CONTRIBUŢII LA ÎMBUNĂTĂŢIREA PERFORMANŢEI SISTEMELOR CU MICROPROCESOR

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea „Politehnica” din Timişoara
în domeniul ŞTIINŢA CALCULATOARELOR
de către

ing. Răzvan Traian Chevereşan

Conducător ştiintific: Prof.dr.ing. Ştefan Holban
Referenţi ştiinţifici: Prof.dr.ing.mat. Dumitru Dan Burdescu
Prof.dr. Alexandru Cicortaş
Prof.dr.ing. Mircea Popa

Ziua susţinerii tezei: 5.01.2010

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica" din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Cuvânt înainte

Teza de doctorat a fost elaborata pe parcursul activităţii mele în cadrul societăţii Sun Microsystems, SUA.

Lucrarea se adresează tuturor celor interesaţi în problemele legate de performanţa sistemelor computaţionale sub diferitele aspecte ale acestei. În condiţiile în care solicitările utilizatorilor devin din ce în ce mai ridicate, îmbunătăţirea performantei sistemelor de calcul presupune realizarea unor arhitecturi optimizate din punctul de vedre al alocării de resurse necesare, precum şi gestionarea, dimensionarea si managementul acestor resurse. Pentru realizarea acestui deziderat sunt importante: acurateţea mijloacele destinate evaluării performantei, tehnicile şi metodele utilizate, cunoaşterea aspectelor specifice legate de modul de rulare a diferitelor aplicatii, precum şi îmbunătăţirea mecanismelor hardware de mărire a vitezei de procesare. În teză sunt prezentate preocuparile, propunerile şi soluţiile autorului legate de acest subiect.

Consider că lucrarea constitue, pentru cei implicaţi în activitatea de evaluare a performanţei microprocesoarelor, un suport ştiinţific pentru cercetări viitoare legate de optimizarea factorilor cu implicaţie asupra performanţei, cât şi un suport practic prin soluţiile şi concluziile prezentate.

Doresc să mulţumesc domnului profesor dr.ing. Ştefan Holban pentru îndrumarea, coordonarea şi ajutorul pe care mi l-a acordat pe tot parcursul edificării acestei lucrări, domnului profesor dr.ing.Mikko Lipasti pentru iniţierea şi indrumarea în tainele arhitecturii microprocesoarelor, domnilor ing. Matthew Ramsay, ing. Tariq-Magdon Ismail, ing. Christopher Feucht pentru importanta contribuţie pe care au adus-o în primul rând la edificarea FAF iar în al doilea rând pentru contribuţia şi participarea în calitate de co-autori împreună cu dr.ing Ilya Sharapov, dr.ing. Robert Kroeger, dr.ing. Guy Delamarter la articolele publicate. De asemenea doresc să mulţumesc colectivului de cadre didactice de la Universitatea „Politehnica" din Timişoara care m-au format ca inginer cu menţiune specială domnului profesor dr.ing. Toma L. Dragomir.

Dedic această lucrare părinţilor mei pentru suportul moral şi ajutorul pe care mi l-au acordat pe întreg parcursul anilor de studiu în scopul finalizării prezentei lucrări.

Timişoara, ianuarie 2010 Răzvan Traian Chevereşan

Chevereşan, Răzvan Traian

Contribuţii la îmbunătăţirea performanţei sistemelor cu microprocesor

Teze de doctorat ale UTP, Seria 10, Nr. 26, Editura Politehnica, 2010, 202 pagini, 51 figuri, 10 tabele.

ISSN: 1842-7707

ISBN: 978-606-554-043-9

Cuvinte cheie:
Microprocesor, Analiza performanţei, Utilitare de analiză, Caracterizarea aplicaţiilor, Îmbunătăţire viteză de procesare, Sisteme cu microprocesor.

Rezumat:
Teza abordează problema performanţei sistemelor cu microprocesor prezentând soluţii în trei importante aspecte ale acesteia:

Îmbunătăţirea structurii şi a calităţii mijloacelor prin care se face evaluarea performanţei; prezentând un nou concept pentru realizarea unui utilitar de analiză prin simulare a microprocesoarelor şi a aplicaţiilor *software* denumit "Flexible Analysis Framework" (FAF), care decurge ca o consecinţă a faptului că analiza făcută asupra utilitarelor folosite în mod curent relevă faptul că acestea au o utilizare limitată.

Cercetarea modului de comportare în rulare şi a cerinţelor impuse structurilor *hardware*, de aplicaţiile care implică un regim computaţional dificil; în acest sens este prezentat un studiu comparativ detaliat şi se face o evaluare complexă asupra comportamentului în rulare a unui set semnificativ de aplicaţii, ce acoperă spaţiul ştiinţific şi cel comercial, care din punct de vedere computaţional solicită intens componentele sistemului de calcul.

Conceperea unor mecanisme care să conducă la îmbunătăţirea vitezei de lucru a microprocesorului; legat de acesta se propune un mecanism complet nou, nespeculativ care poate servi îmbunătăţirii performanţei, sunt descrise două posibilităţi în care mecanismul poate fi implementat în *hardware,* într-un microprocesor *out-of-order*, iar celălalt într-un microprocesor CMT. Este prezentat un studiu asupra potenţialului şi a limitelor mecanismului propus în a genera informaţie parţială, despre regiştrii operanzi ai instrucţiunilor. Calităţile mecanismului sunt evidenţiate în urma unui studiu de caracterizare a soluţiei de implementare a acestuia ca o etapă a pipeline-ului unui microprocesor superscalar.

Cuprins

1. Introducere.

Industria IT a avut pe parcursul a numai 15-20 de ani o evoluţie spectaculoasă. Tehnologia actuală permite dublarea numărului de tranzistori ce pot fi plasaţi într-un circuit integrat la fiecare doi ani, confirmând ceea ce Moore unul din fondatorii firmei Intel, a postulat prin legea ce-i poartă numele („Moore's Law" [1.1]), cantitatea de memorie ce poate fi integrată în chip este din din ce în ce mai mai mare, viteza de lucru a memoriei şi viteza de operare a dispozitivelor I/O au crescut simţitor.

În acelaşi timp, alimentată fiind de capacitatea tot mai mare de procesare a platformelor IT actuale, cerinţele utilizatorilor de sisteme de calcul au crescut exponenţial. Rularea simultană a mai multor aplicaţii, sau a unor aplicaţii paralele a devenit o realitate a zilelor noastre. Aplicaţii cu mii sau milioane de *thread*-uri paralele au devenit uzuale atât în domeniile ştiinţifice cât şi în cele comerciale. În plus de aceasta tehnicile de virtualizare permit rularea mai multor sisteme de operare eterogene, în paralel pe aceeaşi platformă *hardware*. Aceste aplicaţii din ce în ce mai complexe, solicită la maxim sistemele computaţionale. Performanţa sistemelor IT actuale este rezultatul unui lung şir de studii, cercetări, proiecte, căutări de soluţii noi inovative, acumulări, şi îmbunătăţiri, atât în domeniul *hardware*-ului cât şi în domeniul *software*-ului. În Anexa 1 este prezentată grafic evoluţia în timp a microprocesoarelor [1.2], în funcţie de principalul element care a constituit suportul esenţial al dezvoltării lor, numărul de tranzistori conţinuţi în *chip*.

Proiectaţii microprocesoarelor actuale trebuie să răspundă la două principale provocări, pe de o parte satisfacerea cerinţelor tot mai ridicate ale utilizatorilor, iar pe de altă parte, gestionarea raţională, la un nivel de performanţă ridicat, a resurselor pe care tehnologia actuală le oferă.

Performanţa este etalonul ce vinde sistemele computaţionale şi din acest motiv constituie unul dintre domeniile de permanentă actualitate. Firmele producătoare alocă resurse importante atât umane cât şi materiale pentru a îmbunătăţi competitivitatea platformelor produse.

Imbunătăţirea performanţei presupune realizarea unor arhitecturi optimizate din punctul de vedere al resurselor necesare, al unităţilor funcţionale şi de control,

dimensionarea şi amplasarea în chip a acestora, al dotării cu mijloace pentru depăşirea evenimentelor cu implicaţii negative asupra performanţei, care apar în timpul procesării unei aplicaţii, precum şi managementul acestor resurse. Pe de altă parte presupune studiul amanunţit al modului de comportare în rulare a unor aplicaţii, care devin din ce în ce mai complexe, precum şi al cerinţelor acestor aplcaţii în relaţia cu sistemul pe care rulează. De asemenea, se pune un mare accent pe elaborarea unor tehnici de compilare ce urmăresc optimizarea acelor componente ale aplicaţiilor ce fac ca acestea să ruleze la un nivel scăzut de performanţă pe platformele *hardware* existente.

Lucrarea de faţă abordează acest domeniu de cercetare de permanentă actualitate în domeniul IT: prezentând preocupările autorului legate de îmbunătăţirea performanţei sistemelor computaţionale.

1.1. Obiectivele tezei.

În cadrul tezei sunt prezentate, atât la nivel teoretic cât şi aplicativ, rezultatele obţinute în trei direcţii de dezvoltare a conceptului de performanţă, care constitue şi obiectivele lucrării, după cum urmează:

a) **Îmbunătăţirea structurii şi a calităţii mijloacelor prin care se face evaluarea performanţei.**

Legat de acest obiectiv propun un nou concept pentru proiectarea unui utilitar de analiză prin simulare a microprocesoarelor şi a aplicaţiilor soft denumit **"Flexible Analysis Framework" (FAF)** [1.3], care în contextul utilitarelor utilizate curent în acest scop, se individualizează printr-o serie de atribute, calificate drept noutăţi.

b) **Cercetarea modului de comportare în rulare şi a cerinţelor impuse structurilor *hardware*, de aplicaţiile care implică un regim computaţional dificil.**

Referitor la acest subiect este prezentat un studiu comparativ asupra modului de comportare în rulare a unor aplicaţii care presupun un regim computaţional dificil, specific aplicaţiilor din domeniul ştiinţific şi comercial, în care sunt evaluate şi comparate caracteristicile ce au influenţă asupra performanţei, precum şi unele aspecte legate de dimensionarea diferitelor componente arhitecturale.

c) **Conceperea unor mecanisme care să conducă la îmbunătăţirea vitezei de de lucru a microprocesorului.**

În conexiune cu acestă idee propun un mecanism *hardware* nou, nespeculativ care să conducă la rezolvarea dependenţelor de date, şi a dependenţelor de control, ce apar în cazul procesării instrucţiunilor, denumit **"Implication Engine" (IE).**

1.2 Structura lucrării.

Teza este structurată în opt capitole, bibliografie şi anexe.

Capitolul 1. prezintă pe scurt câteva consideraţii generale legate de dezvoltarea microprocesoarelor în ultimele decenii. Sunt prezentate obiectivele ce vor fi abordate pe parcursul tezei, şi în final este prezentată structura tezei.

Capitolul 2. introduce noţiunea de performanţă şi face o prezentare succintă a principalilor indicatori de performanţă folosiţi pentru evaluarea sistemelor cu microprocesor. Este de asemenea prezentată o sinteză a celor mai importante mijloace şi tehnici utilizate pentru evaluarea ei. În acest context sunt descrise principalele tipuri de simulatoare folosite în proiectare şi se fac aprecieri critice asupra metodelor de simulare şi a diferitelor categorii de aplicaţii de testare.

Se face o trecere în revistă a paşilor parcurşi în procesul de evaluare a performanţei platformelor *hardware* noi, începând cu scrierea codului sursă al simulatorului de performanţă şi terminând cu analiza efectivă. Toate acestea au ca scop să definescă şi să reţină atenţia asupra unor noţiuni, unităţi de măsură şi aspecte legate de performanţa micropocesoarelor şi a mijloacelor prin care se face evaluarea ei.

Capitolul 3. tratează, în prima parte, unele apecte cu conotaţie critică asupra utilitarelor utilizate în activitatea de evaluare a performanţei, continuând cu prezentarea unui nou concept ce stă la baza realizării unui nou utilitar, realizat de un colectiv din care face parte şi autorul tezei. Noua structură *Flexible Analysis*

Framework (FAF) a fost proiectată pentru a rezolva unele dintre neajunsurile utilitarelor uzuale şi se caracterizează prin flexibilitate şi versatilitate, fiind în acelaşi timp uşor de folosit şi extins. Calităţile şi potenţialul structurii de analiză sunt scoase în evidenţă, prin exemple, în două ipostaze de utilizare.

- Utilizarea FAF în cadrul simulatorului de evaluare a performanţei, situaţie în care sunt prezentate o serie de date interesante legate de corelarea unor evenimente apărute în urma execuţiei instrucţiunilor, cu funcţiile cele mai frecvent executate dintr-un anumit set de programe de evaluare.
- Utizarea FAF pentru analiza fişierelor *trace,* fiind prezentate date despre modul în care instrucţiunile specifice diferitelor aplicaţii accesează diversele segmente de memorie.

Capitolul 4. se referă la unul dintre aspectele importante în proiectarea unor microprocesoare noi şi anume caracterizarea din punct de vedere computaţional a aplicaţiilor ce urmează să fie rulate de către acestea, în vederea identificării acelor caracteristici ce ar putea face ca ele să ruleze la un nivel redus de performanţă.

Este prezentată metodologia actuală de lucru, utilizată de producătorii de microprocesoare şi sisteme computaţionale, pentru atingerea acestui scop, fiind scoase în evidenţă infrastructura folosită şi o serie de etape parcurse în acest proces.

Astfel, sunt descrise metode actuale de generare, evaluare şi validare a fişierelor de *trace* şi se prezintă un studiu comparativ detaliat asupra comportamentului în rulare a unui set semnificativ de aplicaţii, ce acoperă spaţiul programelor de calcul matematic intensiv şi cel comercial, care din punct de vedere computaţional solicită intens componentele sistemului de calcul.

Sunt prezentate rezultate pentru o gamă variată de caracteristici de performanţă cum sunt: distribuţia instrucţiunilor, localizarea accesărilor memoriei, sensibilitatea aplicaţiilor la dimensiunile şi structura memoriei *cache*, etc. distribuţia intercomunicărilor de date şi impactul precitirii datelor asupra performanţei.

Capitolul 5 prezintă stadiul actual al tehnicilor de îmbunătăţire a microprocesoarelor superscalar ce implementează modul de execuţie *out-of-order*. În partea de început se descrie modul de procesare *out-of-order* ce exploatează ILP

(*Instruction Level Parallelism*) şi principalii factori ce limitează ILP. Se face o trecere în revistă a principalelor *task*-uri efectuate în fiecare etapă a *pipeline*-ului unui microprocesor superscalar clasic, precum şi a principalelor mecanisme utilizate de microprocesoarele *out-of-order* moderne pentru rezolvarea conflictelor de date şi control, ce reprezintă principalii factori limitatori ai performanţei procesarii *out-of-order*.

Se face o analiză cantitativă a beneficiilor rezultate în urma implementării unor structuri *hardware* consacrate, destinate îmbunătăţirii performanţei. În final sunt prezentate direcţii de ultimă oră folosite în proiectarea microprocesoarelor actuale în vederea creşterii performanţei.

