplificare să considerăm o secvenţă de program C şi secvenţa de program obiect obţinută prin compilare:

```
a[ i ] = a[ i ]+1;
if (a[ i ] < 100) {
        count = count +1;
        b -> sum =  b -> sum + a[ i ];}

1:              SLL     R1, i, 2
2:              ADD     R1, R1, base_a
3:              LD      R2, (R1); a[ i ]
```

```
4:          ADD   R2, R2, 1; a[ i ]+1
5:          ST    R2, (R1); asignare a[ i ]
6:          CPLT  R1, R2, 100; a[ i ] < 100?
7:          JMPF  R1, LABEL
8:          ADD   count, count, 1
9:          ADD   R1, base_b, s_off; adresa lui b -> sum
10:         LD    R3, (R1)
11:         ADD   R3, R3, R2
12:         ST    R3, (R1)
LABEL:
```

Pentru a putea aplica TS compilatorul trebuie să aibă criterii rezonabile de predicție a salturilor condiționate, în vederea construirii căilor cu cea mai mare probabilitate de execuție. Există aici o mare experiență înglobată în aceste compilatoare [Cho95,Cha95,Joh91,Hen96]. În general, predicția software se face pe baza informațiilor rezultate din execuția programului neoptimizat sau a altor algoritmi euristici înglobați în compilator.

Execuția secvenței anterioare pe un procesor superscalar care decodifică 4 instrucțiuni simultan și deține 5 unități de execuție se va face ca în Tab.3.15 . S-a considerat că saltul condiționat nu se va face și că procesorul execută In Order.

DECODIFICARE				EXECUȚIE					
I0	I1	I2	I3	ALU1	ALU2	SHF	LS	BRN	CICLU
1	2	3	4						1
						1			2
				2					3
							3		4
									5
5	6	7	8		4				6
				6			5		7
9	10	11	12		8			7	8
				9					9
							10		10
									11
					11				12
							12		13

Tab. 3.15

Datorită paralelismului limitat al acestei secvențe se va obține o rată de execuție de doar o instr. / ciclu. Graful de control al acestei secvențe este complus din doar 2 basic

block-uri ca în figura 3.12.

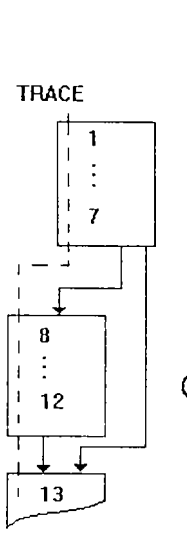


Fig. 3.12

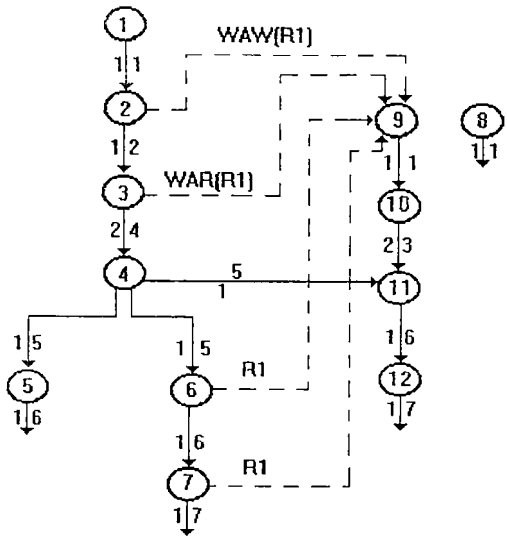


Fig. 3.13

Se va considera că programul compilator a ales calea prin care saltul nu se face spre a fi optimizată. Aşadar, instrucţiunea de salt va fi tratată în acest caz ca oricare alta. Graful precedenţelor de date asociat secvenţei anterioare de program este prezentat în fig. 3.13.

Pentru a elimina dependenţele WAR şi WAW între cele 2 ramuri paralele, vom redenumi R1 cu un alt registru disponibil (RS1) pe ramura 9, 10, 11, 12. De remarcat că această problemă se datorează alocatorului de regiştri din cadrul compilatorului. Aplicând acum algoritmul LS pe graful dependenţelor de date, obţinem următoarea ordine inversă de execuţie:

NOD / PRIORITATE
12 / 7, 7 / 7
5 / 6, 6 /6, 11 / 6
4 / 5
10 / 3
3 / 4, 9 / 1
2 / 2
1 / 1

Tab. 3.16

Aşadar, execuţia se va face în 7 cicli ca mai jos:

1)1	1:	SLL R1, i, 2
2)2	2:	ADD R1, R1, base_a
3)3,9	3:	LD R2, (R1)
9)10	9:	ADD RS1, base_b, s_off
5)4,8	10:	LD R3, (RS1)
6)5,6,11	4:	ADD R2, R2, 1
7)12,7	8:	ADD count, count, 1
	5:	ST R2, (R1)
	6:	CPLT R1, R2, 100
	11:	ADD R3, R3, R2
	12:	ST R3, (RS1)
	7:	JMPF R1, LABEL
	LABEL:	

Aşadar prin suprapunerea operaţiilor din cele 2 basic block-uri s-a obţinut o rată de procesare de 1.71 instr. / ciclu. Problema care apare însă este: ce se întâmplă dacă saltul se face?

Răspunsul ar consta în utilizarea unor coduri de compensaţie pentru predicţiile incorecte. Tehnica TS presupune mutarea instrucţiunilor dintr-un basic block în altul şi asta presupune anumite compensaţii în vederea păstrării corectitudinii programului reorganizat. Prezentăm mai jos compensaţiile necesare atunci când o instrucţiune e mutată dintr-un basic block într-unul succesor respectiv predecesor (v. fig.3.14).

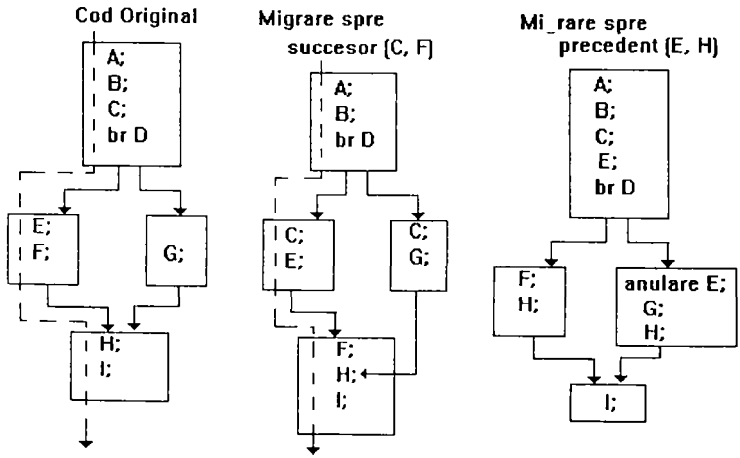


Fig. 3.14

În exemplul prezentat am avut o migrare a întregului bloc 8-12 în blocul

precedent 1-7, deci compilatorul va trebui să introducă în al 2-lea bloc instrucțiuni de compensare în vederea anulării unor operații, absolut necesare atunci când saltul se face, ca mai jos:

```
1:      SLL   R1, i, 2
2:      ADD   R1, R1, base_a
3:      LD    R2, (R1)
9:      ADD   RS1, base_b, s_off
10:     LD    R3, (RS1)
4:      ADD   R2, R2, 1
8:      ADD   count, count, 1
5:      ST    R2, (R10)
6:      CPLT  R1, R2, 100
11:     ADD   RS3, R3, R2
12:     ST    RS3, (RS1)
7:      JMPF  R1, C2
C1:     JMP   LABEL
C2:     SUB   count, count, 1
C3:     ST    R3, (RS1)
LABEL:
```

În vederea anulării operației b -> sum = b -> sum + a[i] în cazul în care saltul se face, s-a introdus de către compilator linia C3. Pentru ca această anulare să fie posibilă a fost necesară redenumirea registrului R3 cu un altul disponibil (RS3) în instrucțiunile I11 și I12. Altfel, instrucțiunea I11 ar fi alterat R3 și deci anularea asignării variabilei b -> sum în memorie ar fi fost imposibilă.

De remarcat similitudinea între corecțiile soft și tehnicile hardware de execuție speculativă prin care se redenumesc dinamic resursele procesorului. În [Joh91], se prezintă o serie de caracteristici arhitecturale cu scopul facilizării implementării tehnicii TS.

Compensarea este eficientă numai dacă ciclii suplimentari introduși prin aceasta nu depășesc numărul ciclilor eliminați prin tehnica TS. O compensare eficientă presupune [Joh91]:

- O acuratețe ridicată a predicției branch-urilor de către compilator. În acest sens programele numerice se pretează mai bine la TS decât cele de uz general.
- Procentaj relativ scăzut al instrucțiunilor de ramificație din program.
- Codurile de compensație să fie posibile și fezabile.
- Paralelism hardware pronunțat în vederea procesării codurilor de compensare cu introducerea unui număr minim de cicli suplimentari.

Într-un program dat, reorganizatorul selectează căile pentru scheduling, utilizând

tehnici de predicție software. Pentru început se aplică algoritmul TS pentru calea cea mai probabilă de a fi executată. Pe timpul acestui proces se vor adăuga coduri de compensare. Apoi se selectează următoarea cale considerată cea mai probabilă. Aceasta va fi de asemenea reorganizată prin tehnica TS. Reorganizarea se referă inclusiv la posibilele coduri compensatoare introduse de către procesul anterior de reorganizare. Algoritmul TS se va repeta pentru fiecare cale în parte.

Pentru limitarea timpului de reorganizare, referitor la căile mai puțin frecvente în execuție, se poate renunța la optimizarea globală în favoarea optimizării exclusiv a basic block-urilor componente prin algoritmul LS. Nu există încă criterii clare care să stabilească momentul în care optimizarea TS să fie abandonată.

3.8 OPTIMIZAREA BUCLELOR DE PROGRAM ÎN CADRUL PROCESOARELOR MEM

3.8.1 TEHNICA "LOOP UNROLLING"

Pentru ilustrarea principiului, să considerăm bucla de program:

```
for (i = 0; i < 64; i + +) {
sum + = a[ i ];
}
```

Din compilarea acestei bucle rezultă următoarea secvență de program obiect:

```
LOOP:   1:LD   R1, (a_ptr)
        2:ADD a_ptr, a_ptr, 4
        3:CLT count, a_ptr, end_ptr
        4:ADD sum, sum, R1
        5:JMPT      count, LOOP
```

Execuția a două iterații succesive pe un procesor superscalar "in order issue" se va desfășura ca în tabelul de mai jos:

ALU1	ALU2	SHF	LS	BRN	CICLU
2			1		1
	3				2
4					3
				5	4
2			1		5
	3				6
4					7
				5	8

Tab. 3.17

S-a considerat că procesorul aduce 4 instrucțiuni simultan și deține 5 unități de execuție distincte (BRN - unitate de execuție a instrucțiunilor de ramificație). Se remarcă faptul că în acest caz rata de procesare este de 1.25 instr./ciclu sau altfel spus de 0.25 bucle/ciclu.

După aplicarea tehnicii Loop Unrolling [Hen96,Vin96, Joh91] de 2 ori secvența anterioară devine:

```
LOOP:   1:   LD    R1, (a_ptr)
        2:   ADD   a_ptr, a_ptr, 4
        3:   ADD   sum, sum, R1
        4:   LD    R1, (a_ptr)
        5:   ADD   a_ptr, a_ptr, 4
        6:   ADD   sum, sum, R1
        7:   CLT   count, a_ptr, end_ptr
        8:   JMPT  count, LOOP
```

Reorganizând basic block-ul precedent prin metoda LS obținem următoarea secvență:

```
LOOP: 1:   LD    R1, (a_ptr)
      2:   ADD   a_ptr, a_ptr, 4
      4:   LD    R2, (a_ptr)
      5:   ADD   a_ptr, a_ptr, 4
      7:   CLT   count, a_ptr, end_ptr
      3:   ADD   sum, sum, R1
      6:   ADD   sum, sum, R2
      8:   JMPT  count, LOOP
```

Execuția acestei bucle pe același procesor superscalar se va face ca în figura 3.18:

ALU1	ALU2	SHF	LS	BRN	CICLU
2			1		1
	5		4		2
7	3				3
6				8	4

Tab. 3.18

S-a obținut o rată de procesare de 2 instr. / ciclu sau 0.5 bucle / ciclu, deci practic performanța s-a dublat față de exemplul precedent.

Se pune problema: ce se întâmplă dacă numărul de iterații este necunoscut în momentul compilării? Pentru aceasta să considerăm secvența de mai jos:

```
for (i = 0; i < n ; i + +) {
        sum + = a[ i ];
```

}

Această secvență va fi compilată ținând cont și de aplicarea tehnicii LU ca mai jos:

```
LOOP:   1:   LD    R1, (a_ptr)
        2:   ADD   a_ptr, a_ptr, 4
        3:   ADD   sum, sum, R1
        4:   CLT   count, a_ptr, end_ptr
        5:   JMPF  count, EXIT
        6:   LD    R1, (a_ptr)
        7:   ADD   a_ptr, a_ptr, 4
        8:   ADD   sum, sum, R1
        9:   CLT   count, a_ptr, end_ptr
        10:  JPMT  count, LOOP
EXIT:
```

Graful de control corespunzător acestei secvențe este prezentat mai jos:

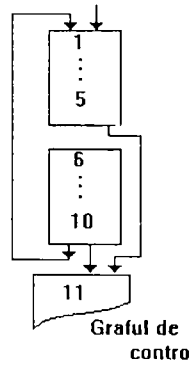


Fig. 3.15

Așadar, în astfel de cazuri după aplicarea tehnicii LU se obține o cale formată din două sau mai multe basic block-uri concatenate. Pentru optimizare se poate aplica algoritmul TS asupra grafului de control, dar aceasta poate complica și reduce sever eficiența tehnicii loop unrolling.

3.8.2 TEHNICA "SOFTWARE PIPELINING"

Tehnica software pipelining (TSP) este utilizată în reorganizarea buclelor de program astfel încât fiecare iterație a buclei de program reorganizată să conțină instrucțiuni aparținând unor iterații diferite din bucla originală. Această reorganizare are

drept scop scăderea timpului de execuție al buclei prin eliminarea hazardurilor intrinseci, eliminând astfel stagnările inutile pe timpul procesării instrucțiunilor.

Prezentăm principiul acestei tehnici bazat pe un exemplu simplu preluat din [Hen96]. Aşadar să considerăm secvența de program următoare:

```
LOOP:   LD  F0, 0(R1)
        ADD  F4, F0, F2
        SD  0(R1), F4
        SUBI  R1, R1, #8
        BNEZ  R1, LOOP
```

Bucla realizează modificarea unui tablou de numere reprezentate în virgulă mobilă, prin adăugarea unei constante conținută în registrul F2 la fiecare dintre acestea.

De remarcat că hazardurile RAW între instrucțiunile I1, I2 respectiv I2, I3 determină stagnări în procesarea pipeline a instrucțiunilor acestei bucle.

Într-o primă fază TSP desfăşoară, însă doar în mod simbolic, bucla în iterații succesive eliminând actualizarea contorului şi saltul înapoi.

```
iterația i:     LD  F0, 0(R1)
                ADD  F4, F0, F2
                SD  0(R1), F4
iterația i+1:   LD  F0, 0(R1)
                ADD  F4, F0, F2
                SD  0(R1), F4
iterația i+2:   LD  F0, 0(R1)
                ADD  F4, F0, F1
                SD  0(R1), F4
```

În a doua fază, instrucțiunile selectate din cele trei iterații se grupează în cadrul unei noi bucle ca mai jos:

```
LOOP:   SD  0(R1), F4;      memorează în M(i)
        ADD  F4, F0, F2;    modifică scalarul M(i-1)
        LD  F0, - 16(R1);   încarcă elementul M(i-2)
        SUBI  R1, R1, #8;
        BNEZ  R1, LOOP;
```

Observăm că prin pipeline-izarea instrucțiunilor din cadrul acestei bucle reorganizate s-au eliminat hazardurile anterioare şi deci stagnările implicate de către acestea. Bucla se execută pe un procesor pipeline scalar în doar 5 impulsuri de tact, adică la viteză maximă asumând o predicție perfectă a saltului.

De remarcat că pe un procesor superscalar cu suficiente unități de execuție primele 3 instrucțiuni s-ar putea executa simultan, ceea ce pe varianta originală a buclei nu se putea din cauza hazardurilor RAW. Aşadar TSP este deosebit de eficientă pe

procesoarele cu execuţie multiplă a instrucţiunilor [Hen96,Fis81].

De multe ori este necesară intervenţia hardului sau a softului pentru eliminarea posibilelor hazarduri WAR care apar în cadrul buclei reorganizate. Totuşi, pentru a fi funcţională, bucla anterioară are nevoie de un preambul şi respectiv un postambul. Preambulul necesar constă în execuţia instrucţiunilor din iteraţiile 1 şi 2 care nu au fost executate în cadrul buclei (LD - 1, LD - 2, ADD - 1). Analog, postambulul constă în execuţia instrucţiunilor care nu au fost executate în ultimele 2 iteraţii (ADD - ultima, SD - ultimele 2 iteraţii).

Alături de tehnica LU, TSP este des utilizată în optimizarea buclelor de program. Spre deosebire de LU, aceasta consumă mai puţin spaţiu de cod. Frecvent, cele 2 tehnici sunt combinate în vederea creşterii performanţei.

Tehnica LU se concentrează pe eliminarea codurilor redundante din cadrul buclelor (actualizări contoare, loop-uri). Bucla se procesează la viteză maximă doar pe parcursul iteraţiilor desfăşurate. TSP încearcă în schimb, obţinerea unei viteze mari de execuţie pe întreg timpul procesării buclei.

4. METODE NOI DE EVALUARE A PERFORMANȚELOR ȘI DETERMINARE A UNOR PARAMETRI OPTIMI DE PROIECTARE ÎN ARHITECTURILE PIPELINE ȘI SUPERSCALARE

În cele ce urmează se va prezenta un set de analize, teoretice dar și pe bază de analiză numerică, asupra performanței arhitecturilor RISC superscalare cu busuri și memorii cache separate (I-CACHE, D-CACHE) și respectiv unificate. Această problemă este extrem de puțin dezbătută în literatura de specialitate, iar multe aspecte nu sunt tratate deloc. Majoritatea specialiștilor consideră ca evidentă superioritatea globală a cache-urilor separate implementate în cadrul procesoarelor RISC superscalare și VLIW. În implementările comerciale cunoaștem doar o singură excepție în acest sens: arhitectura RISC superscalară Power PC 601, care implementază busuri și memorii cache unificate pe instrucțiuni și date.

În mod tradițional, cercetătorii abordează problemele de evaluare a performanțelor arhitecturilor pipeline, superscalare și VLIW, prin intermediul unor ample programe de simulare. Pentru aceasta este necesară dezvoltarea următoarelor instrumente: un **compilator C** - arhitectura cercetată, un **scheduler** pentru optimizarea programului sursă generat prin compilare, un **simulator parametrizabil** al arhitecturii și în fine, o **suită de benchmark-uri** care după compilare și optimizare, vor fi procesate pe simulatorul parametrizabil. Este clar că deși și-au dovedit în timp utilitatea, aceste tehnici sunt deosebit de laborioase necesitând lucrul în echipă al mai multor specialiști pe o perioadă de câțiva ani. În plus, o rezervă la această abordare este dată de **gradul de reprezentativitate al benchmark-urilor** utilizate în simulare și care niciodată nu vor putea epuiza totalitatea aplicațiilor posibile în activitatea de zi cu zi. De asemenea, aceste complexe instrumente de investigare se pretează doar la o anumită arhitectură particulară.

Ca o alternativă, se va încerca dezvoltarea unor modele analitice în evaluarea performanței. În prezenta lucrare, se vor dezvolta modele analitice bazate pe conceptele de **vector de coliziune** și **automat finit,** destinate în particular comparării performanțelor cache-urilor separate și respectiv unificate. De asemenea vom dezvolta în același scop și modele hibride de natură teoretică și empirică.

Avantajul unei asemenea abordări constă în evitarea clasicelor metode bazate pe simulare, consumatoare de timp, resurse materiale și umane enorme. Pe baza unor metode analitice ca cele dezvoltate aici se pot lua decizii rapide cu privire la proiectarea optimală și performanța unor module funcționale din cadrul arhitecturii. Un alt avantaj ar consta în faptul că rezultatele obținute prin aceste metode sunt practic independente de benchmark-urile utilizate în simulări, inevitabil limitate și particulare. De asemenea, metodele acestea sunt practic independente de multe particularități ale arhitecturii cercetate.

Revenind la una din problemele deschise ce urmează să fie abordate aici, se precizează câteva aspecte. Este cunoscut faptul că memoriile cache cu busuri separate dețin avantajul față de cele unificate de a nu determina procese de coliziune. În cazul unificatelor acest lucru este inevitabil datorită conflictelor de acces la spațiile de instrucțiuni și date, având în vedere procesarea pipeline RISC a instrucțiunilor [Smi95, Hen96, Sto93, VinL96]. Din acest punct de vedere performanța unificatelor este mai scăzută.

În acest sens, se consideră spre exemplificare un procesor scalar pipeline de tip RISC idealizat, având rata de hit în cache-uri de 100%, latența tuturor instrucțiunilor de un tact, predicție perfectă a branch-urilor, fără dependențe între instrucțiuni, etc.

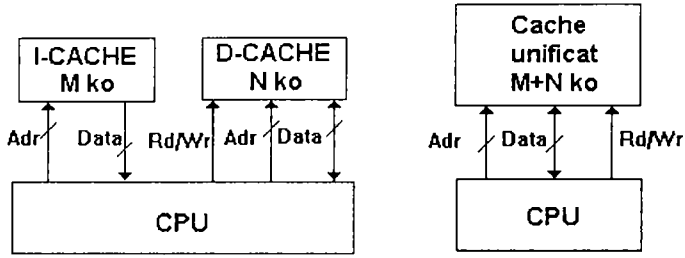


Figura 4.1

Un astfel de procesor, având busuri și cache-uri separate ar obține o ratã de procesare IR (Issue Rate) idealã de o instrucțiune per tact. Un procesor similar, având însă cache-uri unificate ar obține o ratã ideală :

IR = 1/ (1+Pmem) [instr./tact], (4.1) , unde:

Pmem = % instrucțiunilor LOAD/STORE **active** dintr-un program dat.

Cum Pmem = 0.4 tipic, rezultă ca IR scade dramatic de la o instrucțiune/tact la 0.71 instr./tact, arhitecturile unificate determinând deci o scădere a performanței cu 30% în acest caz. Acest tip de argument, simplu, este unul forte în justificarea superiorității separatelor [Hen96, Sta96, Bha96].

Totuși, în opinia autorului, există cîteva argumente care pot face din arhitecturile unificate arhitecturi deosebit de atractive. În primul rând aceste busuri și cache-uri unificate **simplifică hardware-ul** procesorului, întrucât necesită numai un set de logică de control, numai un registru de adrese și decodificator de adrese și numai o magistrală de legătură între CPU și memorie (v. Fig.4.1). Din punct de vedere al complexității proiectării și implementării reale, cache- urile unificate sunt **mai facil de integrat** decât cele separate [Rya93, IBM93]. În al doilea rând, latențele anumitor instrucțiuni (de exemplu LOAD) pot **masca efectele defavorabile ale coliziunilor** [VinL96]. În fine, un al treilea avantaj constă în faptul că **separatele dețin o rată de miss cu 5-14% mai mare decât cele unificate,** acestea din urmă oferind deci o mai bună utilizare, raportat la capacități egale. Acest ultim aspect însă, este extrem de puțin investigat totuși, la ora actuală.

Se prezintă în continuare, în Fig.4.2, preluat din [Hen96], ratele medii de miss rezultate în urma unor simulări laborioase realizate pe stații DEC 5000 (Alpha 21064) ale benchmark-urilor SPEC 92. S-au considerat cache-uri mapate direct, cele mai facil de integrat în microprocesor, având 32 octeți/bloc. Procentajul referințelor la I-CACHE a fost de 75%, considerat tipic.

O posibilă explicație, în opinia autorului, a ratei de hit superioare în spațiul de instrucțiuni, are la bază 2 motive: **frecvența net superioară a acceselor la I-CACHE** și respectiv o **mai bună compactizare a spațiului de instrucțiuni** în raport cu cel de date în cadrul benchmark-urilor SPEC 92 și probabil în majoritatea programelor uzuale.

Aceste motive ar explica în bună măsură şi rata de hit superioară corespunzătoare cache-urilor unificate. În plus, cache- urile unificate, spre deosebire de cele separate, permit o **balansare dinamică între spaţiile de acces la instrucţiuni şi respectiv date** [Rya93]. De altfel şi această problemă a balansării optimale între I-Cache şi D-Cache este o problemă deschisă în opinia autorului. Drept dovadă, se prezintă capacităţile I şi D-Cache alocate în cadrul unor microprocesoare MEM actuale.

- AMD K5: 16ko 4 way set associative I-Cache, 8ko D-Cache.
- Pentium Pro şi Alpha 21164: 8ko I-Cache, 8ko D-Cache.
- MIPS R 10000: 32ko I-Cache , 32ko D-Cache.
- Ultra SPARC: 16ko I-Cache, 16ko D-Cache; Super SPARC: 20ko I-Cache, 16ko D-Cache.
- IBM RS/6000: 8ko I-Cache, 64ko D-Cache

Contradicţia rezidă în faptul că toate acestea sunt microprocesoare de uz general şi deci trebuie să execute practic aceleaşi programe!

Probabil din aceste motive, multe (dar nu toate!) procesoare RISC superscalare comerciale au cache-ul de instrucţiuni de capacitate mai mare decât cel de date.

Capacitate ko	I-CACHE %	D-CACHE %	CACHE Unificat %
1	3.06	24.61	13.34
2	2.26	20.57	9.78
4	1.78	15.94	7.24
8	1.10	10.19	4.57
16	0.64	6.47	2.87
32	0.39	4.82	1.99
64	0.15	3.77	1.35
128	0.02	2.88	0.95

Figura 4.2

De exemplu, măsurat pe benchmark- urile SPEC 92, microprocesorul superscalar Super SPARC (SUN) atinge o rată de hit în I-Cache de 98% iar în D-Cache de 90% [Sun92].

Singura excepţie după cum am mai arătat o constituie microprocesorul superscalar Power PC 601 [Smi94, IBM93, Weis94] care combină un bus şi un cache unificat de mare capacitate (32 ko), având o rată de fetch maximă de 8 instrucţiuni, un buffer de prefetch de 8 instrucţiuni şi posibilitatea de a lansa în execuţie maxim 3 instrucţiuni din acest buffer de prefetch. Din păcate nicăieri nu se justifică alegerea acestei soluţii (departe de a fi optimă după cum se va putea constata) şi parametri, pe o bază teoretică sau/şi de simulare, ci doar pe faptul că e mai simplu aşa şi pe considerente cel mult de " bun simţ".

În continuare se vor prezenta câteva metode originale de investigare a performanţei cache-urilor şi busurilor unificate pe instrucţiuni şi date (I/D) comparativ cu cele separate, implementate în cadrul arhitecturilor cu execuţii multiple ale instrucţiunilor. De asemenea se vor dezvolta câteva consideraţii legate de determinarea unor condiţii analitice, referitoare la o proiectare optimală a acestor arhitecturi. Se precizează că- cu excepţia ultimelor două metode prezentate în 4.3 şi 4.5 - toate celelalte metode dezvoltate aici sunt de tip cvasi-analitic (analiză teoretică şi numerică) şi abordează această problematică exclusiv prin prisma **proceselor de coliziune** implicate de către arhitecturile cu busuri unificate (fără a aborda deci efectiv problema cache-urilor ci doar

aceea a busurilor. În Cap. 5 însă, se va aborda problema în toată complexitatea ei, pe bază de simulare). Penultima abordare însă, din paragraful 4.3, încearcă tot pe o bază teoretică, de data aceasta mai simplă, o **tratare globală** a comparației analitice între cache-urile unificate și cele separate ținând cont de toate fenomenele care apar aici (coliziuni și respectiv procese de hit/miss în cache-uri). În ultimul paragraf, se prezintă câteva considerații de natură analitică relative la stabilirea unor parametri optimali pentru o arhitectură superscalară, pe baza principiului "procese multe, resurse hardware puține".

4.1. EVALUĂRI PE BAZA CONCEPTULUI VECTORILOR DE COLIZIUNE

4.1.1. PERFORMANȚA UNUI PROCESOR SUPERSCALAR CU BUSURI UNIFICATE

Scopul principal constă în determinarea cantitativă a scăderii de performanță pentru o arhitectură superscalară având busuri unificate (și implicit cache-uri unificate) față de una cu busuri separate (și implicit cache-uri separate, v. Fig.4.1). Se vor avea în vedere deci, exclusiv procesele de coliziune implicate de către primul tip de arhitectură.

Se consideră o arhitectură simplificată de procesor superscalar având rata de fetch a instrucțiunilor (FR) de 2 instrucțiuni, rata maximă de execuție (IRmax) tot de 2 instrucțiuni și rata de hit în cache de 100%. Se consideră de asemenea că procesorul deține un cache unificat având două porturi de citire -scriere (realist) și că include un mecanism perfect de predicție a branch-urilor.

Fiecare instrucțiune este procesată într-o structură pipeline având 4 stagii cu următoarea semnificație:

IF - fetch instrucțiune din I-CACHE;

ID - decodificare instrucțiune, selecție operanzi, calcul adrese în cazul instrucțiunilor LOAD/STORE și de salt;

ALU/MEM - fază de execuție, respectiv acces la memoria de date;

WB - înscriere a rezultatului în registrul destinație.

Desigur că în cazul procesării pipeline, fazele IF și MEM nu se pot suprapune datorită busurilor de legătură unificate între CPU și cache.

Din punct de vedere al proceselor de coliziune (IF, MEM), există 2 tipuri de instrucțiuni distincte: instrucțiunile LOAD/STORE, singurele care exploatează nivelul MEM (grupul A) și respectiv restul instrucțiunilor mașinii care deci nu exploatează nivelul MEM (grupul B). Bazat pe conceptul de **vector de coliziune atașat** unei structuri pipeline [Sto93, VinL96], prezentat de altfel și în capitolul 2 al acestei lucrări, se poate scrie:

$$VC(A,A) = VC(AB) = 010 \qquad (4.2)$$

$$VC(B,A) = VC(B,B) = 000 \qquad (4.3)$$

Prin VC(i,j) am notat vectorul de coliziune al unei instrucțiuni curente din grupul i urmată de o instrucțiune din grupul j, unde i,j aparține {A,B}.

Având în vedere amintitele caracteristici ale arhitecturii simplificate considerate, am dezvoltat două automate de fetch a instrucțiunilor pentru această arhitectură. În cele 2 figuri următoare se prezintă principiul utilizat în construcția acestor automate, prin reprezentarea unor tranziții tipice. Automatele reprezentate complet pot fi găsite în

lucrările [VinL96, Vint96] În fiecare reprezentare, în dreptunghi este înscrisă instrucțiunea dublă adusă din I-CACHE (AA, AB, BA, BB) și cei doi vectori de coliziune aferenți. Din fiecare stare ies 4 arce, în conformitate cu cele 4 posibilități de aducere a următoarei instrucțiuni duble. Pe fiecare arc este înscris un număr reprezentând numărul de tacte necesare până la startarea aducerii următoarei instrucțiuni. Acest număr este funcție de vectorii de coliziune care definesc starea respectivă a automatului.

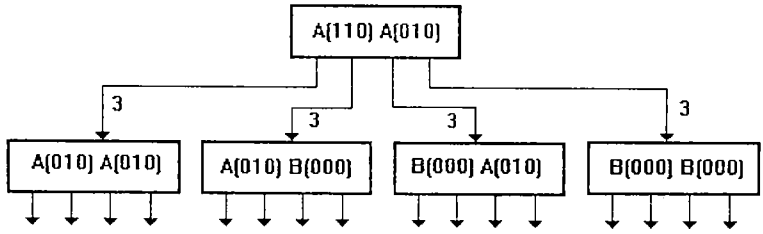


Figura 4.3

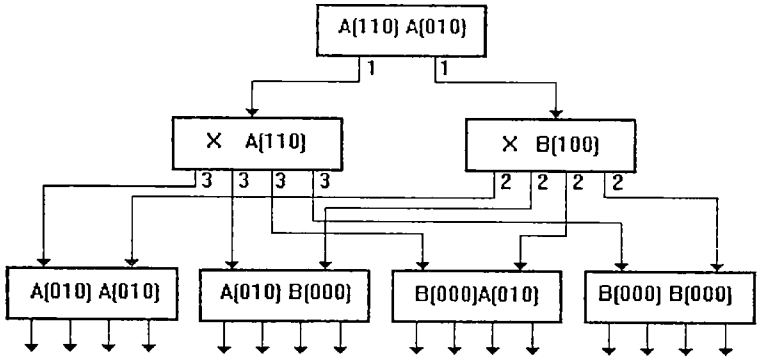


Figura 4.4

Am dezvoltat două modele de astfel de automate, al doilea fiind mai agresiv în strategia de aducere a instrucțiunilor după cum se poate observa (v. Figura 4.4).