Capitolul 6 se adresează celui de-al treilea aspect de îmbunătăţire a performanţei şi anume proiectarea unor arhitecturi optimizate în scopul depăşirii evenimentelor cu conotaţie negativă asupra vitezei de execuţie a aplicaţiilor *software*. Se propune şi se investighează posibilităţile şi limitările unei noi structuri *hardware* numită IE (*Implication Engine*) a cărei implementare poate contribui semnificativ la îmbunătăţirea vitezei de execuţie a microprocesorului. Sunt prezentate premisele care au stat la baza proiectării noului mecanism şi anume predicţia valorilor şi algoritmul PODEM de testare a circuitelor combinaţionale şi este descris pe larg modul în care IE acţionează asupra lanţului de instrucţiuni dependente, pentru a genera informaţie parţială despre regiştrii operanzi şi despre rezultatul fiecărei instrucţiuni. Sunt evidenţiate sursele de informaţie parţială, precum şi zonele unde informaţia obţinută în urma utilizării IE poate fi folosită în scopul creşterii performanţei. În cele din urmă sunt prezentate detalii despre două moduri posibile de implementare în *hardware* a IE, într-un microprocesor superscalar şi într-un microprocesor CMP

Capitolul 7 evaluează şi caracterizează mecanismul IE pentru a scoate în evidenţă potenţialul pe care acesta îl oferă în îmbunătăţirea performanţei microprocesoarelor. Este prezentat mediul de simulare folosit bazat pe suita de simulatoare SimpleScalar [5.19] precum şi etapele parcurse pentru implementarea şi integrarea IE în mediul de simulare. Datele de caracterizare a IE au fost obţinute în urma folosirii unei structuri de analiză gen FAF. Pentru a avea o imagine completă a beneficiilor implementării IE capacitatea acestuia a fost evaluată în două cazuri

diferite de implementare, când rezultatele instrucţiunilor sunt retransmise IE şi când acesta funcţionează de sine stătător, scoţându-se în evidenţă avantajele şi dezavantajele uneia sau celeilalte implementări.

Capitolul 8 prezintă concluzii, principalele contribuţii originale ale tezei şi dezvoltările ulterioare ale subiectelor abordate.

2. Predicţia şi evaluarea performanţei microprocesoarelor.

2.1. Definirea performanţei.

Performanţa sistemelor computaţionale se defineşte prin cantitatea de muncă utilă efectuată de un sistem computaţional, raportată la timpul şi resursele pe care le utilizează. Cu alte cuvinte conceptul de performanţă, în cazul echipamentelor numerice, are în vedere cel puţin unul dintre următorii indicatori: viteză, capacitate (volumul de muncă), fiabilitate şi cost. Nivelul global de performanţă al unui procesor este apreciat după modul în care proiectantul a reuşit optimizarea indicatorilor enunţaţi, pentru a obţine rezultatul cel mai bun.

2.1.1. Mărimi asociate performanţei. Indicatori de performanţă.

În decursul timpului, odată cu evoluţia sistemelor cu microprocesor viziunea asupra performanţei a suferit şi ea modificări, utilizâdu-se diferiţi indicatori pentru cuantificarea performanţei acestora.

Fiecărui procesor îi este propriu un ceas care determină cu precizie momentul în care se produc „evenimentele" ce au loc în *hardware*. Intervalele dintre două momente discrete de timp marcate de ceas sunt numite perioade de *clock*. În legătură cu acesta a fost definit ca un prim indicator de performanţă:

- **Frecvenţa de *clock*** măsurată în MHz, sau în GHz. În multe cazuri însă, numai frecvenţa de *clock* a procesorului, luată individual ca indicator de performanţă, nu reflectă în mod fidel performanţa efectivă a sistemului. Frecvenţa de *clock* este influenţată de către o serie de alţi factori, care ţin de microarhitectura internă a procesorului, cantitatea şi viteza de acces a

- memoriei, viteza de acces a datelor de pe disc, eficienţa sistemului de operare şi nu în ultimul rând de structura aplicaţiilor *software* ce sunt executate.

La ora actuală în legătură cu performanţa microprocesoarelor sunt acceptaţi doi indicatori globali:

- **Timp de execuţie**, (*execution time*) care mai apare sub denumirea de *elapsed time* şi care reprezintă timpul fizic necesar pentru execuţia unui *task* de la iniţializarea şi până la finalizarea lui. *Execution time* cuprinde timpul petrecut pentru execuţia programului utilizator, timpul petrecut în sistemul de operare ca urmare a execuţiei acestui program, precum şi timpul în care procesorul aşteaptă finalizarea operaţiunilor de I/O. (Pentru utilizatorul unui calculator personal, acesta este probabil indicatorul cel mai important).
- **Volumul de muncă**, *(throughput)* reprezintă volumul de date procesate, dintr-unul sau mai multe procese ce se desfăşoară concomitent, într-un interval de timp determinat. (Pentru utilizatorul unui server multiprocesor, desigur acesta va fi factorul hotărâtor).

Dacă luăm în considerare ca indicator de performanţă, numai timpul de execuţie, este evident faptul că performanţa va fi dată de inversul acestuia, adică:

$$Performanta = \frac{1}{Timp\ de\ executie} \qquad (1)$$

Putem spune deci că, măsura performanţei unui procesor este dependentă de timpul de execuţie al diverselor aplicaţii *software* şi are un nivel cu atât mai ridicat cu cât acest timp este mai mic.

În multe cazuri datorită faptului că resursele fizice ale procesorului sunt utilizate concomitent de mai multe programe, trebuie făcută distincţia între *execution time* şi timpul dedicat exclusiv executării unui anumit program. Pentru a face această distincţie a fost introdus indicatorul:

- **CPU_{time}** *(Central Processing Unit time)* timpul de execuţie al unităţii centrale, acesta este timpul efectiv dedicat execuţiei unui program şi nu include timpul petrecut în operaţiile de I/O, sau în execuţia altor programe. CPU_{time}, exprimat în secunde, este indicatorul cel mai utilizat în practică pentru evaluarea vitezei

de execuţie a unui procesor. Dependenţa acestei mărimi, de numărul total de perioade de *clock* necesare execuţiei unui program şi frecvenţa de *clock* a procesorului în cauză, este exprimată de relaţia:

$$CPU_{time} = \frac{Nr.\ total\ \ perioade\ \ de\ \ clock}{Frecventa\ \ de\ \ clock} \qquad (2)$$

Un alt mod de a exprima timpul de execuţie al unui program CPU_{time}, se poate face prin multiplicarea numărului total de instrucţiuni din program, cu timpul mediu necesar execuţiei unei instrucţiuni exprimat în perioade de *clock*. Acest indicator permite compararea performanţei diferitelor procesoare când execută aceeaşi aplicaţie *software*. Pentru aceasta a fost introdusă mărimea:

- **CPI** *(cycles per instruction)* care reprezintă valoarea medie a numărului de perioade de *clock* petrecute în procesarea unei instrucţiuni, dintr-un anumit program. Această mărime permite compararea diferitelor posibilităţi de implementare al aceluiaşi set de instrucţiuni.

În practică pentru evaluarea performanţei procesoarelor mai este utilizat şi indicatorul:

- **IPC** (*instructions per clock*) sau numărul total de instrucţiuni executate într-o perioadă de *clock*. Datorită faptului că **IPC**-ul poate varia semnificativ în funcţie de structura internă a diferitelor blocuri componente ale unui microprocesor, cum ar fi de exemplu predictorul instrucţiunilor de control, utilizarea acestuia ca un indicator al performanţei, nu mai este atât de răspândită în ultima vreme. Există totuşi o serie de lucrări de specialitate recente [2.1] care încearcă să reinstituie **IPC**-ul ca un indicator valid al performanţei.

Având în vedere mărimile definite mai sus, dacă notăm cu:

NI - numărul total de instrucţiuni executate într-o aplicaţie;

CT - numărul total de perioade de *clock* petrecute în execuţia unui program;

PC - durata unui *clock*;

se pot scrie următoarele relaţii [2.2] :

$$CPI = CT / NI \qquad (3)$$

$$IPC = 1/CPI \qquad (4)$$

$$CPU_{time} = CT * PC \quad (5)$$

Dacă în relaţia (5) înlocuim valoarea lui CT = CPI * NI aşa cum rezultă din relaţia (3), atunci relaţia (5) mai poate fi scrisă sub forma:

$$\mathbf{CPU_{time} = NI * CPI * PC} \quad (6)$$

Relaţia (6) reuneşte trei indicatori importanţi în aprecierea performanţei (numărul total de istrucţiuni ale unui program; valoarea medie a numărului de perioade de *clock* necesare procesării unei instrucţiuni din acel program; durata unui *clock*) şi permite compararea diverselor soluţii arhitecturale alternative, fiind considerată relaţia de bază pentru evaluarea performanţei.

Pe lângă aceste mărimi, pentru evaluarea performanţei unor subcategorii de aplicaţii *software,* în practică se folosesc o serie de alţi indicatori, dintre care se vor aminti:

- **MFLOPS** (*Million Floating Point Operations per Second*) - indicator utilizat pentru evaluarea aplicaţiilor ştiinţifice.

$$MFLOPS = \frac{Nr.\,total\ instr.\,in\ virgula\ mobila}{Timp\,executie * 10^6} \quad (7)$$

- **Transactions per minute** – indicator folosit pentru evaluarea performanţei aplicaţilor web.

2.2. Mijloace pentru evaluarea performanţei.

Scopul mijloacelor pentru evaluarea performanţei este obţinerea unor informaţii sau date cuantificabile, în legătură cu comportarea microprocesorului sau a aplicaţiei rulate de acesta, care în final să permită:

- aprecieri şi comparţii ale nivelului de performanţă atins.
- decizii pertinente legate de modificările sau optimizările necesare, atât la nivel hard cât şi la nivel soft.

Estimarea şi îmbunătăţirea performanţei cu care un procesor aflat în fază de proiect se va comporta în mediul în care va trebui să lucreze, nu ar fi posibilă fără

utilizare unor mijloace performante de analiză şi evaluare. Pentru ca datele furnizate să fie cât mai apropiate de valorile reale aceste mijlace de caracterizare trebuie să îndeplinească câteva condiţii. Principalele cerinţe ce trebuie îndeplinite de mijloacele pentru estimarea performanţei sunt prezenate în tabelul 2.1. [2.3].

Nr. crt.	CERINŢA	COMENTARII
1	**Să fie precise.**	Datele furnizate trebuie să fie datele reale.
2	**Să fie non-invazive.**	Procesul de evaluare trebuie să nu altereze modul de funcţionare al sistemului, sau să-i afecteze performanţa.
3	**Să fie ieftine.**	Atât proiectarea cat şi utilizarea mijlocului de evaluare trebuie să nu implice mijloace financiare importante.
4	**Să fie uşor de modificat sau extins.**	Microprocesoarele şi sistemele cu microprocesor evoluează continuu şi din această cauză este foarte important ca utilitarele de evaluare a performanţei să poată fi adaptate uşor noilor cerinţe.
5	**Să nu necesite accesul la codul sursă al aplicaţiilor.**	La unele aplicaţii accesul la codul sursă este prohibit (vezi aplicaţiile comerciale).
6	**Să aibă capabilităţi pentru măsurarea atât a activităţii sistemului de operare, cât şi a aplicaţiei utilizator.**	Doar în cazul rulării aplicaţiilor ştiinţifice, de calcul numeric intensiv, evaluarea activităţii aplicaţiei utilizator poate oferi informaţii suficiente despre performanţa unui anumit sistem. În cazul aplicaţiilor ce accesează baze de date (aplicaţiile web şi aplicaţile Java) există o componentă semnificativă din partea sistemului de operare, motiv pentru care utilitarele trebuie să poată oferi informaţii şi despre acesta.
7	**Să poată fi utilizate pentru o gamă largă de aplicaţii.**	–
8	**Să fie uşor de utilizat.**	Cu cât activităţile necesare utilizării unei structuri de analiză sunt mai complexe cu atât probabilitatea apariţiei unei erori umane este mai mare.
9	**Să fie rapide.**	Dacă un sistem de evaluare a performanţei este încet, aplicaţiile ce rulează câteva ore pe *hardware*-ul real vor rula săptămâni în cadrul mediului de simulare.
10	**Să ofere posibilităţi de control.**	Trebuie să permită controlul asupra caracteristicilor ce sunt cuantificate. Este important să măsoare doar acele caracteristici ce prezintă interes pentru proiectant.
11	**Să aibă capacitatea de a caracteriza inclusiv sistemele multiprocesor şi multi-*thread*.**	–
12	**Să fie utilizabile în diversele faze de proiectare a procesorului.**	Cerinţă impusă pentru obţinerea unor date unitare.

Tabelul 2.1. Codiţiile ce trebuie îndeplinite de mijloacele pentru evaluarea performanţei.

După cum se remarcă unele dintre cerinţele impuse mijloacelor pentru evaluarea performanţei sunt contradictorii (spre exemplu: rapid/ieftin şi precis) motiv pentru care proiectantul acestora trebuie să găsească un optim al acestora.

Tehnicile legate de predicţia şi analiza performanţei, dispun la ora actuală de mijloacele de evaluare prezentate în figura 2.1.

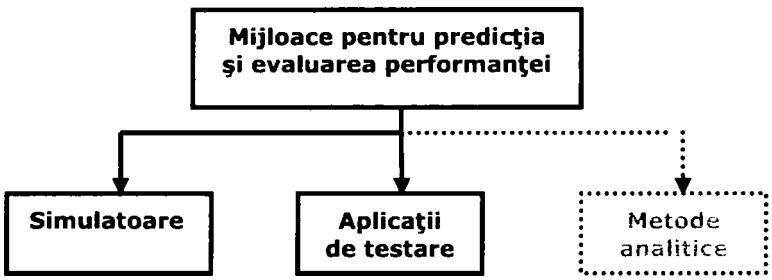


Figura 2.1. Evidenţierea mijloacelor pentru evaluarea performanţei.

Metodele analitice care se bazează pe modele probabilistice, nu vor fi utilizate în cadrul acestei lucrări.

Pe lângă mijloacele prezentate în figură mai trebuie menţionată şi o sub-categorie conexă; utilitarele pentru analiza performanţei, care prin felul în care sunt concepute şi prin facilităţile de utilizare oferite, uşurează obţinerea datelor vizate, scurtând semnificativ timpul necesar obţinerii lor. Acestea vor constitui subiectul unui capitol aparte.

2.2.1.Simulatoare de procesor.

Simulatoarele pot fi definite ca fiind nişte programe scrise într-un limbaj de nivel înalt "**C**", "**C++**" sau "**Java**" [2.4], care reproduc detaliat, în *software,* microarhitectura unui viitor *chip*. Prin rularea unor programe în această arhitectură simulată şi prin colectarea unor date despre: modul de accesare a memoriei, tipurile de instrucţiuni dintr-o aplicaţie şi frecvenţa acestora, numărul de apelări către sistemul de operare, activitatea I/O etc., este posibilă aprecierea nivelului de performanţă cu care un procesor, sau sistem, va rula o anumită aplicaţie.

Simulările, utilizate în proiectarea de microprocesoare oferă avantaje certe; dintre care cele mai importante sunt:

- fiind structuri soft, uşor de modificat, oferă un spaţiu de explorare nelimitat pentru proiectanţii de microprocesoare;
- nivelul de abstractizare al arhitecturii simulate, poate fi stabilit în funcţie de complexitatea analizei ce urmează a fi efectuată;
- oferă posibilităţi de verificare a arhitecturilor structurilor *hardware*, înainte ca acestea să fie disponibile fizic;
- permit studiul performanţei în execuţie a unor configuraţii alternative, pentru modulele ce alcătuiesc microprocesorul.

Aşa cum este ilustrat în figura 2.2, simulatoarele se împart în principal în două categorii:

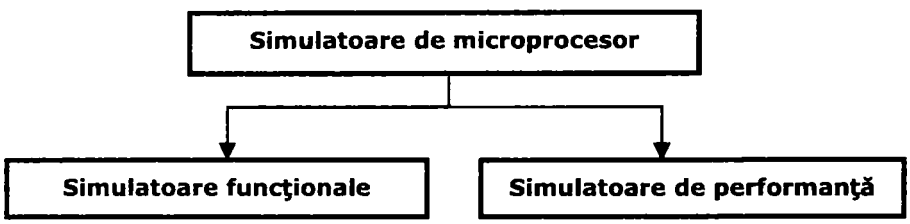


Figura 2.2. Tipuri de simulatoare.