În continuare scopul a fost să se determine o relație de tipul:

FR = IR = f(Pmem), exprimat în [instr./tact] (4.4),

unde FR și IR reprezintă ratele medii de fetch respectiv execuție a instrucțiunilor iar Pmem reprezintă procentajul instrucțiunilor LOAD/STORE active din program.

În acest scop, s-a scris un program simulator pe baza algoritmilor de fetch implementați în cadrul celor două automate. Acest program cere utilizatorului parametrul Pmem. Odată furnizat acest parametru programul generează automat 10 "programe" (fișiere text), fiecare dintre acestea conținând câte 1000 de "instrucțiuni" A și B dintre care Pmem*100% sunt instrucțiuni din grupul A, distribuite practic aleator în cadrul fișierului. În continuare pe baza acestor fișiere generate, se simulează procesarea pe baza

celor două modele de automate anterior prezentate. În fiecare caz se calculează rata medie de procesare IR [instr./tact]. Interesant este faptul că toate cele 10 rate IR obținute pentru un anumit procentaj Pmem, sunt cvasiidentice, fiind **practic independente de distribuția instrucțiunilor** din grupul A. Așadar IR depinde doar de procentajul instrucțiunilor A în program.

Se prezintă funcțiile IR = f(Pmem) așa cum au fost obținute pentru cele două modele analizate(Figura 4.5).

Pentru Pmem = 30% (tipic), rata de procesare a modelului mai agresiv este IR = 1.49 instr./tact față de IR = 2 instr./tact cât ar procesa în aceleași condiții un procesor având cache-uri separate pe instrucțiuni și date. Așadar rezultă o degradare cu cca 26% a performanțelor, datorită proceselor de coliziune implicate.

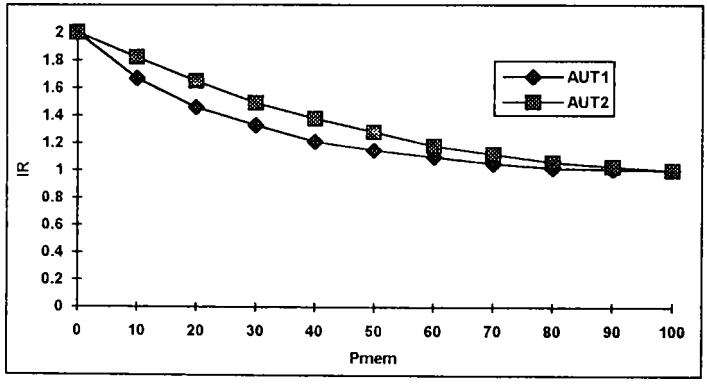


Figura 4.5

În tabelul următor se prezintă ratele de procesare obținute în baza modelului mai agresiv de automat (Figura 4.6), în urma "execuției" unor secvențe de program de tip "trace", de această dată deterministe deci, de 100 instrucțiuni, aparținând celor 8 benchmark-uri consacrate, dezvoltate la Universitatea din Stanford. Se prezintă mai jos caracteristicile trace-urilor analizate și ratele de procesare obținute.

Benchmark	Distrib. LOAD/STORE %	IR
puzzle	16	1.72
sort	24	1.56
matrix	20	1.65
tree	30	1.51
tower	38	1.42
queen	30	1.47
permute	40	1.35
bubble	26	1.51

Figura 4.6

De remarcat și în aceste evaluări făcute pe secvențe reale ale unor programe consacrate de test, o deplină concordanță cu rezultatele evaluărilor anterioare, bazate pe distribuții cvasialeatoare. S-a obținut o performanță medie de 1.52 instr./tact (la o medie

de 26% a distribuției instrucțiunilor LOAD/STORE în cadrul secvențelor de program considerate), adică o degradare cu cca. 24% a performanței raportat la un procesor echivalent dar cu busuri și cache-uri separate.Detalii suplimentare despre această problemă pot fi găsite în [VinL96].

Totuși, modelele și metoda utilizată sunt relativ simpliste, rezultatele obținute fiind considerate doar orientative. Mai mult, aceste rezultate sunt defavorabile arhitecturilor cu busuri unificate întrucît metoda utilizată nu are în vedere fenomenul de prefetch. Acesta poate "ascunde" mult din așteptările provocate de către procesele de coliziune.

Un alt punct slab al prezentei abordări constă în faptul că nu am considerat dependențele posibile și probabile între instrucțiunile care urmau a fi lansate în execuție și care limitează serios rata de procesare. Această simplificare este însă comună ambelor modele de arhitecturi comparate, în plus, a interesat exclusiv problema abordată.

Oricum, în continuare se va aborda aceeași problemă însă pe baza unui model mai realist, parametrizabil și care să țină cont de observațiile anterior formulate. După cum se va putea constata, diferența reală de performanță între arhitecturile cu cache-uri separate și respectiv cele unificate, devine mult mai mică, făcând în opinia autorului din arhitecturile unificate o opțiune posibil superioară în anumite condiții, prin prisma raportului performanță/ cost implicat.

4.1.2.INFLUENȚA INSTRUCȚIUNILOR LOAD ASUPRA PERFORMANȚEI ARHITECTURILOR UNIFICATE

În continuare, bazat pe același concept al vectorilor de coliziune [Sto93], se va încerca determinarea într-un mod cantitativ, a influenței defavorabile a instrucțiunilor de tip LOAD asupra performanței unui procesor RISC scalar idealizat, având busuri unificate pe instrucțiuni și date.

Pentru aceasta, se va considera o arhitectură scalară RISC având 5 stagii de procesare pipeline-izate (IF, ID, ALU, MEM, WB). De asemenea, se consideră un LDS (Load Delay Sloat) de o instrucțiune (tipic), precum și o unitate funcțională de LOAD nepipeline-izată.

Un astfel de procesor RISC, având busuri și cache-uri separate în spațiile de instrucțiuni și date (arhitectură Harvard) și care are latențe ale instrucțiunilor și memoriilor de un tact, ar realiza o rată de procesare:

$$IRs = 1/(1 + Pl) \qquad (4.5)$$

Prin **Pl** am notat procentajul instrucțiunilor LOAD active dintr-un program dat. Am considerat că procesorul temporizează un tact, prin hardware sau software scheduling, în cazul fiecărei instrucțiuni LOAD executată.

În continuare se propune determinarea unei relații de tipul IRu = f(Pl, Ps), pentru o arhitectură non-Harvard, unde Ps reprezintă procentajul instrucțiunilor STORE active dintr-un program dat. Evident că: Pl + Ps = Pmem.

Modelul de procesare pentru o astfel de arhitectură cu busuri unificate, este prezentat în continuare în figura 4.7.

Prin L, S și X am notat o instrucțiune LOAD, STORE sau alt tip de instrucțiune adusă la finele fazei IF. L(0010) și S(0010) semnifică faptul că o instrucțiune inițială în program de tip LOAD sau STORE, poate determina peste două impulsuri de tact o coliziune la busurile unificate și la cache-ul comun, prin suprapunerea fazelor MEM

aferente acestor instrucțiuni cu faza IF aferentă uneia dintre instrucțiunile următoare. Desigur că în filozofia RISC doar instrucțiunile LOAD/STORE pot determina asemenea procese de coliziune.

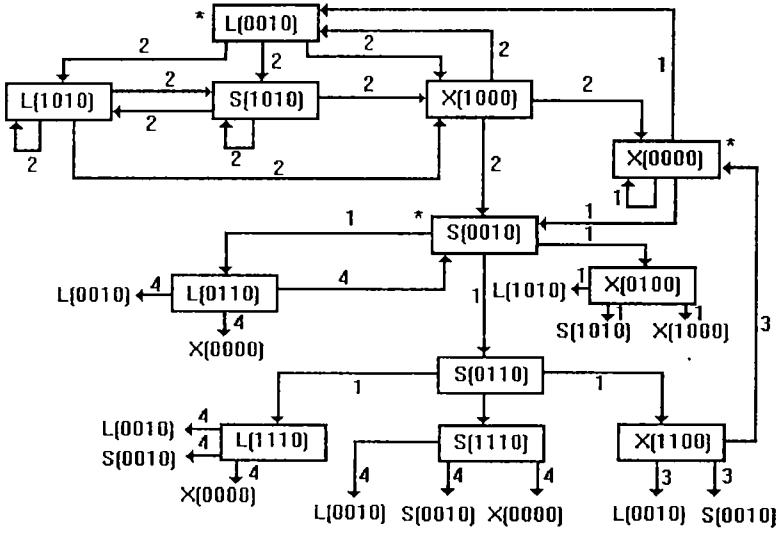


Figura 4.7

Numărul înscris pe cele 3 arce care ies din fiecare stare semnifică numărul de tacte după care se poate aduce o nouă instrucțiune (L, S sau X), relativ la instrucțiunea curentă. Acesta este principiul de construcție al grafului anterior, care modelează procesarea instrucțiunilor pe un procesor scalar RISC de tip non-Harvard. După cum se observă, graful are 12 stări Si i = 1,...,12. Probabilitățile Pi ca automatul să se afle într-o anumită stare Si se pot determina pe baza celor 12 ecuații liniare omogene de echilibru precum și a relației de normalizare:

$$\sum_{i=1}^{12} Pi = 1 \qquad (4.6)$$

Automatul poate starta din una dintre cele 3 stări marcate cu (*). Din punct de vedere teoretic avem:

$$IRu = 1/\sum_{i=1}^{12} Pi * Ni, \qquad (4.7), \text{ unde:}$$

Ni = latența aferentă stării Si

Pe baza acestui model s-au realizat simulări pe o vastă arie de programe în scopul determinării ratei de procesare IRu = f(Pl, Ps). În graficul următor se prezintă rezultatele

obținute pentru procentaje Pl = Ps cuprinse între 0% și 45% relativ la o arhitectură non-Harvard de procesor.

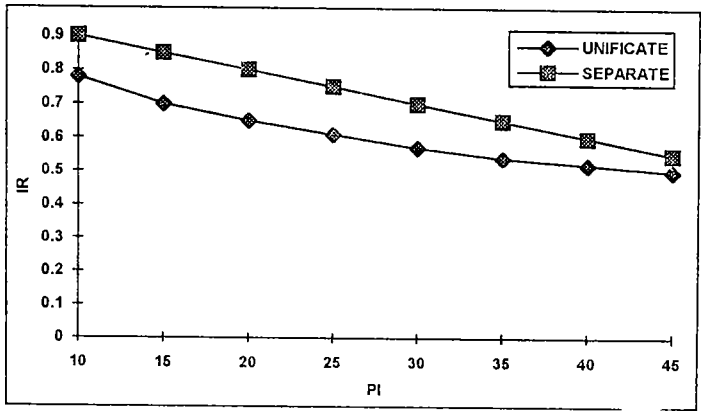


Figura 4.8

De asemenea se prezintă IR = f(Pmem) pentru o arhitectură Harvard, în scopul determinării diferențelor de performanță între cele două opțiuni arhitecturale.

Se poate concluziona că pentru procentaje tipice de Pmem = 30% (dintre care 50% LOAD-uri), performanța unui procesor RISC non-Harvard față de unul Harvard scade cu până la 18%, exclusiv din cauza proceselor de coliziune, dar acest fapt ar putea fi compensat dintr-un punct de vedere global de avantajele introduse în hardware printr-o astfel de arhitectură precum și de faptul că arhitecturile unificate de cache-uri oferă o rată de hit superioară față de cele separate cu cca 10-15% [Hen96].Pe de altă parte, instrucțiunile LOAD pot masca în parte procesele de coliziune la busurile comune (v. Fig.4.7). Această analiză a abordat problema exclusiv din punct de vedere al proceselor de coliziune introduse de către arhitecturile unificate. Detalii metodologice și rezultate suplimentare despre această investigație pot fi găsite în [VinL96].

4.2. NOI ABORDĂRI ANALITICE ALE ARHITECTURILOR MEM CU BUSURI UNIFICATE PE INSTRUCȚIUNI ȘI DATE UTILIZÂND AUTOMATE FINITE DE STARE

4.2.1. PRINCIPIUL PRIMEI METODE DE ANALIZĂ PROPUSĂ

Se propune dezvoltarea unui model teoretic de evaluare a performanței arhitecturilor pipeline, bazat pe modelarea acestor arhitecturi folosind **automate finite de stare.** În particular se va aplica acest model în mod concret, în scopul evaluării comparative a procesoarelor RISC superscalare având busuri și cache-uri unificate respectiv separate I/D, continuând astfel investigarea începută anterior pe o bază mai complexă și mai realistă.

De precizat că evaluarea comparativă se va baza preponderent pe estimarea cantitativă a pierderii de performanță implicată de busurile unificate și datorată coliziunilor între fazele IF și MEM.

Faţă de modelul precedent, noutatea constă în apariţia unui **buffer de prefetch de capacitate parametrizabilă** între nivelele IF şi ID din cadrul structurii pipeline (v. Fig.4.9). Am investigat 2 tipuri de structuri pipeline tipice pentru arhitecturile actuale: una având 4 stagii pipeline (IF, ID, ALU/MEM, WB) şi alta având 5 stagii, cu stagii separate ALU şi MEM (IF, ID, ALU, MEM, WB). Ca şi în cazul precedent am considerat latenţa tuturor instrucţiunilor de un tact şi o predicţie perfectă a branch-urilor. Unul dintre obiectivele importante ale acestei investigaţii a fost să se determine fracţiunea din timpul total al procesorului cât bufferul de prefech poate furniza instrucţiuni spre execuţie pe timpul coliziunilor.

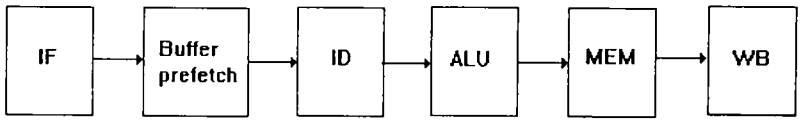


Figura 4.9

În cazul structurilor pipeline cu 4 stagii, fiecare stare a automatului conţine 2 componente ortogonale S(IB) şi S(ID). Componenta S(IB) reprezintă numărul de instrucţiuni memorate curent în bufferul de prefetch iar componenta S(ID) reprezintă o valoare binară care dacă este '1' arată că în faza ID se află o instrucţiune LOAD/STORE. Aşadar componenta S(ID) arată dacă este '1' că în următorul impuls de tact se va face un acces la spaţiul de date şi prin urmare procesul de prefetch va trebui stagnat.

Analog, în cazul structurilor pipeline cu 5 stagii, fiecare stare a automatului va conţine 3 componente ortogonale: S(IB), S(ID) şi S(ALU). Primele două au o semnificaţie identică cu cea descrisă anterior. Componenta S(ALU) indică dacă este '1' că în faza ALU a structurii pipeline se află o instrucţiune LOAD/STORE şi deci peste un tact va apare un potenţial conflict la memorie care se va rezolva prin stagnarea prefetch-ului în tactul următor. În acest caz, S(ID) va semnifica dacă e '1' că peste 2 tacte vom avea un conflict structural la memorie.

Automatele dezvoltate vor avea 3 parametri importanţi: rata de fetch a instrucţiunilor (**FR-Fetch Rate**), rata maximă de execuţie a instrucţiunilor (**IR-Issue Rate**) şi respectiv capacitatea bufferului de prefetch (**IBS-Instruction Buffer Size**).

Ca un exemplu, se prezintă automatul aferent unui procesor cu 4 stagii având FR = 2, IR = 2 şi IBS = 4. Automatul conţine 10 stări în acest caz. Probabilităţile de tranziţie din fiecare stare pot fi calculate funcţie de doar 2 parametri: **P(2instr)**, adică probabilitatea ca să se trimită în execuţie din bufferul de prefetch 2 instrucţiuni şi care implicit sunt independente, şi respectiv **Pmem**, care reprezintă probabilitatea ca instrucţiunea trimisă spre execuţie să fie una de tip LOAD/STORE. Se consideră că dacă bufferul de prefetch nu este gol, întotdeauna există cel puţin o instrucţiune care să poată fi executată. Aşadar, dacă notăm P(1instr) probabilitatea ca să fie trimisă spre execuţie o singură instrucţiune, avem egalitatea:

$$P(1instr) + P(2instr) = 1 \qquad (4.8)$$

Cu aceste precizări prezentăm în cazul particular anterior descris, automatul aferent acestei arhitecturi.

Stare prez.	Tranziție	Stare urm.	Probab. tranz.
1. 0,0	Fetch 2, Ex 0	2, 0	1
2. 0,1	Fetch 0, Ex 0	0, 0	1
3. 1,0	Fetch 2, Ex1	2,0	1-Pmem
		2,1	Pmem
4. 1,1	Fetch 0, Ex 1	0, 0	1 - Pmem
		0,1	Pmem
5. 2,0	Fetch 2, Ex 1	3,0	P(1) (1-Pmem)
		3,1	P(1) Pmem
	Fetch 2, Ex 2	2,0	$P(2) (1-Pmem)^2$
		2,1	$P(2) (1-(1-Pmem)^2)$
6. 2,1	Fetch 0, Ex 1	1,0	P(1) (1-Pmem)
		1,1	P(1) Pmem
	Fetch 0, Ex 2	0,0	$P(2) (1-Pmem)^2$
		0,1	$P(2) (1-(1-Pmem)^2)$
7. 3,0	Fetch 2, Ex 1	4,0	P(1) (1-Pmem)
		4,1	P(1) Pmem
	Fetch 2, Ex 2	3,0	$P(2)(1-Pmem)^2$
		3,1	$P(2) (1-(1-Pmem)^2)$
8. 3,1	Fetch 0, Ex 1	2,0	P(1) (1-Pmem)
		2,1	P(1) Pmem
	Fetch 0, Ex 2	1,0	$P(2) (1-Pmem)^2$
		1,1	$P(2) (1-(1-Pmem)^2)$
9. 4,0	Fetch 0, Ex 1	3,0	P(1) (1-Pmem)
		3,1	P(1) Pmem
	Fetch 2, Ex 2	4,0	$P(2) (1-Pmem)^2$
		4,1	$P(2) (1-(1-Pmem)^2)$
10. 4,1	Fetch 0, Ex 1	3,0	P(1) (1-Pmem)
		3,1	P(1) Pmem
	Fetch 0, Ex 2	2,0	$P(2) (1-Pmem)^2$
		2,1	$P(2) (1-(1-Pmem)^2)$

Pe baza acestui automat se poate scrie următorul sistem de 10 ecuații liniare și omogene. Am notat: Pi - probabilitatea ca automatul să fie în starea i, a = P(1instr), b= Pmem.

$$
\begin{aligned}
P1 &= (1 - a) (1 - b)^2 P6 + P2 + (1 - a) P4 \\
P2 &= b P4 + (1 - a) (1 - (1 - b)^2) P6 \\
P3 &= a (1 - b) P6 + (1 - a) (1 - b)^2 P8 \\
P4 &= a b P6 + (1 - a) (1 - (1 - b)^2) P8 \\
P5 &= P1 + (1 - b) P3 + (1 - a) (1 - b)^2 (P5 + P10) + a (1 - b) P8 \\
P6 &= b P3 + a b P8 + (1 - a) (1 - (1 - b)^2) (P5 + P10) \\
P7 &= a (1 - b) P5 + (1 - a) (1 - b)^2 P7 + a(1 - b) (P9 + P10) \\
P8 &= a b P5 + a b (P9 + P10) + (1 - a) (1 - (1 - b)^2) P7 \\
P9 &= a (1 - P7 + (1 - a) (1 - b)^2 P9 \\
P10 &= a b P7 + (1 - a) (1 - (1 - b)^2) P9
\end{aligned}
\qquad (4.9)
$$

Desigur că există relația de normalizare:

$$\sum_{i=1}^{10} Pi = 1 \qquad (4.10)$$

Pe această bază se pot determina după soluționarea sistemului 2 parametri deosebit de importanți: **rata medie de execuție a instrucțiunilor (IRu)** și respectiv procentajul de timp cât bufferul de prefetch este gol **(IBE)** și deci nu mai poate "masca" eventualele procese de coliziune.

În acest caz particular, acești 2 parametri sunt:

$$IRu = P3 + P4 + (2 - P(1instr))\,(P5 + P6 + P7 + P8 + P9 + P10) \qquad (4.11)$$
$$IBE = (P1 + P2)\ 100\% \qquad (4.12)$$

De observat că rata medie de procesare pentru o arhitectură identică dar cu memorii și busuri separate (I-CACHE, D-CACHE) este:

$$\mathbf{IRs} = 2 - P(1instr) \qquad (4.13)$$

Într-un mod analog pentru o structură pipeline cu 5 stagii și aceleași caracteristici (FR = 2, IR = 2, IBS = 4) s-ar obține un automat similar. Se prezintă în acest caz doar un exemplu de tranziție dintr-o stare în alta în cazul acestui automat.

Stare prez.	**Tranziție**	**Stare urm.**	**Probab. tranz.**
3,1,0	Fetch2, Ex1	4,0,1	P(1) (1-Pmem)
		4,1,1	P(1) Pmem
	Fetch2, Ex2	3,0,1	$P(2)\ (1\text{-}Pmem)^2$
		3,1,1	$P(2)(1\text{-}\ (1\text{-}Pmem)^2)$

În acest caz particular vom obține însă un sistem de 20 ecuații liniare și omogene.

În general, pe baza automatului de tranziții se obține un sistem de N ecuații liniare și omogene furnizat de **ecuația matricială de echilibru**:

$$I\,P\,I = I\,P\,I * I\,T\,I, \qquad (4.14), \text{ unde:}$$
$$I\,P\,I = I\,P1\ P2\ \ldots\ldots\ PN\,I \text{ iar}$$
$$I\,T\,I = II\ t_{ij}\ II \text{ unde } i, j \text{ aparțin } \{1, 2, \ldots, N\}$$

S-au utilizat următoarele notații :

Pi = probabilitatea ca automatul să se afle în starea i;
N = numărul total de stări; N = (IBS + 1) * 4;
t_{ij} = probabilitatea de tranziție din starea i în starea j a automatului.

Cum automatul trebuie să fie la un moment dat în una dintre cele N stări, este îndeplinită **relația de normalizare:**

$$\sum_{i=1}^{N} Pi = 1 \qquad (4.15)$$

Sistemul de N ecuații omogene se poate rezolva prin metode numerice bine cunoscute. După rezolvarea acestuia, probabilitățile obținute vor fi folosite pentru calculul ratei medii de procesare (IRu) și respectiv a fracțiunii de timp cât bufferul de prefetch este gol și deci nu poate "ascunde" procesele de coliziune apărute.

Așadar, la modul general, se vor calcula următorii parametri importanți:

IRu = f(P(2instr), Pmem, P1, ..., PN); (4.16)
IRs = 2 - P(1instr); (4.17)
IBE = f(P1,... ,PN) 100%. (4.18)

În final, pe baza acestor relații se vor putea compara ratele medii de procesare obținute pentru diverse arhitecturi, stabilindu-se astfel o comparație cantitativă, pe bază teoretică, a celor 2 tipuri de arhitecturi: cu busuri de memorie unificate respectiv separate I/D. S-au folosit modele care au necesitat rezolvarea unor sisteme de ecuații liniare omogene de până la 70 de ecuații.

4.2.1.1. REZULTATE OBȚINUTE PE BAZA PRIMEI METODE

În primul rând s-a dorit să se compare performanțele relative pentru 2 arhitecturi cu busuri unificate: una cu 4 stagii, iar cealaltă cu 5 stagii. Pentru ambele arhitecturi am considerat FR = IR = 2, IBS = 4 și Pmem = 0.2. După cum se observă în figura 4.10 performanțele sunt cvasiidentice în ambele cazuri. Teoretic, performanțele arhitecturii cu 4 stagii sunt sensibil mai bune, însă considerăm că această superioritate este practic neglijabilă. Putem deci concluziona că performanța celor 2 modele este practic aceeași așa că în continuare vom prezenta doar rezultatele obținute pentru modelul cu 5 stagii de procesare. Mai mult, în baza altor investigații realiste pe care le-am făcut, a rezultat că **performanța scade nesemnificativ cu creșterea numărului de stagii din structura pipeline,** din punctul de vedere abordat aici.

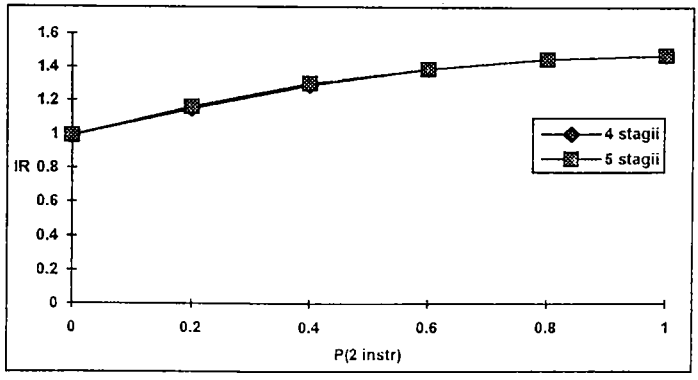


Figura 4.10

În cele ce urmează se vor prezenta curbele IRu = f(P(2instr)) pentru diferite proporții ale instrucțiunilor LOAD/STORE în program (Pmem variabil). În figura următoare (4.11) se prezintă rezultatele obținute considerând FR = IR = 2 și IBS = 4. În mod natural creșterea valorii parametrului Pmem determină degradarea performanței. În

particular, pentru Pmem = 0, performanța unificatelor este identică cu cea a separatelor, întrucât nu există probleme de coliziune care sa degradeze performanța primelor.

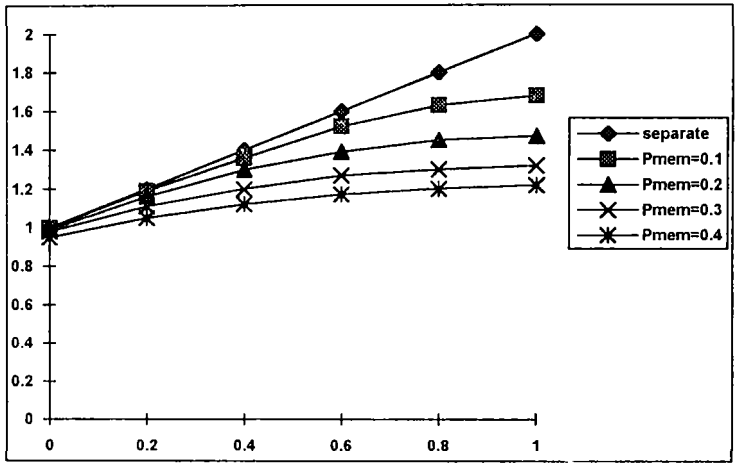


Figura 4.11

Pentru valori tipice, de exemplu Pmem = 0.4 și P(2instr) = 0.5, performanța arhitecturilor separate este de 1.5 instr./tact iar în cazul celor unificate ea scade la 1.14 instr./tact, adică cu 24%. După cum se va arăta în continuare, această diferență de performanță poate fi redusă fie prin mărirea parametrului IBS, fie prin mărirea FR, fie prin mărirea simultană a ambilor parametri. În figurile următoare se prezintă influența capacității bufferului de prefetch asupra performanței arhitecturilor unificate. S-au ales în ambele cazuri IR = 2 și FR = 4, iar proporțiile instrucțiunilor LOAD/STORE în cadrul programelor de 30% respectiv 40%, valori tipice de altfel.

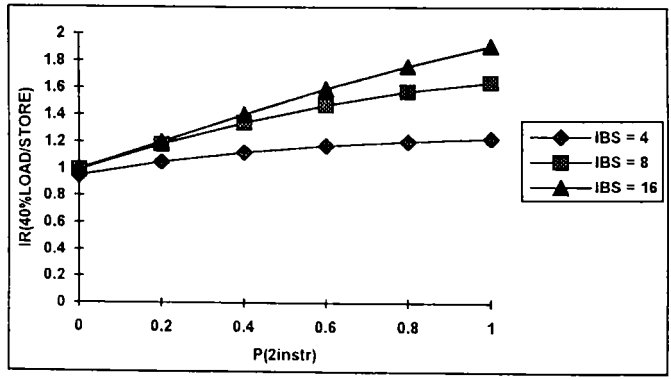


Figura 4.12

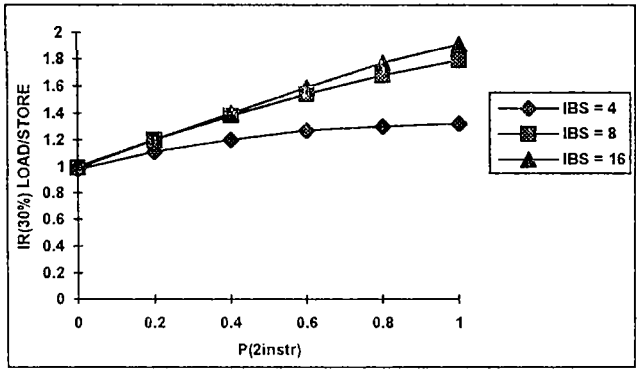


Figura 4.13

Important este în acest caz că pentru acciaşi parametri tipici aleşi în cazul anterior, anume Pmem = 0.4 şi P(2instr) = 0.5, performanţa arhitecturilor unificate atinge 1.48 instr./tact cu numai 1.2% inferioară arhitecturilor separate echivalente. De asemenea, performanţa medie creşte cu 10% de la un IBS = 4 la un IBS = 16.

În continuare se prezintă influenţa ratei de fetch a instrucţiunilor (FR) asupra performanţei arhitecturii, practic în aceleaşi condiţii ca şi în cazul anterior analizat (IBS = 16, Pmem = 40% şi IR = 2).

În acest caz, diferenţa de performanţă între arhitecturile unificate şi cele separate în condiţii identice (Pmem = 0.4, P(2instr) = 0.5) este de doar 0.38%, adică realmente nesemnificativă. Având în vedere utilizarea mai bună a memoriilor cache unificate, este posibil ca per ansamblu acestea să se comporte chiar mai bine în condiţiile de proiectare impuse prin parametrii aleşi (FR = 8, IR = 2, IBS = 16), condiţii considerate ca fiind **deosebit de realiste.**

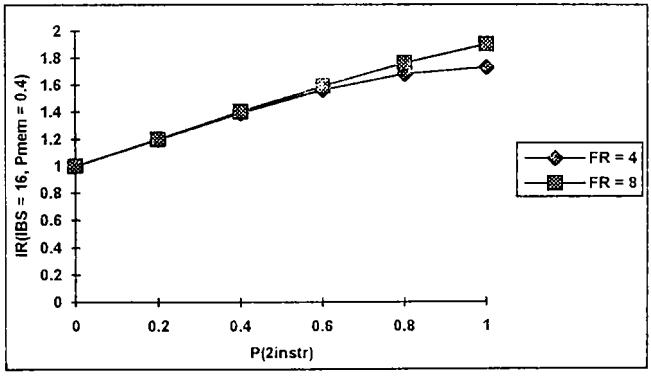


Figura 4.14

Cred că în ciuda părerii practic unanime, o proiectare având parametrii IR = N, FR = 2N,..., 4N, IBS = 2FR,.....,4FR şi combinată cu un algoritm eficient de prefetch poate conduce arhitecturile unificate la performanţe comparabile sau chiar superioare celor separate. În plus, există simplificări atrăgătoare în proiectarea hardware a acestora.

Figura următoare confirmă faptul că alegând capacitatea bufferului de prefetch de 16 instrucțiuni și o rată de fetch de 4 sau 8 instrucțiuni, în ipoteza susținută de practică în care P(2instr) < 0.8, bufferul de prefetch rămâne gol mai puțin de 4% din timpul total.

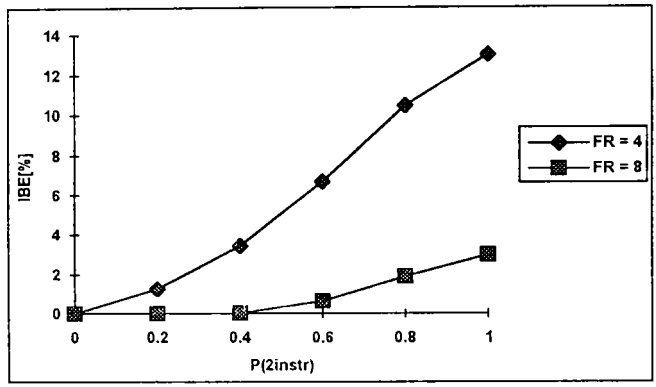


Figura 4.15

Acest lucru confirmă faptul că o strategie performantă de prefetch dublată de o acuratețe ridicată în predicția branch-urilor, maschează practic total procesele de coliziune la I-CACHE și D-CACHE, făcând astfel deosebit de atractivă arhitectura unificată.

În următoarele 3 figuri, se prezintă evaluări cantitative ale ratei medii de procesare a instrucțiunilor (IR) funcție de P(2instr) pentru diferite modele de procesoare. În fiecare caz, estimările sunt făcute pentru diferite distribuții considerate rezonabile ale instrucțiunilor LOAD/STORE în program [Mil89]. De asemenea, în fiecare caz se prezintă și rata de procesare pentru cache-uri separate în scopul de a oferi o viziune sintetică și rapidă asupra diferenței de performanță între cele două arhitecturi.

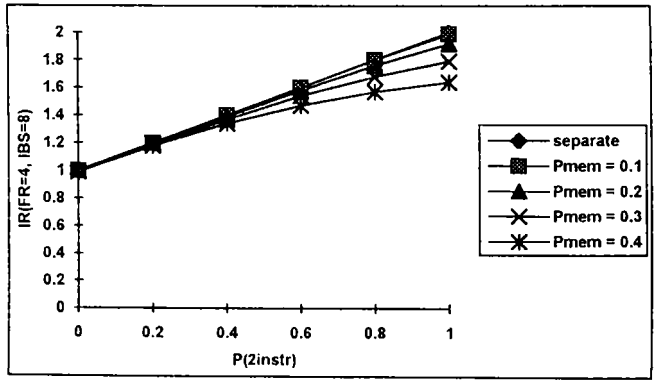


Figura 4.16

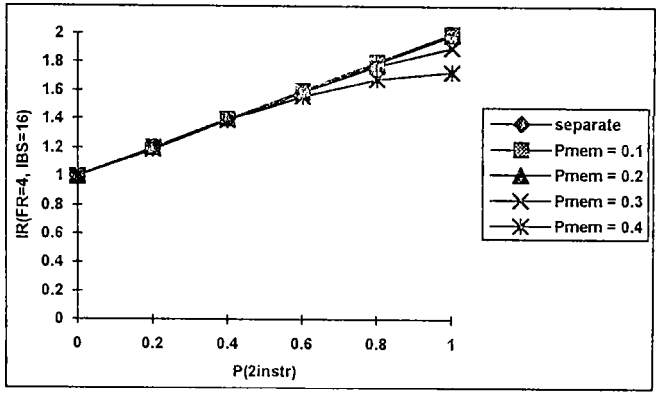


Figura 4.17

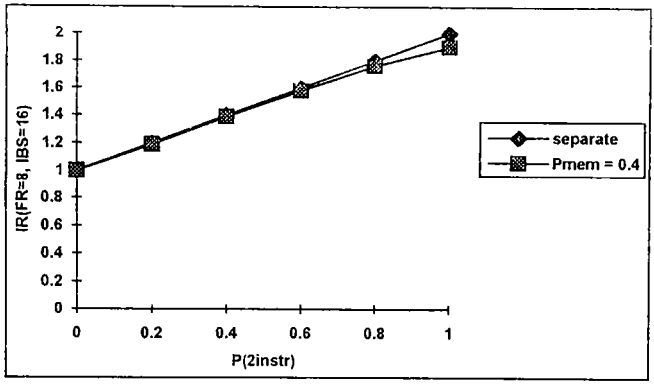


Figura 4.18

Este clar că o proiectare realistă (de exemplu, FR = 8, IRmax = 2, IBS =16) poate anula diferența de performanță între cele două variante.

În concluzie consider că asemenea metode de evaluare a performanțelor arhitecturilor cu paralelism în procesarea instrucțiunilor pot constitui o **alternativă utilă** la abordările convenționale bazate pe simulări ample. În particular, am arătat că pierderea de performanță a arhitecturilor unificate comparativ cu cele separate, datorată coliziunilor, **poate fi redusă semnificativ** prin **mecanisme adecvate de prefetch** și o **alegere corespunzătoare a parametrilor FR, IR și IBS.** În plus, acestea prezintă avantajul față de cache-urile separate de a avea o rată de hit superioară cu cca 10%. Astfel de metode teoretice de evaluare, au avantajul evitării unor simulări care necesită timp și resurse enorme. Ele pot fi deosebit de utile în alegerea unor variante arhitecturale optimale în raport cu scopul propus.

Totuși, un punct mai slab al prezentei abordări ar fi următorul: în practică, un branch predicționat de exemplu, în mod incorect că nu se va face, determină golirea bufferului de prefetch. Prezenta abordare nu "prinde" acest aspect deși această simplificare s-a făcut pentru ambele modele arhitecturale comparate. În continuare se va dezvolta metodologia în vederea rezolvării acestei probleme.

4.2.2. O METODĂ ÎMBUNĂTĂŢITĂ DE EVALUARE ANALITICĂ A ARHITECTURILOR MEM CU BUSURI UNIFICATE PE INSTRUCŢIUNI ŞI DATE

Aşadar, un model teoretic mai realist, ar trebui să introducă un nou parametru **Pb, ca semnificând probabilitatea ca să se lanseze în execuţie o instrucţiune de salt condiţionat predicţionat incorect.** Noul model de automat va tranziţiona dintr-o stare în alta pentru o structură pipeline cu 4 stagii, după modelul de mai jos.

Stare prez.	**Tranziţie**	**Stare urm.**	**Probab. tranz.**
3,0	Fetch 2,Ex.1	4,0	P(1)(1-Pmem-Pb)
		4,1	P(1)Pmem
		0,0	P(1)Pb
	Fetch 2,Ex.2	3,0	$P(2)\ (1-Pmem-Pb)^2$
		3,1	$P(2)\ (2Pmem(1-Pmem-Pb)+Pmem^2)$
		0,0	$P(2)\ (2\ Pb(1-Pb) + Pb^2)$

O verificare- fie şi incompletă- a corectitudinii acestui model rezidă în observaţia că adunând toate probabilităţile de tranziţie aferente acestei stări, suma este 1. De remarcat, faţă de modelul anterior unde se putea considera cu o bună aproximare ca parametrii aferenţi sunt cvasiindependenţi, că în acest caz parametrii Pb şi Pmem depind unul de altul, adică de exemplu, dacă la un moment dat P(1instr)=1, atunci avem Pb*Pmem=0.

Desigur, complexitatea modelului creşte considerabil în acest caz. Uzual, probabilitatea Pb < 15% , conform referinţelor bibliografice [Per93, Yeh92].

O problemă dificilă dar necesară acestui demers, constă în determinarea unei relaţii analitice pentru IRs, în cazul unui model echivalent cu cel anterior dar având busuri separate. Acest lucru este absolut necesar comparaţiei. Pentru aceasta se prezintă următorul raţionament:

- dacă **P(1instr)=1 şi Pb=0**, atunci se poate executa o instrucţiune.
- dacă **P(1instr)=1 şi Pb=1**, se execută branch-ul în tactul curent, iar apoi datorită golirii bufferului de prefetch nu se execută nici o instrucţiune, deci se execută în medie 0.5 instrucţiuni
- dacă **P(2instr)=1 şi $(1-Pb)^2=1$**, se execută 2 instrucţiuni
- dacă **P(2instr)=1 şi $Pb^2 + 2Pb = 1$**, atunci se execută în tactul curent 2 instrucţiuni, iar în tactul următor, datorită golirii bufferului de prefetch, nu se mai execută nici o instrucţiune, deci putem considera că se execută în medie 1 instrucţiune în această situaţie.

În acest caz, pe baza celor de mai sus, cu o bună aproximaţie, rata medie de procesare pentru o arhitectură cu busuri separate (IRsc) ar putea fi dată de relaţia:

$$\mathbf{IRs = P(1instr)\ (1 - 0.5\ Pb) + P(2instr)\ (3Pb^2 - 2Pb + 2)} \quad \mathbf{(4.19)}$$

Pentru Pb=0, din relaţia (4.19) se obţine relaţia (4.13), ceea ce este corect, verificându-se generalizarea modelului precedent.

4.2.2.1 REZULTATE OBȚINUTE PE BAZA METODEI

În figura următoare (fig.4.19), se prezintă comparativ ratele de procesare obținute pentru o arhitectură Harvard şi respectiv non-Harvard, pe baza modelelor teoretice dezvoltate în paragraful 4.2.2.

Pentru Pb = 20% şi pentru valori tipice ale parametrului P(2instr) de 0.4 şi respectiv 0.6, se obține IRu = 1.01 şi IRs = 1.22 (creştere cu 20%), respectiv IRu = 1.09 şi IRs = 1.39 (creştere cu 27%).

Pentru Pb = 5% şi pentru aceleaşi valori tipice ale lui P(2instr), se obține IRu = 1.15 şi IRs = 1.34 (creştere cu 17%) respectiv IRu = 1.22 şi IRs = 1.52 (creştere cu 25%).

De remarcat că **scăderea lui Pb determină performanțe superioare** şi totodată permite **micşorarea decalajului de performanță** între cele două opțiuni analizate. Totuşi acest decalaj de performanță cuprins între 17 şi 27% este unul semnificativ, aceasta din cauza parametrilor FR, IR şi IBS nefavorabili arhitecturii non-Harvard după cum am arătat într-un paragraf anterior.

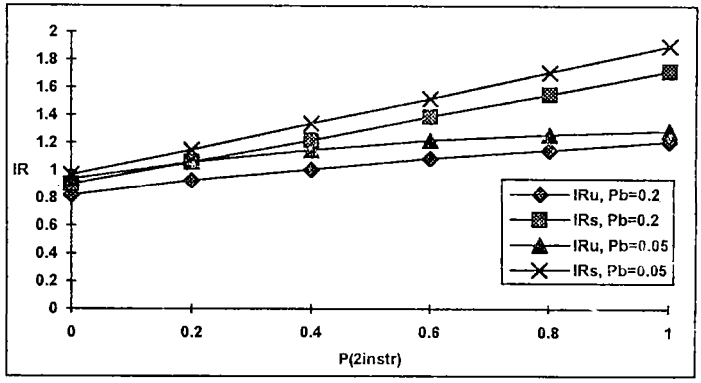


Figura 4.19

În figura următoare (fig.4.20) se prezintă aceleaşi performanțe comparative, de data aceasta însă cu parametrii FR = 4, IR = 2 şi IBS = 8, mai realist aleşi şi care conduc la rezultate superioare. De asemenea s-a ales Pmem = 30%.

Pentru aceleaşi valori tipice ale lui P(2instr) de 0.4 şi 0.6, am obținut pentru Pb = 20%, IRu = 1.09 şi IRs = 1.22 (creştere cu 12%) şi respectiv IRu = 1.21 şi IRs = 1.39 (creştere cu 15%). Pentru Pb = 5%, s-au obținut IRu = 1.28 şi IRs = 1.34 (creştere cu 5%), respectiv IRu = 1.43 şi IRs = 1.52 (creştere cu 6%).

Aşadar în acest caz diferențele de performanță între arhitectura Harvard respectiv non-Harvard, devin mai mici. Iarăşi de remarcat că scăderea lui Pb determină micşorarea decalajului şi important, **creşterea gradului de paralelism** P(2instr) determină la rândul său **mărirea decalajului de performanță în favoarea arhitecturilor cu busuri separate I/D.**

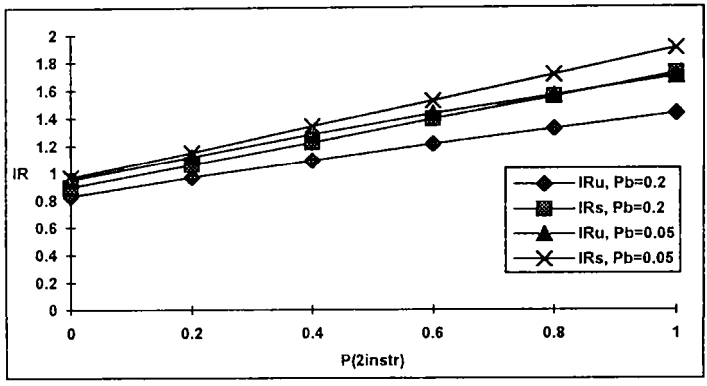


Figura 4.20

În cele două figuri următoare 4.21 şi 4.22 se prezintă evoluţia parametrului IBE pentru o arhitectură superscalară caracterizată de FR = IR = 2, IBS = 4 şi respectiv pentru o alta având FR = 4, IR = 2, IBS = 8. În ambele cazuri am ales Pmem = 30%, tipic. Evoluţia procentajului de timp cât bufferul de prefetch este gol, se prezintă pentru valori diferite şi realiste ale parametrului Pb.

Cum era şi de aşteptat, datorită parametrilor de proiectare FR, IR, IBS mai realist aleşi în al doilea caz se obţin valori superioare pentru IBE.

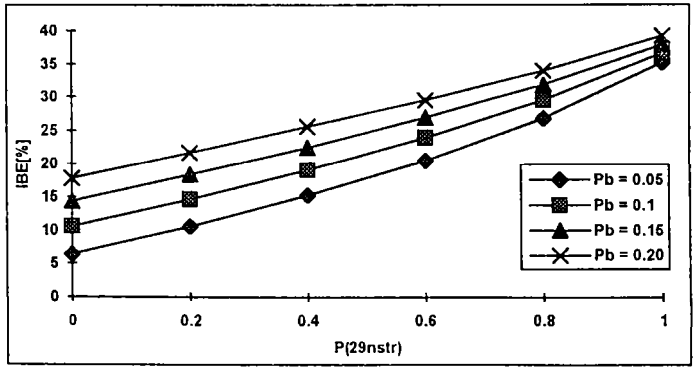


Figura 4.21

Astfel, de exemplu, pentru Pb = 15% şi P(2instr) = 0.6, valori tipice, în primul caz se obţine IBE = 27% iar în al doilea de 19.5%, ceea ce subliniază încă o dată importanţa unei alegeri adecvate a parametrilor FR, IR, IBS. Interesant, **creşterea lui Pb pare a contribui mai semnificativ la degradarea lui IBE în cel de-al doilea caz (FR = 4, IR = 2, IBS = 8).**

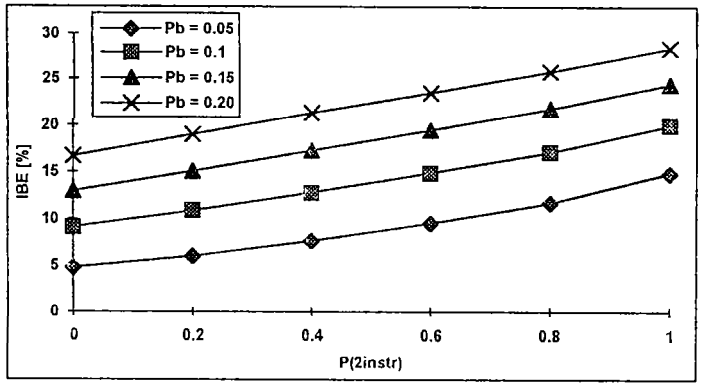


Figura 4.22

În figura care urmează (fig. 4.23), se prezintă într-un mod cantitativ, pentru o arhitectură non-Harvard cu FR = 4, IR = 2, IBS = 8 şi Pmem = 30%, variaţia performanţei (IRu) funcţie de diferite valori tipice ale parametrului Pb.

În acest caz de exemplu, pentru P(2instr) = 0.4 şi Pb = 5% rezultă IRu = 1.28 iar pentru P(2instr) = 0.4 şi PB = 20% se obţine IRu = 1.09, deci o scădere cu 17% a performanţei, ceea ce nu este deloc surprinzător.

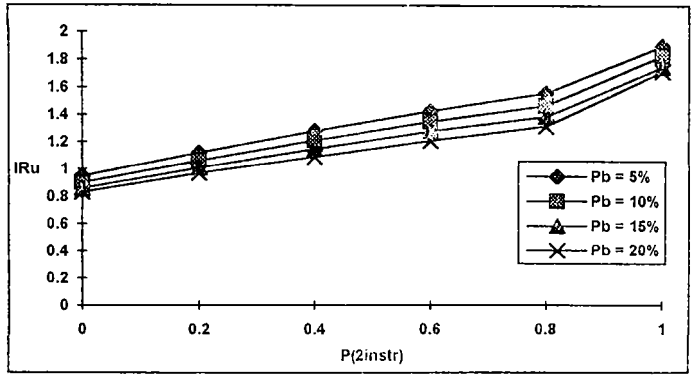


Figura 4.23

De asemenea, P(2instr)>0.8, determină un **salt brusc de performanţă.**

În continuare se prezintă pentru o arhitectură non-Harvard având FR = 4, IR = 2, IBS = 8 şi pentru un Pb = 0.1 tipic, variaţia performanţei funcţie de două valori extreme, realiste, ale parametrului Pmem.

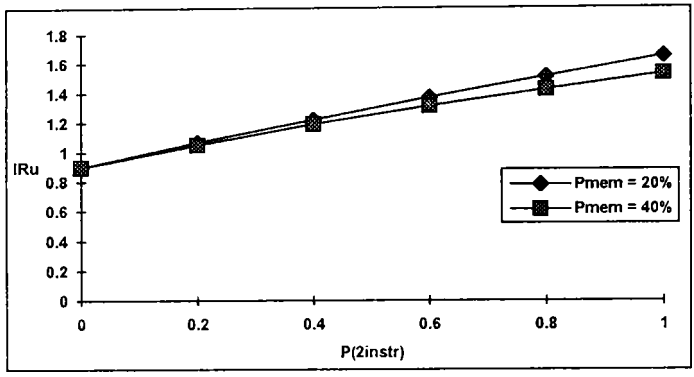


Figura 4.24

Interesant, o variaţie de la 20% la 40% a lui Pmem **nu produce o degradare semnificativă a performanţei,** ceea ce este semnificativ.

Degradarea maximă a lui IRu este de la 1.65 la 1.53 instr./tact, adică de cca 7%. În realitate, diferenţa medie de performanţă pentru cei doi parametri Pmem este mult mai mică, de cca 4%. Aceasta constituie încă o dovadă că o judicioasă alegere a parametrilor FR, IR şi IBS poate masca puternic efectele negative de performanţă pe care le implică coliziunile la busul unificat.

În fine, următorul grafic pune în evidenţă creşterea cantitativă a performanţei datorată exclusiv creşterii parametrilor FR şi IBS, pentru o arhitectură non-Harvard de procesor superscalar.

Astfel, pentru P(2instr) = 0.4 se obţine IRu = 1.21 respectiv 1.10 (creştere cu 10%), iar pentru P(2instr) = 0.6 se obţine IRu = 1.35 şi respectiv 1.16, deci o creştere de 16%. Iarăşi interesant, creşterea gradului de paralelism P(2instr) favorizează în acest caz al doilea model (FR = 4, IBS = 8) comparativ cu primul (FR = 2, IBS = 4).

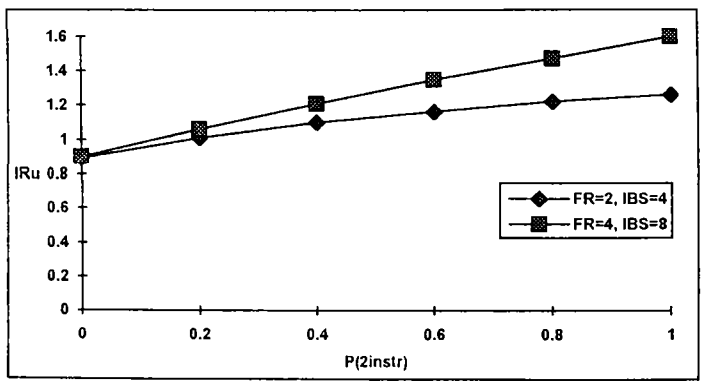


Figura 4.25

În concluzie, o alegere adecvată a parametrilor FR, IR şi IBS poate face din arhitecturile cu busuri unificate I/D o variantă atrăgătoare. De asemenea, scăderea parametrului Pb prin **strategii performante de predicţie** precum şi **creşterea gradului**

de paralelism prin metode performante de scheduling static şi dinamic, conduc la performanţe ridicate ale acestor arhitecturi considerate până acum ca fiind ineficiente în cadrul arhitecturilor MEM.

O altă problemă deosebit de interesantă şi care se ridică este următoarea: pare evident că un prefetch agresiv determină creşterea performanţei procesorului. Ironia constă în faptul că, acelaşi prefetch în cazul predicţiilor eronate ale branch-urilor, poate genera 2 fenomene nedorite: mărirea inutilă a ratei de miss în cache-uri şi respectiv necesitatea golirii bufferului de prefetch şi a anulării anumitor execuţii. Altfel spus, se aduc anticipat instrucţiuni inutile execuţiei, doar spre a fi golite din buffer. Acestea determină scăderea performanţei procesorului. Există puţine abordări, dacă totuşi există, ale acestei probleme, iar acest model poate răspunde la această problemă datorită **generalităţii sale.**

În fine, toate aceste concluzii teoretice vor trebui comparate cu cele obţinute pe bază de simulare. Astfel, se va avea un control asupra acestor metode teoretice şi se va putea stabili până la ce punct ele rămân realiste. Acestă investigaţie complexă bazată pe simulare va fi startată în continuare, în capitolul 5.

Pe de altă parte, trebuie avut în vedere faptul că şi aceste modele teoretice ar putea implica la un moment dat o complexitate ridicată a calculului. În acest sens, complexitatea de calcul a acestui model este de ordinul $O(N^3)$. Aşadar calculul devine relativ complex atunci când numărul de stări N creşte simţitor dar oricum **nesemnificativ în comparaţie cu simularea software.**

4.3. UN MODEL ANALITIC DE EVALUARE GLOBALĂ A PERFORMANŢEI CACHE-URILOR IN ARHITECTURILE SUPERSCALARE

În cele ce urmează ne propunem determinarea unui model cvasiteoretic simplu de comparare a eficienţei memoriilor cache unificate respectiv separate, implementate într-o arhitectură RISC superscalară. Până în acest moment am abordat această problemă exclusiv din punct de vedere al coliziunilor la busurile unificate. Demersul nostru se va baza pe un indicator de performanţă specific global, numit **timp mediu de acces la cache**. În final, acest indicator va trebui să înglobeze **ratele de miss** specifice, **procesul de prefetch** din I-CACHE iar în cazul cache-urilor unificate **şi procesele de coliziune** implicate. Aşadar, se încearcă înglobarea într-un singur indicator, pe bază teoretică, a principalelor fenomene legate de problematica memoriilor cache separate şi respectiv unificate.

Vom dezvolta aşadar doi indicatori: **Ts şi Tu** cu semnificaţia de timp mediu de acces la cache-urile separate, respectiv unificate. Comparaţia asupra performanţelor se va putea face pe baza maximului dintre cei doi parametri astfel: dacă max(Ts, Tu) = Tu, atunci separatele sunt mai eficiente, altfel, unificatele sunt mai eficiente.

Rata medie de miss într-o arhitectură de procesor cu memorii cache separate pe spaţiile de instrucţiuni şi date este dată de relaţia:

$$MRs = IC\ MRic + DC\ MRdc, \qquad (4.20)$$ unde:

IC - probabilitatea statistică de acces la spaţiul de instrucţiuni;
DC - probabilitatea statistică de acces la spaţiul de date;
MRic - rata medie de miss în memoriile cache de instrucţiuni;
MRdc - rata medie de miss în memoriile cache de date.

Este evident că IC + DC = 1 şi că practic IC > DC. Valori tipice pentru aceşti parametri sunt: IC = 75%, MRic = 0.02% - 3% şi MRdc = 2.88% - 24%. De observat că DC = Pmem / (1 + Pmem) .

Notând cu **MRu** rata de miss într-o arhitectură de procesor cu memorii cache unificate, simulări laborioase au arătat că MRs > MRu în medie cu cca 10% [Hen96]. Aşadar, din punct de vedere al ratei de hit precum şi al ratei de utilizare, cele unificate sunt mai eficiente.

Timpul mediu de acces exprimat în număr de impulsuri de tact în ipoteza că în caz de hit orice acces la cache durează un singur tact, este:

$$Ts = IC(1 + MRic\ N) + DC(1 + MRdc\ N) = 1 + N\ MRs \qquad (4.21)$$

unde N reprezintă numărul de tacţi necesari accesului la memoria principală în caz de miss în cache-uri. Tipic, N ia valori cuprinse între 10 - 20. În mod analog rezultă:

$$Tu = 1 + N\ MRu \qquad (4.22)$$

Cum MRs > MRu, rezultă că Ts > Tu, aşadar pare o dovadă clară a superiorităţii celor unificate.

Realitatea nu este aceasta, dimpotrivă, datorită proceselor de coliziune pe care unificatele le implică şi care generează aşteptări în accesele la cache-uri. Putem considera în plus, că se introduc în accesarea cache-urilor, datorită proceselor de coliziune, un număr mediu de M tacţi suplimentari, M =1,2,.... Problema care se pune este: în caz de conflict se vor introduce aşteptări în accesele la I-CACHE sau la D-CACHE? În primul caz timpul de acces va fi:

$$Tu = IC(1 + M + MRu\ N) + DC(1 + MRu\ N) = 1 + N\ MRu + IC\ M \qquad (4.23)$$

În al doilea caz, analog, Tu devine:

$$Tu = 1 + MRu\ N + DC\ M \qquad (4.24)$$

Cum IC > DC din relaţiile anterioare rezultă că pare a fi mai bine ca în caz de conflict să dăm prioritate acceselor la I-CACHE. Prin M am notat numărul mediu de tacţi suplimentari introduşi de către procesele de coliziune în accesul la I-CACHE respectiv la D-CACHE. În [Hen96 pagina 385] se adoptă ca model de evaluare a performanţei cel dat de relaţia (4.24) pe motivul că acesta minimizează Tu. Acest model este **greşit şi incomplet** după cum vom arăta în continuare, chiar dacă din punct de vedere pur algebric pare cel optim.

Realitatea este că în practică se preferă a se da prioritate acceselor la D-CACHE în caz de conflict [Rya93, IBM93, Weis94]. În continuare vom lămuri de ce se întâmplă acest lucru.

Dacă am da prioritate acceselor la I-CACHE, este posibil ca o instrucţiune LOAD/STORE aflată în bufferul de prefetch să aştepte indefinit lansarea în execuţie (până când acest buffer s-ar umple). Acest lucru este inacceptabil, întrucât în fond nu ne interesează atât minimizarea Tu, ci mai important, creşterea ratei de procesare a instrucţiunilor. Modelul prezentat nu are în vedere procesul de prefetch implementat în majoritatea procesoarelor cu scopul suprapunerii aducerii instrucţiunilor cu execuţia lor pipeline-izată.

Cu oarecare aproximaţie putem considera că t% din timpul total, accesele la I-CACHE sunt transparente din punct de vedere al performanţei, unde t% reprezintă % de timp cât numărul instrucţiunilor din bufferul de prefetch este mai mare decât rata maximă de lansare în execuţie a instrucţiunilor. Altfel spus, t% reprezintă procentajul de timp cât procesorul deţine suficiente instrucţiuni în bufferul de prefetch a. î. să proceseze la o rată maximă. Cu o bună aproximare s-ar putea considera că t = acurateţea predicţiei branch-urilor. Aşadar, numai (1 - t) % din accesele la I-CACHE se "simt" din punct de vedere al ratei de procesare a instrucţiunilor.

Printre altele, t este funcţie de rata de fetch (FR), rata maximă de lansare în execuţie (IR) şi capacitatea bufferului de prefetch (IBS). O alegere optimală a acestor 3 parametri în sensul sugerat anterior precum şi o strategie agresivă de prefetch, vor maximiza parametrul t şi deci rata de procesare medie (IR). Desigur că parametrul t depinde direct şi de acurateţea predicţiei branch-urilor întrucât în caz de predicţie eronată bufferul de prefetch trebuie golit. Cum aceste branch-uri sunt predicţionate corect prin scheme clasice de tip BTB (Branch Target Buffer), în proporţii de peste 90% (schemele adaptive de predicţionare comunică succese în proporţie de 97% [Yeh92]), rezultă că cel mult 10% din timp bufferul de prefetch este gol. Putem deci considera o valoare maximă pentru parametrul (1 - t) cuprinsă între 0.2 - 0.4 ,în cazuri clar defavorabile pentru unificate. Având în vedere performanţele actuale obţinute în predicţia branch-urilor, parametrul t poate fi considerat ca variind între 0 şi 0.1.

Dacă cele de mai sus le introducem în relaţia (4.23) obţinem :

$$Tu = (1 - t)\ IC\ (1 + M + MRu\ N) + DC\ (1 + MRu\ N) \qquad (4.25)$$

Dacă cele de mai sus le introducem în relaţia(4.24) vom obţine următoarea formulă:

$$Tu = (1 - t)\ IC\ (1 + MRu\ N) + DC\ (1 + M + MRu\ N) \qquad (4.26)$$

Acum devine clar şi din punct de vedere formal, de ce se preferă în caz de conflict să se dea prioritate acceselor la D-CACHE, întrucât relaţia (4.25) minimizează parametrul de performanţă Tu, care acum trebuie văzut ca un indicator empiric de performanţă şi nu ca timp de acces în sensul strict. Prin urmare relaţia (4.21) devine:

$$Ts = (1 - t)\ IC\ (1 + MRic\ N) + DC\ (1 + MRdc\ N) \qquad (4.27)$$

Desigur, parametrii MRdc, MRic şi MRu pot fi minimizaţi prin creşterea capacităţii cache-urilor şi respectiv prin mărirea gradului de asociativitate al acestora [Hen96].

În concluzie, o comparare între cele două tipuri de arhitecturi cache s-ar putea face pe baza relaţiilor (4.25) şi (4.27), care deşi cvasiempirice, le considerăm ca fiind **realiste.** Avantajul lor poate consta în **evitarea unor simulări laborioase** în stabilirea arhitecturilor optimale de memorii cache într-o anumită arhitectură.

În final, vom face o evaluare numerică pe baza relaţiilor stabilite. Vom evalua o arhitectură separată având 2 Ko I-CACHE şi 2 Ko D-CACHE cu o arhitectură unificată având 4 Ko. Memoriile cache sunt de tip "direct mapped". Preluat din [Hen96], vom considera MRic = 0.0226, MRdc = 0.2057, MRu = 0.0724, IC = 0.75 şi DC = 0.25. De asemenea, vom considera M = 1, N = 10 şi t = 0.7 (defavorabil pentru cache-urile unificate). În baza relaţiilor (4.25) şi (4.27), se obţine Ts = 1.0400 şi Tu = 1.0439 având aşadar performanţe comparabile.

În aceleaşi condiţii, dar alegând t = 0.75 se obţine Ts = 1.0951 şi Tu = 0.9417, deci un avantaj pentru cele unificate. Maximizarea lui t este deci esenţială în creşterea performanţei arhitecturlior unificate. Această problemă merită cred dezvoltată în continuare.

În concluzie, în ciuda părerii cvasiunanime, arhitecturile de memorii cache unificate pot avea în anumite condiţii **performanţe egale sau chiar superioare celor separate,** oferind în plus anumite simplificări în hardware.

Desigur că pentru o concluzie fermă sunt necesare investigaţii ulterioare atât pe plan teoretic cât şi pe bază de simulare a comportării acestor arhitecturi pe trace-uri ale unor benchmark-uri consacrate. Acesta este de altfel pasul următor al cercetării noastre din această lucrare şi care va da un răspuns definitiv la această problemă. Eficienţa cache-urilor unificate este direct legată de **îmbunătăţirea tehnicilor de prefetch,** de **predicţie a branch-urilor** şi de reducere a efectului defavorabil aferent **proceselor de coliziune.**

4.4. TEHNICI DE PREFETCH PERFORMANTE ÎN ARHITECTURILE MEM

Dificultatea majoră pentru implementarea unor tehnici de prefetch cât mai agresive o constituie instrucţiunile de ramificaţie care modifică secvenţialitatea programului. Întrucât performanţa acestor tehnici de prefetch afectează direct performanţa procesoarelor MEM în general şi în special pe cea a cache-urilor separate/unificate, vom prezenta succint cele mai avansate realizări pe această temă la ora actuală [Wal96, Tem96, Par96, Hen96].