❖ **Simulatoare funcţionale.**

Acest tip de simulatoare execută în mod secvenţial setul de instrucţiuni din alcătuirea unui program şi permit simularea unor sisteme complexe. Pentru a reproduce în detaliu modul cum interacţionează aplicaţia, procesorul şi sistemul de operare, unele dintre aceste simulatoare boot-eaza sistemul de operare care urmează să ruleze pe microprocesorul proiectat şi implementează *driver*-ele diferitelor componete I/O (intrări/ieşiri – plăci de reţea, discuri, etc.)

Un exemplu de astfel de simulator este cel prezentat în figura 2.3. În figură este reprezentat modul de interacţiune între aplicaţia test, simulatorul funcţional şi platforma *hardware* pe care acesta este rulat. Simulatorul practic creează interfaţa între: sistemul *host* (*hardware*-ul şi sistemul de operare în care este rulat simulatorul), sistemul *target* (*hardware*-ul modelat de simulator şi sistemul de operare al acestuia) pe care este rulată aplicaţia. Simulatorul funcţional emulează în

suficient detaliu componentele *hardware* ale unui sistem cu procesor, pentru a face posibilă *boot*-area unui sistem de operare în interiorul acestuia.

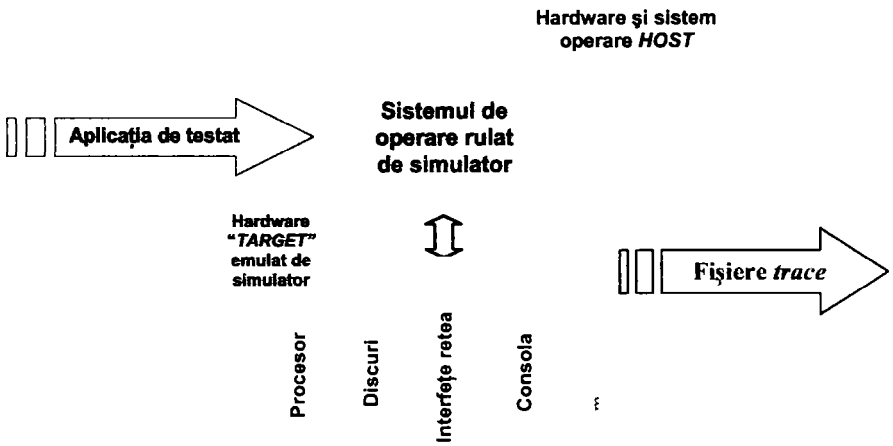


Figura 2.3. Simulatorul funcţional. Relaţia între sistemul *host* şi simulatorul funcţional.

Principalele componente *hardware* emulate sunt: una sau mai multe unităţi centrale (CPU), unitatea de control a memoriei, care cuprinde şi unitatea ce execută translaţia între adresele virtuale generate de procesor şi adresele fizice utilizate de memoria principală. De asemenea pentru a asigura rularea corectă a sistemului de operare *target*, simulatorul emulează o serie de componente I/O cum sunt unităţile de disc, consola şi interfeţele de reţea.

În practică se folosesc o serie de simulatoare funcţionale, dintre care cele mai cunoscute sunt SimOS şi Simics. În afară de acestea majoritatea companiilor ce au ca obiect de activitate fabricarea de microprocesoare deţin cel puţin un simulator funcţional propriu.

SimOS a fost realizat în cadrul universităţii Stanford în scopul studierii comportamentului sistemelor de operare [2.5]. Simulatorul este capabil să *boot*-eze ca sistem de operare ţintă o versiune de Unix realizată şi utilizată de compania Silicon Graphics. Modelul de unitate centrală emulat este bazat pe arhitectura MIPS.

Simics este produs de o companie suedeză numită Virtutech AB [2.6] şi este probabil unul dintre cele mai complexe simulatoare funcţionale existente în domeniu. Este capabil să simuleze mai multe seturi de instrucţiuni cum ar fi Alpha (DEC), PowerPC (IBM), UltraSparc (Sun Microsystems) şi x86-64 (Intel and AMD). Este extrem de versatil în ceea ce priveşte *boot*-area sistemelor de operare *target*, putând să *boot*-eze Linux, Solaris sau Windows. De asemenea este capabil să emuleze mai multe tipuri de unităţi centrale pornind de la un procesor secvenţial în care fiecare instrucţiune durează un tact, până la un procesor *out-of-order*.

❖ **Simulatoare de performanţă.**

Aceste simulatoare modelează în *software* micro-arhitectura procesorului şi poate executa simularea în timp a unor aplicaţii test reprezentative. În urma simulării se înregistrează o serie de parmetri cum ar fi: CPI, număr de *miss*-uri în diferitele niveluri ale memoriei cache, numărul de predicţii greşite a instrucţiunilor de *branch*, etc.; parametrii ce reflectă viteza de execuţie şi modul în care aceste aplicaţii interacţionează cu modulele platformei *hardware*. Aceste simulatoare sunt utilizate în fazele de început ale proiectării pentru evaluarea unor structuri arhitecturale alternative şi alegerea unei structurii optimizate.

În figura 2.4. este prezentat schematic un simulator de performată şi interacţiunea între acesta şi platforma *hardware* reală pe care este rulat.

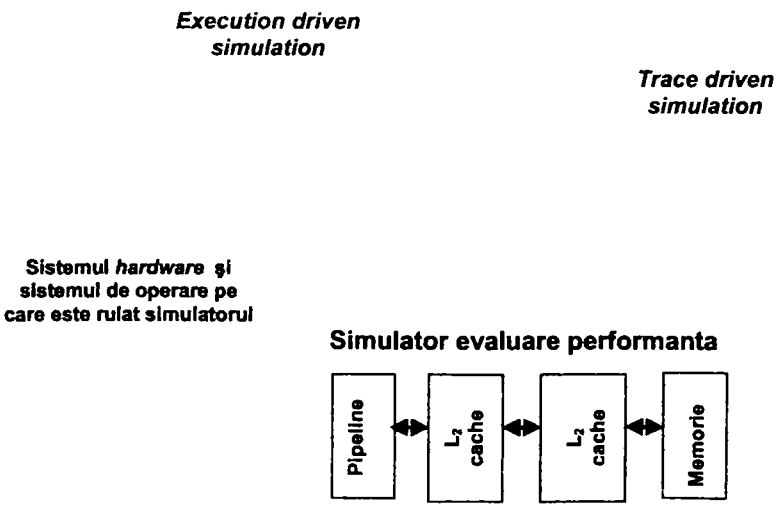


Figura 2.4. Relaţia între simulatorul de performanţă şi sistemul hardware pe care acesta este rulat.

Au fost scoase în evidenţă blocurile principale ale unei arhitecturi în fază de proiect (pipeline, ierarhia de memorie etc.) modelate de acesta şi s-au scos în evidenţă cele două moduri în care acesta poate opera (*trace driven, execution driven).*

2.2.2. Aplicaţii pentru evaluare performanţei.

Aplicaţiile pentru evaluarea performanţei sunt programe de calculator reale, sau programe sintetice (scrise în mod special în acest scop).

Condiţiile impuse unor astfel de programe sunt:

- să furnizeze date reale despre performanţa sistemului analizat,
- să ofere informaţii despre gradul de utilizare al componentelor *hardware* ale procesorului.

În ordinea descrescătoare a acurateţei oferite, clasificarea acestor aplicaţii se face pe patru nivele [2.2] :

- **Aplicaţii reale**: Acestea sunt aplicaţii utilizate în mod curent în diferite domenii şi permit evaluarea performanţei platformei *hardware* pe care sunt rulate.
- **Kernel**: Acestea sunt porţiuni cheie extrase din aplicaţii reale. Sunt utilizate în principal pentru a studia performanţa a doar unor anumite blocuri componente ale platformei *hardware,* cum ar fi de exemplu sistemul de memorie.
- **Programe "jucarie"**: Acestea sunt aplicaţii cu doar câteva linii de cod al căror rezultat este ştiut dinainte de utilizator. Avantajele acestora sunt dimensiunile reduse, uşurinţa utilizării şi rulării.
- **Aplicaţii sintetice**: Acestea sunt similare kernele-lor. Rolul lor este de a sintetiza secvenţa şi distribuţia instrucţiunilor unui set de aplicaţii reale.

Programele de testare utilizate pentru evaluarea performanţei, alegerea acestora, precum şi analiza în cele mai mici detalii ale părţilor componente şi a modului lor de execuţie, este o problemă critică atunci când se doreşte o estimare precisă a performanţei. Din acest motiv în ultima vreme, deoarece permit modificări şi există posibilitatea ca rezultatele obţinute să nu fie cele reale, ultimele trei dintre

tipurile de aplicaţii prezentate, într-o măsură mai mare sau mai mică, sunt discreditate.

Pentru evitarea posibilităţilor de mistificare, cu scopuri comerciale, a datelor despre performanţa reală a sistemelor cu microprocesor, în 1988 a fost fondată *Compania Standard Performance Evaluaţion Corporation (SPEC).* SPEC oferă mai multe pachete de programe standard, destinate testării performanţei, care au devenit obligatorii pentru evaluarea platformelor *hardware* existente sau viitoare. Ulterior au fost create programe speciale pentru testarea diferitelor categorii de aplicaţii. Principalele programe pentru evaluarea performanţei utilizate în practică sunt prezentate în Anexa 2.

Pentru a avea o imagine completă asupra performanţelor unui sistem *hardware* este necesară analizarea unui spectru foarte larg de aplicaţii; de la aplicaţiile de testare de genul **SPECCPU 2000** descrise pe larg în Anexa 2 până la aplicaţii ştiinţifice ori comerciale sau de altă natură, ce sunt rulate de către utilizatori.

2.3. Modul de operare în simulare.

Evaluarea performanţei obţinută în rularea unui program pe un anumit procesor existent se face prin lansarea acelui program şi măsurarea timpului necesar execuţiei lui. Datele cantitative sunt obţinute cu ajutorul unor utilitare puse la dispoziţie de sistemul de operare, sau a unor utilitare specifice obţinerii unui anumit gen de date, ori prin programarea şi citirea regiştrilor de evaluare a performanţei prezenţi în majoritatea microprocesoarelor moderne.

Problema devine însă mai complicată atunci când trebuie prezisă performanţa unei structuri noi aflată încă în faza de concepţie. Ciclul de proiectare al unei asemenea structuri complexe cum este procesorul poate ajunge până la 7 ani şi presupune derularea unor ample programe de cercetare – proiectare şi implică mobilizarea unor importante resurse, umane şi financiare. Datorită complexităţii microchip-urilor moderne, cu arhitecturi tot mai sofisticate, precum şi datorită costurilor foarte mari de proiectare ale acestora, anticiparea performanţei unor astfel de structuri, încă din faza de proiectare, devine o condiţie obligatorie. Pentru ca procesorul vizat să se ridice la nivelul de performanţă dorit, trebuie înţeles încă

din fazele incipiente ale proiectării modul în care acesta va interacţiona cu celelalte componente alę sistemului, cum sunt discurile, componentele de reţea şi nu în ultimul rând sistemul de operare. Evaluarea performanţei diverselor configuraţii şi alegerea celei mai bune variante arhitecturale pentru procesorul "ţintă" reprezintă provocarea la care trebuie să răspundă toţi cei implicaţi în proiectarea lui.

În general proiectarea unui procesor sau a unui sistem computaţional (figura2.5.) [2.7], presupune parcurgerea următoarelor etape:

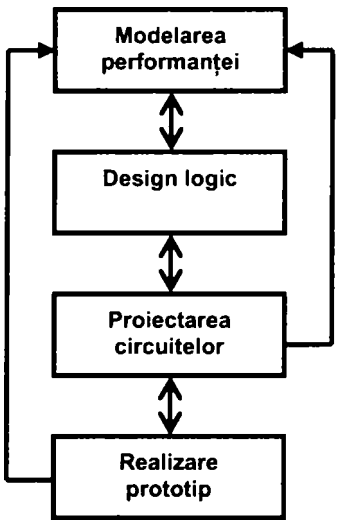


Figura 2.5. Etapele proiectării unui procesor.

- **modelarea performanţei**, acestă fază iniţială are o importanţă deosebită deoarece presupune scrierea unor programe de simulare, care reproduc în detaliu blocurile componente şi modul de funcţionare al microprocesorului. Prin rularea unor aplicaţii pe aceste structuri simulate, este evaluată performanţa diferitele alternative arhitecturale şi în funcţie de rezultatele obţinute sunt realizate modificările necesare.
- **designului logic,** este etapa în care fiecare componentă *hardware* în structura şi complexitatea stabilită în etapa aterioară este descrisă logic cu ajutorul unor programe ca Verilog sau VHDL.
- **proiectarea circuitelor,** etapă în care harta logică a procesorului este sintetizată în porţi logice şi fiecare componentă în parte este aşezată în spaţiul dedicat, în aria *chip*-ului.

- **realizarea prototipului.**

Între aceste faze există o strânsă legătură, iar informaţiile referitoare la durata de răspuns şi alte caracteristici ale diverselor componente sunt furnizate proiectanţilor programelor de simulare pentru a fi luate în considerare, în scopul îmbunătăţirii acurateţei procesului de evaluare a performanţei.

Pentru ca estimările generate de simulator să fie cât mai apropiate de datele de performanţă ale viitorului *hardware*, simulatorul trebuie să reproducă cât mai veridic diferitele blocuri componente ale acestuia. Între acurateţea simulării şi timpul de simulare există însă o dependenţă directă. Din acest punct de vedere la un capăt al spectrului se poate afla un simulator secvenţial ce implementează doar detalii minime legate de arhitectura unui *pipeline*, iar la celalat capăt se află harta procesorului în întregul ei, descrisă în Verilog sau VHDL, aceea care în etapa finală va fi sintetizată în porţi logice. Durata simulării însă este direct proporţională, cu complexitatea blocurilor conţinute de design-ul simulat, putând ajunge în unele cazuri la valori deranjant de mari. Din această cauză proiectanţii simulatoarelor de procesor trebuie să găsească un optim între acurateţe simulării şi viteză de simulare.

În orice condiţii însă, dat fiind faptul că pe baza rezultatelor simulărilor efectuate se iau decizii critice, legate de structura şi componentele unei arhitecturi, este imperios necesar ca datele obţinute să aibă valorile sensibil egale cu valorile reale [2.8]. Pentru realizarea acestui deziderat în practică procesul de simulare implică efectuarea următoarelor activităţi :

a) Validarea acurateţei simulatorului.

Una din modalităţile de validare este aceea de a compara rezultatele simulării cu cele obţinute din simulările blocurilor logice exprimate în Verilog pe măsură ce acestea devin disponibile. O altă metodă de validare este compararea rezultatelor simulatoarelor cu rezultatele obţinute din simulări directe pe procesoarele fizice anterioare din aceeaşi familie. Se utilizează de asemenea o serie de scurte programe de testare scrise în limbaj de asamblare numite "*microbenchmarks*" a căror performanţă poate fi evaluată prin metode analitice.

b) Implementarea mecanismelor microarhitecturale ce vizează creşterea vitezei de execuţie a procesoarelor.

În această etapă proiectanţii exprimă în software, ca parte a simulatorului

de evaluare a performanţei, diversele soluţii microarhitecturale ce au ca scop creşterea vitezei de execuţie.

c) Selectarea parametrilor blocurilor componente ale simulatorului.

Simulatoarele de performanţă actuale pentru a oferi un spaţiu de explorare nelimitat oferă un număr extrem de variat de posibilităţi de configurare a blocurilor ce le alcătuiesc. Modul de configurare poate influenţa decisiv performanţa raportată de simulator, o configurare greşită putând conduce la raportări eronate. În plus de aceasta modul de configurare al unui bloc poate influenţa modul de execuţie al altor blocuri declanşând astfel o reacţie în lanţ. Din această cauză configurarea simulatorului trebuie făcută cu extrem de multă atenţie pentru a reflecta în mod cât mai veridic microarhitectura procesorului ce se intenţionează a fi simulat şi bineînţeles cea care produce rezultatele cele mai favorabile.

d) Alegerea aplicaţiilor de testare.