O soluţie de referinţă în opinia mea este cea propusă şi analizată de către Park şi colaboratori [Par96] şi se bazează pe conceptul novator de **cache NRP** (Non Referenced Prefetch). În principiu se propune ca pe lângă memoria I-CACHE să se mai implementeze o altă memorie CACHE numită NRP. Soluţia are în vedere prefetch-ul ambelor ramuri în cazul unor ramificaţii condiţionate. Blocurile aduse anticipat se memorează în bufferul de instrucţiuni IB. În momentul în care ramura de ramificaţie este determinată, blocul (instrucţiunea) adus inutil în IB va fi memorat în NRP-CACHE. Acest mecanism reduce posibilitatea unor viitoare miss-uri în cache, în cazul în care instrucţiunea de ramifcaţie anterior executată va avea o comportare complementară.

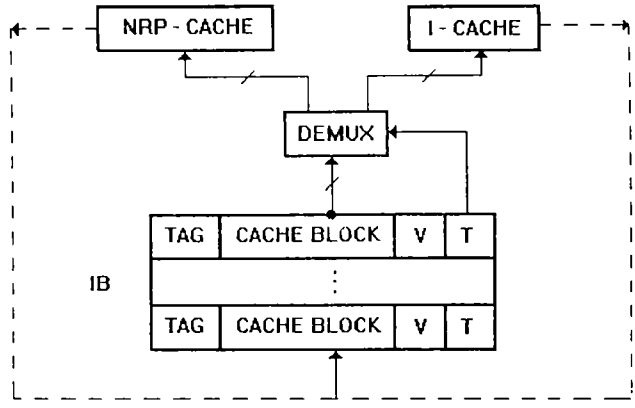


Figura 4.26

Structura memoriilor cache în acest caz, va fi cea prezentată în figura 4.26. Bitul V are rol de validare a informației în cache (accese DMA, conectare în sisteme multimicro, etc.), iar bitul T determină dacă un bloc din IB va fi memorat în NRP-CACHE sau în I-CACHE. La prefetch-ul unui bloc în IB, inițial bitul T este setat. Dacă respectivul bloc va fi procesat de către CPU, bitul T va fi resetat. Dacă în momentul determinării ramurii procesate bitul T = 1, atunci blocul respectiv se va evacua în NRP-CACHE.

Desigur că mecanismul NRP funcționează în strânsă legătură cu predictorul hardware de tip BTB. Mecanismul de prefetch se declanşează numai dacă blocul respectiv nu se află în I-CACHE, NRP-CACHE sau IB.

Cu un NRP-CACHE având capacitatea 1/8 din cea a I-CACHE, ambele de tip mapare directă, se arată că traficul la memoria principală se reduce cu 50-112% față de o arhitectură fără NRP-CACHE. De asemenea, această soluție reduce semnificativ rata de miss în I-CACHE determinând deci o îmbunătățire a timpului mediu de acces [Par96].

O altă soluție recentă [Wal96], are în vedere **depăşirea unei limitări fundamentale** relativă la rata de prefetch în procesoarele superscalare. Dacă p este probabilitatea ca o instrucțiune să determine o ramificație, atunci rata de prefetch medie este limitată la valoarea 1/p. Soluția se bazează pe conceptul nou introdus numit DBTB **(Dual BTB)**, derivat din conceptul de predictor BTB introdus de Lee şi Smith [Lee84]. Fiind dat PC-ul curent, DBTB-ul predicționează adresele de început ale următoarelor două instrucțiuni multiple ce vor fi executate. Având deci în vedere că se predicționează în avans două PC-uri, este necesat un I-CACHE de tip biport şi cu autoaliniere pentru a putea permite două citiri simultane. Astfel, prima instrucțiune multiplă predicționată va putea conține o instrucțiune primitivă de transfer fără a fi necesari doi cicli pentru accesarea din I-CACHE.

Prin această tehnică nouă, rata de prefetch va fi limitată superior de valoarea 2/p, ceea ce este semnificativ. Se arată că astfel pot deveni realiste rate de prefetch de 14 instrucțiuni primitve simultan ceea ce este deosebit de performant.

În concluzie, îmbunătățirea tehnicilor de prefech în arhitecturile MEM sunt determinate esențial de 3 aspecte: acuratețea predicțiilor hardware, arhitectura memoriilor cache şi implementarea unor suporturi hardware adecvate pentru prefetch [Tem96]. Soluțiile arhitecturale actuale în acest domeniu par a fi deosebit de agresive şi eficiente, acesta constituind un puternic argument în favoarea realismului analizelor noastre realizate până în acest moment.

4.5. CONSIDERAȚII PRIVIND ALEGEREA RATEI DE PROCESARE ŞI A NUMĂRULUI DE RESURSE ÎNTR-O ARHITECTURĂ ILP

Să considerăm acum un procesor cu execuții multiple ale instrucțiunilor, de tip RISC. Vom partiționa setul de instrucțiuni G al acestui procesor în s tipuri reprezentative, ortogonale, astfel:

$$G = \{I_1(m1, n1), I_2(m2, n2), ..., I_s(ms, ns)\} \qquad (4.28)$$

Am notat prin:

mk = numărul acceselor simultane de citire permise dintr-un set fizic de regiştri generali pentru o instrucțiune de tipul I_k. Uzual mk = 0, 1, 2, pentru oricare k = 1, .., s.

nk = numărul acceselor simultane de scriere (WB) permise într-un anumit set de regiştri generali pentru o instrucțiune de tipul I_k. Uzual nk = 0, 1 oricare k = 1, ..., s.

Pentru ca tipurile de instrucțiuni I_k să fie ortogonale și deci partiționarea consistentă este necesară îndeplinirea condiției:

$$(mi, ni) \neq (mj, nj)\ ,\ (\forall)\ i \neq j,\ \ i, j \in \{1,..., s\} \qquad (4.29)$$

De asemenea notăm:

Pk = probabilitatea de lansare în execuție a unei instrucțiunide tipul I_k.(mk, nk). Este evident că avem relația de normalizare:

$$\sum_{k=1}^{s} Pk = 1 \qquad (4.30)$$

IR = rata medie de procesare, în general impusă prin proiectarea arhitecturii de procesor [instr./tact].

N = numărul de seturi de regiștri generali. Menționăm că este necesar ca și conținutul acestor seturi de regiștri să fie permanent același pentru a respecta modelul clasic de programare. Presupunem că un set permite două citiri simultane în faza ID din pipeline.

M = numărul de scrieri simultane permis în faza WB a structurii pipeline. Uzual se alege M = 2 - 4. Mărirea lui M produce complicații hardware și datorită time-sharing-ului perioadei de tact necesare pentru scriere în faza WB și implică mărirea perioadei de tact a procesorului. De exemplu, microprocesorul superscalar AMD-K5 compatibil Pentium, are N = 4 și M = 4 [Cri96].

În continuare ne propunem să determinăm câteva condiții relative la alegerea optimală a parametrilor N și IR, prin prisma unui optim performanță-preț.

Putem considera deci că numărul mediu de citiri/tact impuse de către o instrucțiune de tipul I_k este:

$$\text{Ncitiri (k)} = mk\left(1 - \sum_{i=1}^{mk} Xki\ \right), \qquad (4.31), \text{ unde:}$$

Xki = probabilitatea ca o instrucțiune de tipul I_k să obțină în stația de rezervare aferentă un număr de i operanzi sursă prin tehnici de forwarding, fără a mai accesa deci un set de regiștri generali.

Desigur că avem îndeplinită identitatea:

$$1 - \sum_{i=1}^{mk} Xki\ = \text{Xko}, \qquad (4.32), \text{ unde:}$$

Xko = probabilitatea ca o instrucțiune de tipul I_k să nu obțină nici un operand sursă prin forwarding, necesitând deci mk accese de citire la un set de regiștri generali. Așadar:

$$\text{Ncitiri(k)} = mk * \text{Xko} \qquad (4.33)$$

Din considerente de obținere a unei utilizări cât mai ridicate a seturilor de regiştri la citire în faza ID şi totodată de obținere a unei rate medii de procesare IR ridicate, devine necesară impunerea următoarei condiții la citirea din regiştri:

$$IR * \sum_{k=1}^{s} Pk * mk * Xko \leq 2N \qquad (4.34)$$

Din motive analoage, în faza de scriere WB este necesar ca:

$$IR * \sum_{k=1}^{s} Pk * nk \leq M \qquad (4.35)$$

Considerând că există Ok unități funcționale aferente execuției instrucțiunilor de tip I_k, k = 1, ..., s, numărul total de unități funcționale (U) este:

$$U = \sum_{k=1}^{s} Ok \qquad (4.36)$$

Desigur este necesară îndeplinirea condițiilor:

IR * Pk ≤ Ok pentru oricare k = 1, ..., s adică:
IR ≤ U (4.37)

Este deci necesar în alegerea parametrilor IR, N, M şi Ok să se țină cont ca toate relațiile anterior stabilite să fie satisfăcute în vederea unei proiectări adecvate.

În [Wal96] se arată că rata de fetch a instrucțiunilor (FR) într-o arhitectură MEM este fundamental limitată prin relația:

FR ≤ 1 / p , (4.38), unde:

p = probabilitatea unei ramificații în program.

Printr-o arhitectură inteligentă de predicție a branch-urilor numită DBTB (Dual BTB) se arată că parametrul FR ar putea fi semnificativ mărit, noua limitare devenind [Wal96]:

FR ≤ 2 / p (4.39)

Cert este deci că aceste limitări sunt cu atât mai mult valabile şi pentru parametrul IR.

Din cele prezentate până acum rezultă deci că este necesară îndeplinirea următoarelor inegalități:

$$\mathbf{IR \leq Min\ \{\ 1/p,\ M/\ (\sum_{k=1}^{s} Pk * nk)\}} \qquad (4.40)\ \text{şi}$$

$$N \geq (IR/2) * \sum_{k=1}^{s} Pk * mk * Xko \qquad (4.41)$$

Aceste relaţii pot fi **deosebit de utile** în alegerea optimală a parametrilor IR şi N. În plus, prin intermediul lor se pot evita simulări software extrem de laborioase şi enorme consumatoare de timp.

Ca un exemplu, aplicând aceste relaţii pe următoarele date iniţiale de proiectare, considerate ca fiind realiste:

G = {I_1(2, 1) - instr. ALU, I_2(1, 1) - LOAD, I_3(2, 0) - STORE, I_4 (1, 0) - BRANCH}, P = 0.125 [Wal96]; Totodată:

P1 = 0.5, P2 = P3 = 0.15, P4 = 0.2 [Hen96]

De asemenea alegând M = 2, X1o = 0.8 şi X2o = X3o = X4o = 0.3 [Joh91], se obţine IR= 3 şi N = 2, rezultate deosebit de realiste şi care pot fi regăsite uşor în multe implementări uzuale.

În concluzie, s-au obţinut câteva **relaţii analitice originale**, deduse pe baza proceselor de coliziune aferente unei arhitecturi MEM şi care trebuie să fie îndeplinite prin orice proiectare realistă a unor asemenea arhitecturi

5.CONTRIBUŢII PE BAZĂ DE SIMULARE LA ARHITECTURA CACHE-URILOR ÎN CADRUL PROCESOARELOR MEM

5.1 SIMULATOR PENTRU O ARHITECTURĂ SUPERSCALARĂ PARAMETRIZABILĂ. PRINCIPIILE SIMULĂRII.

În cele ce urmează se va prezenta succint un simulator scris în Borland C++ 3.1, versiunea DOS, dezvoltat special cu scopul de a furniza rezultate semnificative relative la performanţa şi structura optimală a memoriilor cache integrate într-o arhitectură superscalară parametrizabilă.

Simulatorul permite atât analiza unor arhitecturi superscalare având busuri şi cache-uri separate pe spaţiile de instrucţiuni şi date (Harvard) cât şi a unor arhitecturi cu busuri şi cache-uri unificate (non-Harvard). Am considerat că printr-o asemenea simulare se pot obţine **rezultate realiste** relative la arhitecturile superscalare Harvard comparativ cu cele non-Harvard. Totodată, simularea permite să se verifice analizele teoretice realizate până în acest moment (v. cap. 4), referitor la această problemă. Se menţionează încă odată, că după stiinţa autorului, bazată pe o laborioasă şi atentă cercetare bibliografică, precum şi pe discuţii cu specialişti de prestigiu din domeniu, **nu există încă abordări teoretice sau pe bază de simulare** referitor la această problemă.

Programul simulator va procesa trace-uri HSA obţinute dintr-un alt simulator, special conceput la Universitatea din Hertfordshire, U.K. [Col95]. Acest simulator însă, nu abordează problema cache-urilor şi deci cu atât mai puţin a celor unificate. Se poate deci considera prezenta contribuţie ca fiind integrată în efortul de dezvoltare al arhitecturii HSA.

O **problemă deschisă** care va fi abordată prin simulare, după abordarea analitică din capitolul anterior, constă în determinarea raportului de performanţă între arhitecturile de memorie Harvard şi respectiv non-Harvard, integrate într-un procesor superscalar. De asemenea, se vor determina într-un mod original, pentru prima dată, parametrii optimi de proiectare pentru aceste două tipuri de arhitecturi cache.

Momentan, simulatorul are în vedere doar cache-urile cu mapare directă, cele mai pretabile în a fi integrate într-un microprocesor [Rya93, Joh91], dar pe viitor se intenţionează dezvoltarea simulatorului şi cu arhitecturi cache având diferite grade de asociativitate.

Principalii parametri ai arhitecturii, aleşi în deplină concordanţă cu nivelul tehnologic al implementărilor actuale, sunt următorii:

- **FR (Fetch Rate)** - rata de fetch instrucţiune din I-CACHE, poate fi de 4, 8 sau 16 instrucţiuni;
- **IBS (Instruction Buffer Size)** - capacitatea bufferului de prefetch (IB) care poate fi de 4, 8, 16, 32 sau 64 instrucţiuni;
- **capacitatea memoriilor cache** care variază între 64 cuvinte şi 16 Kcuvinte. Aceste capacităţi relativ mici ale memoriilor cache se datorează particularităţilor benchmark-urilor Stanford utilizate. Acestea folosesc o zonă restrânsă de instrucţiuni, limitată la cca 2 K şi o zonă de date mai mare dar mai "rarefiată", care se întinde pe un spaţiu de cca 24 K.

Structura de pricipiu a arhitecturii superscalare non-Harvard care a fost simulată este cea din figură (Fig.5.1). Structura Harvard simulată este similară, doar că deţine busuri şi cache-uri separate pe spaţiile de instrucţiuni şi date.

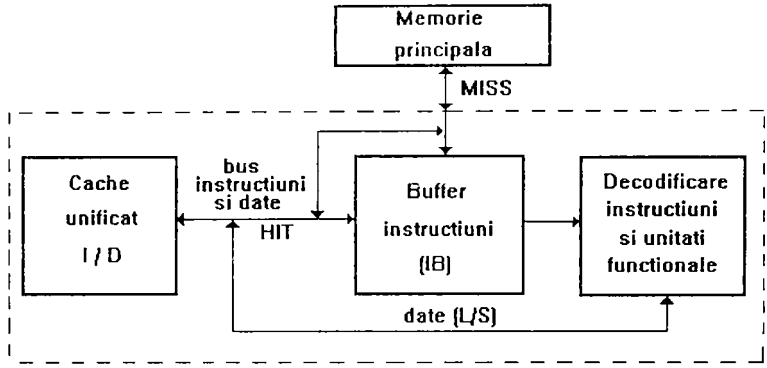


Figura 5.1

Un cuvînt din memoria cache va conține două câmpuri: un câmp de TAG și un bit de validare V. Câmpul de **TAG** conține blocul din memoria principală actualizat în cuvântul din cache de la indexul curent, iar bitul **V** validează informația din cache în sensul în care, inițial, fiecare cuvînt din cache va fi considerat invalid , până la prima actualizare a sa din memoria principală.

Memoria principală se accesează numai la miss în cache și va avea latențe parametrizabile cuprinse între 10-20 tacți procesor (realist la nivelul performanțelor tehnologiilor actuale). La miss în cache, trebuie deci introduse penalizări corespunzătoare în numărul tacților de execuție. În cazul acceselor la spațiul de date, se introduc penalizări numai pentru instrucțiunile LOAD, în cazul instrucțiunilor STORE nemaifiind nevoie datorită procesorului de ieșire (WB) și a bufferelor aferente (vezi cap2, 3).

S-a considerat în mod realist că este posibilă execuția simultană a oricăror două instrucțiuni cu referire la memorie cu restricția ca adresele lor de acces să fie diferite (cache biport pe date). Având în vedere că se lucrează pe trace-uri, această analiză antialias perfectă este evident posibilă.

Întrucât simularea se face pe baza trace-urilor HSA ale benchmark-urilor Stanford, s-a presupus o predicție perfectă a branch-urilor în cadrul simulării. Aceste **trace-uri HSA** sunt disponibile din simulatorul HSA [Col95] sub o formă specială din motive de economie de spațiu pe disc și ele conțin practic doar instrucțiunile de LOAD, STORE și BRANCH (numai cele care au provocat într-adevăr salt), în ordinea procesării lor în cadrul programului respectiv, fiecare cu PC-ul său și cu adresa efectivă corespunzătoare (de salt sau de acces la dată). Chiar și în această formă, un asemenea fișier trace ocupă între 2-4 Mo, el conținând practic câteva sute de mii de instrucțiuni ce au fost executate (între 250.000 și 900.000 de instrucțiuni mașină).

Implementarea simulatorului a fost făcută considerând următoarele:

- se aduc FR instrucțiuni din memorie în IB dacă există un spațiu disponibil în IB mai mare sau egal cu FR;
- execuția instrucțiunilor se face In Order, eludând deci complicații hardware inutile, având în vedere optimizarea de cod software, realizată anterior execuției [Col95, Ste96];
- nu pot fi lansate în execuție decât maxim două instrucțiuni cu referire la memorie în condițiile în care adresele lor de acces sunt diferite;
- în cazul arhitecturii non -Harvard, dacă în tactul curent se lansează în execuție cel puțin o instrucțiune LOAD/STORE, nu se declanșează aducerea de instrucțiuni din cauza

coliziunilor implicate (nu se pot aduce decât exact FR instrucţiuni din cache, din motive de aliniere adrese);
- dacă apare miss în spaţiul de instrucţiuni pe perioada "lungă" a accesării memoriei principale, se vor executa în continuare instrucţiuni din IB, cu condiţia să nu fie cu referire la memorie. În schimb, prefetch-ul va fi "îngheţat" pe această perioadă.
- nu s-au considerat hazarduri de date între instrucţiuni întrucât problema studiată este cu totul alta şi de asemenea s-a considerat latenţa de execuţie a tuturor instrucţiunilor egală cu un tact, pentru a nu complica practic inutil, simularea.

Bufferul de prefetch (IB) este implementat sub forma unei stive de tip FIFO, facilitând astfel operaţia de fetch instrucţiune *(tail)* şi respectiv lansare în execuţie *(head)*. Cuvintele din IB conţin 3 câmpuri:

- **tip**: tipul instrucţiunii (B- Branch, L- Load, S-Store, O- Alta)
- **adresa_instr:** adresa instrucţiunii din program (PC-ul aferent)
- **adresa_urmat**: adresa următoarei instrucţiuni din program (pt. instr. tip B şi O) respectiv adresa "ţintă" de memorie pentru instrucţiunile LOAD/STORE

Dacă adresa_urmat a unei instrucţiuni B este una nealiniată cu FR, se va alinia şi se va aduce noua instrucţiune multiplă de la adresa aliniată, considerându-se astfel o implementare cache realistă, în care accesele pot fi realizate doar la adrese multiplu de FR. Evident că în acest caz, instrucţiunile inutil aduse, nu se vor lansa şi contoriza în procesare.

Execuţia simulatorului se face în **următorii paşi succesivi**:

1. Solicitare parametri de intrare ai structurii superscalare şi anume: FR, dimensiune IB, IR, capacitate cache(-uri), nume fişier trace utilizat (*.trc), număr tacte penalizare la miss în cache (N).

2. Creare fişier care simulează funcţionarea cache-ului (CACHE.DM) şi iniţializare cache (peste tot bit V şi TAG=0).

3. Lansare în execuţie a procesării trace-ului
- se verifică conflictele LOAD/STORE din zona de "issue teoretic" (egală cu IR) din IB.
- dacă în zona de "issue real" din IB nu au existat instrucţiuni LOAD/STORE , atunci se va executa fetch instrucţiune, în ipoteza că spaţiul disponibil din IB o permite. Instrucţiunile vor fi citite din fişierul trace respectând logica de program. În caz de miss în cache, se penalizează timpul de execuţie şi dacă este posibil se vor executa în continuare instrucţiuni din IB.
- dacă în zona de "issue real"(număr instr. lansate anterior), au existat instrucţiuni LOAD/STORE, se iniţiază doar lansarea în execuţie a instrucţiunilor, cu penalizare a performanţelor în caz de miss. Din motive de conflict, în acest caz nu este posibil prefetch-ul simultan .

Simulatorul implementat **generează următoarele rezultate** , considerate ca fiind deosebit de importante pentru analiza propusă:
- număr de instrucţiuni procesate, număr total de tacte, rata medie de procesare (IR)
- rata de hit/ miss în cache-uri
- procentajul din timpul total cât IB este gol (IBE%)
- procentajul instrucţiunilor LOAD/STORE din trace
- procentajul fetch-urilor de instrucţiuni raportat la timpul total de execuţie
- procentajul din numărul tactelor cât există alias-uri la memorie

În ciuda predicţiei perfecte a branch-urilor, indicatorul IBE nu este trivial datorită posibilităţii apariţiei miss-urilor în cache-uri şi latenţelor ridicate ale memoriei principale(N), care pot conduce la golirea bufferului IB.

Toate aceste rezultate sunt afişate pe monitor şi scrise într-un fişier (REZULT.SIM), permiţându-se astfel prelucrarea lor ulterioară.

Eventuale viitoare dezvoltări ale acestei cercetări pot avea în vedere implementarea unor cache-uri cu un grad sporit de asociativitate şi implementarea hazardurilor de date între instrucţiuni.

În continuare se prezintă câteva dintre rezultatele originale obţinute precum şi o analiză a acestora prin prisma unor obiective vizând proiectarea optimală şi evaluarea de performanţă. De asemenea, se va concluziona în ce măsură modelele teoretico- analitice dezvoltate până în acest moment corespund concluziilor rezultate din aceste laborioase simulări.

5.2. REZULTATE OBŢINUTE PRIN SIMULARE. ANALIZĂ ŞI CONCLUZII

Simulatorul anterior prezentat a fost exploatat utilizând trace-uri ale benchmark-urilor Stanford, consacrate în acest domeniu. În tabelul următor, se prezintă distribuţia instrucţiunilor cu referire la memorie şi procentajele acceselor în spaţiile de instrucţiuni şi date, în cadrul acestor benchmark-uri.

Benckmark	Pmem %	IC %
Bubble	30%	75%
Matrix	29%	75%
Perm	40%	70%
Puzzle	17%	85%
Queens	33%	76%
Sort	24%	80%
Tower	38%	69%
Tree	27%	73%

Figura 5.2

Se remarcă reprezentativitatea benchmark-urilor Stanford şi din acest punct de vedere [Mil89, Ste96].

În continuare se prezintă rezultate experimentale obţinute, în scopul unei comparaţii pe bază de simulare între arhitecturile cache unificate şi respectiv cele separate.

Se precizează că toate aceste rezultate au fost obţinute pentru N = 10 (numărul tactelor necesare procesorului pentru un acces la memorie în caz de miss), acolo unde nu se precizează altfel.

În figura de mai jos, 5.3, comparaţia între unificate şi separate se realizează în condiţiile FR = 8, IR = 4, IBS = 32 şi capacitate cache de 64 locaţii.

În acest caz, se prezină ratele medii de procesare (IRmed) obţinute pentru fiecare benchmark în parte. Astfel, ratele medii armonice (HM), practic unanim utilizate în literatura de specialitate, obţinute au fost IRu = 0.48 instr./tact, respectiv IRS = 0.49 instr./tact, cvasiidentice. Se menţionează din nou că aceşti indicatori înglobează toate procesele importante implicate de aceste arhitecturi cache. De asemenea se precizează că, în cazul arhitecturilor separate, capacităţile cache-urilor pe instrucţiuni şi date sunt considerate identice (32, 32).

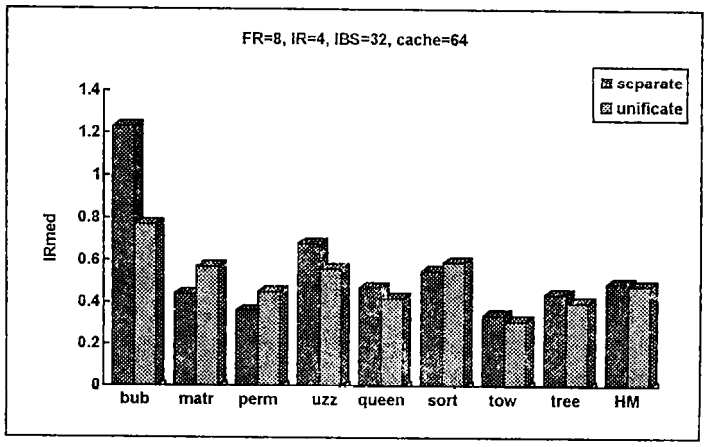


Figura 5.3

În cele două figuri următoare, Fig.5.4, 5.5, se prezintă tot IR mediu comparativ pentru cele două arhitecturi cache, dar considerând capacitatea acestora mărită la 512 şi respectiv la 2k locaţii.

În conformitate cu cele concluzionate în capitolul 4, aceşti parametri FR, IR, IBS sunt deosebit de realişti şi mai ales conduc la o eficienţă ridicată a procesării structurii analizate.

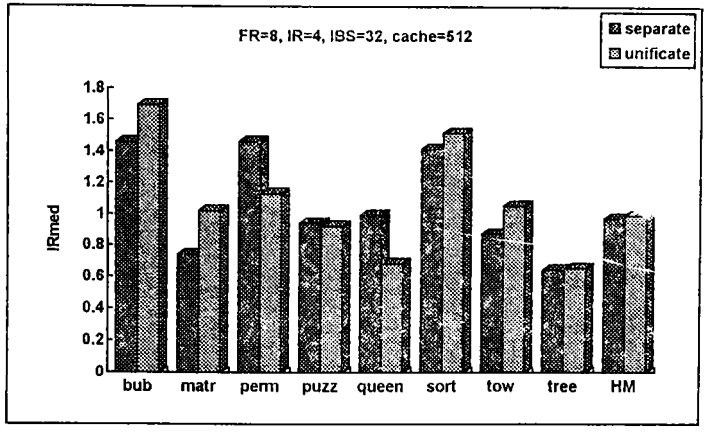


Figura 5.4

În primul caz, s-au obţinut medii IRu = 0.99 şi IRs = 0.97 (superioare cu cca 2 % unificatele!) iar în al doilea caz, IRu=1.20 şi IRs = 1.52 (superioare cele separate).

Se consideră acest al doilea caz caracterizat de un cache de 2k ca fiind relativ nerealist în sensul în care avantajează artificial separatele. Acest fapt se explică prin capacitatea exagerat de mare a acestui cache (2k) în raport cu caracteristicile benchmark-

urilor utilizate (acestea aparţin unui spaţiu de instrucţiuni < 2k), mărindu-se astfel artificios rata de hit în cache şi anulându-se deci avantajul unificatelor pe acest plan.

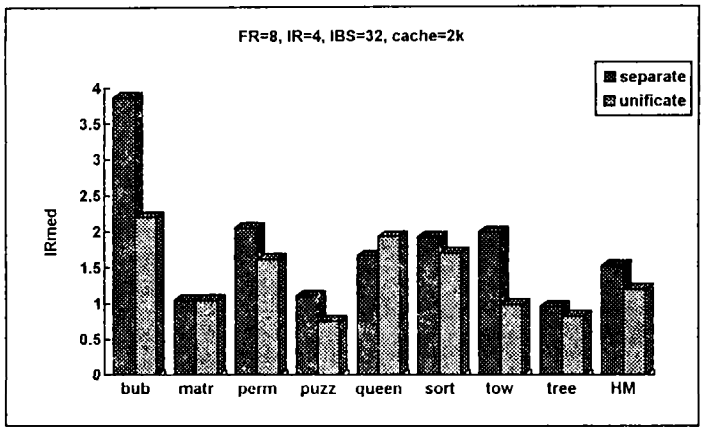


Figura 5.5

Oricum, concluzia obţinută până în acest moment prin simulare este că, în condiţii similare, cele două arhitecturi de memorii cache implică performanţe cvasiidentice, fapt situat în **deplină concordanţă cu estimările analitice** realizate în capitolul 4, paragraful 4.2.1.1. De remarcat că parametrii aleşi sunt deosebit de realişti. În condiţii echitabile de performanţă, cu parametrii FR, IBS, IR şi algoritmi de procesare eficienţi, arhitecturile unificate oferă performanţe identice sau chiar superioare celor separate. Această concluzie arată clar faptul că **arhitecturile unificate trebuiesc reconsiderate**. În plus, după cum am mai arătat, construcţia şi implementarea lor este mai facilă [Rya93, IBM93].

În fig. 5.6 se prezintă performanţe relative la astfel de modele de procesoare neconvenţionale, deosebit de agresive în domeniul prefetch-ului.

Astfel s-a ales FR = 16, parametru posibil de atins într-un viitor apropiat prin tehnici de predicţie de tip DBTB care permit la ora actuală rate realiste de fetch de 14 instrucţiuni [Wal96]. În acest caz s-au obţinut performanţe medii IRs = 0.58 şi IRu = 0.52, un avantaj de 11% pentru cele separate. Creşterea parametrului FR de la 8 la 16 a adus un câştig mediu de performanţă de cca 18%. O excepţie o constituie programul *queens* unde pentru FR = 8 rezultă IRs = 0.47 iar pentru FR = 16 rezultă IRs = 0.43, mai slab. Explicaţia poate consta în numărul mare de branch-uri care se fac specifice acestui program şi care determină aducerea multor instrucţiuni practic inutil, având în vedere rata ridicată de fetch (FR=16). Oricum, încă o dovadă că eficienţa prefetchului este esenţială.

În continuare, s-a dorit să se cerceteze comportamentul arhitecturilor unificate/separate dacă capacitatea cache se măreşte de la 64 locaţii (fig. 5.6) la 512 locaţii (fig. 5.7). În acest caz, se obţin performanţe medii IRs = 1.21 şi IRu = 1.10 cu 108% respectiv 111% , substanţial superioare faţă de ratele obţinute în cazul precedent.

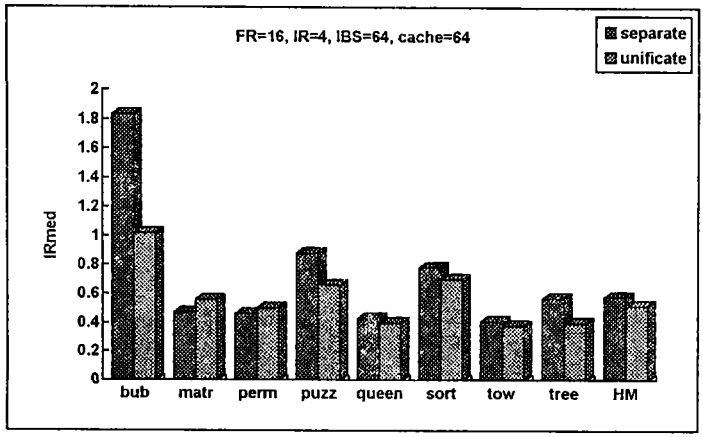


Figura 5.6

Rezultatele conţinute în aceste două grafice (5.6, 5.7) pot fi considerate realiste având în vedere faptul că spaţiul de instrucţiuni al benchmark-urilor Stanford este practic limitat mult sub 2k. În aceste condiţii capacităţi cache cuprinse între 64 cuvinte şi 512 cuvinte sunt posibile şi realiste.

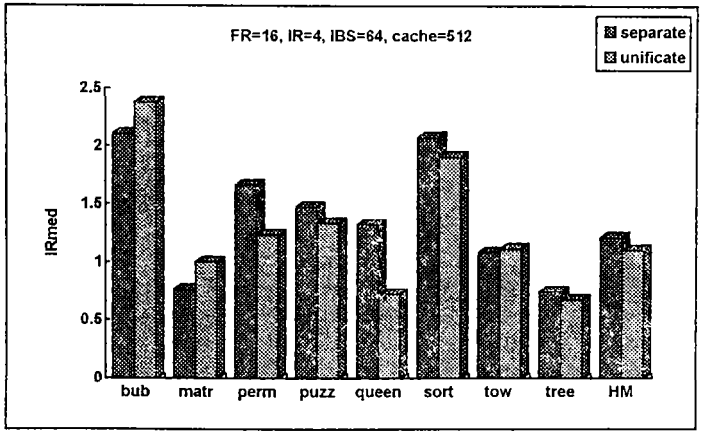


Figura 5.7

În figura 5.8, numai pentru arhitecturi de tip Harvard, se prezintă în mod cantitativ influenţa unor capacităţi diferite ale memoriilor cache asupra performanţei. Pentru capacităţi de 64, 512 şi 2k locaţii, se obţin performanţe medii respectiv de 0.49, 0.97, 1.52.

Creşterile de performanţă medii (fig. 5.8) sunt semnificative şi anume de 97% respectiv de 56% de la un nivel al capacităţii cache la altul.

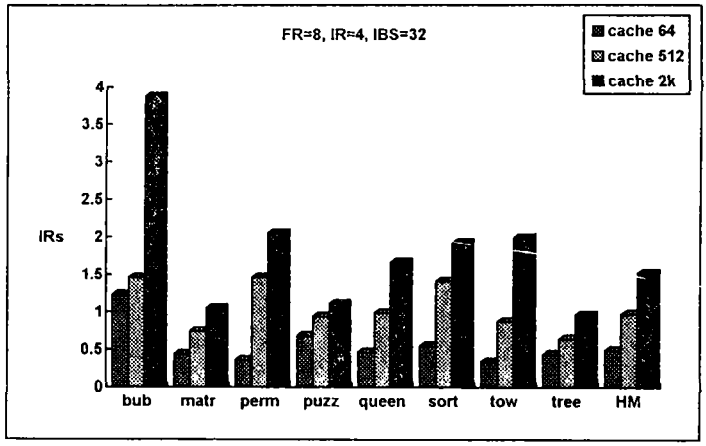


Figura 5.8

În aceleaşi condiţii, însă în cazul unor arhitecturi non-Harvard, se obţin performanţe medii IRu = 0.48, IRu = 0.99 şi IRu = 1.20 (fig. 5.9).

Aşadar în acest caz se constată din nou creşteri semnificative de 106% şi respectiv 21%. Având în vedere că un cache de capacitate 2k este nerealist de mare datorită caracteristicilor programelor Stanford şi ţinând cont de rezultatele anterioare, devine clar că **mărirea capacităţii cache în limite rezonabile avantajează unificatele**. Acest lucru este explicabil pe baza creşterii ratei de hit datorată măririi capacităţii cache.

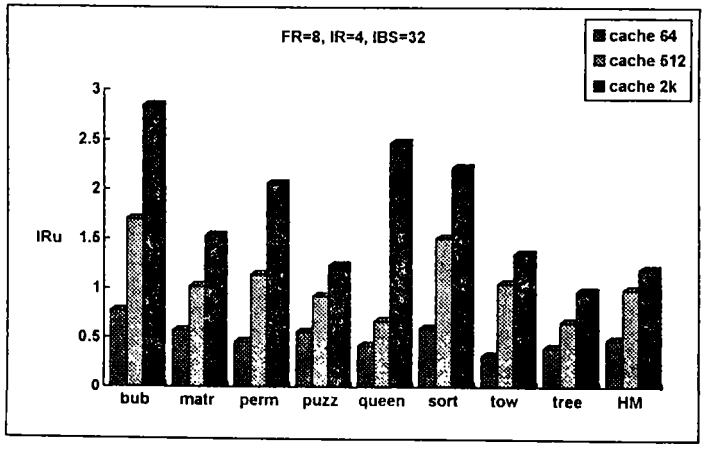


Figura 5.9

În cele două figuri următoare (fig. 5.10., fig. 5.11), se prezintă influenţa măririi capacităţii bufferului de prefetch asupra indicatorilor de performanţă globali IRs şi IRu.

În cazul Harvard, o creştere a lui IBS de la 8 la 64, determină o creştere medie a indicatorului IRs de la 0.64 la 0.66, nesemnificativă. În celălalt caz, acelaşi fapt determină

o creştere medie a lui IRu de la 0.73 la 0.74, de asemenea nesemnificativă. În ambele cazuri rezultă că IBSoptim = 8, altfel spus **IBSoptim = 2FR**, rezultat aflat în **deplină concordanţă cu cele obţinute pe bază pur teoretică** în capitolul 4 (vezi fig. 4.13).

Acest lucru este demonstrat şi de faptul că pentru FR = 8, IR = 4, cache = 512 locaţii, s-a obţinut un IBSoptim = 16 (mărirea acestuia determină o creştere asimptotică a performanţei), adică tot 2FR.

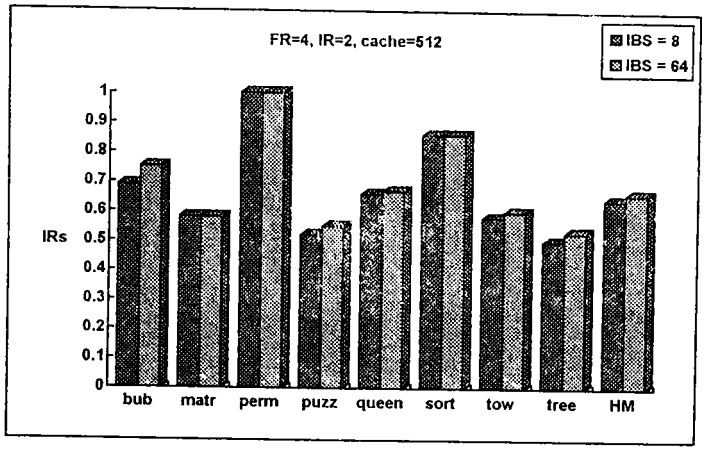


Figura 5.10

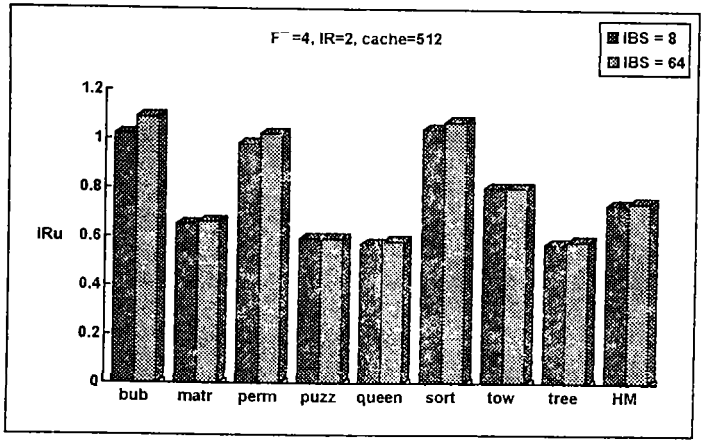


Figura 5.11

În continuare (fig. 5.12) se prezintă influenţa capacităţii cache asupra ratei de miss în cache-ul de date. S-a considerat că I-CACHE şi D-CACHE au capacităţi egale. Pentru capacităţi de 64, 512 şi 4k s-au obţinut procentaje de miss de 50%, 24% şi 0.3% respectiv.

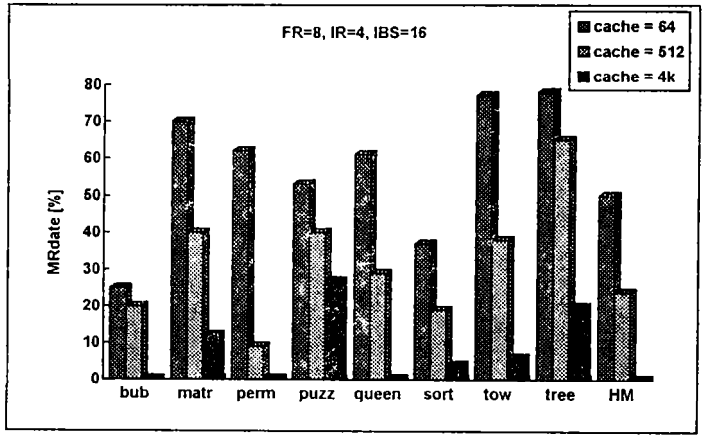


Figura 5.12

Devine clar că pentru capacităţi realiste ale D-CACHE, cuprinse între 64 şi 512 în acest caz, performanţa şi gradul de utilizare ale acestora sunt relativ scăzute. Acest rezultat este în deplină concordanţă cu cele obţinute prin intermediul altor cercetări, puţine, oarecum similare [Hen96].

În continuare în fig. 5.13 se prezintă efectul pe care îl are asupra performanţei dezechilibrarea capacităţii cache-urilor în arhitecturile de tip Harvard. Astfel pentru capacităţi egale de 512 locaţii se obţine IRs = 1.27 iar pentru I-CACHE = 128 şi D-CACHE = 996 se obţine IRs = 1.10. Aşadar o dezechilibrare a capacităţii în favoarea cache-ului de date conduce la o degradare a performanţei cu cca 15%, fapt confirmat de altfel şi prin alte experimente realizate cu acest simulator.

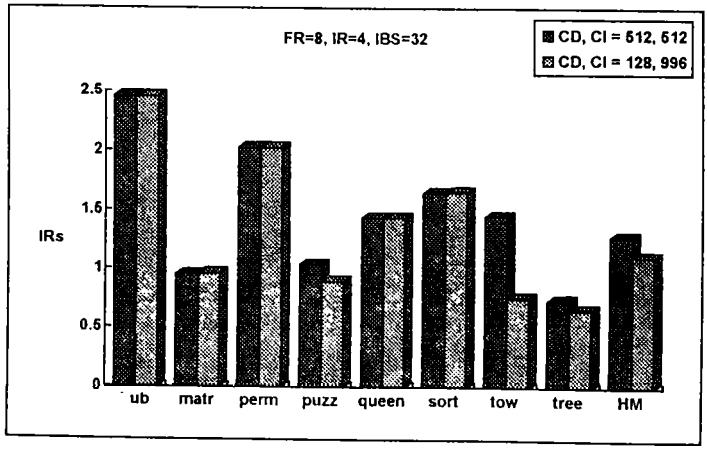


Figura 5.13

Pe baza a numeroase simulări a rezultat clar că nu este recomandabilă nici dezechilibrarea capacității în favoarea D-CACHE în scopul optimizării de performanță, cel puțin pentru programele Stanford și pentru capacități rezonabile, limitate tehnologic, ale cache-urilor.

În opinia autorului capacitățile egale implică performanțe optime. Desigur că acest lucru este valabil numai în condiții relativ restrictive impuse capacității totale a cache-ului (v. în continuare). În aceleași condiții cu cele din fig. 5.13 dar considerând I-CACHE = 768 și D-CACHE = 256 s-a obținut IRsmediu = 0.97, nesatisfăcător.

În continuare se dorește analiza capacităților optime ale I-Cache respectiv D-Cache în condiții nerestrictive ale capacității totale.

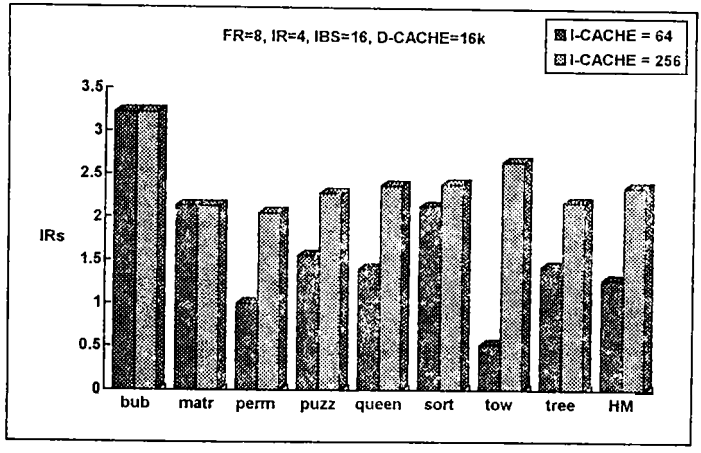


Figura 5.14

În figura 5.14 se prezintă variația performanței IRs, considerând un cache de date practic nelimitat (cvasiideal) și variind capacitatea I-Cache. Astfel, pentru un I-Cache de 64 locații, se obține performanța medie armonică IRs= 1.28 iar pentru I-Cache de 256 locații, IRs= 2.36.

Așadar se constată o creștere a performanței în acest caz cu cca 84%. O capacitate a I-Cache mai mare de 256 locații, practic nu mai mărește performanța ceea ce arată că această capacitate este cea optimă din punct de vedere al performanței, în condiții mai puțin restrictive ale capacității D-Cache.

În figura 5.15 se prezintă rezultatele problemei reciproce, și anume de a determina capacitatea optimă a D-Cache în condiții nerestrictive ale capacității I-Cache (16k). Aici, rezultatele optime IRs se obțin pentru un D-Cache de 4k după cum se poate observa. Mai precis s-au obținut pentru capacități D-Cache de 256, 1k, 4k, performanțe medii de 0.95, 1.48 și 2.06 respectiv. O mărire a capacității memoriei cache de date peste 4k, determină creșteri asimptotice ale performanței.

Concluzia este una interesantă, întrucât a rezultat că în condiții mai puțin restrictive ale capacității totale a memoriei cache, I-Cache optim este de 256 locații iar D-Cache optim de 4k locații !. Această concluzie îndreptățește și **justifică alegerea lui D-Cache mult mai mare decât cea a lui I-Cache** în anumite procesoare RISC superscalare (IBM RS/6000, 8k I-Cache, 64k D-Cache), în condiții practic nerestrictive. În schimb, concluzia aceasta **contrazice alegeri inverse**, întâlnite și ele în anumite realizări comerciale (AMD K5, SuperSPARC). Evident că rezultatele se bazează pe

reprezentativitatea deosebită a programelor Stanford utilizate pentru aplicațiile de uz general.

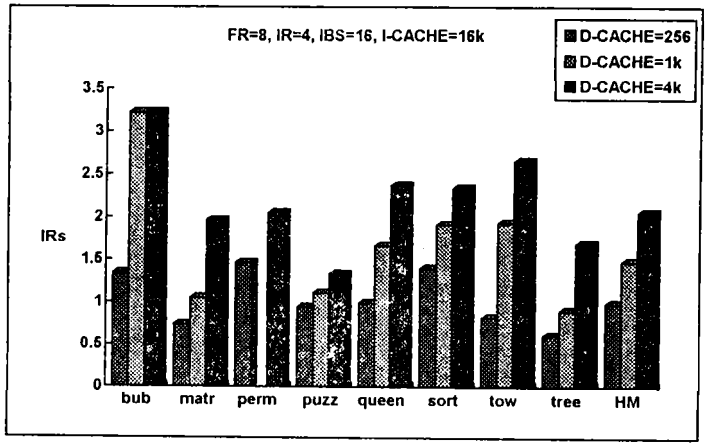


Figura 5.15

În figura 5.16 se prezintă rata medie globală de miss pentru diferite cache-uri unificate implementate. Astfel, pentru capacități de 64, 512, 4k s-au obținut rate de miss medii de 40%, 12%, 2.3% respectiv, ceea ce demonstrează clar că **gradul de utilizare al cache-urilor unificate este superior** (a se compara cu Fig.5.12).

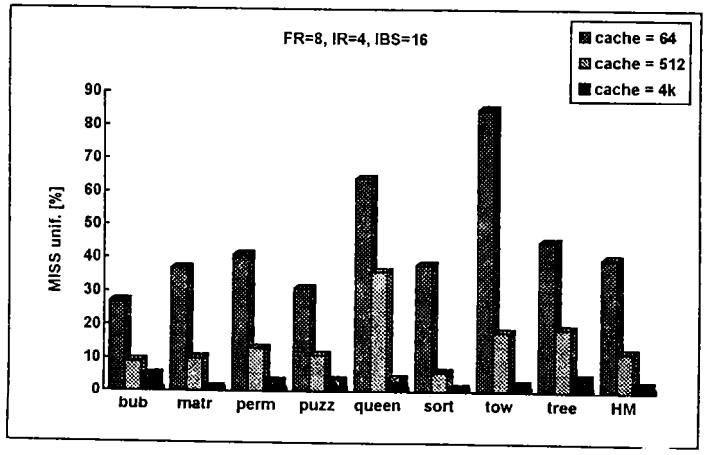


Figura 5.16

În figura următoare (fig. 5.17) se prezintă IBE unificate pentru FR = 4 și FR = 8. În medie s-a obținut IBE = 61% respectiv IBE = 27% rezultând deci prin această creștere posibilă a parametrului FR o îmbunătățire semnificativă a lui IBE. Rezultatul acesta obținut prin simulare se găsește într-o foarte bună concordanță cu cele obținute în capitolul 4 pe bază pur analitică (vezi fig. 4.21, fig. 4.22).

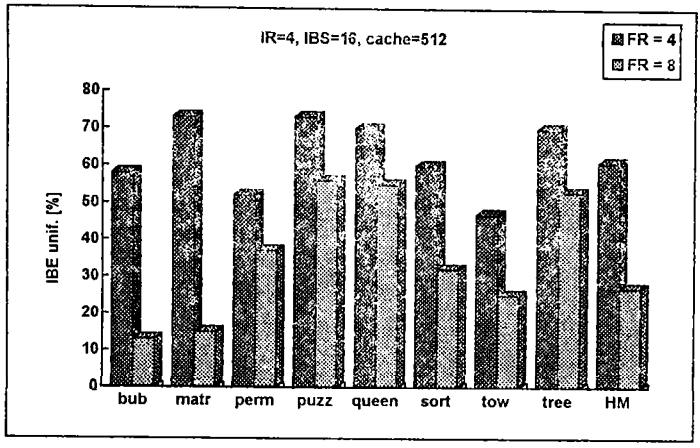


Figura 5.17

IBE mediu în condiții similare într-o arhitectură Harvard simulată, s-a obținut de cca 7%, net superior datorită faptului că aici nu există coliziuni.

În figura următoare (Fig.5.18), se prezintă procentajul din timpul total al procesorului cu cache-uri unificate, cât acesta declanşează cicluri de aducere a instrucțiunilor. Rezultatele se prezintă aici comparativ pentru 2 modele: unul "slab" caracterizat prin FR = 4, IBS = 8, IR = 4, cache = 64 şi altul "tare", adică caracterizat de parametrii FR = 8, IBS = 32, IR = 4, cache = 16k.

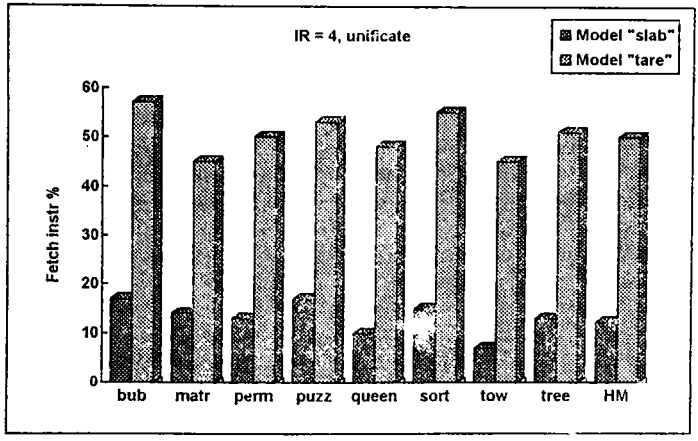


Figura 5.18

S-a obținut pentru cel "slab" 12.3% în medie, iar pentru cel "tare" 50% în medie. Aşadar, în cazul modelului "tare" prefetch-ul este performant datorită optimalității parametrilor arhitecturii, care diminuează din procesele de coliziune şi miss-urile în cache. În acest caz aducerea de instrucțiuni se realizează în mod performant.

În fine, ultima analiză (fig. 5.19) abordează problematica influenței latenței memoriei principale asupra performanței arhitecturilor unificate de cache.

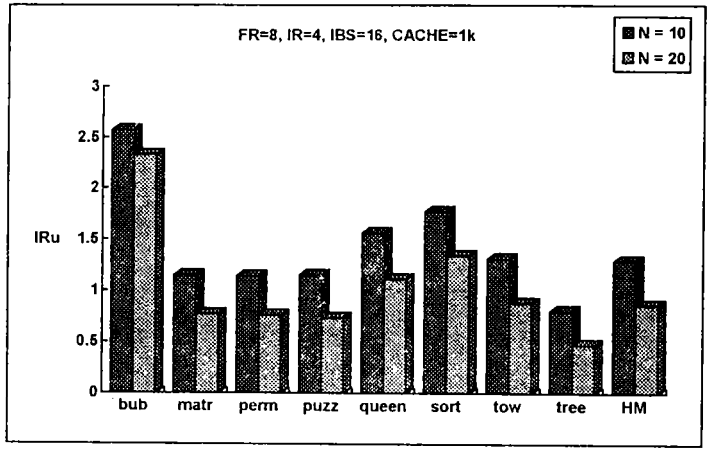


Figura 5.19

Se prezintă rezultatele obținute pentru o latență N de 10 tacte, respectiv 20 tacte. IRu medii obținute sunt de 1.29 respectiv 0.86 instr./tact. Așadar se constată o reducere a performanței cu cca 50% în acest caz determinată de dublarea latenței memoriei principale. Rezultă deci o influență semnificativă a latenței memoriei principale în cadrul acestei arhitecturi ,favorizată și de procesele de coliziune implicate.

Se observă că metrica preferată în evaluarea de performanță a acestor arhitecturi neconvenționale a fost una globală, relevantă și anume rata medie de procesare, preferată metricii locale, nu întotdeauna de mare relevanță, numită rata medie de hit/miss în cache.

În concluzie, această cercetare în domeniul compromisului Harvard / non-Harvard într-un procesor MEM, este într-o deplină concordanță prin rezultatele pe care le furnizează cu cea realizată la nivel teoretic în capitolul 4. Prin intermediul acestor investigații, arhitecturile unificate apar într-o postură favorabilă, nouă, care le face atractive și din punct de vedere al performanței. Până în prezent era vehiculată ideea - nejustificată nicăieri în mod riguros - că acestea sunt depășite, datorită conflictelor structurale pe care le implică și deci a performanței scăzute. De asemenea s-a determinat o metodă de simulare valabilă pentru determinarea principalilor parametri arhitecturali optimali pentru o arhitectură superscalară eficientă.

6. CONTRIBUȚII LA PROBLEMATICA PREDICȚIEI BRANCH-URILOR ÎN ARHITECTURILE SUPERSCALARE.

În acest capitol vom prezenta două contribuții referitoare la importanta problemă a predicției instrucțiunilor de ramificație în arhitecturile pipeline și cu paralelism spațial la nivelul instrucțiunilor.

Prima cercetare încearcă să ofere un **model analitic general** de estimare a performanțelor schemelor de predicție. Acest lucru este deosebit de important întrucât există extrem de puține asemenea modele prezentate în literatură, toate fiind deficitare în opinia autorului. Utilitatea acestui model este evidentă, nemaifiind necesare simulări laborioase, cel puțin într-o primă iterație a procesului de proiectare.

A 2-a investigație, abordează pe bază de **simulare software,** problema extrem de interesantă și dezbătută, a celor mai performante scheme de predicție la ora actuală, cele corelate pe 2 nivele. Se încearcă pentru prima dată, integrarea unei asemenea predicții hardware în cadrul arhitecturii HSA, dezvoltată la Universitatea din Hertfordshire, UK. Menționăm că până în prezent, această arhitectură se baza pe tehnici pur software de tip compensare "Branch Delay Slot", total nerecomandate la o arhitectură superscalară [Col94,Col95, Ste96]. De asemenea se prezintă rezultatele obținute, aflate în deplină concordanță cu cele publicate în literatura de specialitate recentă.

6.1. UN NOU MODEL ANALITIC DE ESTIMARE A PREDICȚIILOR BTB (BRANCH TARGET BUFFER)

Multiple sunt necesitățile dezvoltării unor modele analitice generale de estimare a performanțelor tehnicilor hardware de predicție a branch-urilor. La ora actuală, majoritatea estimărilor de performanță și deci de proiectare în acest domeniu, se bazează pe extrem de laborioase simulări ale unor arhitecturi BTB puternic parametrizate. Acestea presupun dezvoltarea unor instrumente sofisticate și dificil de creat, considerate ca fiind absolut necesare, cum ar fi: crosscompilator, reorganizator de program și simulator propriu-zis. Toate acestea se aplică unei arhitecturi particulare și unor benchmark-uri inevitabil particulare și ele. Astfel de abordări ale problemei proiectării și evaluării de performanță pentru diferite tehnici BTB pot fi găsite în multe cercetări [Lee84, Per93, Dub91, Yeh92] și pot fi considerate ca fiind deja "clasice".

Există puține abordări ale problemei, care să încerce determinarea unor **modele analitice generale** aferente tehnicilor de predicție de tip BTB, unanim recunoscute și implementate la ora actuală. Asemenea încercări pot fi totuși găsite în [Hen96, Per93] și-cele mai mature și mai performante- respectiv în monografia [Cra92]. De altfel, aici, se și justifică pe larg necesitatea imperioasă a dezvoltării unor asemenea metode care ar putea permite evaluări rapide, generale și ușor adaptabile la modificări arhitecturale majore în cadrul arhitecturii cercetate.

În continuare se va prezenta un model analitic general și original de analiză pentru arhitecturile cu predicție de tip BTB, implementate în procesoarele scalare RISC și respectiv în cele MEM. Modelul dezvoltat aici conține anumite principii utilizate de către cel mai prolific cercetător în acest domeniu, ***Harvey G. Cragon,*** în monografia sa [Cra92]. Din păcate, modelele lui *Cragon* sunt ineficiente și parțial nerealiste, întrucât presupun că o instrucțiune de ramificație care nu determină salt propriu-zis, trebuie introdusă în structura BTB chiar dacă anterior nu s-a aflat acolo. După cum s-a arătat și în [Per93], acest lucru este inutil și în plus implică reducerea performanței arhitecturii în

mod considerabil. De asemenea, ***Cragon*** abordează problema exclusiv în cazul particular al unor arhitecturi scalare de procesoare.

Modelul dezvoltat în continuare abordează problematica unui BTB clasic, integrat într-o arhitectură scalară sau de tip MEM. Analiza se bazează pe investigarea detaliată a următoarelor 4 cazuri posibile: saltul nu se află în BTB şi se va face, saltul nu se află în BTB şi nu se va face, saltul se află în BTB şi este predicţionat corect şi respectiv saltul se află în BTB dar este predicţionat incorect. În continuare se va prezenta o analiza detaliată a fiecăruia din aceste cazuri în parte.

1. Saltul nu se află în BTB

Deoarece instrucţiunea de salt/apel nu se află în BTB în momentul aducerii sale din I-Cache, în mod implicit va fi prezisă că nu se va face, procesarea continuând secvenţial în acest caz. Aici se disting 2 subcazuri :

a. Saltul se va face (P=1)

Am notat prin P probabilitatea ca instrucţiunea de salt să se facă. Uzual, pe programe generale, P ia valoarea tipică de 0.67 [Mil89, Cra92]. De asemenea se fac următoarele notaţii:

N = nivelul de procesare din structura pipeline în care se determină condiţiile de salt şi adresa efectivă destinaţie a instrucţiunii de salt. În continuare, presupunem N > 2.

d = numărul de cicli (tacţi) necesari pentru citire (accesare) din BTB. În general, d = 0, căutarea în BTB făcându-se chiar în faza IF.

u = numărul de cicli necesari pentru scriere (actualizare) în BTB. În general u >= d şi uzual u = 0 sau 1.

i = instrucţiunea următoare în secvenţă, instrucţiunii de salt (B).

B = instrucţiunea de salt curentă

T = instrucţiunea la care se face saltul (destinaţie, ţintă).

C = numărul de cicli necesari pentru refacerea contextului procesorului în cazul unei predicţii incorecte (evacuări, actualizări, etc.).

Cu aceste precizări, în acest caz, secvenţa de procesare a instrucţiunilor se va desfăşura în timp ca în tabelul următor :

1. IF	B	d	i	u	T	T+1	T+2
2.		B	d	i	u	T	
N.			B	d		u	T

Tab. 6.1

Aşadar, în acest caz particular, numărul ciclilor de penalizare datorat instrucţiunii de salt B este dat de relaţia :

$$CP = N - 2 + d + Max (u, C) \qquad (6.1)$$

Actualizarea în acest caz se referă chiar la introducerea saltului în BTB, de îndată ce el s-a făcut.

b. Saltul nu se va face (P = 0)

Deoarece saltul nu se va face şi întrucât el nu se găseşte în BTB, nu are sens să fie introdus în structura BTB deoarece introducerea lui nu ar determina îmbunătăţirea performanţei, ci dimpotrivă. În [Cra92] acest considerent nu se aplică, modelul suferind inutil penalizări şi în acest caz. Procesarea temporală a instrucţiunilor s-ar desfăşura ca mai jos:

1. IF	B	d	i	i+1	i+2
2.		B	d	i	
N.			B	d	i

Tab. 6.2

În acest caz penalizarea este minimă şi este determinată de căutarea în BTB. Relaţia de penalizare este :

$$CP = d \qquad (6.2)$$

2. Saltul se află în BTB

Aici se vor analiza detaliat cele 4 subcazuri posibile şi anume :

a. Ptt = 1

Am notat prin Ptt probabilitatea ca instrucţinea de salt curentă să fie predicţionată că se face şi într-adevăr să se facă.

Aici se disting din nou 2 subcazuri, după cum adresa destinaţie memorată în cuvântul BTB este ori nu este corectă.

a1. Adresa destinaţie din BTB este corectă (Pac = 1).

Notăm cu Pac probabilitatea ca adresa din BTB să fie corectă deci nemodificată în raport cu utilizarea ei anterioară. În practică Pac atinge valori mari cuprinse între 0.9 şi 0.99. În acest subcaz procesarea este descrisă prin tabelul următor.

1. IF	B	d	T	u	T+1	T+2
2.		B	d	T	u	
N.			B	d	T	u

Tab. 6.3

Şi aici, ca şi în cazurile următoare, actualizarea (u) semnifică modificarea stării automatului de predicţie, în conformitate cu acţiunea instrucţiunii de salt. Ciclii de penalizarea sunt daţi de relaţia:

$$CP = d + u \qquad (6.3)$$

a2. Adresa destinaţie din BTB nu este cea corectă (Pac = 0)

Înseamnă deci că adresa efectivă a instrucţiunii destinaţie memorată în BTB este diferită de cea actuală, de exemplu datorită unei adresări indirecte sau indexate a instrucţiunii de salt în care conţinutul regiştrilor respectivi a fost modificat sau a

fenomenului de interferență a salturilor în BTB [Yeh92]. Procesarea instrucțiunilor se desfășoară ca mai jos:

1. IF	B	d	T	u	T*		
2.		B	d	T	u	T*	
N.			B	d		u	T*

Tab.6.4

Am notat prin T* instrucțiunea la care se face într-adevăr saltul, corespunzătoare noii adrese modificate. În acest caz penalizarea va fi:

$$CP = N - 2 + d + Max (u, C) \qquad (6.4)$$

b. Ptn = 1

Am notat prin Ptn probabilitatea ca saltul respectiv să fie prezis că se face şi în realitatea să nu se facă. În acest caz, procesarea temporală a instrucțiunilor este descrisă prin tabelul următor.

1. IF	B	d	T	u	i		
2.		B	d	T	u	i	
N.			B	d		u	i

Tab.6.5

Desigur că şi în acest caz instrucțiunea va rămâne în BTB până la evacuarea sa naturală. Ciclii de penalizarea sunt:

$$CP = N - 2 + d + Max(u, C) \qquad (6.5)$$

c. Pnn = 1

Am notat prin Pnn probabilitatea ca saltul respectiv să fie prezis că nu se face şi într-adevăr să nu se facă. Procesarea este prezentată în Tab.6.6.

1. IF	B	d	i	u	i+1	
2.		B	d	i	u	
N.			B	d	i	u

Tab.6.6

$$CP = d + u \qquad (6.6)$$

d. Pnt = 1

Prin Pnt am notat probabilitatea ca saltul să fie prezis că nu se face şi în realitate se va face.