Aplicaţiile de testare sunt extrem de numeroase, acoperind în mare parte întregul spectru al aplicaţiilor *software*. Datorită timpului necesar simulării este extrem de dificilă obţinerea de date de performanţă pentru toate aplicaţiile de testare ce acoperă un anumit domeniu al aplicaţiilor *software*. Din această cauză este foarte importantă clasificarea acestora şi obţinerea unor liste de aplicaţii cu caracteristici echivalente. Odată realizată aceasta se poate simula doar un subset din fiecare categorie fără a pierde din acoperire.

e) Simularea.

Odată ce simulatorul de performanţă a fost configurat, mecanismele de îmbunătăţire a performanţei au fost exprimate în *software* şi aplicaţiile de testare au fost alese, urmează să se execute simularea propriu-zisă. In această fază simulatorul execută instrucţiune cu instrucţiune o aplicaţie sau o parte dintr-o aplicaţie de testare şi la final raportează o serie de date statistice legate de modul de execuţie.

f) Analiza performanţei.

Aceasta este etapa în care bazat pe datele raportate de simulatorul de performanţă proiectanţii evaluează beneficiul mecanismelor arhitecturale implementate. Aici se compară mărimile de evaluare a performanţei prezentate în [1.8]. În urma acestor studii se decide dacă îmbunătăţirile microarhitecturale implementate au adus beneficiile scontate în ceea ce priveşte creşterea performanţei.

Modurile de simulare sunt în principal în număr de două (figura 2.6.):

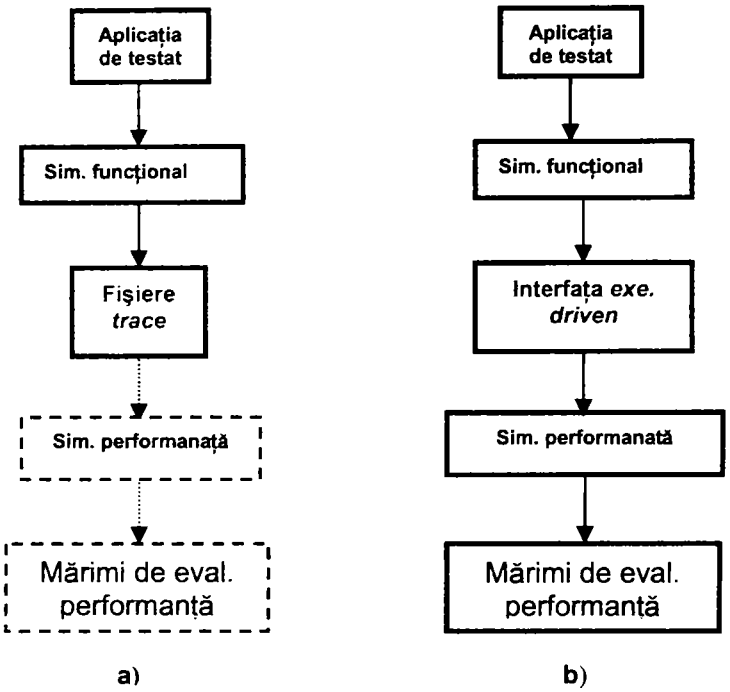


Figura. 2.6. Tipurile de simulări: a) *trace-driven* b) *execution-driven*

❖ **Simulări *trace driven*.**

În acest mod de lucru simulatorul de performanţă citeşte şi execută instrucţiunile dintr-un fişier *trace* ce conţine instrucţiunile ce au fost executate în timpul unei execuţii anterioare a programului evaluat. Colectarea fişierelor *trace* se face în diverse moduri, dintre care amintesc:

- utilizarea unor sonde ce permit captarea traficului pe magistrala de memorie,
- emularea dinamică a instrucţiunilor de accesare a memoriei,
- utilizarea modului *debug* a sistemului de operare în care execuţia fiecărei instrucţiuni cauzează un *trap* în sistemul de operare,
- marcarea statică a aplicaţiei de test cu instrucţiuni ce permit stocarea accesărilor memoriei în fişiere *trace*, simulatoarele funcţionale etc.

Datorită avantajelor pe care le prezintă, cel mai comun mod de obţinere a

fişierelor *trace* este acela de a utiliza simulatoarele funcţionale (figura 2.6.a).

Avantaje:

- simplitatea şi faptul că odată ce au fost colecţionate fişierele *trace* pot fi refolosite.
- reproductibilitatea, două simulări diferite ce utilizează acelaşi fişier *trace* vor executa exact acelaşi set de instrucţiuni.

Dezavantaje:

- dimensiunile mari ale fişierelor *trace,*
- fişierele *trace* conţin doar instrucţiuni de pe direcţia nespeculativă a execuţiei, făcând dificilă simularea scenariilor ce apar în urma predicţiei unei instrucţiuni de control.

❖ **Simulări *execution driven*.**

În acest mod de simulare simulatorul rulează programele de testare instrucţiune cu instrucţiune şi pentru aceasta necesită conectarea simulatorului de performanţă cu unul funcţional. Simulatorul funcţional reproduce fidel starea sistemului în fiecare moment, dar nu are noţiunea duratei de execuţie a fiecărei instrucţiuni, aceasta fiind modelată de simulatorul de performanţă (figura 2.6.b). Atunci când se efectuează un asemenea tip de simulare trebuie avute în vedere avantajele şi dezavantajele ce le implică.

Avantaje:

- executarea directă a programului de testare,
- permite accesul la informaţii asupra stării interne a simulatorului în orice moment.
- pot fi simulate o serie de mecanisme speculative de rezolvare a interdependenţelor de date şi de control.
- pot fi studiate efectele ce apar în sistemele multiprocesor cum ar fi de exemplu, ordinea de acaparare şi eliberare a unui *lock* (o locaţie de memorie specială, utilizată în sincronizarea accesării unei zone critice de memorie, de către mai multe procesoare).

Dezavantaje:

- dificultatea implementării,

- dificultatea simulării unor aplicaţii comerciale, care accesează cantităţi foarte mari de memorie şi disc.

În studiile de evaluare a performanţei proiectanţii de procesoare apelează la una sau cealaltă metodă de simulare în funcţie de tipul analizei ce se doreşte a fi efectuată. Datorită vitezei scăzute a simulărilor *execution driven* acest mod de simulare este utilizat în special în cazurile în care se doreşte evaluarea mecanismelor speculative şi a sistemelor multiprocesor. Pentru toate celelalte studii unde viteza de execuţie este o problemă se foloseşte modul de simulare *trace driven*.

În urma simulărilor sunt obţinute date diverse, referitoare la: distribuţia instrucţiunilor; caracteristicile de accesare a memoriei; reutilizarea locaţiilor de memorie; sensibilitatea aplicaţiilor la structura ierarhiei de memorie; modul de reutilizarea a datelor între diferitele *thread*-uri soft în execuţia variantelor paralelizate a programelor studiate, analiza avantajelor utilizării unor mecanisme de precitire, etc.

2.4. Concluzii.

Scopul acestui capitol este de a introduce noţiunea de performanţă a sistemelor computaţionale, ce reprezintă subiectul principal al acestei lucrări. Sunt introduse mărimile asociate performanţei şi sunt prezentate mijloacele moderne utilizate în activitatea de analiză a ei. Între acestea se evidenţiază simulatoarele de procesor, utilitarele de analiză şi *benchmark*-urile pentru evaluarea performanţei.

Simulatoarele şi utilitarele reprezintă mijloacele principale folosite în evaluarea platformelor *hardware* în fază de concepţie, deoarece fiind instrumente soft oferă proiectanţilor un spaţiu de explorare nelimitat.

Simulatoarele în funcţie de destinaţia lor, sunt clasificate în simulatoare funcţionale şi simulatoare de performanţă. Cele din prima categorie, simulatoarele funcţionale, simulând atât aplicaţia test cât şi sistemul de operare în care aceasta rulează. Principalul rol al acestui tip de simulatoare, este acela de a genera fişierele *trace,* folosite ulterior în analiza performanţei.

Simulatoarele de performanţă implementează în *software* structurile microarhitecturale ale unui microprocesor şi produc informaţii referitoare la nivelul de performanţă la care o aplicaţie rulează pe microprocesorul simulat.

Sunt descrise deasemenea cele două moduri principale de simulare, *trace-driven* şi *execution-driven,* facându-se aprecieri critice legate de avantajele şi dezavantajele folosirii fiecăruia dintre ele.

În cele din urmă, se face o sinteză a etapelor parcurse în simulare, în vederea îmbunătăţirii performanţei unei platforme *hardware* noi.

3. Mijloace folosite în activitatea de caracterizare a aplicaţiilor *software* – un nou concept.

3.1. Aprecieri asupra utilitarelor şi a modului în care acestea sunt folosite în activitatea de analiză a performanţei.

În activitatea de predicţie şi evaluare a performanţei în funcţie de scopul urmărit şi nivelul de complexitate şi abstractizare al analizei efectuate sunt utilizate, pe lângă regiştrii *hardware* şi mijloacele de analiză a performanţei puse la dispoziţie de sistemul de operare, nişte structuri soft denumite generic utilitare. Scopul acestor structuri este acela de a facilita obţinerea informaţiilor dorite şi de a scurta timpul necesar obţinerii lor. Aceste structuri pot funcţiona de sine stătător, sau înserate în simulatoarele destinate evaluării performanţei.

Clase de utilitare:

Utilitarele folosite pentru analiza performanţei sunt de o mare diversitate, de la cele ce oferă doar analiză specializată a fişierelor *trace*, până la cele ce oferă o analiză detaliată a modului de execuţie a programelor, atât pe platformele *hardware* reale, cât şi a celor aflate în fază de proiectare [3.1], [3.2]. Dacă vom analiza însă aceste utilitare prin prisma cerinţelor cărora trebuie să le răspundă un mijloc de analiză a performanţei (prezentate în Tabelul 2.1.) se pot face următoarele aprecieri:

1) Dacă luăm în considerare utilitare ca *Sun Studio Performance Analyzer* [3.3]; *Intel Vtune Performance Analyzer* [3.4] (unele dintre cele mai utilizate structuri pentru analiza performanţei), vom constata că permit doar colectarea informaţiilor oferite de către regiştrii *hardware* de analiză a performanţei, cu care este dotat procesorul. Acest fapt limitează utilizarea acestora numai la analiza

performanţei platformelor *hardware* existente, deci nu pot fi utilizate pentru analiza platformelor aflate în fază de proiect.

2) O altă clasă de utilitare care include *MemSpy* [3.5], *CPROF* [3.6] şi *gprof* [3.7], pot fi utilizate pentru analiza interacţiunii aplicaţiilor cu memoriile *cache*. Mai precis, permit maparea *miss*-urilor într-o memorie *cache* de o anumită configuraţie spre liniile sau structurile de program care le-au generat, acest lucru însă este posibil numai cu condiţia instrumentării codului sursă al programului în cauză, cu macro-uri specifice. Necesitatea accesului la codul sursă al aplicaţiilor studiate şi recompilarea acestora, reprezintă o piedică importantă în folosirea utilitarelor de acest tip în analizarea aplicaţiilor comerciale, datorită accesului prohibitiv la codul sursă al acestor aplicaţii. În plus, acest tip de utilitare nu poate fi folosit pentru analiza activităţii generată în sistemul de operare ca urmare a rulării aplicaţiei respective.

3) Utilitarele cum sunt *Dtrace* [3.8] şi *KernInst* [3.9] permit analizarea activităţii sistemului de operare, dar o pot face doar în timp real, în timpul în care acesta rulează pe platforma *hardware* analizată.

4) Alte programe de analiză a performanţei cum sunt *TraceVis* [3.10] şi *TAXI* [3.11] permit maparea anumitor evenimente din execuţia unui program, către codul sursă al programului testat, dar pentru acesta ambele necesită modificarea simulatorului de performanţă. Acest fapt face ca utilizarea lor să fie limitată, din cauza diviziunii existente între expertiza analiştilor de *software* şi cea a proiectanţilor simulatoarelor de performanţă.

Concluzionând asupra celor arătate, se constată că utilitarele la care s-a făcut referire, au una sau mai multe dezavantaje; dezavantaje care în mai toate cazurile le limitează aplicabilitatea la analize de ordin general.

Rezumând, aceste dezavantaje sunt:

- necesită suport *hardware* (pot analiza doar sistemele *harware* existente);
- necesită acces la codul sursă al programului precum şi modificări sau recompiiări ale acestuia (pentru unele aplicaţii accesul este

prohibitiv);

- analiza este limitată doar la anumite platforme *hardware,* extinderea sau recompilarea lor fiind o operaţie dificilă;
- analiza sistemului de operare poate fi făcută numai în timp real;
- necesită modificarea simulatorului de performanţă al platformelor *hardware* studiate.

Pe lângă problemele legate de posibilităţile de folosire a unor utilitare, în activitatea practică se poate constata că apar şi alte aspecte cu conotaţie negativă (unele cu caracter obiectiv, altele cu caracter subiectiv), datorate modului de organizare a activităţii de evaluare a performanţei, ca parte a activităţilor de proiectare. Aceste aspecte apar mai ales datorită segmentării expertizei în analiza performanţei, între diversele departamente participante la proiectarea procesorului. În diferitele faze de proiectare, diferitele compartimente participante, utilizează diferite utilitare pentru analiza performanţei, creându-se în acest fel o barieră tehnică în calea accesului tuturor celor interesaţi la informaţii într-un format unitar.

Adesea arhitecţilor li se oferă date despre performanţa unor aplicaţii pe platformele *hardware* existente, fiind lăsaţi să extrapoleze singuri modul de comportarea al acestor aplicaţii pe *chip*-urile pe care le proiectează. Acest lucru, evident, poate conduce la estimări eronate sau chiar erori.

Nu în rare cazuri utilitarele folosite sunt specifice doar unor anumite arhitecturi, sau numai unor simulatoare. Din această cauză mulţi utilizatori preferă să rescrie pornind de la zero un nou utilitar, în loc să reutilizeze unele deja existente, dar care trebuie modificate, fapt ce reprezintă un efort suplimentar care conduce şi la o anumită ineficienţă în proiectare.

Pe lângă aceasta, utilizatorii care deşi cunosc detaliile intime despre structura şi alcătuirea programelor de testare pe care le utilizează, pentru a putea folosi utilitarele de analiză disponibile şi pentru a le putea îmbunătăţii, sau adapta, trebuie să facă efortul de a înţelege structura şi funcţionarea acestora. Fiecare grup de proiectare dispune, spre exemplu, de o serie de utilitare menite să analizeze fişierele *trace*. Dar dintre acestea puţine utilitare sunt suficient de flexibile, pentru a fi utilizate fără a fi modificate atât pentru analiza fişierelor *trace,* cât şi înserate în simulatoarele de performanţă, între diverse etape ale *pipeline*-ului.

3.2. Flexible Analysis Framework. Descriere.

Pentru rezolvarea unora dintre neajunsurile prezentate anterior, pe care le-am constatat personal în activitatea practică, neajunsuri legate de faptul că: unele din utilitarele folosite în mod curent pot fi utilizate doar pe platformele *harware* existente, necesită accesul la codul sursă al programului, precum şi modificări ori recompilări ale acestuia, sau necesită modificarea simulatorului de performanţă a platformei *hardware* studiate, la care se adaugă neajunsurile de natură subiectivă costatate în activitatea de evaluare a performanţei; s-a născut ideea unui nou concept de analiză denumit "*Flexible Analysis Framework*" (FAF).