1. IF	B	d	i	u	T	T+1	T+2
2.		B	d	i	u	T	T+1
N.			B	d		u	T

Tab.6.7

În acest caz penalizarea este:

$$CP = N - 2 + d + Max(u, C) \qquad (6.7)$$

Superpoziționând toți acești cicli de penalizare (6.1-6.7) obținuți în cazurile particulare prezentate anterior, obținem numărul mediu de tacți de penalizare introduși prin execuția unei anumite instrucțiuni de ramificație (CPM):

$$CPM = (N - 2 + d + Max(u,c))\,((1 - Pbtb)P + Pbtb(Ptt(1 - Pac) + Ptn + Pnt)) + Pbtb(d + u)\,(Ptt * Pac + Pnn) + d(1 - Pbtb)\,(1 - P) \qquad (6.8)$$

Am notat prin Pbtb, probabilitatea ca instrucțiunea de salt adusă curent din I-CACHE, să se afle memorată în BTB.

Este clar că obiectivul principal al proiectării este minimizarea expresiei lui CPM și deci maximizarea performanței. Minimizarea lui CPM se obține prin minimizarea parametrilor N, d, u, C, Pnt, Ptn. Este desigur dificilă o soluție analitică pentru CPM minim în condițiile variațiilor rezonabile ale diverșilor parametri.

În cazul unui procesor pipeline scalar, performanța sa (rata de procesare) este dată de relația:

$$IR = 1 / (1 + Pb * CPM), [instr./tact] \qquad (6.9)$$

unde:

Pb = probabilitatea ca instrucțiunea curentă să fie o instrucțiune de salt

În cazul unui procesor pipeline superscalar (MEM), putem scrie:

$$CPI = CPIideal + CPM * Pb \qquad (6.10)$$

unde:

CPIideal = numărul mediu de cicli/instr. dacă ciclii de penalizare introduși de branch-uri sunt nuli și dacă predicția branch-urilor se consideră perfectă. Evident că CPI < 1. Rezultă imediat că în cazul arhitecturilor superscalare, rata de procesare (IR) este dată de relația:

$$IR = 1 / CPI \qquad (6.11)$$

O formulă aproximativă dar utilă a parametrului CPM, se poate deriva din cea exactă, anterior obținută, pe baza unor valori particulare considerate realiste a unora din multiplii parametri componenți și anume [Cha94, Per93, Cra92]:

d = 0, u = 1, C = 2, Pac = 1 (0.98 în realitate [Cra92]).

Cu aceste particularizări, de altfel realiste, obținem:

$$CPM = N((1-Pbtb)P + Pbtb(1-Ap)) + Pbtb\,Ap, \qquad (6.12)$$ unde:

Ap=Ptt + Pnn şi reprezintă **acurateţea predicţiei.**

În acest moment devine interesant de precizat performanţa unui procesor superscalar (MEM) care s-ar obţine în lipsa oricărei predicţii a branch- urilor. În acest caz rezultă imediat, în mod succesiv, relaţiile:

$$CPM = P(N - 1) + C \qquad (6.13)$$
$$CPI = CPideal + Pb * CPM \qquad (6.14)$$
$$IR = 1 / CPI \qquad (6.15)$$

În concordanţă cu [Per93, Cra92], se pot particulariza următorii parametri consideraţi realişti şi reprezentativi:

$N = 3$, $Pb = 0.317$, $P = 0.67$, $Pbtb = 0.98$ şi $CPIideal = 0.25$

În acest caz performanţa arhitecturii superscalare în funcţie de gradul de acurateţe al predicţiei hardware (Ap) este prezentată în graficul următor, pe baza relaţiilor anterior determinate. Spre comparare, se prezintă o arhitectură superscalară cu predicţie BTB faţă de una fără predicţie integrată.

De remarcat că pentru Ap = 0.7 rezultă IR = 1.31, iar pentru Ap = 1 rezultă IR = 1.76 adică o creştere a performanţei cu peste 34%. De altfel de la Ap = 0.9 la Ap = 1.0, performanţa creşte cu 12%, deci relativ semnificativ.

La o acurateţe a predicţiei de 80%, extrem de uşor de atins, performanţa arhitecturii creşte cu 88% faţă de cazul fără predicţie, iar la o acurateţe de 90%, absolut comună, performanţa creşte cu 106% faţă de cazul fără predicţie. Este adevărat însă că parametrii Pbtb şi Ap nu sunt complet independenţi cum de altfel se consideră şi în [Cra92], ambii depinzând de capacitatea BTB-ului, strategia de predicţie utilizată, structura informaţiei din BTB, etc. Totuşi, consider asemenea grafice ca fiind deosebit de utile în estimarea rapidă a performanţelor asociate diverşilor parametri arhitecturali.

Ca un alt exemplu de aplicare a relaţiilor analitice obţinute, considerând aceiaşi parametri aleşi şi în plus acurateţea Ap = 0.93 tipică, obţinem o performanţă realistă: IRreal = 1.62 instr./tact.

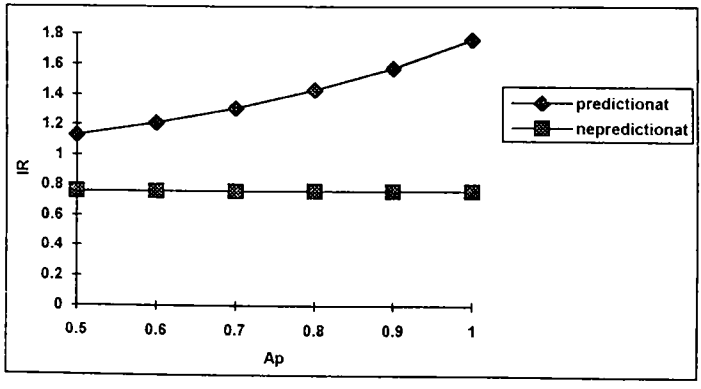


Figura 6.1

De menționat că în cazul unei predicții perfecte Ap s-ar obține IRperfect = 1.76 instr./tact, față de IRideal = 4 instr./tact și asta numai datorită penalizărilor de accesare ale BTB-ului, relativ scăzute și întârzierii calculării adresei de salt (N = 3)!

De asemenea, se observă clar că o acuratețe a predicției de 93%, foarte bună în schemele BTB neadaptive, diminuează performanța față de cazul unei acurateți perfecte a predicției cu cca 9% în condițiile date, ceea ce este în acord cu alte cercetări [Yeh92]. Există deci potențial de îmbunătățire a performanței arhitecturilor superscalare prin tehnici de predicție performante. Încă din 1992 s-au făcut pași importanți în acest sens prin dezvoltarea unor scheme de predicție adaptive pe 2 nivele corelate, care ating o acuratețe de până la 97% măsurat pe benchmark-urile SPEC.

În concluzie, s-a reușit o modelare analitică realistă a performanței arhitecturilor MEM în funcție de parametrii caracterisitici ai tehnicilor de predicție hardware a branch-urilor. Avantajele acestei metode față de cele clasice bazate pe simulare, constau în faptul că este generală, oarecum independentă de mașină și generează rapid parametrii legați de optimizarea predicției , fără să implice simulări laborioase.

Prin particularizări realiste ale diverșilor parametri, metoda furnizează rezultate concordante cu cele publicate în literatura de specialitate și obținute pe bază de simulare.

6.2. SIMULATOR DE PREDICTOR HARDWARE CORELAT PE 2 NIVELE

În acest paragraf se prezintă un simulator de tip "trace driven simulation", destinat predicției hardware adaptive pe două nivele, a instrucțiunilor de ramificație. De precizat că acest tip de scheme de predicție, după cum deja am arătat în cap.2, sunt cele mai performante la ora actuală în cadrul arhitecturilor MEM [Yeh92, Hen96, CheC96].

Acest predictor hardware este integrat, pentru prima dată, în cadrul arhitecturii de procesor HSA care, inițial, nu a fost gândită să realizeze predicția hardware a ramificațiilor [Col95, Ste96], aceasta constituind ideea autorului acestei lucrări. Schema de principiu a predictorului implementat este prezentată în figura următoare (6.1).

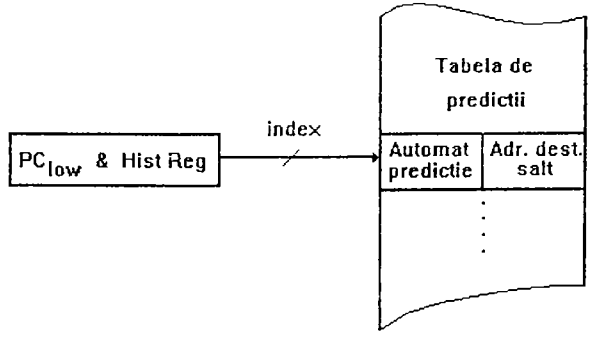


Figura 6.2

Se va urmări deci analizarea în premieră, a fezabilității unui asifel de predictor integrat în cadrul arhitecturii HSA.

Hist Reg reprezintă "registrul istorie" al predicțiilor și conține valori binare semnificând comportarea ultimelor k instrucțiuni de ramificație. În cadrul simulatorului

dezvoltat, Hist Reg are o lungime variabilă cuprinsă între 6 şi 14 biţi şi este văzut de program ca un întreg pe 2 octeţi prelucrat la nivel de bit pentru fiecare rang al registrului. De asemenea s-a parametrizat şi lungimea variabilei PClow, utilizată în adresarea tabelei de predicţii.

Pentru tabela de predicţii s-a folosit o structură vectorială de înregistrări. Fiecare înregistrare memorează adresa destinaţie a saltului şi respectiv starea automatului de predicţie asociată contextului la acel moment dat. În cadrul simulatorului schema de automat de predicţie utilizat poate fi stabilită iniţial de către utilizator. Astfel se pot alege automate având între 2 şi 16 stări.

Programul procesează trace-uri HSA speciale, provenite din compilarea şi executarea benchmark-urilor Stanford. Aceste trace-uri speciale vor conţine după cum este şi firesc, toate instrucţiunile de ramificaţie din benchmark, în ordinea executării lor. Fiecare dintre aceste instrucţiuni de ramificaţie din trace, au asociate PC-ul corespunzător şi respectiv adresa destinaţie a saltului, esenţială pentru verificarea corectitudinii predicţiei.

În realitate trace-urile HSA conţin doar branch-urile care s-au făcut, din motive de economie de spaţiu, după cum am mai arătat. Au trebuit deci generate pe baza acestora şi a surselor în asamblare, trace-uri speciale conţinând şi salturile ineficetive.

În principiu, simulatorul dezvoltat funcţionează astfel:

1. Iniţializare simulator

Aici, utilizatorul va stabili numărul biţilor ce caracterizează registrul Hist Reg, PClow, precum şi tipul automatului de predicţie din cadrul tabelei de predicţii.

Tot acum se iniţializează cu zero adresele destinaţie şi starea automatului de predicţie utilizat.

2. Simularea propriu-zisă

Se stabileşte de către utilizator benchmark-ul de tip trace care va fi utilizat. Din acest benchmark, se citesc secvenţial instrucţiunile de ramificaţie şi se compară predicţia reală din trace cu cea propusă din tabelă. Aici pot să apară 3 cazuri distincte: predicţie corectă, predicţie incorectă, predicţie incorectă datorată exclusiv adresei de salt incorecte din tabelă. Acest ultim caz se poate datora faptului că adresa de salt din tabelă a fost modificată de exemplu de către un alt salt, având astfel un **fenomen de interferenţă al salturilor** dar pot fi şi alte cauze posibile.

În continuare se vor actualiza corespunzător registrul Hist Reg şi respectiv locaţia folosită din tabela de predicţii.

3. Generarea de rezultate

La finele simulării propriu-zise se generează rezultate semnificative precum numărul total de salturi executate, procentajul de predicţii corecte, incorecte şi respectiv afectate de interferenţe ale salturilor.

În continuare se vor prezenta şi analiza câteva dintre rezultatele obţinute prin exploatarea acestui simulator de predictor hardware pe două nivele.

6.3. REZULTATE OBŢINUTE PENTRU O SCHEMĂ CORELATĂ DE PREDICŢIE, INTEGRATĂ ÎN PROCESORUL HSA

În continuare se prezintă câteva rezultate semnificative obţinute prin exploatarea simulatorului anterior descris pe 5 din suita benchmark-urilor Stanford. Mai jos se prezintă procentajul instrucţiunilor de ramificaţie în cadrul acestor programe precum şi procentajul din numărul total al instrucţiunilor de salt care într-adevăr să determine un

salt în program. Rezultă că în cadrul acestor programe, în medie 15% din instrucțiuni sunt ramificații. Dintre acestea, cca 78% se fac.

Benchmark	% branch-uri	% salt efectiv
bubble	20	75
matrix	9	97
perm	15	80
sort	17	65
tower	15	77

Figura 6.3

În figura următoare (figura 6.4) se prezintă procentajul predicțiilor eronate, obținute prin exploatarea simulatorului pe benchmark-urile respective. Simularea s-a făcut considerând o tabelă de predicții având capacitatea 16k, considerând registrul Hist Reg de 10, 8 și 6 biți.

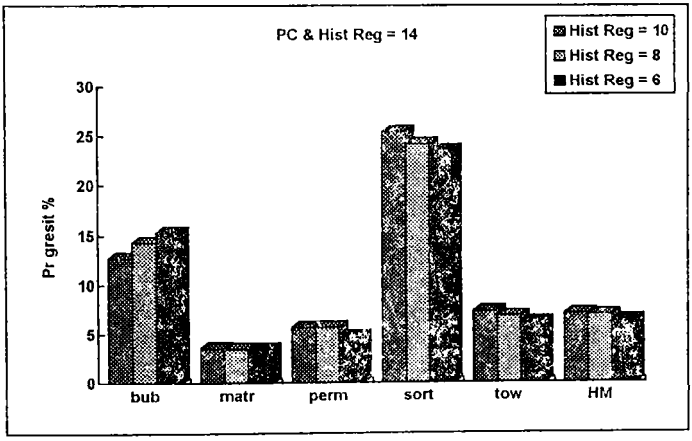


Figura 6.4

În aceste condiții s-au obținut rate medii (armonice-HM) de miss de 7.06%, 6.95% și 6.4% respectiv. Dacă s-ar fi ajuns la Hist Reg pe 4 biți, s-ar fi obținut o rată medie de miss de 7.07% ceea ce arată clar faptul că, performanța optimă se obține pentru Hist Reg pe 6 biți și anume o acuratețe a predicției de 93.6%, comparabilă cu cele obținute prin cercetări consacrate [Yeh92, Per93]. Normal, cel mai bine s-a comportat benchmark-ul *matrix* (predicție corectă în 96.5% din cazuri) întrucât aici 97% din salturi se fac. În plus, acestea sunt deosebit de predictibile, ca în toate programele cu un caracter numeric accentuat de altfel.

O problemă interesantă care s-a dorit studiată a fost următoarea: în ce măsură predicțiile, altfel corecte, sunt afectate de modificarea adresei efective în tabela de predicții? La această întrebare se răspunde în graficul din figura următoare (fig. 6.5).

Simularea s-a realizat considerând Hist Reg de 10, 8 și 6 biți și s-au obținut adrese efective modificate în 5.16%, 4.89% și 4.73% din cazuri respectiv (medii armonice). Din nou optimul se obține și din acest punct de vedere pentru Hist Reg pe 6 biți (simulări pentru Hist Reg pe 4 biți generează adresă greșită în 4.8% din cazuri).

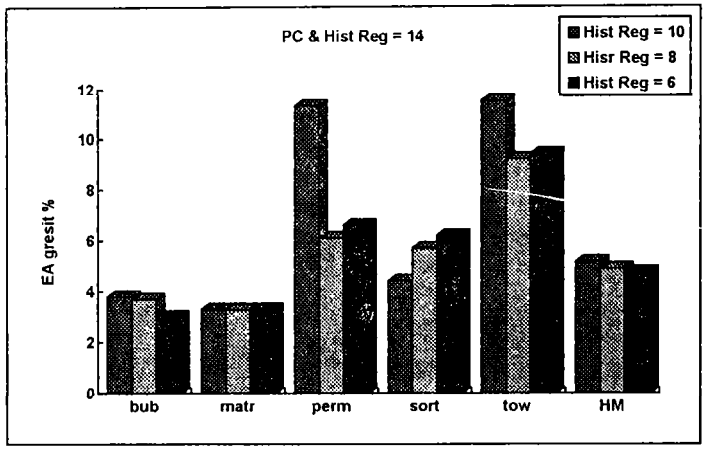


Figura 6.5

În continuare (fig. 6.6) se prezintă rata predicțiilor eronate pentru un model având tabela de predicții doar de 1k. În acest caz, pentru Hist Reg pe 6, 4 și 2 biți s-au obținut rate medii de 7.59%, 7.08% și respectiv 7.42%. Rezultă deci o acurateţe medie maximă de 92.92% a predicțiilor în acest caz, obținută pentru Hist Reg pe 4 biți.

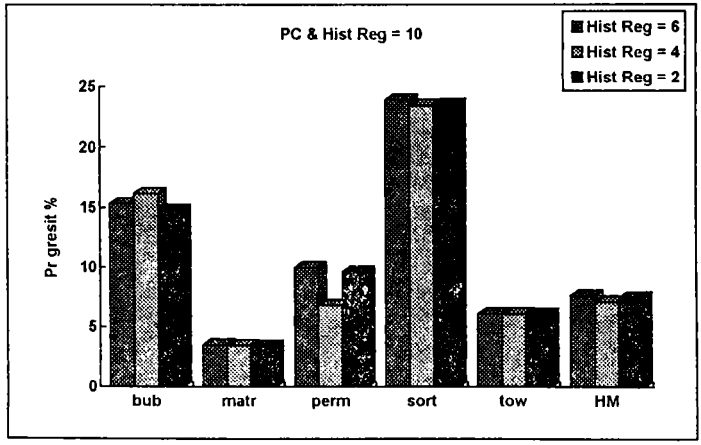


Figura 6.6

În figura următoare (fig. 6.7) se prezintă în aceleași condiții cu cele precedente, procentajul din totalul predicțiilor care caracterizează predicții incorecte din cauza modificării adresei saltului. S-au obținut în acest caz procentaje medii de 5.48%, 4.92% și 5.33% din totalul predicțiilor. Și aici performanța minimă se obține pentru Hist Reg pe 4 biți.

Din cele prezentate, precum și din alte cazuri particulare explorate cu ajutorul simulatorului implementat, rezultă că optimul între gradul de corelare (Hist Reg) și

capacitatea tabelei (adresată prin intermediul PClow & Hist Reg), se obține pentru lungimi ale Hist Reg de cca 40% din numărul total al biților de adresare tabelă.

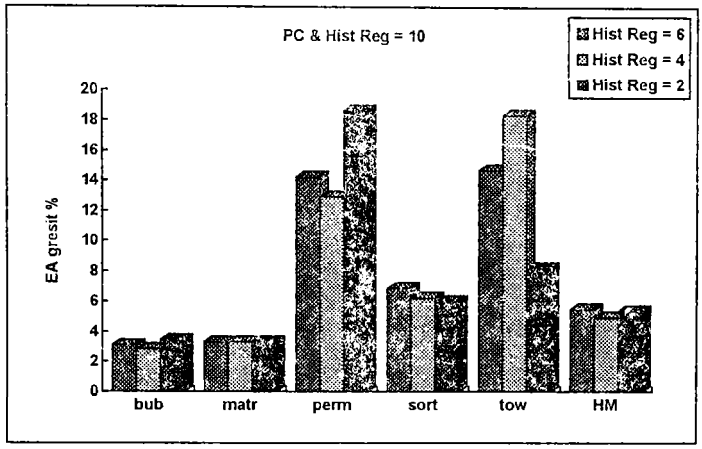


Figura 6.7

Altfel spus, aici stabileşte simularea compromisul optim între 2 procese complementare: **gradul de corelare** al saltului (Hist Reg "mare") şi respectiv **gradul de localizare** al saltului (PClow "mare"). Suma celor doi parametri este fixată prin proiectare, aşadar compromisul optim trebuie găsit. Rezultatele acestei simulări demonstrează foarte clar **următorul proces**: un grad de localizare scăzut determină interferenţe ale unor salturi diferite la aceeaşi locaţie din tabelă rezultând predicţii eronate pe motiv de adresă de salt alterată, deci scade performanţa. Pe de altă parte un grad de localizare ridicat determină un grad de corelare scăzut şi deci schema devine inefectivă în cazul unor salturi corelate, datorită nesituării adecvate în context a predicţiei. Şi în acest caz performanţa globală scade. Optimul este un compromis între aceste două situaţii extreme, după cum a şi rezultat.

Simulatorul construit generează soluţia optimală pentru orice schemă corelată de predicţii.

În concluzie, un astfel de predictor integrat în cadrul procesorului HSA conduce la rezultate foarte bune, pe deplin comparabile cu cele prezentate în literatură şi constituie o alternativă superioară compensării statice a BDS-ului, propuse în cadrul acestui procesor [Col95,Ste96]. Simulatorul construit conduce la soluţia constructivă optimă de schemă de predicţie corelată pe 2 nivele, integrată într-o arhitectură superscalară.

7. CERCETĂRI ÎN OPTIMIZAREA PROGRAMELOR PE ARHITECTURILE MEM. EVALUĂRI CANTITATIVE ASUPRA "LIMITELOR" OPTIMIZĂRILOR STATICE

În cele ce urmează vom prezenta 2 investigații, ambele referitoare la optimizarea programelor în arhitecturile cu paralelism redus.

Prima cercetare abordează problema optimizării unităților secvențiale de program în cadrul unei arhitecturi superscalare simplificate și parametrizabile, definită de către autor. Cercetarea se bazează pe un scheduler/ simulator special conceput pentru acest scop, iar algoritmul de optimizare este unul de tip "List Scheduling". Se prezintă comparativ și într-un mod cantitativ, beneficiile obținute prin scheduling static local. De asemenea, prin prisma rezultatelor obținute, se prezintă limitele optimizărilor locale în raport cu cele globale.

A 2-a investigație încearcă să răspundă la o problemă, credem noi, fundamentală și anume: mai putem spera rezultate spectaculoase de la acest domeniu? Altfel spus, **mai există potențial de performanță neexploatat** relativ la schedulingul static? După cum se va vedea, pe o bază de simulare de asemenea cantitativă, răspunsul meu va fi unul optimist.

7.1. INVESTIGAȚII ÎN OPTIMIZAREA BASIC-BLOCK-URILOR PENTRU EXECUȚIA PE ARHITECTURI SUPERSCALARE

7.1.1. INTRODUCERE. O ARHITECTURĂ SUPERSCALARĂ SIMPLIFICATĂ

Problema optimizării programelor în vederea procesării lor pe arhitecturi superscalare și VLIW este una de mare interes în cercetarea actuală. Optimizarea se referă în general la 2 aspecte: optimizarea locală, adică în cadrul unităților secvențiale de program (basic-block-uri) și respectiv optimizarea globală, adică a întregului program. Această a doua abordare constituie momentan o problemă deschisă, nefiind încă rezolvată la modul general și nici din punct de vedere al optimalității algoritmilor utilizați [Joh91, Ebci 88, Ste96]. Nu se va aborda această problemă în prezentul studiu, limitându-mă la investigarea câtorva aspecte legate de **optimizările locale,** în cadrul basic-block-urilor.

După cum am prezentat în [Vin96], problema optimizării programelor în vederea execuției lor optime implică în general următoarele etape succesive mai importante:
- determinarea grafului de control al programului de analizat. Aceasta se rezolvă pe baza unor algoritmi de partiționare a programului în basic-block-uri;
- determinarea grafurilor dependențelor de date și respectiv precedențelor pentru fiecare unitate;
- optimizarea locală a basic-block-urilor. Aici se utilizează în general algoritmi de tip "List Scheduling" (LS) pe care îi vom folosi și noi în continuare;
- optimizarea globală a programului. Aceasta se bazează pe algoritmi determiniști de tip "Trace Scheduling" [Hen96, Joh91] sau pe baza unor algoritmi euristici de tip "Percolation" [Col95].

În continuare se va prezenta o investigație în domeniul optimizării basic-block-urilor pentru o arhitectură RISC superscalară deosebit de simplă, definită de autor și

configurabilă. Schema bloc de principiu a acestui procesor superscalar virtual este prezentată în figura 7.1.

Fiecare instrucțiune este procesată pipeline în 5 stagii de execuție succesive (IF, ID, ALU, MEM, WB). Arhitectura de memorie este una clasică, de tip Harvard (I-CACHE, D-CACHE). Bufferul de prefetch s-a considerat de capacitate practic nelimitată. Procesarea instrucțiunilor s-a considerat a fi de tip "In Order" aspect absolut necesar având în vedere că programele sunt preoptimizate prin software. Procesorul deține 3 grupe de unități de execuție independente (ALU, SHIFT și unitatea LOAD/STORE) și 2 seturi de regiștri generali, unul principal și altul secundar, acesta din urmă fiind utilizat în renaming-ul impus de algoritmul LS în vederea eliminării hazardurilor de date de tip WAR și WAW(Write After Read, Write After Write). Cade în sarcina decodificatorului să aloce în mod dinamic instrucțiunile din bufferul de prefetch spre unitățile de execuție.

Arhitectura este configurabilă după cum urmează:

- rata de fetch a instrucțiunilor cuprinsă între 2 și 6 instrucțiuni simultan;

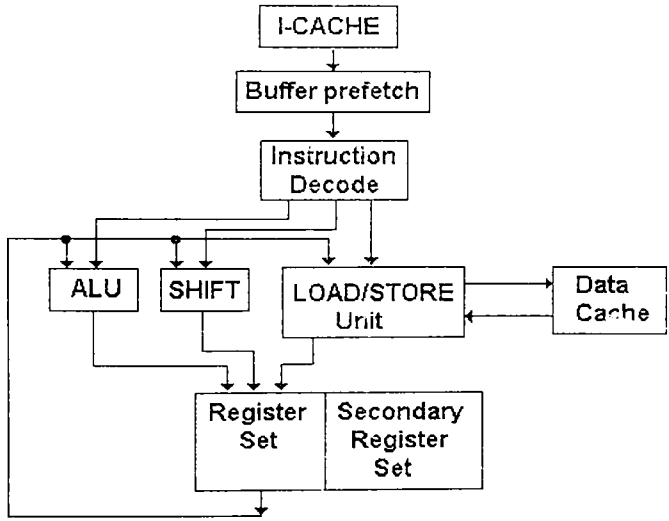


Figura 7.1

- numărul de unități ALU cuprins între 1 și 4;
- numărul de unități pentru deplasări și rotiri SHIFT între 1 și 2;
- o singură unitate LOAD/STORE;
- numărul de regiștri generali cuprins între 32 și 64.

Am considerat următoarele tipuri de instrucțiuni tipice, capabile de a fi executate de către această arhitectură:

- instrucțiuni ALU având sintaxa: ALU Ri, Rj, Rk sau ALU Ri, Rj, #const (Ri destinație);
- instrucțiuni de tip LOAD având sintaxa LD Ri, (Rj) offset;
- instrucțiuni de tip STORE având sintaxa ST Rj, (Ri) offset;
- instrucțiuni de deplasare și rotire având sintaxa SHIFT Ri, Rj, #const și semantica: procesează (Rj) cu <#const > biți și introdu rezultatul în Ri.

Se precizează că toate instrucțiunile cu excepția celor de tip LOAD au latența în execuție de un tact. Instrucțiunile LOAD au latența de două tacte. Unitatea LOAD s-a considerat a fi pipeline-izată.

Obiectivul principal a fost acela de a aborda cantitativ eficiența schedulingului prin metoda LS pe această arhitectură simplifcată. Pentru aceasta, s-a implementat un scheduler şi un simulator în limbajul C pentru acest procesor parametrizabil.

7.1.2. PRINCIPIUL METODEI DE OPTIMIZARE UTILIZATĂ

Se prezintă pe scurt algoritmul LS utilizat în optimizare de către programul scheduler/simulator implementat. Se consideră pentru exemplificare un procesor RISC superscalar având 2 unități ALU, o unitate SHIFT şi o unitate LOAD/STORE. De asemenea se consideră rata de fetch a instrucțiunilor ca fiind de 4. Benchmark-ul pe care vom face exemplificarea algoritmului (b1) este primul dintre cele 7 benchmark-uri pe care le-am exploatat prin programul scheduler şi de optimizare. În continuare prezentăm acest program:

```
0:    LD      R1, (R0)8
1:    SHIFT   R3, R2, #2
2:    ALU     R1, R9, R10
3:    ALU     R1, R1, R3
4:    SHIFT   R2, R3, #4
5:    ALU     R2, R1, R2
6:    ALU     R1, R6, R8
7:    ALU     R1, R1, R7
8:    SHIFT   R1, R1, #6
9:    LOAD    R1, (R1)2
10:   LOAD    R4, (R4)0
11:   ALU     R1, R4, R1
12:   ALU     R1, R1, R2
13:   ST      R1, (R4)0
14:   ST      R3, (R1)2
15:   ST      R2, (R0)16
```

În urma analizei acestui program, schedulerul construieşte graful dependențelor/precedențelor de date ca în figura următoare (7.2):

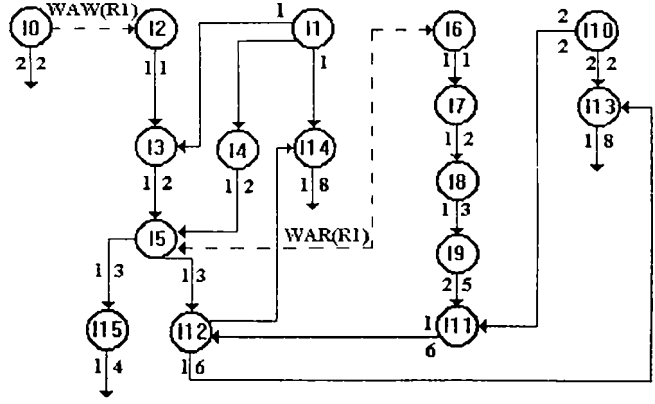


Figura 7.2.

Având în vedere resursele disponibile ale procesorului, acesta va executa programul b1 aşa cum se sugerează în tabelul 7.3

De menţionat că pentru aceasta, s-a implementat un algoritm de renaming al regiştrilor, în vederea eliminării hazardurilor de date de tip WAR şi WAW. Astfel de exemplu, pentru acest program, în varianta optimizată, pe ramura 6-11 a grafului dependenţelor/precedenţelor se face redenumirea registrului R1 din setul principal cu un registru disponibil din setul secundar (Rt1).

Tact	Instr./prioritate
1	13/8
2	14/8
3	12/6, 15/4
4	5/3, 11/6, 10/2
5	3/2, 4/2
6	2/1, 1/1, 8/3, 9/5
7	7/2, 0/2
8	6/1

Cycle	ALU 1	ALU 2	SHIFT	LOAD/STORE
1	I6			
2		I7		I0
3	I2		I8	I9
4		I3	I4	
5	I5	I11	I1	I10
6		I12		I15
7				I14
8				I13

Figura 7.3

De asemenea, instrucţiunea 12 devine în varianta optimizată: ALU R1, Rt1, R2.

Se poate remarca faptul că rata de procesare pentru programul iniţial este de 1.14 instr./tact în timp ce rata pentru programul optimizat ajunge la 1.875 instr./tact, rezultând deci o creştere a performanţei prin scheduling cu 64% în acest caz. Totodată se observă uşor că utilizarea resurselor hardware este îmbunătăţită radical prin scheduling. Utilizarea ALU a ajuns la 75%, a unităţii SHIFT la 38% şi a memoriei de date la 75%.

În continuare se vor prezenta câteva rezultate obţinute prin intermediul schedulerului şi simulatorului implementat. Menţionăm că, asumând o predicţie perfectă a instrucţiunilor de salt condiţionat, cu anumite extinderi şi perfecţionări, simulatorul s-ar putea utiliza şi pentru determinări cantitative în optimizarea globală.

Acest program permite configurarea arhitecturii în limitele expuse, editarea şi încărcarea unui program de test, optimizarea acestuia, simularea execuţiei sale, etc. La ieşire, programul generează date statistice edificatoare privind performaţele obţinute în urma procesării, cum ar fi: rate de procesare pentru programele iniţiale şi optimizate, coeficienţi de utilizare pentru diversele resurse hardware, etc.

7.1.3. REZULTATE OBȚINUTE ÎN OPTIMIZAREA BASIC-BLOCK - URILOR PE ARHITECTURA SUPERSCALARĂ SIMPLIFICATĂ

Se prezintă rezultatele obținute prin optimizarea și simularea în execuție a 7 benchmark-uri (b1, ...,b7), de tip basic-block. Acestea au fost preluate din cadrul unor benchmark-uri consacrate precum Stanford, SPECint, etc.

Structura acestor benchmark-uri este tipică din punct de vedere al distribuției grupelor de instrucțiuni și este prezentată în tabelul următor.

Benchmark	Nr. de instr.	% ALU	% SHIFT	% LOAD/STORE
b1	16	44	19	37
b2	14	29	35	36
b3	12	66	9	25
b4	10	40	20	40
b5	8	62	13	25
b6	27	41	19	40
b7	12	58	17	25
Medie.	12.44	45.12	16.25	31.13

Figura 7.4

În primul rând am testat o configurație minimală, având rata de fetch 4, o unitate ALU, o unitate SHIFT și o unitate LOAD/STORE. După cum se observă în figura următoare, s-a obținut o rată medie de procesare măsurată pe testele inițiale de 1.12 instr./tact.

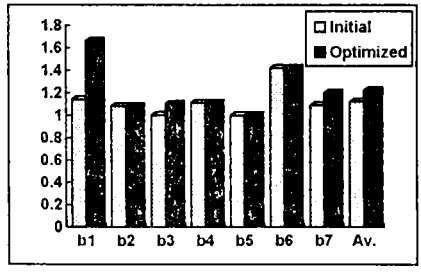


Figura 7.5

În schimb, rata de procesare după optimizare a crescut la 1.22 instr./tact în medie, deci o creștere medie de 9%.

A doua configurație arhitecturală testată, numită "tipică" întrucât caracterizează multe procesoare superscalare de referință [Joh91], s-a deosebit de cea minimală doar prin faptul că deține 2 unități ALU în loc de una singură. După cum se observă în figura 7.6. , rata de procesare pentru aceleași benchmark-uri a crescut spectaculos după scheduling la 1.50 instr./tact, adică cu 33% mai mult decât în cazul configurației minimale. În schimb performanța a rămas aceeași pentru benchmark-urile neoptimizate.

În fine, a 3-a configurație arhitecturală, maximală din punct de vedere al programului implementat, conține 4 unități ALU, 2 unități SHIFT și o unitate

LOAD/STORE. În acest caz rata de fetch a fost mărită la 6 instrucțiuni. Se observă că rata de procesare după scheduling a ajuns la 1.53 instr./tact, adică cu 35% mai mare față de configurația minimală, dar crescută nesemnificativ în comparație cu rata de procesare obținută pentru arhitectura tipică.(v. Fig. 7.7.)

S-au mai testat și alte arhitecturi intermediare, concluzia fiind aceeași și în deplină concordanță cu referințele bibliografice: arhitectura tipică pare a fi optimă din punct de vedere al raportului performanță/cost din punct de vedere al procesării optimale a basic-block-urilor.

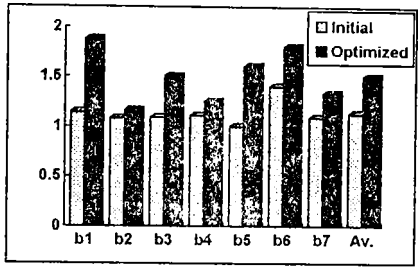


Figura 7.6.

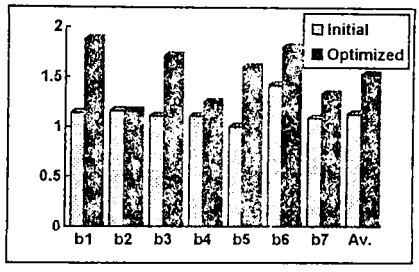


Figura 7.7

În continuare, s-a pus problema determinării coeficienților de utilizare aferenți resurselor hardware. Este clar că prin scheduling utilizarea acestor resurse devine mai bună. Am definit coeficientul de utilizare al unei anumite resurse hardware (Ku) astfel:

$Ku = (Nu * 100\%) / N * Eu$ (7.1), unde:

Nu = număr de instrucțiuni care utilizează în faza de execuție resursa u.
N = număr total impulsuri de tact în care se execută programul.
Eu = numărul de unități de execuție de un anumit tip.
Rezultă imediat $0\% < Ku < 100\%$.

Coeficienții de utilizare aferenți resurselor ALU, SHIFT și LOAD/STORE sunt prezentați în figurile pentru benchmark-urile în cauză.

Acești coeficienți se prezintă comparativ pentru programele inițiale și cele optimizate. În medie coeficientul de utilizare ALU este de 28% pentru programele neoptimizate și crește la 40% după optimizare. În mod analog, Kshift mediu este 21% respectiv 27% iar Kload/store mediu crește de la 41% la 54% respectiv.

Ca şi în cazul ratelor de procesare şi în cazul coeficienţilor de utilizare aferenţi resurselor hardware performanţa nu se îmbunătăţeşte practic atunci când se trece la variante hardware mai complexe. Astfel, de exemplu, pentru configuraţia maximă, utilizarea resurselor nu se va îmbunătăţi în cazul programelor iniţiale comparativ cu varianta tipică. Pentru programele optimizate, utilizarea resurselor se va îmbunătăţi în mod nesemnifivativ cu un procentaj de (1.53/1.50 -1)*100% = 2%.

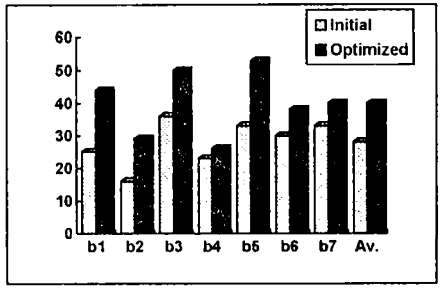


Figura 7.8

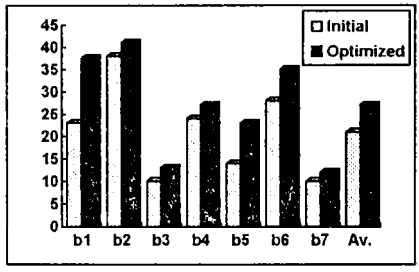


Figura 7.9

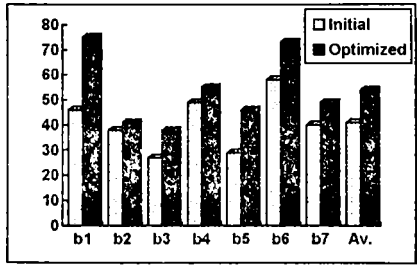


Figura 7.10

În concluzie, şi din acest punct de vedere configuraţia tipică pare a fi optimă prin prisma raportului performanţă/cost.

Ratele de procesare aferente unui procesor pipeline scalar compatibil, sunt prezentate în figura 7.11.

Prin urmare acest procesor execută benchmark-urile neoptimizate cu o rată medie de 0.88 instr./tact. Rezultă deci că arhitectura superscalară tipică este cu 28% (fără scheduling) şi respectiv cu 70% (cu scheduling) mai rapidă decât arhitectura scalară echivalentă care execută programele iniţiale. Prin scheduling global se comunică creşteri de performanţă superioare de cca 300-400% [Ste96, Hwu95]. După cum rezultă şi din această investigaţie, rezultatele optimizării basic-block-urilor sunt relativ modeste ducând la creşteri de performanţă cuprinse între 9% şă 35%.

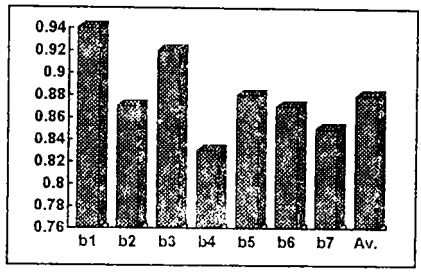


Figura 7.11.

Aceasta se explică în principal prin faptul că nivelul de paralelism al acestor unităţi secvenţiale de program este limitat la 2-3 instrucţiuni după cum se arată în numeroase studii. Rezultă ca sunt necesare metode de scheduling global cât mai agresive şi eficiente. Acestea comunică creşteri de performanţe deosebite ajungându-se până la 200%-300%, dar în schimb cresc necesităţile de memorare ale programelor de 2-3 ori [Col95, Na93].

În acest sens, prezentăm pe scurt un experiment pe care l-am făcut utilizând schedulerul şi simulatorul HSA (Hatfield Superscalar Architecture), ambele dezvoltate la Universitatea din Hertfordshire, U.K. [Ste96, Col95]. Schedulerul este unul global şi conţine metode originale de optimizare de tip "percolation" pentru o arhitectură superscalară puternic parametrizabilă (HSA-Hatfield Superscalar Architecture).

Prezentăm mai jos ratele de procesare obţinute prin simularea a 5 dintre benchmark-urile Stanford, cu şi fără scheduling. Simularea a fost făcută pentru un model maxim al arhitecturii HSA conţinând câte 16 unităţi de execuţie pentru fiecare grup semnificativ de instrucţiuni. Toate instrucţiunile au latenţa de un impuls de tact cu excepţia celor de înmulţire şi împărţire.

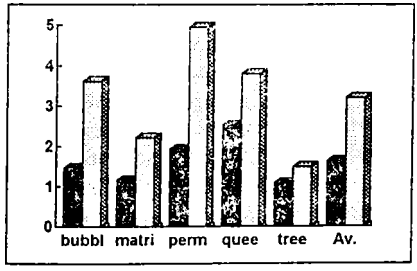


Figura 7.12.

A rezultat că gradul de utilizare al resurselor hardware este foarte scăzut în acest caz. De exemplu, dintre cele 16 unități ALU primele 3 sunt utilizate în medie cca 32% din timp, următoarele 4 cca 1% din timp iar restul de 9 unități ALU sunt neutilizate.

Ratele medii de procesare pentru programele Stanford neoptimizate sunt 1.63 instr./tact, iar după scheduling ajung la 3.21 instr./tact, deci o creştere a performanței de 97% față de 33% cât s-a obținut prin optimizarea doar a basic-block-urilor în această investigație.

Aceasta constituie încă o dovadă că pentru a obține performanțe deosebite schedulingul trebuie să fie de tip global, exploatând deci paralelismul întregului program şi nu doar cel din cadrul unităților secvențiale de program.

O problemă interesantă şi puțin dezbătută în literatură este aceea de a determina **gradul ideal de paralelism exploatabil** de către un procesor cu execuții multiple ale instrucțiunilor. Cu alte cuvinte, există oare suficient paralelism care să justifice cercetări viitoare în scheduling şi care să poată duce la rezultate spectaculoase? Asupra acestei probleme se va concentra paragraful următor, prezentând o metodologie originală şi rezultatele aferente.

7.2. INVESTIGAȚII ASUPRA LIMITELOR DE PARALELISM ÎNTR-O ARHITECTURĂ SUPERSCALARĂ

7.2.1. PRINCIPIUL METODEI UTILIZATE ÎN INVESTIGARE

În ultimii ani se manifestă un interes deosebit pe plan mondial în dezvoltarea unor metode şi algoritmi de scheduling static global pentru arhitecturile MEM. Aceste schedulere - unele chiar integrate în compilatoare [Hwu95] - asamblează în aşa-numite grupuri, instrucțiuni independente din program, în scopul execuției simultane a instrucțiunilor aparținând aceluiaşi grup. Această investigație realizată de grupul de la Universitatea din Hertfordshire împreună cu subsemnatul, se bazează pe arhitectura HSA (Hatfield Superscalar Arhitecture) şi a fost prezentată în detaliu în [PSVin97].Există totuşi câteva aspecte ce se vor prezenta în premieră în această lucrare (de ex. instrucțiunile combinate). Arhitectura HSA dezvoltată la Universitatea din Hertfordshire, U.K., reprezintă o arhitectură superscalară- VLIW hibridă, care aduce anticipat din I-CACHE instrucțiuni multiple într-un buffer de prefech. În fiecare tact, logica de decodificare trimite "In-Order" spre execuție din bufferul de prefech cât mai multe instrucțiuni independente pentru a fi executate în paralel. Optimizarea programelor se face static printr-un scheduler special conceput [Col95, Ste96].

Scopul acestei cercetări este de a "măsura" gradul de ILP (Instruction Level Parallelism) existent în benckmark-urile Stanford, compilate special pentru arhitectura HSA utilizând compilatorul GNU CC [Col96]. Aceste 8 benchmark-uri au fost scrise în C şi propuse de către John Hennessy de la Universitatea din Stanford, U.S.A., cu scopul de a constitui **"numitorul comun"** în evaluarea performanțelor arhitecturilor ILP. Acestea sunt considerate deosebit de reprezentative pentru aplicațiile de uz general (non-numerice) şi realizează aplicații generale precum: sortări (bubble, tree şi sort), aplicații puternic recursive (perm, puzzle, tower-problema turnurilor din Hanoi), şi alte aplicații clasice (matrix- procesări de matrici, queens - problema de şah a celor 8 regine). În urma compilării acestor benchmark-uri C, s-au obținut programe asamblare HSA (*.ins).

Principiul metodei utilizate în investigare se bazează pe implementarea unui simulator TDS (**Trace Driven Simulator**), care să lucreze pe trace-urile HSA (*.trc) ale bench-urilor Stanford. Aceste **trace-uri** reprezentând în principiu toate instrucțiunile

maşină HSA dintr-un program, scrise în ordinea execuţiei lor şi memorate într-un fişier, s-au obţinut pe baza simulatorului HSA dezvoltat anterior [Col95]. Acest simulator procesează benchmark-urile Stanford gata asamblate şi generează parametrii completi aferenţi procesării precum şi trace-urile în diverse forme. De precizat că trace-urile utilizate conţin între cca 200.000 şi 900.000 de instrucţiuni maşină HSA.

În principiu TDS analizează secvenţial toate instrucţiunile dintr-un anumit trace HSA. Fiecărei instrucţiuni i se asociază un **parametru numit PIT** (Parallel Instruction Time), semnificând numărul impulsului de tact în care instrucţiunea respectivă poate fi lansată în execuţia propriu-zisă. Aceasta înseamnă că în acel moment, operanzii sursă aferenţi instrucţiunii respective sunt disponibili. Dacă o instrucţiune următoare este dependentă RAW printr-un registru sau printr-o variabilă de memorie de instrucţiunea curentă, atunci ei i se va aloca un nou PIT dat de relaţia:

$$PITnou = PITvechi + L, \qquad (7.2)$$ unde:

L = latenţa instrucţiunii curente

Acest proces de alocare PIT continuă în mod similar până la finele trace-ului.

Instrucţiuni arbitrar plasate în trace pot avea acelaşi PIT semnificând deci faptul că pot fi executate în paralel.

Se consideră că arhitectura are resurse (unităţi funcţionale, regiştri, etc.) infinite astfel încât oricât de multe instrucţiuni independente pot fi executate în paralel la un moment dat. De asemenea, se ignoră hazardurile false de tip WAR şi WAW, considerându-se deci un "renaming" perfect, analiză anti-alias perfectă şi o predicţie perfectă a branch-urilor (**model ORACLE**). În final se obţine gradul teoretic de paralelism disponibil: **IRteoretic = N/ PITmax,** unde N = numărul total de instrucţiuni din trace. De remarcat că dacă un asemenea model idealizat ar deţine mecanisme de forwarding prin implementarea unor algoritmi de tip Tomasulo, s-ar reduce la maximum posibil citirile din seturile de regiştri. Astfel, s-ar diminua deci hazardurile structurale la regiştri.

Acest indicator este esenţial întrucât va lămuri dacă există suficient paralelism care să justifice în continuare cercetările în scheduling, întrucât realizările actuale cele mai performante comunică rate de procesare de până la 6 instr./tact [Col95]. După cum se va vedea, răspunsul va fi unul pozitiv. O altă problemă, implicată de cele prezentate până acum, este următoarea: de ce nu se obţine IRteoretic în practică? Răspunsul este: datorită unor **limitări fundamentale**, obiective dar şi datorită unor **limitări artificiale**. O limitare fundamentală se referă la chiar conceptul de scheduling static. Acesta este nevoit să fie uneori, inevitabil, conservator datorită informaţiilor necunoscute în momentul compilării programului [Fra92, Col95]. Dintre celelalte limitări fundamentale amintim: hazardurile structurale, de date şi de control.

Limitările artificiale sunt date de "conservatorismul", teoretic evitabil, al schedulerelor actuale şi după cum vom arăta, limitează serios performanţa acestora.

De exemplu, buclele (loops) constituie o astfel de limitare. Multe schedulere forţează execuţia serială a iteraţiilor unei bucle de program deşi ar fi posibilă paralelizarea acestor iteraţii prin tehnici deja cunoscute şi prezentate aici, precum cele de "loop unrolling" sau "trace scheduling". De asemenea, majoritatea schedulerelor actuale nu permit execuţia instrucţiunilor dintr-o buclă până când toate instrucţiunile precedente buclei nu s-au executat. Analog, la ieşirea din buclă. O limitare similară cu cea introdusă de bucle o introduc procedurile.

Un alt exemplu îl constituie reorganizarea statică (execuţia Out of Order) a instrucţiunilor LOAD/STORE. Schedulerele actuale nu permit sau permit în limite foarte

strînse acest lucru, întrucât problema dezambiguizării (analiză antialias) referinţelor la memorie nu este încă pe deplin rezolvată [Nic89, Hua94]. Din păcate un scheduler pur static nu poate distinge întotdeauna dacă două referinţe la memorie sunt permanent diferite pe timpul execuţiei programului.

În fine, o altă limitare de acest tip o constituie latenţa mare a unor instrucţiuni sau memorii care se aşteaptă să fie reduse în viitor prin progrese arhitecturale sau/şi tehnologice.

În cele ce urmează se vor cuantifica pierderile de performanţă introduse prin aceste limitări, demonstrând totodată că există suficient potenţial în acest domeniu în care cercetările sunt doar la început.

Se menţionează că există câteva referinţe bibliografice care abordează această problematică [Lam92, Wal91]. Din păcate concluziile obţinute nu concordă datorită unor metodologii de lucru foarte diferite. Astfel, de exemplu Wall consideră că rata maximă de procesare pe un procesor superscalar nu poate depăşi 7 instr./tact. Acest lucru se datorează faptului că în modelul său schedulingul este exclusiv dinamic realizându-se de fapt exclusiv prin hardware. Având în vedere capacitatea limitată a bufferului de prefech, rezultatul obţinut este absolut normal. Alţii, pe modele mai agresive şi prin scheduling static comunică potenţiale mult mai optimiste cuprinse între 90 şi 158 instr./tact [Lam92].

7.2.2. REZULTATE OBŢINUTE ÎN DETERMINAREA GRADULUI TEORETIC ILP ŞI A INFLUENŢEI DIFERITELOR LIMITĂRI ASUPRA ACESTUIA

7.2.2.1. GRADUL TEORETIC DE PARALELISM

S-a considerat un procesor HSA cu resurse infinite, predictor de branch-uri perfect, renaming perfect al regiştrilor şi dezambiguizare perfectă. Aşadar, timpul de execuţie este restricţionat doar de către dependenţele reale de date. Latenţa tuturor instrucţiunilor este de un tact cu excepţia celor de tip DIV care este 32 tacte şi respectiv MUL, 3 tacte.

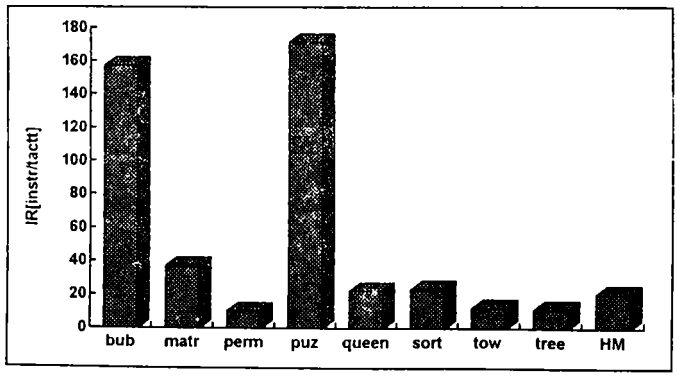


Figura 7.13

Aşadar, pe un model hibrid superscalar- VLIW idealizat, media armonică a ratelor de procesare este de 19.45 instr./tact (media aritmetică ar fi de 55 instr./tact). Toate raportările ulterioare se vor face relativ la acest model de bază.

7.2.2.2. INSTRUCȚIUNI COMBINATE

În [Vas93] se arată că s-a reușit proiectarea și implementarea în tehnologie CMOS a unor unități ALU complexe cu 3 intrări și care nu impun mărirea perioadei de tact a procesorului comparativ cu o unitate ALU clasică. Acest fapt a condus la ideea unor instrucțiuni **ALU combinate** care să conțină trei operanzi sursă în loc de doar doi. Așadar, ar cădea în sarcina schedulerului să combine 2 instrucțiuni ALU dependente RAW într-una singură combinată. Mai precis o secvență de 2 instrucțiuni dependente RAW printr-un registru, ca mai jos:

ADD R1, R2, R3
ADD R5, R1, R9

va fi transformată de către scheduler într-o instrucțiune combinată care va avea același timp de execuție: ADD R5, R2, R3, R9.

Această tehnică, posibil de aplicat atât prin hardware cât și prin software, ar putea fi deosebit de agresivă întrucât ar acționa asupra unei limitări considerată până acum fundamentală și deci imposibil de depășit.

Se prezintă în continuare câteva evaluări cantitative ale acestei tehnici noi, pe arhitectura HSA și trace-urile Stanford. Modelarea s-a bazat pe atribuirea aceluiași PIT pentru 2 instrucțiuni dependente RAW.

După cum era de așteptat, instrucțiunile combinate generează o creștere semnificativă a performanței, mai precis cu 60% față de modelul precedent, obținându-se o medie armonică de 31.27 instr./tact. Consider acest câștig ca fiind suficient de ridicat încât ideea instrucțiunilor combinate, implementabilă atât prin scheduler static cât și prin hardware, să "prindă teren" în viitor.

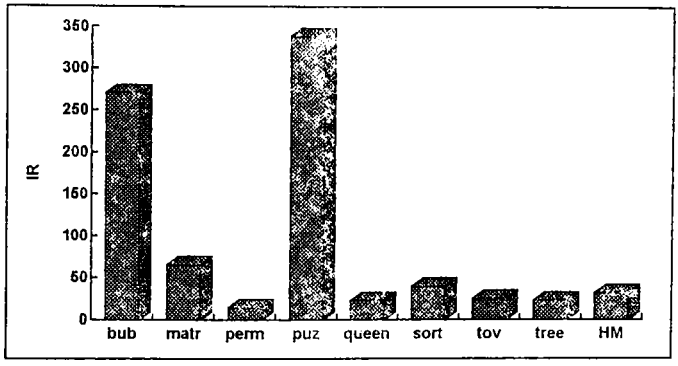


Figura 7.14

În continuare, ne propunem se vor determina pierderile cantitative de performanță pe care diversele limitări artificiale le implică și să analizăm aceste rezultate.

7.2.2.3. ÎNCĂ O LIMITARE: LATENȚA UNOR INSTRUCȚIUNI

Se va determina deci creșterea de performanță în ipoteza că toate instrucțiunile mașină ar avea latența de un tact. Cu alte cuvinte, voi încerca să răspund la următoarea

întrebare: vor aduce viitoarele progrese în eficiența algoritmilor de înmulțire și împărțire progrese semnificative?

Răspunsul cantitativ la această întrebare este dat de următoarea figură, obținută pe baza metodei de analiză și a simulatorului implementat în acest scop.

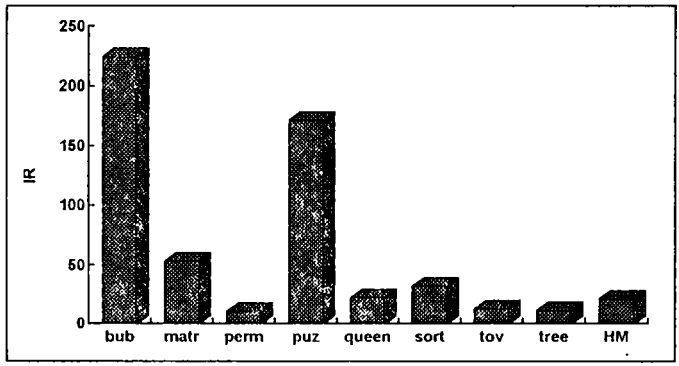


Figura 7.15

S-a obținut o medie armonică de 20.57 instr./tact, deci constatăm o creștere cu doar 5.7% față de modelul de bază, ceea ce arată clar că reducerea latenței instrucțiunilor DIV și MUL nu va putea aduce creșteri spectaculoase de performanță în programele de uz general.

7.2.2.4. LIMITĂRI DATORATE BUCLELOR DE PROGRAM

Se va încerca să se determine cantitativ, degradarea ratei de procesare ideală atunci când există simultan două limitări, specifice schedulerelor actuale: instrucțiunile dintr-o buclă nu se vor lansa în execuție până când toate instrucțiunile anterioare nu se vor fi terminat și respectiv instrucțiunile care urmează unei bucle nu se vor lansa în execuție până când toate instrucțiunile din buclă nu s-au lansat deja în execuție.

Cu această restricție am obținut următoarele rezultate:

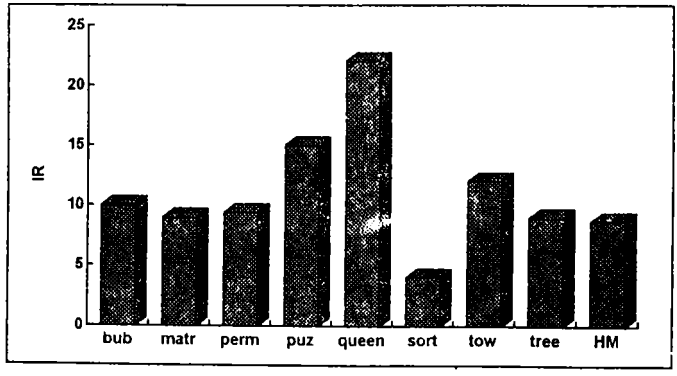


Figura 7.16

Rezultă deci IR = 8.72 instr./tact, față de idealul IR = 19.45 instr./tact, adică o degradare a performanței cu 123%, ceea ce reprezintă enorm. Iată de ce se impun în mod absolut necesar noi tehnici de paralelizare ale buclelor de program, întrucât acestea reprezintă o limitare majoră în calea obținerii unor performanțe superioare în schedulingul global.

În continuare se va determina degradarea de performanță introdusă de fiecare componentă: limitarea la intrarea în buclă și respectiv la ieșirea din buclă pentru a analiza contribuția fiecăreia în parte la degradarea ratei de procesare.

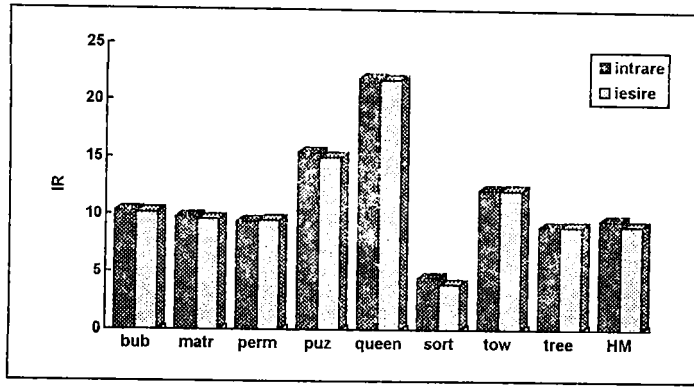


Figura 7.17

Se constată că ratele medii armonice sunt cvasiegale în ambele cazuri, adică 9.61 respectiv 9.19 instr./tact, ceea ce arată că ambele restricții sunt practic la fel de importante.

O altă limitare ar consta în forțarea execuției seriale a tuturor iterațiilor unei bucle, așadar ignorarea paralelizărilor în interiorul buclei prin tehnici de tip LU și TS. Rezultatele obținute în acest caz se vor prezenta în continuare (Fig.7.18).

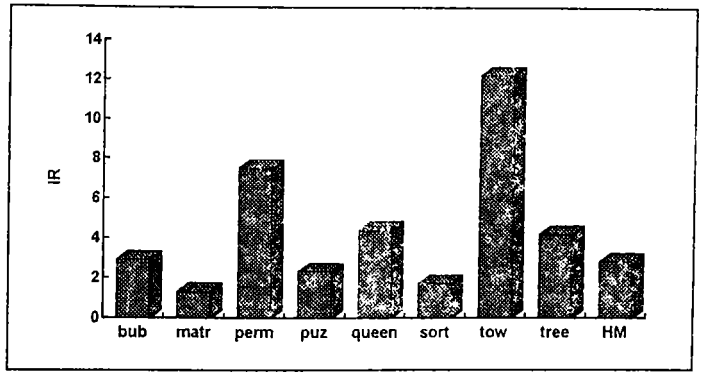


Figura 7.18

Scăderea de performanță devine acum realmente dramatică întrucât de la o rată ideală de 19.45 instr./tact s-a ajuns la una de 2.84 instr./tact, ceea ce înseamnă o

degradare a performanței cu 584%! Această degradare se datorează exclusiv serializării execuției buclelor.

7.2.2.5. LIMITAREA TOTALĂ A DEZAMBIGUIZĂRII REFERINȚELOR LA MEMORIE. EVALUĂRI CANTITATIVE

Se dorește aici, să se determine degradarea de performanță relativă la modelul de bază, pe care o implică o execuție In Order a instrucțiunilor de tip LOAD și STORE. Aceasta este o caracteristică a majorității schedulerelor actuale.

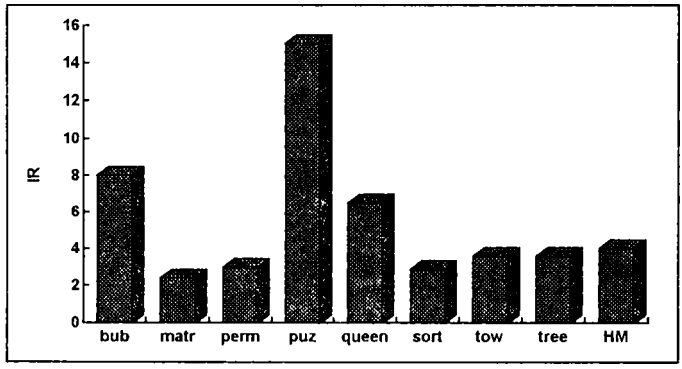


Figura 7.19

Cu alte cuvinte, am modelat o procesare a instrucțiunilor, fără nici un mecanism de dezambiguizare a referințelor la memorie, deci cu forțarea execuției In Order a instrucțiunilor LOAD respectiv STORE. Menționez că majoritatea schedulerelor actuale nu au implementate mecanisme de dezambiguizare.

Din nou se observă o scădere de performanță extrem de ridicată, de la 19.45 la doar 4.00 instr./tact, deci o reducere a performanței cu 380%. Se impun deci clar metode mai puternice de dezambiguizare a referințelor la memorie comparativ cu cele deja existente.

7.2.2.6. MACROINSTRUCȚIUNI SAU PROCEDURI?

Se știe că procedurile implică salvări/restaurări laborioase de contexte și totodată sunt mari consumatoare de timp. "In lining"-ul acestora ar mări semnificativ lungimea codului, dar ar micșora timpul de execuție. În cele ce urmează prezentăm rezultatele obținute pentru un in- lining perfect al tuturor procedurilor. Acest model ignoră toate instrucțiunile de salvare/restaurare asociate procedurilor existente în programele benchmark utilizate.

Rezultatele obținute prin simularea acestei idei, sunt prezentate în figura 7.20.

Se constată o creștere a gradului de paralelism de la 19.45 la 28.24 instr./tact, adică cu 45% mai mult, ceea ce este semnificativ. Concluzia autorului este că trebuiesc găsite aici soluții de compromis de tip in lining selectiv, pe baze euristice, întrucât se pare că beneficiile asupra performanței ar putea să fie majore. O euristică relativă la această selecție ar trebui să țină cont în opinia mea de lungimea procedurii și de cât de des este ea

apelată. De asemenea, în acelaşi sens, este important dacă procedura respectivă mai apelează la rândul ei alte proceduri.