Scopul principal avut în vedere la conceperea noii structuri, a fost acela de a realiza o structură de analiză uşor de înţeles şi de utilizat, cu un grad înalt de universalitate, care să nu implice activităţi laborioase când ar fi necesară o eventuală modificare a ei şi care să nu afecteze în niciun fel structura simulatorului, la care aceasta este conectată.

Pentru realizarea dezideratului propus, "Flexible Analysis Framework" (FAF), a fost conceput ca un cadru software a cărui infrastructură este realizată din module independente cu funcţii specializate, destinate analizei diverselor „evenimente" urmărite în timpul simulării, interconectabile şi interschimbabile. În funcţie de parametrii urmăriţi în timpul simulării, tipul şi gradul de complexitate al analizei efectuate, modulele realizate conform noului concept, se conectează între ele pentru a realiza, un aşa numit "arbore de analiză", care devine în acest fel o unitate funcţională, cu o configuraţie adecvată scopului urmărit.

La conceperea FAF precum şi la proiectarea şi realizarea noii structurii, a cărui aplicabilitate practică a fost deja dovedită, autorul prezentei lucrări a avut o importantă contribuţie.

3.2.1. Caracteristici FAF

Conceperea noii structuri *Flexible Analysis Framework* în maniera prezentată se materializează în câteva caracteristici ce o individualizează în contextul larg al structurilor de analiză folosite în prezent şi îi conferă următoarele avantaje:

a). Flexibilitate şi versatilitate.

Arhitectura modulară a noii structuri permite configurarea acesteia în blocuri funcţionale diverse ce pot fi modificate şi adaptate atât pentru analiza activităţii sistemului de operare, cât şi pentru analiza unui spectru foarte larg de aplicaţii, inclusiv a celor ştiinţifice şi comerciale, putând fi utilizată pentru evaluare performanţei, atât pe platformele existente cât şi pe cele în faza de proiectare. Modulele care alcătuiesc FAF au fost proiectate pentru a putea fi folosite, fără modificări, atât pentru analizarea fişierelor *trace*, cât şi ca parte componentă a unor simulatoare de performanţă, sau a ierarhiei de memorie. FAF permite, spre exemplu, analiza detaliată a „evenimentelor" din ierarhia de memorie, sau din *pipeline*, precum şi *map*-area acestora spre structurile din codul sursă, în aceeaşi măsură în care permite caracterizarea la nivel înalt a modului de funcţionare al bazelor de date, din aplicaţiile comerciale.

b). Reconfigurare dinamică.

Arborele de analiză poate fi reconfigurat dinamic, conform cu „evenimentele" apărute în informaţia de analizat. În acest fel se poate realiza, într-o singură simulare, urmărirea simultană a mai multor parametrii, sau obţinerea unor date diverse despre un anumit parametru, cât şi o gamă variată a tipurilor de analiză, fără ca structura principală a programelor de testat să fie modificată.

c). Uşor de înţeles.

Design-ul noii structuri orientat pe obiecte conferă acestuia posibilitatea ca diferitele module să fie scrise izolat unele de altele, fără a fi necesare cunoştinţe despre arhitectura întregului utilitar, sau despre arhitectura simulatorului de performanţă în care acesta urmează a fi utilizat. Acest fapt este deosebit de important atunci când în situaţii speciale de analiză se pune problema unei eventuale adaptări sau modificări a structurii de analiză.

d). Uşor de utilizat.

FAF structurat sub formă de arbore, se conectează la simulatoarele de

performanţă, facând posibilă obţinerea informaţiilor dorite, fără a fi necesară o documentare prealabilă, aprofundată, asupra structurii de analiză, deoarece atât pentru a o folosi cât şi pentru a o extinde, utilizatorul trebuie să înţeleagă doar o mică parte din modul de organizare şi codul sursă al acesteia. Aceast fapt contribuie într-o mare măsură la eliminarea timpilor auxiliari, şi scurtarea timpului necesar generării profilului aplicaţiilor studiate, sau a ciclului de proiectare.

e). Influenţă nesemnificativă asupra timpului de simulare.

Pentru a avea o imagine asupra măsurii în care FAF influenţează timpul necesar obţinerii datelor de interes (element important avut în vedere la proiectarea acestuia), în comparaţie cu timpul necesar pentru a obţine aceleaşi date folosind utilitarele uzuale s-au efectuat diverse teste comparative. Testele efectuate, aşa cum era de aşteptat au relevat faptul că şi în cazul utilizării FAF, viteza de lucru este dependentă de frecvenţa evenimentelor analizate (*miss*-urile din nivelul unu al memoriei *cache* sunt mai frecvente decât cele din TLB). În cele mai defavorabile cazuri, atunci când este înserat în simulatorul de performanţă, scăderea vitezei de execuţie este de maxim 5%. Această scădere însă este compensată de faptul că FAF permite într-o singură simulare urmărirea mai multor parametrii, sau obţinerea unor date diverse despre un anumit parametru. La acestea dacă se adugă şi faptul că timpul necesar pentru ai înţelege modul de funcţionare este minim, la o apreciere globală se constată de fapt o îmbunătăţire a timpului necesar generării datelor de interes. În situaţia în care FAF este utilizat doar pentru caracterizarea fişierelor *trace*, indiferent de complexitatea arborelui de caracterizare, el nu are influenţă asupra vitezei de lucru a simulatorului.

f) Grad ridicat de universalitate.

Modul în care a fost conceput FAF îi conferă acestuia un grad înalt de universalitate motiv pentru care poate constitui o punte de legătură între proiectanţii de *hardware* şi a celor de *software* permiţându-le să schimbe informaţii despre modul în care aplicaţiile vor rula pe platformele proiectate, încă din fazele incipiente ale proiectării. În acest fel se reduce semnificativ, sau chiar se elimină, necesitatea extrapolării comportării diverselor aplicaţii pe platformele *hardware* existente, deoarece datele pot fi obţinute uşor, direct din simulatoarele de performanţă ale noilor arhitecturi reducându-se semnificativ atât ciclul de proiectare al *hardware*-ului în sine, cât şi al *software*-ului ce urmează să ruleze pe respectivele platforme.

3.2.2. Clasele principale şi arhitectura FAF

Infrastructura noii structuri de analiză a fost concepută ca o structură modulară fiind constituită din două categorii de module, care stau la baza realizării oricărui "arbore de analiză", denumite:

- *analyzers* (module pentru analiză şi clasificare),
- *profilers* (module pentru înregistrare şi/sau contorizare).

În figura 3.1. este reprezentată diagrama UML (*Unified Modeling Language*) [3.12], sau diagrama claselor Java. Figura ilustrează relaţia de generalizare existentă între clasa „părinte" (*supertype*) *Module* şi clasele „copii" (*subtype*), *Profiler* si *Analyzer* care sunt o formă specializată a clasei părinte.

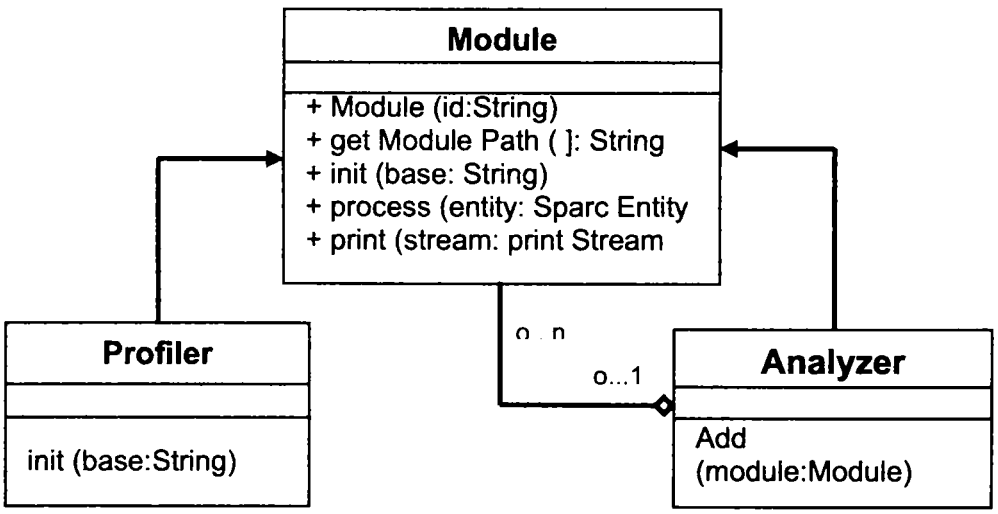


Figura 3.1. Diagrama UML: *Analyzer*-ele si *Profiler*-ele extind clasa abstracta *Module*

Este prezentată de asemenea relaţia de asociere între clasa *Module* şi clasa *Analyzer* şi anume: clasa *Analyzer* este un container de *Module,* mai precis clasa *Module* este conţinută în clasa *Analyzer*. Relaţia dintre acestea este o relaţie de agregare. În cazul în care clasa container este distrusă toate clasele care le conţine sunt distruse la rândul lor.

- ***Analyzers.***

Analyzer-ele sunt modulele ce formează "ramurile arborelui" şi nu poat fi utilizate decât la acest nivel. Pentru efectuarea unor analize amănunţite, *analyzer*-ele pot fi conectate între ele.

Analyzer-ul acţionează asupra şirului de informaţii ce urmează a fi analizat de nivelurile inferioare, în sensul că:

- *analyzer*-ul poate să clasifice şirul informaţiilor de analizat, divizându-l în subşiruri ce urmează a fi analizate separat, de către modulele de la nivelurile inferioare ale arborelui de analiză. Clasificarea şirului de informaţii se face în funcţie de anumite caracteristici cum ar fi, identificatorul *thread*-ului soft, sau identificatorul procesorului, etc. În acest fel caracteristica urmărită este analizată separat, pentru fiecare procesor sau *thread*, conform indicatorului ataşat.
- *analyzer*-ul este capabil să marcheze şirul de informaţii cu date adiţionale, ce pot fi utilizate de către modulele aflate la nivelurile inferioare din arborele de analiză. De exemplu: *analyzer*-ul funcţiilor, poate marca şirul de informaţii cu numele funcţiei corespunzătoare, urmând ca modulele de la nivelurile inferioare ale arborelui, să utilizeze această informaţie fără a mai fi necesară *map*-area, între şirul de informaţie şi numele funcţiei în cauză;
- *analyzer*-ul are capacitatea, de a transforma şirul de informaţie prin modificarea unor câmpuri ale acestora. Spre exemplu: *analyzer*-ul ce caracterizează modul în care se face alocarea memoriei fizice, poate modifica intervalele de memorie studiate şi să re-*map*-eze şirul de informaţii, urmând ca analiza la nivelurile inferioare să se facă cu noile *map*-ari.

- ***Profilers.***

Profiler-ul este un modul ce formează "frunzele arborelui" de analiză, care identifică, şi/sau cuantifică, anumite evenimente din şirul de informaţii de analizat, trimis spre el de către *analyzer*-e. Spre deosebire de *analyzer*-e, *profiler*-ele nu au capacitatea de decizie, motiv pentru care nu pot fi conectate între ele. Conectarea lor este posibilă doar cu unul dintre *analyzer*-e. Deşi *profiler*-ele pot fi privite ca fiind simple numărătoare de evenimente, conectate în arborele de analiză, ele au o utilitate deosebită. Spre exemplu, acelaşi tip de *profiler,* dedicat analizării cantităţii de memorie accesată de un şir de instrucţiuni, poate fi conectat la *analyzer*-e cu specific diferit, generând în acest fel date diverse. (În figura 3.2. *profiler*–ul *memory footprint* oferă într-o singură simulare, fără a necesita modificări, informaţii

despre creşterea cantităţii de memorie accesată, atât pentru fiecare funcţie a codului sursă, cât şi pentru fiecare segment de memorie).

Profiler-ele pot fi proiectate pentru a servi unor necesităţi specifice, fără a necesita cunoştinţe despre structura *analyzere*-lor, la care ar urma să fie conectate. Spre exemplu *Profiler*-ul pentru verificarea corectitudinii secvenţei PC-urilor a fost scris în mod special, pentru a testa acurateţea simulatorului funcţional şi a modului în care acesta generează fişierele de *trace*.

În Anexa 3 sunt prezentate principalele *analyzer*-e si *profiler*-e fiind indicată denumirea şi destinaţia fiecăruia. Detaliat, pe parcursul lucrării, se vor prezenta numai acele module la care se va face referire directă.

3.2.3. Arborele de analiză, modul de operare.

Modul în care se construieşte un arbore de analiză este prezentat figura 3.2.

Arhitectura arborelui de analiză are la bază modulul cu funcţionalitatea cea mai generală, dezvoltându-se apoi cu module tot mai specializate, pentru ca în final să fie conectat modulul pentru cuantificare. Informaţia de analizat ce conţine date referitoare la tipurile de instrucţiuni, adresele de memorie accesate, conţinutul regiştrilor *hardware,* sau alte date specifice, urmărite în simularea făcută, sunt transmise spre modulele utilitarului, care fiecare la nivelul său, decodifică informaţia şi o procesează sau o cuantifică, în funcţie de destinaţia şi abilităţile proprii, fără a afecta în nici un fel modul de execuţie al simulatorului în cauză.

Se remarcă faptul că din punctul de vedere al construcţiei arborelui de analiză, *analyzer*-ele pot fi conectate cu alte *analyzer*-e, pentru clasificarea mai specializată a datelor din şirul de informaţie, sau cu *profiler*-e, pentru a efectua cuantificarea datelor urmărite.

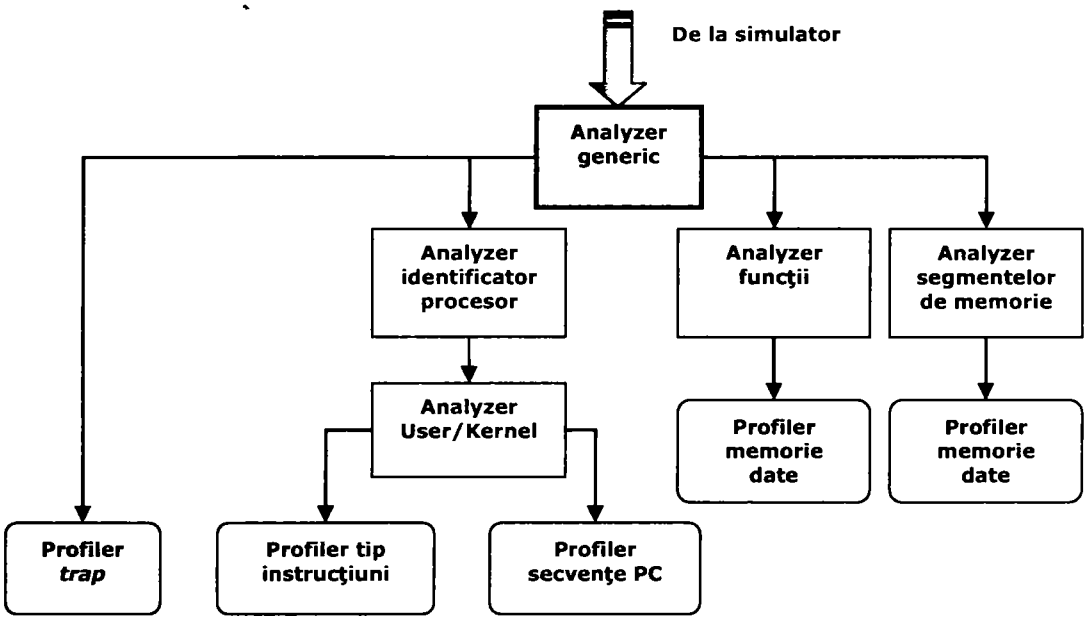


Figura 3.2. Configurarea arborelui de analiză. În figură sunt prezentate posibilităţile de conectarea a analyzer -elor şi a profiler-elor. Analyzer-ele pot fi conectate între ele; profiler-ele se pot conecta numai cu analyzer-e; acelaşi tip de profiler poate fi conectat la analyzer-e diferite.

Fiecare modul din arborele de analiză, execută o subanaliză în funcţie de tipul de informaţie ce îi este transmis de către modulul părinte (modulul ierarhic superior din structura arborelui). În final, fiecare modul, începând cu modulul ce reprezintă rădăcina arborelui, apelează recursiv modulele din sub-arborele propriu, pentru generarea raportului cuprinzând rezultatele analizei efectuate.

Aşa cum a fost configurat arborele de analiză din centru-stânga figurii 3.2. (culoare galbenă), în situaţia sistemelor multiprocesor, va produce pentru fiecare procesor al sistemului, o cuantificare a tipurilor diferite de instrucţiuni întâlnite, atât în codul sursă al programului respectiv, cât şi în sistemul de operare. Prin modul de configurare a ramurii din dreapta a arborelui de analiză (culoare verde) este pusă în evidenţă o altă caracteristică a utilitarului, concretizată în faptul că permite utilizarea aceluiaşi tip de modul, fără a-i fi modificată structura (se utilizează acelaşi tip de *profiler,* conectat la diferite *analyzer*-e*)*, pentru obţinerea unor date diferite.

Arborele de analiză poate fi reconfigurat dinamic, în funcţie de „evenimentele" apărute în informaţia de analizat. În funcţie de evenimentul apărut *analyzer*-ul ataşează un *switch* informaţiei şi în funcţie de acest *switch* se modifică tipul şi/sau profunzimea analizei prin adăugarea sau suprimarea unor ramuri ale arborelui de analiză configurat iniţial. (Profunzimea analizei este direct proporţională cu numărul de parametrii urmăriţi simultan într-o simulare.) Dacă luăm în considerare *analyzer*-ul funcţiilor, (Figura 3.2. culoare verde), acesta are capacitatea de a clona o copie a *profiler*-ului său, care să permită obţinerea de informaţii despre volumul de memorie accesat, separat, pentru fiecare funcţie din codul sursă al aplicaţiei de analizat.

Datorită versatilităţii FAF pot fi create mai multe copii ale arborelui de analiză, fiind posibil în felul acesta să fie executate simultan mai multe tipuri de caracterizări într-o singură simulare.

Una dintre cele mai importante caracteristici ale FAF este accea că structurat sub forma de arbore, poate fi încărcat în interiorul simulatoarelor de performanţă, fapt ce permite evaluarea în detaliu a diferitelor părţi componente ale *pipeline*-ului unui viitor *chip*, precum şi a modului în care acesta interacţionează cu celelalte componente din sistem. În activitatea practică de simulare spre exemplu, este destul de dificilă identificarea funcţiei care contribuie la cele mai multe *miss*-uri în TLB. De asemenea este dificilă găsirea corespondenţei dintre segmentele de memorie ale unei aplicaţii şi modul de comportare al anumitor tipuri de ierarhii ale memoriei *cache*. Prin modul în care a fost conceput, FAF simplifică semnificativ obţinerea acestor informaţii.

FAF este echipat cu toate mecanismele necesare pentru a opera de sine stătător asupra fişierelor *trace* fiind prevăzut şi cu un mecanism care citeşte şi extrage şirurile de informaţii de analizat din acest tip de fişiere.

3.2.4. Inserarea arborelui de analiză în simulatoarele de performanţă.

Conectarea arborelui de analiză în simulatoarele de performanţă se face cu ajutorul unei clase Java, numită *Probe*. Această clasă este structurată sub forma unui tabel *hash,* în care sunt stocate numele diverselor *probe* asociate arborilor de analiză ce urmează a fi înseraţi în simulatorul de performanţă. Pentru a uşura modul de utilizare, clasa *probe* conţine doar trei metode intitulate sugestiv *attach*; *exec* şi *print*. Codul sursă de definire a clasei *Probe* este prezentat în figura 3.3.

```
.....

public class Probe {                          //Declaratia clasei Probe
        public static void attach(String probeName, Analyzer analyzer)
        public static void exec(String probeName, SpartEntity sparcEtity)
        public static void print(String probeName, PrintStream printstream)
}
.....
```

Figura 3.3. Codul sursă al clasei *Probe*.

Metoda ***attach*** ataşează analyzer-ul aflat la rădăcina arborelui de analiza unei probe specificate prin *ProbeName*.

Metoda ***exec*** transmite şirul de date de analizat, (care în exemplul de mai sus este stocat în *SparcEntity*, un câmp specific arhitecturii SPARC [3.13]), arborelui de analiză identificat prin *ProbeName*.

Metoda ***print*** apelează recursiv metodele *print* ale tuturor modulelor componente ale arborelui de analiză ataşat *ProbeName*-ului şi printează datele procesate de acestea.

În figura 3.4. este prezentat un exemplu de cod sursă, extras dintr-un simulator de procesor scris în Java, în care am scos în evidenţă secvenţele de program necesare pentru setarea arborilor FAF în cadrul simulatorului.

```
import faf.*;                                   // Importă clasele FAF
public class ProcessorTest
        public static void main(String[] args) {
        ...
                fafSetup();                     // Metoda ce configurează arborele de analiză
                                                FAF impreună cu probele. Întodeauna este
                                                apelată în funcţia main(). Este prezentată în
                                                detaliu în cele ce urmează
        ...
        }
        public static void fafSetup() {                 // Declară arborele de analiza FAF
                CpuAnalyzer traceCPUAnalyzer = new CpuAnalyzer(params);
                OpcodeCount traceOpcode = new OpcodeCount(params);
                traceCPUAnalyzer.add(traceOpcode);
                traceCPUAnalyzer.init(null);    //Initializeaza arborele de analiza  FAF

        Probe.attach("Trace-Profile", traceCPUAnalyzer, true);    // Ataseaza arborele de
                                                                  analiza FAF unei probe
        ...
```

Figura 3.4. Exemplu de cod sursă pentru modul de înserare a FAF în simulatorul de performanţă.

În ceea ce priveşte apelarea arborilor de analiză în interiorul simulatoarelor de performanţă în punctele de interes, ea se face prin apelarea *metodei exec* a clasei *probe* ce identifică arborele de analiză în cauză, aşa cum este prezentat în figura 3.5.

```
....
Probe.exec("<Probe name>", (SparcEntity)instr);     // Transmite sirul de date de
                                                        analizat arborelui de analiza
                                                        identificat prin <Nume proba>
....
```

Figura 3.5. Exemplu de apelare a arborelui de analiză.

3.3. Posibilităţi de utilizare.

Iniţial noua structură de analiză (FAF) edificată conform celor prezentate anterior a fost utilizată pentru obţinerea unor date de caracterizare pentru diverse aplicaţii ce rulează pe platforme *hardware* existente, în vederea comparări acestor date, cu date obţinute pe platformele aflate în fază de proiect. În prezent are utilizări multiple, putând fi folosit, atât în cadrul simulatoarelor de performanţă cât şi pentru validarea calităţii simulatoarelor funcţionale.

Având în vedere biblioteca de module de care dispune, noua structură de analiză poate fi utilizată pentru:

- analiza fişierelor *trace*:
  - datele statistice obţinute sunt folosite pentru a verifica dacă fişierele *trace*, colectate din simulatoarele funcţionale, reflectă în mod corect execuţia aplicaţiei pe *hardware*-ul existent şi dacă aceste fişiere conţin o zonă reprezentativă pentru execuţia aplicaţiei în cauză.
  - obţinerea de informaţii despre calitatea şi conţinutul fişierelor *trace*, utilizate de către proiectanţii viitoarelor sisteme procesor;
  - studiul modului de accesare a locaţiilor de memorie pentru toată gama aplicaţiilor *software* atât în sistemele uniprocesor cât şi în cele multiprocesor sau CMT;

- în cadrul simulatoarelor de performanţă:
  - pentru maparea diferitelor evenimente (miss-uri în ierarhia de memorie cache, predicţii greşite a instrucţiunilor de *branch*) spre funcţiile din codul sursă care le-au generat;
  - pentru maparea diferitelor evenimente (miss-uri în ierarhia de memorie cache, predicţii greşite a instrucţiunilor de *branch*) spre segmentele de memorie responsabile pentru ele;
  - colectarea de date statistice ce oferă informaţii critice necesare în amplasarea datelor în memoria principală locală a unui procesor sau a altor procesoare în sistemele multiprocesor distribuite.

- pentru validarea rezultatelor generate de simulatoarele de performanţă, în modul de simulare *execution driven.* Problema validării acestor rezultate este deosebit de complexă iar datele obţinute prin utilizarea FAF oferă un mod de validare elegant şi relativ simplu. Distribuţia instrucţiunilor ce modifică starea arhitecturală a unui procesor, executate exclusiv în urma rulării aplicaţiei utilizator, trebuie să fie identică atât în fişierele de *trace* cât şi în cadrul simulatoarelor de performanţă ce execută în modul *execution driven,* atâta timp cât este executată aceaşi porţiune de cod. FAF permite obţinerea distribuţiei instrucţiunilor în ambele tipuri de simulare.

Pentru confirmarea veridicităţii datelor furnizate de FAF s-au efectuat o serie de rulări test pentru un număr larg de aplicaţii din diverse domenii software, folosind simulatoare funcţionale ce emulează o platformă *hardware* şi un sistem de operare existente. Pe această structură emulată au fost analizate fişiere *trace* cu ajutorul FAF, obţinâdu-se o serie de date de caracterizare ca de exemplu: distribuţia instrucţiunilor, numărul de apeluri ale sistemului de operare, numărul de *trap*-uri etc. Pe aceiaşi platformă *hardware* reală de data acesta (în cazul anterior simulată), au fost colectate date similare obţinute cu ajutorul regiştrilor de performanţă, cu care este echipat procesorul, sau cu ajutorul utilitarelor puse la dispoziţie de sistemul de operare rulat. În urma comparării datelor obţinute s-a constatat o concordaţă perfectă în cele două cazuri, validându-se în acest fel veridicitatea datelor furnizate de FAF, precum şi multiplele lui posibilităţi de utilizare.

3.4. Exemple de utilizare.

În continuare, voi prezenta două exemple de utilizare precum şi rezultatele obţinute cu FAF, în două situaţii: folosirea FAF în cadrul simulatorului de performanţă, şi folosirea utilitarului pentru analiza fişierelor *trace.*

Rezultatele prezentate în cele ce urmează, au fost obţinute pe o platformă Solaris? UltraSparc III, în urma rulări unor *benchmark*-uri ce fac parte din suita SPEC (Anexa 2).

3.4.1. Utilizarea FAF în cadrul simulatoarelor de performanţă.

Scopul analizei este evidenţierea abilităţii FAF de a fi înserat în cadrul simulatorului de performanţă, precum şi capacitatea acestuia de a genera date despre multiplele caracteristici ale aplicaţiei, într-o singură simulare.

În figura 3.6. este prezentat un exemplu al modului în care FAF este inserat în simulatorul de performanţă pentru obţinerea unor date despre evenimentele urmărite în interiorul unui *pipeline* şi pentru a scoate în evidenţă capacitatea utilitarului de *map*-are a evenimentelor importante cum sunt, predicţia greşită a instrucţiunilor condiţionale şi *miss*-urile în memoriile *cache,* înapoi spre codul sursă care le-a generat.

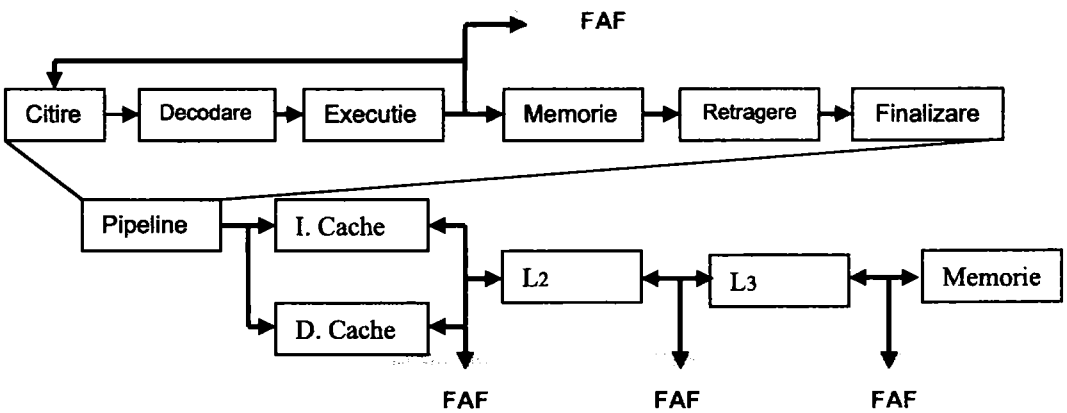


Figura 3.6 Integrarea arborelui de analiză într-un simulator de performanţă.

Configurarea simulatorului de performanţă utilizat pentru generarea datelor, este prezentată în tabelul 3.1.

Caracteristici	Valoare
Fetch/Xlate/Decode/Issue/Commit	16/8/8/8/8
Lungime *pipeline*	6 etape
Ciclii executie: alu/imul/idiv/branch/fmul/fdiv	1/8/40/1/8/30
Load Buffer – Store Buffer	16 intrari : 16 intrari, 2 bancuri
Predictor instr. condiţionale: Stiva adrese revenire	Gshare, 14 bit istorie,16 K intrari: 32 intrari
TLB & DTLB	64 intrari, 4 cai, LRU
Memorie cache instrucţiuni	64 KB, 4 cai, LRU, 64B linii, 2 ciclii. timp de acces
Memorie cache date	64 KB, 4 cai, LRU, 64B linii, 4 ciclii. timp de acces
Memorie L2 Cache	1 MB, 4 cai, LRU, 64B linii, 20 ciclii. timp de acces,WB
Memorie L3 Cache (în afara chip-ului)	16 MB, 16 cai, LRU, 128B linii, 80 ciclii timp de acces,WB
Memorie principala	300 ciclii timp de acces

Tabelul 3.1. Configurarea simulatorului de performanţă în care este integrat FAF.

Caracteristicile urmărite vor fi: cele mai frecvent executate funcţii; procentul de *miss*-uri în toate nivelurile memoriei *cache*; predicţiile greşite ale instrucţiunilor de control condiţionale cauzate de funcţiile cele mai frecvent executate.

Arborele de analiză utilizat pentru a obţine datele dorite are structura din figura 3.7.

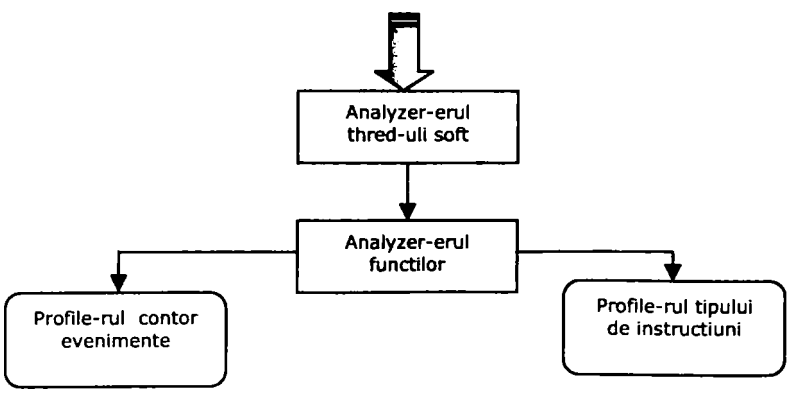


Figura 3.7 Configurarea FAF când lucrează în simulatorul de performanţă.