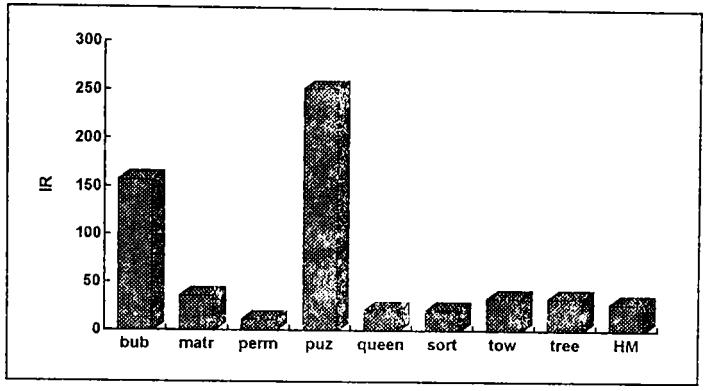


Figura 7.20

În concluzie, în concordanţă cu aceste investigaţii, performanţa arhitecturilor MEM poate fi serios îmbunătăţită în opinia autorului prin dezvoltarea următoarelor tehnici:

- tehnici agresive de **paralelizare a buclelor** de program ;
- tehnici de utilizare hard/soft a **instrucţiunilor combinate** în vederea eliminării dependenţelor reale de date. Tot aici considerăm ca deosebit de utilă o viitoare generalizare a acestor mecanisme de instrucţiuni combinate.
- tehnici de **in lining selectiv** al procedurilor
- noi **tehnici de dezambiguizare** a referinţelor la memorie

Mai multe detalii asupra metodologiei şi rezultatelor obţinute pot fi găsite în [PSVin97].

8. CONCLUZII

Această lucrare se încadrează în domeniul arhitecturilor cu paralelism la nivelul instrucțiunilor mașină (ILP).

Problema principală abordată pe parcursul lucrării a constat în investigarea pe bază teoretică și de simulare a performanței unor structuri ILP puternic parametrizabile și, în urma analizei și înțelegerii semnificației rezultatelor obținute, determinarea unor soluții optime de proiectare și funcționare a acestor arhitecturi.

Lucrarea conține 139 de figuri dintre care 68 de figuri cu rezultatele cercetărilor efectuate și respectiv 67 relații analitice, multe stabilite de către autor pentru prima dată.

În continuare, se vor prezenta pe capitole contribuțiile originale aduse în cadrul prezentei teze, cu specificarea paragrafului, acolo unde este cazul.

Capitolul 2

- o sinteză a modelelor de procesare de tip pipeline RISC, cu principalele lor caracteristici. Aici, s-a evidențiat arhitectura internă a acestor procesoare, principiile și tehnicile procesării pipeline a instrucțiunilor, problematica și clasificarea hazardurilor implicate prin procesarea pipeline, soluții hard/soft consacrate, etc;
- o sinteză a celor mai performante strategii hard/soft actuale de predicție a ramificațiilor în cadrul arhitecturilor pipeline scalare și superscalare.

Capitolul 3

- o sinteză a principiilor, tehnicilor și algoritmilor hard/soft de procesare paralelă în cadrul arhitecturilor MEM de tip superscalar și respectiv VLIW. S-a insistat pe analiza critică a acestor metode, așa cum se conturează ele pe baza cercetării bibliografice realizate.

Capitolul 4

- o sinteză a posibilelor avantaje ale arhitecturilor cache unificate, implementate în cadrul procesoarelor MEM ;
- o metodă analitică de estimare comparativă a arhitecturilor cu busuri separate, respectiv unificate pe spațiile de instrucțiuni și date, bazată pe conceptul de vector de coliziune. Metoda permite evaluări cantitative rapide asupra performanței acestor două arhitecturi de procesor superscalar și scutește simulări software laborioase (4.1.1);
- un program care permite aplicarea metodei precedente și care generează indicatori de performanță utili în urma procesării unor benchmark-uri, în acord cu principiile metodei originale dezvoltate;
- o metodă analitică de estimare a influenței defavorabile a instrucțiunilor LOAD asupra performanței arhitecturilor scalare pipeline cu busuri unificate respectiv separate pe instrucțiuni și date. Metoda se bazează pe același concept al vectorilor de coliziune (4.1.2);
- un simulator în vederea aplicării metodei anterioare asupra unor benchmark-uri speciale. Programul pune în evidență în mod cantitativ degradarea de performanță cauzată de LOAD- uri pentru diferiți parametri arhitecturali, în acord cu principiul metodei;

- o metodă analitică bazată pe conceptul de automat finit de stare, pentru abordarea arhitecturilor MEM cu busuri unificate respectiv separate. Se presupune o predicție perfectă a instrucțiunilor de ramificație. În final, metoda se reduce la rezolvarea unor sisteme laborioase de ecuații liniare și omogene, caracterizate și de o metrică de normalizare. Se furnizează parametrii optimali de proiectare precum FR (*Fetch Rate*), IR (*Issue Rate*), IBS (*Instruction Buffer Size*). Se prezintă rezultate cantitative obținute prin aplicarea acestei metode (4.2.1);

- o metodă analitică, ce constituie de fapt o generalizare a metodei precedente și care în plus ia în considerare într-un mod realist problematica predicției hardware a ramificațiilor în cadrul arhitecturii (4.2.2), de analiză și evaluare a arhitecturilor MEM. Se stabilește în premieră o relație analitică a performanței arhitecturilor superscalare de tip Harvard cu predicție hardware încorporată (relația 4.19). De asemenea se prezintă evaluări cantitative detaliate realizate pe baza acestei metode precum și concluzii relativ la o proiectare optimală a arhitecturilor superscalare Harvard respectiv unificate;

- un model analitic de evaluare globală a performanței cache-urilor separate/unificate într-o arhitectură superscalară. Modelul înglobează fenomene precum rate de miss, procese de prefetch, procese de coliziune, etc. (4.3). În final, se arată cum se pot obține anumiți parametri optimi de proiectare prin aplicarea metodologiei propuse. Modelul subliniază, practic pentru prima dată, atractivitatea arhitecturilor cache unificate din punct de vedere al performanței potențiale globale pe baza unei metrici globale de performanță stabilită;

- câteva considerații analitice privind alegerea unor parametri funcționali optimi pentru o arhitectură de tip MEM (4.5). Relațiile obținute pot fi deosebit de utile în faza de proiectare în care încă nu sunt disponibile instrumente complexe de simulare, dar în care trebuie totuși precizate în linii mari caracteristicile arhitecturii.

Capitolul 5

- un simulator al unei arhitecturi superscalare puternic parametrizabile în vederea optimizării arhitecturii cache. Procesorul superscalar poate utiliza o arhitectură cache Harvard sau unificată. Simulatorul lucrează pe principiul "trace driven simulation" și generează în premieră, parametrii optimali de funcționare pentru ambele variante. Există o coincidență perfectă între rezultatele generate prin simulare și cele obținute teoretic în capitolul 4. Simularea însă generează rezultate mai realiste și mai ample. Se dă astfel pentru prima dată un răspuns complet (analitic + simulare) la problematica arhitecturilor cache unificate și respectiv la anumite probleme deschise legate de arhitecturile superscalare Harvard.

Capitolul 6

- un model analitic general de analiză a schemelor de predicție a ramificațiilor de tip "Branch Target Buffer". Se pornește de la justificarea necesității unor asemenea modele și de la un set de modele - din păcate incomplete și eronate - elaborate de *H. Cragon* (6.1). În final se obține pentru prima dată pe plan analitic, o metrică de performanță globală și originală(relația 6.8), care prin particularizări realiste generează concluzii deosebit de utile;

- un simulator de predictor hardware de tip corelat pe două nivele (6.2, 6.3). Simulatorul generează soluția constructivă optimală pentru orice schemă corelată de predictor. De asemenea rezultă concluzii cantitative și calitative asupra compromisului optim ce trebuie realizat în cadrul unor asemenea scheme.

- un scheduler/ simulator destinat optimizării execuției basic-block-urilor într-o arhitectură superscalară virtuală și parametrizabilă. Se prezintă performanțele obținute prin aplicarea unui algoritm de optimizare de tip "List Scheduling". Prin diverse parametrizări ale resurselor arhitecturii, rezultă compromisul optim din punct de vedere al structurii hardware a procesorului (7.1);

- o cercetare bazată pe simulare a potențialului teoretic maxim care ar putea fi atins prin scheduling static. Se investighează apoi în mod cantitativ, influența diferitelor bariere artificiale care apar în calea atingerii acestei performanțe ideale. De asemenea se propun și se evaluează soluții noi de îmbunătățire a performanței schedulingului (7.2). Pe această bază, se concluzionează principalele tehnici de scheduling static care trebuie îmbunătățite în viitor.

Concluziile acestei lucrări sugerează sugerează în esență, necesitatea unor analize teoretice mai profunde, mai generale și mai sistematizate în acest domeniu, dublate și verificate prin simulări laborioase. Această dualitate de investigare de tip "analiză + simulare", a reprezentat esența acestei cercetări. Cred că acesta este un fapt pozitiv în abordarea domeniului ILP, aflat aproape exclusiv sub dominația unor investigații empirice, teoretic nemature, bazate practic exclusiv pe simulare și care nu oferă un cadru sistematizat și general de analiză. Asemenea simulări sunt inevitabil particulare și pot duce prin extrapolarea rezultatelor la concluzii greșite dacă sunt utilizate în mod exclusiv și dacă nu sunt dublate de o bază analitică relativ riguroasă. Desigur că și aspectul reciproc este probabil adevărat.

Cercetările abordate în această teză ar putea fi continuate în scopul rezolvării altor probleme deschise, interesante și conexe cu cele prezentate aici, pe care domeniul ILP le pune. Astfel de exemplu, cadrul general construit în capitolele 4 și 5, ar putea fi aplicat întocmai în analiza compromisurilor optimale legate de numărul seturilor de regiștrii generali, unități funcționale, busuri interne, etc. De asemenea, cuantificarea efectelor pe care algoritmii de tip Tomasulo îi au asupra reducerii presiunii la setul general de regiștri poate constitui un obiectiv la care un răspuns definitiv îl va da simularea. Aceeași problemă ar putea fi analizată din punct de vedere teoretic pe baza unei metodologii derivată din cea prezentată în paragraful 4.5.

În același sens dezvoltarea simulatorului din capitolul 5 pentru memorii cache având diferite grade de asociativitate ar putea constitui de asemenea o provocare interesantă.

De menționat că pe parcursul elaborării prezentei lucrări, autorul a publicat câteva articole conexe, în reviste de specialitate din țară și străinătate, precum și o monografie relativă la problematica ILP.

În final, pe baza celor rezumate mai sus, doresc să remarc aria relativ largă a contribuțiilor acestei teze în cadrul domeniului ILP, întrucât s-au abordat de la probleme legate de structuri hardware optimale până la probleme legate de optimizarea programelor aferente acestor arhitecturi. Consider că și metodologiile dezvoltate și folosite în investigare sunt relativ diverse și flexibile.

BIBLIOGRAFIE

1.[Abno94] Abnous A., Bagherzadeh N.- Pipelining and Bypassing in a VLIW Processor, IEEE Trans. Parallel and Distributed Systems, No 6, 1994

2.[Aco86] Acosta R., Kjelstrup J., Torng H.- An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors, IEEE Trans. on Computers, No 9, 1986

3.[Alt96] Altman E., Guang G.- Optimal Software Pipelining Through Enumeration of Schedules, EuroPar Conf., Lyon,Aug., 1996

4.[Aik88] Aiken A., Nicolau Al.- A Development Environment for Horizontal Microcode, IEEE Trans. on Software Eng., No 5,1988

5.[AMD89] Advanced Micro Device- 29K Family Data Book, AMD, 1989

6.[And93] Anderson D., Shanley T.- Pentium Processor System Architecture, Richardson TX: Mindshare Press, 1993

7.[Bal95] Bala V., Rubin N.- Efficient Instruction Scheduling Using Finite State Automata, Proceedings of MICRO 28, IEEE, 1995

8.[Beck93] Becker M.,s.a.- The Power Pc 601 Microprocessor, IEEE Micro, October, 1993

9.[Bha96] Bhandarkar D.- Alpha, Implementations and Architecture, Digital Equipment Co Press, 1996

10.[Bre94] Breach S., Vijaykumar T., Sohi G.- The Anatomy of the Register File in a Multiscalar Processor, MICRO'27, No 11, 1994

11.[But91] Butler M., s.a.- Single Instruction Stream is Greater than Two, Proc. 18th Ann. Int'l Symp. on Computer Architecture, 1991

12.[Cha 94] Chang P.Y., s.a.- Branch Classification: A New Mechanism for Improving Branch Predictor Performance, MICRO-27 Conf., San Jose, CA, Nov., 1994

13.[Cha95] Chang P.P., s.a.- The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors, IEEE Trans. on Comp., No.3, 1995

14.[Chang95] Chang P.P., s.a.- Three Architectural Models for Compiler Controlled Speculative Execution, No 4, 1995

15.[ChaM96] Chang M., Lai F.- Efficient Exploitation of ILP for Superscalar Processors by the Conjugate Register File Scheme, IEEE Trans. on Computers, No 3, 1996

16.[Che92] Chen T., Baer J. - Reducing Memory Latency Via Non - Blocking and Prefetching Caches, 5 th ASPLOS, 1992.

17.[CheC96] Chen C., King C.- Designing Dynamic Two- Level Branch Predictors Based on Pattern Locality, EuroPar Conf., Lyon, 1996

18.[Cho95] Chou H., Chung P.- An Optimal Instruction Scheduler for Superscalar Processor, IEEE Trans. on Par. and Distr. Syst., No.3, 1995

19.[Cir95] Circello J., s.a.- The Superscalar Architecture of the MC 68060, IEEE Micro, April, 1995

20.[Col93] Collins R.- Developing a Simulator For the Hatfield Superscalar Processor, Technical Report No 172, December,1993

21.[Col93b] Collins R.- Toward a Minimal Superscalar Implementation, Technical Report No UHCS-93-N1, Univ. of Herts, UK, 1993

22.[Col94] Collins R., Steven G.B.- An Explictly Declared Delayed Branch Mechanism for a Superscalar Architecture, Microprocessing and Microprogramming vol.40, 1994

23. [Col95] Collins R. - Exploiting Parallelism in a Superscalar Architecture, PhD thesis, University of Hertfordshire, U.K., 1996.

24.[Col96] Collins R., Steven G.- Instruction Scheduling for a Superscalar Architecture, Euromicro Conference, Prague, Sept., 1996

25.[Cor93] Corporaal H.- MOVE 32 INT Architecture and Programmer's Reference Manual, TR 1-68340-44, Delft University, Holland, 1993

26.[Cor95] Corporaal H., Hoogerbrugge J.- Code Generation for TTA, Kluwer Academic Publishers, 1995

27.[Corp93] Corporaal H., Arend P.- MOVE 32 INT a Sea of Gates Realization of a High Performance TTA, Microprocessing and Microprogramming, Sept., 1993

28.[Cra92] Cragon H.G.- Branch Strategy Taxonomy and Performance Models, IEEE Computer Society Press, 1992

29.[Cri96] Cristie D.- Developing the AMD- K5 Architecture, IEEE Micro, April, 1996

30.[DEC91] Digital Equipment Co- Digital's RISC, Architecture Technical Handbook, DEC, 1991

31.[Dit87] Ditzel D., McLellan H.- Branch Folding in the CRISP Microprocessor, Proc. 14th Ann. Int'l Symp. Computer Architecture, 1987

32.[Dub91] Dubey P., Flynn M.- Branch Strategies: Modeling and Optimization, IEEE Trans. on Comp., No 10, 1991

33.[Dub95] Dubey P., Adams G., Flynn M.- Evaluating Performance Tradeoffs Between Fine-Grained and Coarse-Grained Alternatives, IEEE Trans. on Parallel and Distributed Systems, No 1, 1995

34.[Dwy92] Dwyer H., Torng H.- An Out of Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts, Proc. 25th Ann. Symp. Microarchitecture, 1992

35.[Ebci88] Ebcioglu K.- Some Design Ideas for a VLIW Architecture, Parallel Processing, Elsevier Science Publishers, 1988

36.[Ebci94] Ebcioglu K., s.a.- VLIW Compilation Techniques in a Superscalar Environment, ACM SIGPLAN 94-6/94, Orlando, Florida, 1994

37.[Edm95] Edmonson J., s.a.- Superscalar Instruction Execution in the 21164 Alpha Microprocessor, IEEE Micro, 1995

38.[Eic92] Eickemeyer R., Vassiliadis S.- Interlock Collapsing ALU for Increased ILP, 25th Ann. Int'l Symp. on Microarchitecture, Dec., Portland Oregon,1992

39.[Els95] Elston C. s.a.- HADES: Towards the Design og an Aynchronous Superscalar Processor, Conference on Asynchronous Design Methodologies, May, London, 1995

40.[Far86] Farling S.M., Hennessy J.- Reducing the Cost of Branching, Proc. 13th Ann. Int'l Symp. Comp. Architecture, 1986

41.[Fag95] Fagin B., Mital A.- The Performance of Counter and Correlation Based Schemes for BTB's, IEEE Trans. on Comp., No 12, 1995

42.[Fil96] Filho E., Fernandes E., Wolfe A.- Functionality Distribution on a Superscalar Architecture, EuroPar Conf., Lyon, Aug., 1996

43.[Fis81] Fisher J.A.- Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. on Comp., No 7, 1981

44. [Fis91] Fisher J. A., Rau B. R. - Instruction Level Parallel Processing, Science, vol. 253, No 5025, 1991.

45.[Fly96] Flynn M.- Computer Architecture and Organization, Prentice-Hall, 1996

46.[Flyn96] Flynn M.- Parallel Processors were the Future and may yet be, in Computer, IEEE, 1996

47.[Fra 92] Franklin M., Sohi G.- The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism, Proceedings 19th Ann. Int. Conf. on Comp. Archit., New York, 1992

48.[Gon93] Gonzales A., Llaberia J.- Reducing Branch Delay to Zero in Pipelined Processors, IEEE Trans. on Computers, No 3, 1993

49.[Goo96] Goossens B., Vu T.D.- On-Chip Multiprocessing, EuroPar Conf., Lyon, 1996

50.[Gro90] Grohoski G.F.- Machine Organization of the IBM RISC System/6000 Processor, IBM J. Research and Development, No 4, 1992

51.[Has95] Hasegawa A., s.a.- SH3: High Code Density, Low Power, IEEE Micro, December, 1995

52.[Hea94] Heath S.- Power PC A Practical Companion, Butterworth-Heinemann, 1994

53.[Hen96] Hennessy J., Patterson D. - Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 2 - nd Edition, 1996.

54.[Hor90] Horst R., Harris R., Jardine R.- Multiple Instruction Issue in the Nonstop Cyclone Processor, Proc. 17th Ann. Int'l Symp. Computer Architecture, 1990

55.[HP93] Hewlett Packard Co- PA- RISC Architecture, 1994

56.[Hua 93] Hua K., Liu L., Peir J.- Designing High Performance Processors Using Real Address Prediction, IEEE Trans. on Computers, No 9, 1993

57.[Hua94] Huang A., s.a.- Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation, Int. Symposium of Computer Architecture, Chicago, April, 1994

58.[Hwa84] Hwang K., Briggs F.- Computer Architecture and Parallel Processing, Mc-Graw Hill, New York, 1994

59.[Hwa93] Hwang K.- Advanced Computer Architecture. Parallelism, Scalability, Programability., McGraw-Hill, New-York, 1993

60.[Hwu89] Hwu W.,Conte T., Chang P.- Comparing Software and Hardware Schemes for Reducing the Cost of Branches, Proc 16th Ann. Int'l Symp. computer Architecture, 1989

61.[Hwu92] Hwu W., Chang P.- Efficient Instruction Sequencing with Inline Target Insertion, IEEE Trans. on Computers, No 12, 1992

62.[Hwu93] Hwu W.- The Superblock: An Effective Technique for VLIW and Superscalar Compilation, Journal of Supercomputing, Vol.7, 1993

63.[IBM93] IBM Microelectronics- Power PC Advance Information, Technical Report, Rev.1, 1993

64.[IEE90] IEEE- Microprocessors and Microsystems, Special Issue: Applying and Implementing RISC, Vol.14, No.6, 1990

65.[INM90] INMOS- The Transputer Data Book, Second Edition, Inmos co, 1990

66.[Joh91] Johnson M. - Superscalar Microprocessor Design, Prentice Hall, 1991.

67.[Jon91] Jones R., Allan V.- Software Pipelining: An Evaluation of Enhanced Pipelining, ACM Conf., Nov., 1991

68.[Jou94] Jourdan S., s.a.- A High Out of Order Symetric Superpipeline Superscalar Microprocessor, Euromicro Conference , Liverpool, Sept., 1994

69.[Joup89] Joupi N., Wall D.- Available Instruction Level Parallelism for Superscalar and Superpipelined Machines, 3rd Int'l Conf. ASPLOS, Port Oregon, 1989

70.[Kat92] Kato T., Ono T., Bagherzadeh N.- Performance Analysis and Design Methodology for a Scalable Superscalar Architecture, 25th Ann. Int'l Symp. on Microarchitecture, Portland, Oregon, USA, Dec., 1992

71.[Kea95] Kearney D., Bergmann N.- Performance Evaluation of Asynchronous Logic Pipelines with Data Dependent Processing Delays, Conference on Asynchronous Design Methodologies, May, London, 1995

72.[Kug91] Kuga M., Murakami K., Tomita S.- DSNS: Yet Another Superscalar Processor Architecture, Computer Architecture News, June, 1991

73.[Kuri94] Kurian L., Hulina P., Coraor L.- Memory Latency Effects in Decoupled Architectures, IEEE Trans. on Comp., No 10, 1994

74.[Lam92] Lam M., Wilson R.- Limits of Control Flow on Parallelism, Proc. 19th Ann. Int'l Symp. Computer Architecture, 1992

75.[Lee84] Lee J., Smith A.J.- Branch Prediction Strategies and Branch Target Buffer Design, Computer, No 1, 1984

76.[LeeR91] Lee R., Kwok A., Briggs F.- The Floating Point Performance of the Superscalar SPARC Processor, ASPLOS Conf., April, 1991

77.[Lo96] Lo R., s.a.- Aggregate Operation Movement: A Min-Cut Approach to Global Code Motion, EuroPar Conf., Lyon, 1996

78.[Lip92] Liptay J.S.- Design of the IBM Enterprise System/9000 High End Processor, IBM J. Research and Development, No 4, 1992

79.[Mal92] Mahlke S., s.a.- Effective Compiler Support for Predicated Execution Using the Hyperblock, Proc. 25th Ann. Int'l Symp. Microarchitecture, 1992

80.[McCra94] McCrackin D.- Practical Delay Enforced Multistream Control of Deeply Pipelined Processors, IEEE Trans. on Computers, No 3, 1994

81.[McG] McGeady S.- The i960CA Superscalar Implementation of the 80960 Architecture, Digest of Papers Spring Compcon, Feb.1990

82.[Mil89] Milutinovic V. (editor) - High Level Language Computer Architecture, Computer Science Press, 1989.

83.[Mil87] Milutinovic V., Gimarc C.- A Survey of RISC Processors and Computers of the Mid- 1980's, Computer, Sept., 1987

84.[Milu87] Milutinovic V., s.a.- Architecture/Compiler Synergism in GaAs Computer Systems, Computer, May, 1987

85.[Mon90] Montoye R., s.a. - Design of the IBM RISC System / 6000 Floating Point Execution Unit, IBM J. Research and Development, Vol. 34., No. 1, 1990.

86.[Moo92] Moon S., Ebcioglu K.- An Efficient Resource Constrained Global Scheduling Technique for Superscalar and VLIW Processors, MICRO' 25 Conf., Port Oregon, December, 1992

87.[Mot91] MOTOROLA Co- MC 88100 RISC Superscalar Microprocessor, User's Manual, Second Edition, Prentice- Hall, 1991

88. [Na93] Nakatani T, Ebcioglu K. - Making Compaction Based Parallelisation Affordable, IEEE Tr.On Parallel and Distr. Systems, No.9, 1993.

89.[Nad95] Nadehara K., s.a.- Low Power Multimedia RISC, IEEE Micro, December, 1995

90.[Nai95] Nair R.- Optimal 2-Bit Branch Predictors, IEEE Trans. on Comp., No 5, 1995

91.[Nee96] Neefs H., s.a.- An Analytical Model for Performance Estimation of Modern Data-Flow Style Scheduling Microprocessors, Euromicro Conference, Prague, 1996

92.[NeeH96] Neefs H.,s.a.- Microarchitectural Issues of a Fixed Length Block Structured Instruction Set Architectute, Euromicro Conference, Prague, 1996

93.[Nic89] Nicolau A.- Run Time Disambiguation: Coping with Statically Unpredictable Dependencies, IEEE Trans. on Comp., No 5, 1989

94.[Pap96] Papworth D.-Tuning The Pentium Pro Microarchitecture, IEEE Micro, April, 1996

95.[Par96] Park G.H., s.a.-Prefetching, IEE Proc. Comput. Digit. Tech., Vol.143, No 1, 1996

96.[Pat82] Patterson D., Sequin C.- A VLSI RISC, Computer, September, 1982

97.[Pat94] Patterson D., Hennessy J. - Computer Organization and Design, The Hardware/ Software Interface, Morgan Kaufmann, 1994

98.[Patt85] Patt Y., Hwu W., Shebanow M.- HPS a New Microarchitecture, Proceedings of the 18th Annual Workshop on Microprogramming, 1985

99.[Per93] Perleberg C., Smith A. J. - Branch Target Buffer Design and Optimization, IEEE Trans. Computers, No. 4, 1993.

100.[Pne94] Pnevmatikatos D., Sohi G.- Guarded Execution and Branch Prediction in Dynamic ILP Processors, IEEE Conf., Chicago, April, 1994

101.[Pop92] Popa M.- Introducere in arhitecturi paralele si neconventionale, Ed. Timisoara, 1992

102.[Pot96] Potter R.- Instruction Scheduling: The Way Forward, Technical Report, Univ. of Herts, UK, 1996

103.[Pot97] Potter R., Steven G.B., **Vintan L.** - Investigating the Limits of Fine Grained Parallelism in a Statically Scheduled Superscalar Architecture, in revista Theoretical Computer Science, Lyon, France, No 8, 1997 (acceptata, in curs de publicare)

104.[Pou93] Pountain D.- Pentium: More RISC than CISC, Byte, Sept., 1993

105.[Rim96] Rim M., Jain R.- Valid Transformation: A New Class of Loop Transformations, IEEE Trans. on Parallel and Distributed Systems, No 4, 1996

106.[Rutt96] Ruttenberg J., s.a.- Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler, ACM PLDI No 5, 1996

107.[Rya93] Ryan B.- RISC Drives Power PC, in Byte, August, 1993

108.[San96] Sanchez F., Cortadella J.- RESIS: A New Methodology for Register Optimization in Software Pipelining, EuroPar Conf., Lyon, 1996

109.[Sch96] Schmidt S.- Global Instruction Scheduling- A Practical Approach, EuroPar Conf., Lyon, 1996

110.[Sia96] Siamak A., Sachs H., Duvvuru S.- An Architecture for High Instruction Level Parallelism, Technical Report, Sun Micro Inc., Mountain view, CA, 1996

111.[Sig96] Sigmund U., Ungerer T.- Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor, EuroPar Conf., Lyon, Aug., 1996

112.[Smi88] Smith J.E., Pleszkun A.R.- Implementing Precise Interrupts in Pipelined Processors, IEEE Trans. Computers, No 5, 1988

113.[Smi89] Smith M.D., Johnson M., Horowitz M.- Limits on the Multiple Istruction Issue , Int'l Conf. ASPLOS, Port Oregon, 1989

114. [Smi94] Smith J.E., Weiss S. - Power - PC 601 and Alpha 21064: A Tale of Two RISCs, Computer, June, 1994

115. [Smi95] Smith J., Sohi G. - The Microarchitecture of Superscalar Processors, Technical Report Madison University, USA, August, 1995.

116.[Soh90] Sohi G.S.- Instruction Issue Logic for High- Performance, Interruptible, Multiple Functional Unit Pipelined Computers, IEEE Trans. on Computers, No 3, 1990

117.[Spra95] Sprangle E., Patt Y.- Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme, MICRO'27 Conf, ACM, 1995

118.[Spro92] Sproull R., s.a.- Counterflow Pipeline Processor Architecture, Technical Report, Sun Micro Inc., Mountain View, CA, 1992

119.[Sta96] Stallings W.- Computer Organization and Architecture, Fourth Edition, Prentice-Hall, 1996

120.[Ste91] Steven G.- A Novel Effective Address Calculation Mechanism for RISC Microprocessors, ACM SIGARCH, No 4, 1991

121.[Ste92] Steven G.B. s.a.- i HARP: A Multiple Instruction Issue Processor, IEE Proceedings, Vol.139, No.5, 1992

122.[SteF93] Steven F.L., s.a.- An Evaluation of the Architecture Features of the i HARP Processor, Univ. of Hertfordshire, UK, Technical Report No. 170, 1993

123[SteF95] Steven F.L., s.a.- Using A Resource Limited Instruction Scheduler to Evaluate the i HARP Processor, IEE Proceedings, Vol.142, No.1, 1995

124. [Ste96] Steven G. B., s.a. - A Superscalar Architecture to Exploit ILP, Euromicro Conference, 2-5 september, Prague, 1996.

125.[SteVi 96] Steven G. B., **Vintan L.** - Modelling Superscalar Pipelines with Finite State Machines, Microprocessing and Microprogramming, No. 12, IEEE Computer Society Press, 1996.

126.[Sto93] Stone H. - High Performance Computer Architecture, Addison Wesley, 1993.

127.[SUN92] Sun Co- The Super SPARC Microprocessor, Sun Microsystems Inc, 1992

128.[Sut89] Sutherland I.- Micropipelines, Communications of the ACM, Vol 32, No 6, 1989

129.[Tem96] Temam O.- Streaming Prefetch, EuroPar Conf., Lyon, Aug., 1996

130.[Thom94] Thompson T., Ryan B.- Power PC 620 Soars, Byte, August, 1993

131.[Tom67] Tomasulo R.- An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal, Vol.11, 1967

132.[Tre96] Tremblay M., O'Connor M.- Ultra Sparc I: A Four Issue Processor Supporting Multimedia, IEEE Micro, April, 1996

133.[Tung96] Tung B., Kleinrock L.- Using Finite State Automata to produce Self Optimization and Self Control, IEEE Trans. on Parallel and Distributed Systems, No 4, 1996

134.[Tys94] Tyson G.S.- The Effects of Predicatef Execution on Branch Prediction, MICRO-27, San Jose, CA, Nov., 1994

135.[Uht91] Uht A.K.- A Theory of Reduced and Minimal Procedural Dependencies, IEEE Trans. Computers, No 6, 1991

136.[Vas93] Vassiliadis S., Phillips J., Blaner B. - Interlock Collapsing ALUs, IEEE Trans. Computers, No. 7, 1993.

137.[Vin95] Vintan L.- Sisteme SIMD si MIMD: perspectiva programatorului, in rev. BYTE (ed. romana), nr.2, 1995

138.[VinL95] Vintan L.- About Two Superscalar Processor Models: Performance Evaluations, Acta Universitatis Cibiniensis, seria Calculatoare, Ed. Univ., Sibiu, 1995

139.[Vin96] Vintan L. - Exploatarea Paralelismului in Microprocesoarele Avansate, Ed. Universitatii Sibiu, ISBN 973-9280-00-5, 1996.

140.[VinL96] Vintan L. - About Some Non-Harvard Processor Models, Bul. St. al Fac. de Calculatoare, Tom 41 (55), Timisoara, 1996.

141.[Vint96] Vintan L.- Evaluari de performanta in cadrul arhitecturilor cu paralelism la nivelul instructiunilor, Referat de doctorat nr.2, Fac. de Calculatoare, Timisoara, 1996

142.[Vin97] Vintan L.- A Global Approach For Two Cache Architectures, Acta Universitatis Cibiniensis, seria Calculatoare, Vol. 25, Ed. Univ. Sibiu, 1997

143.[Wal91] Wall D.- Limits of Instruction Level Parallelism, ASPLOS Conf., 1991

144.[Wall96] Wallace S., Bagherzadeh N.- Instruction Fetching Mechanisms for Superscalar Microprocessors, EuroPar Conf., Lyon, 1996

145.[Wang93] Wang C., Emnett F.- Implementing Precise Interruptions in Pipelined RISC Processors, IEEE Micro, August, 1993

146.[War90] Warren H.S.- Instruction Scheduling for the IBM RISC System/6000 Processor, IBM J. Research and Development, No 1, 1990

147.[Weis93] Weiss S.- Optimizing a Superscalar Machine to Run Vector Code, IEEE Parallel and Distributed Technology, May, 1993

148.[Weis94] Weiss S., Smith J.- POWER and Power PC, San Francisco: Morgan Kaufmann, 1994

149.[Yea96] Yeager K.- The MIPS R 10000 Superscalar Microprocessor, IEEE Micro, April, 1996

150.[Yeh92] Yeh T., Patt Y. - Alternative Implementations of Two Level Adaptive Branch Prediction, 19 th Ann. Int.'L Symp. Computer Architecture, 1992.