Modulele componente sunt:

- ***analyzer*-ul *thred*-ului soft;** are rolul de a clasifica informaţia de analizat în funcţie de identificatorul *thread*-ului respectiv (programele actuale pot avea mai multe *thread*-uri soft, ce sunt executate în paralel, fiecare având un identificator unic).
- ***analyzer*-ul funcţiilor;** are rolul de a extrage PC-urile din şirul de informaţii de analizat şi a face conexiunea cu funcţia de provenienţă din codul sursă, atât pentru programul utilizator, cât şi pentru sistemul de operare. *Analyzer-ul* permite *map*-area anumitor evenimente cum ar fi: *miss* în TLB, *miss* în diferitele memorii *cache*, sau predicţia greşită a instrucţiunilor de control condiţionale spre codul sursă, responsabil pentru generarea lor. Asemeni *analyzer*-ului segmentelor de memorie acest *analyzer* citeşte un fişier de configurare ce conţine informaţie extrasă din tabelul de simboluri al aplicaţiei respective, şi care permite *map*-area codului sursă al fiecărei funcţii, către un interval de adrese din memoria virtuală a sistemului *hardware* simulat.
- ***profiler*-ul tipului de instrucţiuni;** rolul acestuia este acela de a decodifica fiecare instrucţiune şi a contoriza de câte ori a fost întâlnit în timpul procesării, un anumit tip de instrucţiune.
- ***profiler*-ul contor evenimente;** identifică şi cuantifică numărul de evenimente de un anumit tip în *pipeline.*

În figura 3.8. sunt prezentate, pentru opt programe de testare din suita SPEC, cele trei evenimente luate în considerare: topul celor mai frecvent executate funcţii, procentul de *miss*-uri în memoria *cache* şi predicţiile greşite ale instrucţiunilor de control condiţionale, cauzate de funcţiile cele mai frecvent executate.

Analizând cele prezentate în figură se pot face următoarele observaţii:

a) în cazul rulării aplicaţiei *mcf* numărul total de instrucţiuni executate în funcţia ***primal_start_artificial()*** poate fi considerat nesemnificativ în raport cu numărul total de instrucţiuni executate de aplicaţie, (ele reprezintă aproximativ 1%); datele de analiză arată însă că această funcţie este responsabilă pentru mai mult de 90% din totalul *miss*-urile în memoria *cache* de la nivelul L3.

Această constatare ne permite să conchidem că funcţia ***primal_start_artificial()*** este unul din principalii factori responsabili pentru o performanţă scăzută în execuţia aplicaţiei respective;

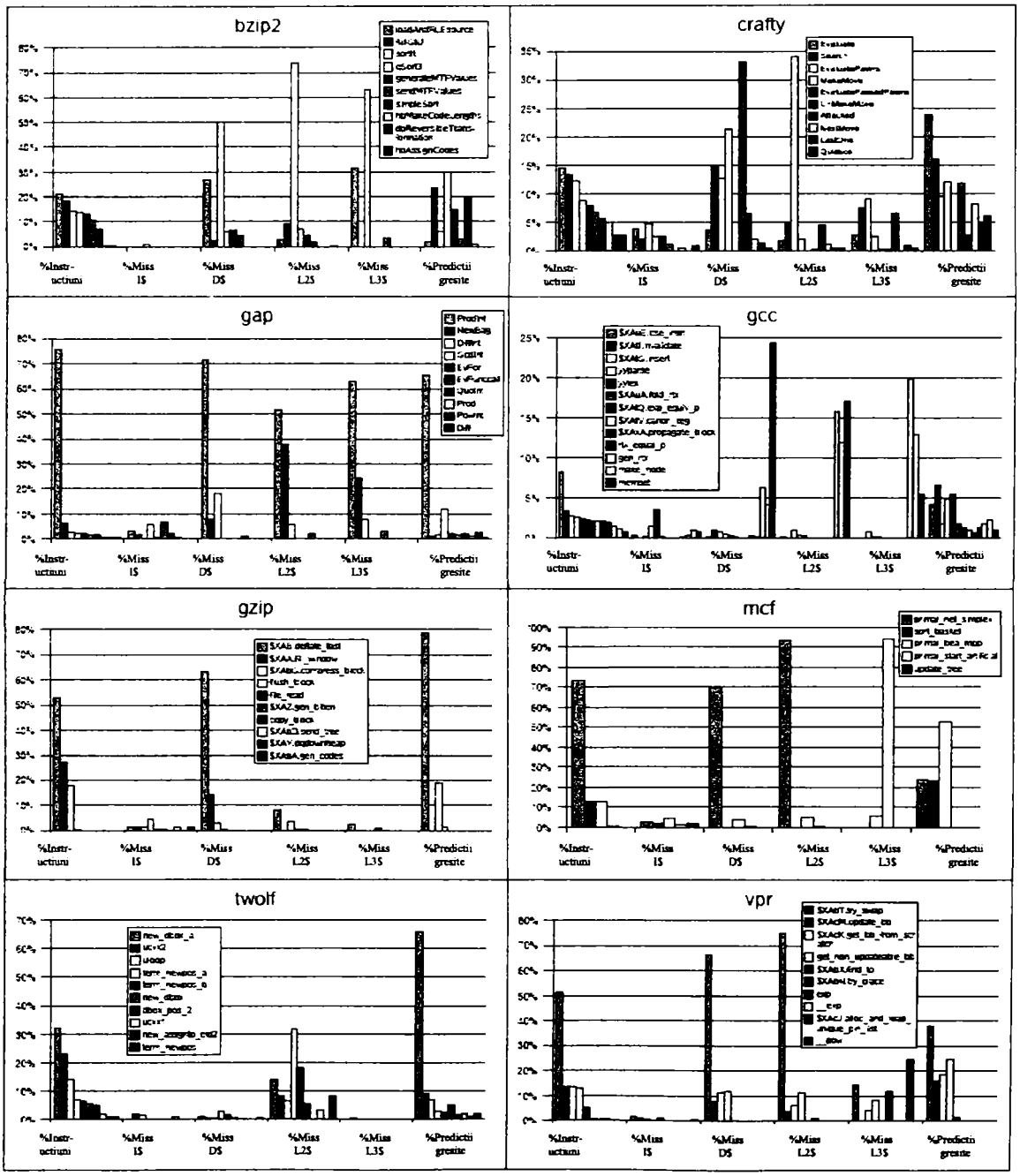


Figura 3.8. Analiza evenimentelor din pipeline la nivelul funcţiilor din codul sursă. Setul de bare din extrema stânga reprezintă distribuţia instrucţiunilor. Celelalte seturi de bare reprezintă contribuţiile fiecărei funcţii la numărul total de miss-uri în memoriile cache de instrucţiuni, date, L2 si L3 precum şi predicţiile greşite ale instrucţiunilor de control.

din predicţiile greşite ale instrucţiunilor de control condiţionale;

c) în cazul programului ***bzip2;*** funcţia ***sortIt()*** totalizează doar 14% din numărul total de instrucţiuni executate şi cauzează o mare parte din traficul spre memorie;

d) în cazul programului ***gcc*** trei funcţii ce totalizează mai puţin de 4% din numărul total de instrucţiuni ***get_rtx( ); make_node( ); memset( );*** sunt responsabile pentru majoritatea *miss*-urilor în ierarhia de memorie.

Informaţii de genul celor prezentate în figura 3.8, sunt utilizate de:

- proiectanţii noilor microprocesoare, deoarece:
  - pot să înţeleagă ce tipuri de funcţii vor fi executate cu performanţe reduse, pe platformele proiectate de ei ;
  - permit luarea unor decizii în cunoştinţă de cauză despre necesitatea adăugării de structuri *hardware* noi, sau despre modificarea celor existente.
- proiectanţii de *software* şi compilatoare, deoarece:
  - în absenţa platformei hardware fizice pot demara acţiunile de îmbunătăţire a acestora pentru a putea obţine performanţe maxime pe platformele harware noi încă din faza de început a proiectării, putând în felul acesta să influenţeze deciziile legate de structura microarhitecturii proiectate.

3.4.2. Utilizarea FAF pentru analiza fişierelor *trace*.

Scopul analizei prezentate este obţinerea de date despre modul în care instrucţiunile specifice diferitelor aplicaţii accesează diversele segmente de memorie.

Spaţiul memoriei virtuale alocat unei anumite aplicaţii de către sistemul de operare Solaris, cuprinde cel puţin patru segmente de memorie [3.14] segmentul text, segmentul date, spaţiul *heap* şi spaţiul *stack*. Segmentele text şi date sunt *map*-ate direct din executabilul aflat pe disc şi conţin instrucţiunile ce urmează a fi executate, precum şi variabilele iniţializate în aplicaţia testată. Zona de *heap* este o zona de memorie alocată dinamic, utilizată ca spaţiu de „ciornă" de către aplicaţie în timpul execuţiei şi poate creşte la cererea acesteia. Când o funcţie este apelată în timpul execuţiei unui program, *stack*-ului îi este adăugată o fereastră ce conţine PC-ul, argumentele funcţiei şi variabilele locale. Asemeni segmentului de *heap* zona alocată *stack*- ului creşte la cererea aplicaţiei.

Datorită faptului că segmentele de memorie, deservesc diferite scopuri pe parcursul execuţiei unui program, modul lor de accesare poate fi foarte diferit.

Pentru obţinerea datelor configurarea arborelui de analiză este prezentat în figura 3.9.

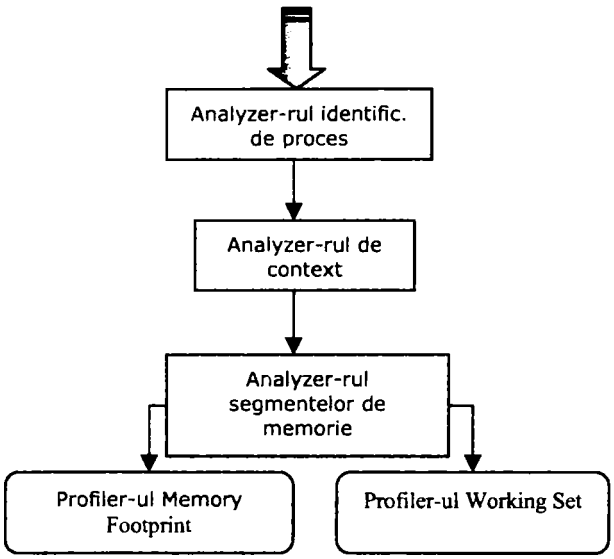


Figura 3.9. Configurarea FAF pentru analiza fisierelor *trace.*

Modulele utilizate sunt următoarele:

- ***analyzer-ul* de procesor;** este utilizat în sistemele multiprocesor şi are rolul de a decodifica şirul de informaţii, pe care le separă şi le clasifică în funcţie de identificatorul aferent procesorului care le-a generat.
- ***analyzer-ul* de context;** este utilizat pentru a extrage identificatorii de context din şirul de informaţie de analizat, pe care o clasifică în funcţie de aceşti identificatori. În acest fel este posibilă caracterizarea, în parte, a fiecărui proces ce rulează în paralel (unele sisteme de operare permit mai multor programe să ruleze în paralel şi pentru a le distinge le atribuie identificatori unici de context).
- ***analyzer-ul* segmentelor de memorie;** împarte şirul informaţiilor de analizat în funcţie de segmentul de memorie virtuală pe care acestea le apelează (*stack, heap, text, data, library,* etc.). Utilitarul permite analiza atât a segmentelor de memorie alocate programului utilizator cât şi cele ale

sistemului de operare. Modulul oferă o modalitate de corelare a diferitelor evenimente cum sunt *miss* în TLB sau în memoriile cache, cu segmentul de memorie responsabil pentru aceste evenimente. Pentru caracterizare analyzer-ul de segment citeşte un fişier de configurare ce conţine mapările segmentelor de memorie ale tuturor proceselor active în sistem la un moment dat.

- ***profiler*-ul *memory footprint***; Decodifică adresa fizică de accesare a memoriei, pentru citirea fiecărei instrucţiuni şi raportează informaţii despre creşterea în timp a cantităţii de memorie accesate în urma execuţiei respectivului set de instrucţiuni. (Cantitatea de memorie accesată, reprezintă numărul total de cuvinte unice accesat de o anumită secvenţă de instrucţiuni).
- ***profiler*-ul *working set***; Decodifică adresa fizică de memorie accesată de instrucţiunile de citire/scriere din memorie şi raportează informaţii despre cantitatea de memorie accesată în timp, de o anumită secvenţă de instrucţiuni.

În figura 3.10. am prezentat rezultatele obţinute pentru fiecare segment de memorie în parte.

În urma analizei făcute se pot face următoarele aprecieri:

a) Segmentele *date*, *heap* şi *stack,* prezintă caracteristici de creştere diferite, fapt ce nu ar fi putut fi decelat dacă le-am fi analizat împreună. De exemplu segmentul de *heap* în ***crafty*** nu creşte semnificativ, în schimb aplicaţia accesează în mod constant date noi din segmentul date.

b) Pentru citirea instrucţiunilor din segmentul text, în cazul tuturor aplicaţiilor de testare studiate, se remarcă o creştere în trepte a volumului de memorie accesat, excepţie făcând zona de memorie alocată *stack*-ului care rămâne constantă.

Acest lucru indică faptul, că deşi aplicaţiile urmează direcţii de execuţie diferite, numărul maxim de apeluri de proceduri, este atins încă din faza de început a execuţiei.

c) Aplicaţia ***gap*** are lanţul cel mai lung de apeluri de proceduri, pe când ***gcc***, din punctul de vedere al instrucţiunilor unice executate, utilizează cea mai mare zona de memorie.

d) Caracteristicile de comportare a programelor ***bzip2*** şi ***gzip*** sunt diferite, deşi fac parte din aceeaşi categorie a programelor de testare (programe de comprimare): ***bzip2*** foloseşte un segment mai mare de *heap* şi date decât ***gzip.*** În ***bzip2*** aceste segmente cresc în etape, iar ***gzip*** atinge zona de execuţie stabilă relativ repede.

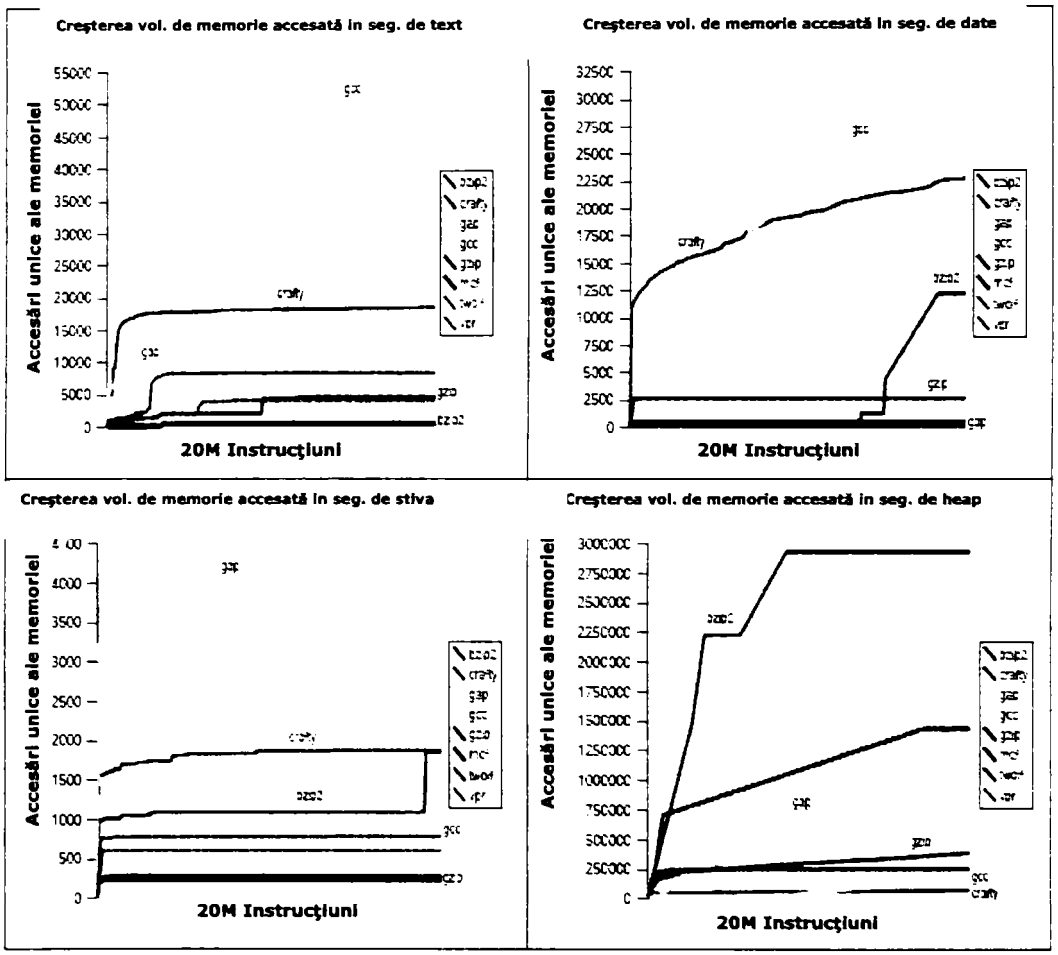


Fig. 3.10. Creşterea volumului de memorie accesată de un subset al aplicaţiilor SPEC CPU2000

Date statistice de acest tip sunt folosite cu dublu scop: pe de o parte pentru a verifica dacă fişierele trace colectate din simulatoarele funcţionale reflectă în mod corect execuţia aplicaţiei pe *hardware*-ul existent, iar pe de altă parte pentru a verifica faptul că în aceste fişiere a fost cuprinsă o zona reprezentativă pentru execuţia aplicaţiei respective.

3.5. Concluzii.

"Flexible Analysis Framework", materializat printr-un utilitar de analiză, a fost edificat ca urmare a necesităţii unei structuri de analiză mai flexibilă, care să elimine o parte din dezavantajele rezultate în urma observaţiilor făcute asupra unor utilitare folosite pentru aprecierea performanţei microprocesoarelor, precum şi a modului în care această activitate se desfăşoară practic.

În contextul larg al structurilor de analiză utilizate în prezent, FAF se individualizează prin faptul că are următoarele avantaje:

a) Structura propusă este proiectată într-o manieră modulară, fapt ce îl face uşor de înţeles şi utilizat, deoarece atât pentru a o folosi cât şi pentru a o extinde, utilizatorul trebuie să înţeleagă doar o mică parte din modul de organizare şi codul sursă al acesteia. Datorită modului în care a fost concepută poate fi adaptată, atât pentru analiza activităţii sistemului de operare, cât şi pentru analiza unui spectru foarte larg de aplicaţii, inclusiv ştiinţifice şi comerciale.

b) Utilizând un număr limitat de module permite o gamă variată de tipuri de analiză şi obţinerea unor date diverse, într-o singură simulare, fară ca structura principală a simulatoarelor şi programelor analizate să fie modificată. Datorită acestui fapt timpul necesar obţinerii datelor referitoare la analiza dorită, este micşorat semnificativ.

O abilitate importantă a noii structuri este faptul că permite obţinerea informaţilor despre modul de rulare al programelor de testare şi evaluare a performanţei, atât pe platformele existente cât şi pe cele în fază de proiect.

c) Modulele care alcătuiesc utilitarul au fost proiectate pentru a putea fi folosite, fară modificări , atât pentru analizarea fişierelor de urmărire a execuţiei, cât şi ca parte componentă a unor simulatoare de performanţă, sau a ierarhiei de memorie. Pe langă aceste caracteristici ale utilitarului în cauză, mai trebuie menţionat şi faptul că folosirea lui permite atât proiectanţilor de *hardware* şi celor de *software*, să schimbe informaţii despre modul în care aplicaţiile vor rula pe platformele proiectate, încă în fazele incipiente ale proiectării.

"Flexible Analysis Framework" structurat sub formă de arbore, poate fi încărcat în interiorul simulatoarelor de performanţă, în acest fel este posibilă analiza detaliată a „evenimentelor" din ierarhia de memorie, sau din *pipelin* şi *map*-area acestora la structurile din codul sursă precum şi analiza modului în care acesta interacţionează cu celelalte componente ale sistemului; în aceiaşi măsură în care permite caracterizarea la nivel înalt a modului de funcţionare al bazelor de date,din aplicaţii comerciale.

Atât avantajele, cât şi multiplele posibilităţi de utilizare a "FAF" au fost dovedite prin exemplele de utilizare prezentate, iar utilitatea lui prin natura diversă a informaţilor (unele dintre ele nu pot fi obţinute prin utilizarea mijloacelor tradiţionale de analiză) şi a modului facil în care pot fi obţinute.

Diversitatea mare a posibilităţilor de utilizare, modul uşor în care acestea pot fi obţinute prin utilizarea FAF reduce semnificativ, sau elimină chiar, necesitatea extrapolării comportării diverselor aplicaţii pe platformele *hardware* existente, deoarece datele pot fi obţinute, direct din simulatoarele de performanţă ale noilor arhitecturi, reducându-se semnificativ atât ciclul de proiectare al *hardware*-ului în sine, cât şi al *software*-ului ce urmează să ruleze pe respectivele platforme.

Testele efectuate pentru a verifica şi totodată pentru a pune în evidenţă calităţile utilitarului, au dovedit multiplele lui posibilităţi de utilizare, precum şi versatilitatea acestuia.

Contribuţii:

1. Introducerea unui nou concept, materializat printr-un utilitar *"Flexible Analysis Framework"* (FAF), destinat evaluării nivelului de performanţă a microprocesoarelor.
2. Îmbunătăţirea condiţiilor de lucru în simulare prin faptul că noul utilitar (FAF), având o structură modulară este uşor de înţeles şi utilizat.
3. Îmbunătăţirea tehnicilor de simulare cu un utilitar ieftin şi non invaziv.
4. Îmbunătăţirea tehnicilor de simulare cu un utilitar care permite o gamă variată de tipuri de analiză şi obţinerea unor date diverse, într-o singură simulare.

5. Îmbunătăţirea tehnicilor de simulare cu un utilitar care permite obţinerea informaţilor despre modul de rulare al programelor de testare şi evaluare a performanţei, atât pe platformele existente cât şi pe cele în fază de proiect, ca parte a simulatoarelor de performanţă, sau a ierarhiei de memorie, cât şi pentru analiza fişierelor *trace*.
6. Demonstrarea modului de utilizare al utilitarului *"Flexible Analysis Framework"* şi a diverselor abilităţi ale acestuia, prin efectuarea unor simulări în condiţii concrete.
7. Conceperea şi aplicarea unei metode de validare a rezultatelor generate de simulatoarele de performanţă când rulează în modul de simulare *execution driven*.

Conceptul FAF şi structura de analiză realizată pe baza acestui concept, este rezultatul conlucrării autorului tezei cu alţi doi colaboratori. În cadrul acestei colaborări contribuţia personală a constat in urmatoarele:

1. Participarea la crearea conceptului FAF.
2. Participarea la proiectarea şi implementarea *analyzer*-ului segmentelor de memorie, *analyzer*-ul de procesor şi a celui de context, a *profiler*-elor ce efectuează analiza localizării accesărilor memoriei şi a altor profilere ca de exemplu cel ce efectuează analiza claselor de instrucţiuni.
3. A proiectat şi implementat individual, *analyzer*-ul funcţiilor şi *profiler*-ul ce contorizează diverse evenimente în *pipeline* precum şi alte *profiler*-e cum este de exemplu profiler-ul ce efectuează analiza distribuţiei instrucţiunilor, cel ce verifică corectitudinea secvenţei PC-urilor şi cel ce analizează alternarea între instrucţiunile de citire/scriere din memorie.
4. Crearea şi implementarea setului de programe necesare obţinerii informaţiilor legate de maparea în memoria virtuală a proceselor şi
5. extragerea adreselor de început şi sfârşit a funcţiilor codului sursă al aplicaţiei.
6. Generarea şi validarea fişierele *trace* care au stat la baza obţinerii datelor experimentale referitoare la modul în care instrucţiunile specifice diferitelor aplicaţii accesează diversele segmente de memorie precum şi generarea rezultatelor referitoare la analiza fişierelor *trace*.

7. Conceperea, realizarea şi integrarea mecanismului prin care arborele de analiză poate fi încorporat în simulatoarele de performanţă.
8. Generarea întregului set de rezultate prezentate în lucrare, referitoare la maparea diferitelor evenimente din pipeline spre funcţiile din codul sursă care le-a generat.

4. Studiu comparativ al particularităţilor de comportare a unor aplicaţii, care implică un regim computaţional intensiv.

4.1. Introducere.

Caracterizarea modului în care diverse aplicaţii (reale, sau de test), rulează pe o anumită platformă *hardware;* înţelegerea detaliilor intime legate de structura şi comportamentul acestora, este una din componentele importante ale activităţii de proiectare a microprocesoarelor [4.1]. Datele obţinute în urma acestei activităţi sunt utilizate pentru:

- luarea unor decizii pertinente referitoare la necesitatea sau oportunitatea adăugării, şi/sau alocării unor mijloace *hardware* noi;
- evaluarea nivelului de utilizare a resurselor din sistem, fapt ce permite luarea măsurilor ce se impun pentru efectuarea optimizărilor, atât la nivelul *hardware*-ului cât şi la nivelul *software*-ului;
- identificarea motivelor datorită cărora anumite aplicaţii rulează cu o viteza scăzută pe anumite structuri *hardware.*

Motivarea studiului.

Modul de comportare în rulare a diverselor aplicaţii este o topică des abordată în literatura de specialitate, însă făcând o analiză a bibliografiei din domeniu se constată următoarele aspecte:

- sunt rare cazurile în care problema este tratată în detaliu iar în general sunt abordate doar o anumită categorie de aplicaţii;
- sunt tratate numai unele din aspectele comportamentale ale aplicaţiilor;
- adesea abordarea problemei se face doar prin prisma unei singure caracteristici de interes;

- în multe cazuri datorită unei infrastructuri limitate ca posibilităţi, nu a fost studiată întreaga gamă de valori pe care le poate lua un anumit parametru;
- sunt puţine studii comparative privind comportarea unor aplicaţii care reclamă un regim computaţional dificil, aşa cum sunt aplicaţile din domeniul ştiinţific sau cele din domeniul comercial;

Toate cele prezenate anterior au costituit tot atâtea motive ce au condus la ideea realizării unui studiu detaliat, asupra unor caracteristici de performanţă pentru o gamă largă de aplicaţii, care să ofere o imagine de ansablu asupra comportamentului acestora.

Abordarea problemei în acest mod, face ca prezentul studiu să se individualizeze în contextul studilor legate de analiza performanţei, aducându-şi contribuţia în acest domeniu, prin obiectivele ce şi le propune.

Obiectivele studiului:

1. Efectuarea unei analize cuprinzătoare şi aprofundată asupra caracteristicilor computaţionale ale unor categorii de aplicaţii care din punctul de vedere al procesorului presupun un regim computaţional dificil, cum sunt:

- *Aplicaţii din domeniul ştiinţific*, caracterizate prin proprietatea de utilizare exhaustivă a unităţilor de calcul. (S-au luat în considerare atât *benchmark*-uri, cât şi aplicaţii reale).
- *Aplicaţii din domeniul comercial,* care se caracterizează printr-un număr ridicat de *thread*-uri soft şi prin procesarea unui volum foarte mare de date.

2. Obţinerea unor date comparative despre caracteristici computaţionale şi de performanţă ale acestor aplicaţii şi punerea în evidenţă a similitudinilor şi diferenţelor dintre ele.

Studiul efectuat oferă date despre variate caracteristici de performanţă, pentru aplicaţile din domeniile ştiinţifice şi comerciale, cum sunt: caracteristicile de accesare a memoriei; reutilizarea locaţiilor de memorie; sensibilitatea aplicaţiilor la structura ierarhiei de memorie; modul de reutilizarea a datelor între diferitele

"*thread*-uri" soft în execuţia variantelor paralel a programelor studiate şi analiza avantajelor utilizării unor mecanisme de precitire.

Pentru aplicaţiile complexe ce au un mod neregulat de execuţie vor fi prezentate rezultate pentru una sau mai multe din fazele cele mai importante ale execuţiei.

Generarea datelor vizate presupune o activitate laboriosă şi o infrastructură elaborată, mai ales în ceea ce priveşte aplicaţiile din domeniul comercial *TPC-C* şi *SAP,* al căror studiul este dificil datorită multiplelor resurse necesare simulării.

Rezultatele prezentate au fost obţinute folosind versiuni nemodificate ale aplicaţiilor amintite; compilarea lor făcându-se folosind optimizări standard. În acest fel executabilele studiate sunt reprezentative pentru modul în care ele sunt utilizate în practică.

Cercetările efectuate şi rezultatele obţinute au constituit obiectul unei lucrări prezentată, (împreună cu alţi colaboratori), la *21st ACM International Conference on Supercomputing.* June 16–20, anul 2007 Seattle, Washington, USA [4.2].

4.2. Caracterizarea aplicaţiilor în literatura de specialitate, mijloace şi metodologie de analiză.

Principalul suport pentru efectuarea prezentului studiu sunt informaţiile obţinute în urma consultării bibliografiei de specialitate. Studiul documentar efectuat a permis decelarea caracteristicilor computaţionale importante, definitorii pentru performanţa realizată în procesarea diverselor aplicaţii, precum şi tehnicile pentru evidenţierea acestor informaţii, tehnici devenite standard, fiind general acceptate de comunitatea producătorilor de tehnică de calcul. Aceste studii sunt bazate, în principal pe rezultatele obţinute din citirea regiştrilor de performanţă puşi la dispoziţie de către *hardware*, dar şi pe procesarea fişierelor de *trace,* colectate în urma executării acestor aplicaţii.

- **Caracteristici definitorii pentru stabilirea nivelului de performanţă.**

Aşa cum rezultă din literatura de specialitate principalele caracteristici cu implicaţii majore asupra performanţei în rulare a unei aplicaţii sunt următoarele:

- ✓ **Distribuţia instrucţiunilor.**

In mod similar cu studiul distribuţiei instrucţiunilor bazat pe fişierele *trace* pe care îl prezint, Rupnow şi alţii [4.3] au prezentat date legate de distribuirea instrucţiunilor pentru programele SPEC-FP (virgulă mobilă). Lucrarea amintită scoate în evidenţă de asemenea importanţa instrucţiunilor cu întregi în programele ştiinţifice.

Lucrarea de faţă extinde unele dintre ideile prezentate în lucrarea precedentă şi totodată trasează o paralelă între aplicaţiile ştiinţifice şi cele comerciale.

✓ **Caracteristicile de accesare a memoriei.**

În mod obişnuit pentru efectuarea caracterizărilor, s-au folosit metode standard de analiză a caracteristicilor de accesare a memoriei, aşa cum au fost descrise în [4.4]. In această lucrare autorii descriu etapele parcurse în analiza modului în care aplicaţiile paralele accesează memoria. Analiza este efectuată folosind simulatoare ale ierarhiei de memorie ce utilizează fişiere de trace colectate cu ajutorul unei structuri ce permite rularea concomitentă a *thread*-urilor aplicaţiilor paralelizate.

✓ **Regularitatea accesării memoriei.**

Evaluarea regularităţii accesării memoriei poate fi efectuată prin analizarea distanţei între două accesări consecutive, aşa cum a fost prezentată în [4.5]. Un aspect interesant al modului de utilizare a memoriei de către diverse aplicaţii, aspect ce a fost studiat în amănunt, este localizarea temporală şi spaţială a accesărilor memoriei, prezentate în [4.6], [4.7], [4.8], [4.9] şi [4.33]. În [4.6] se efectuează un studiu al localizării accesărilor memoriei în aplicaţii comerciale ce accesează baze de date dându-se ca exemplu trei moduri diferite de accesare specifice aplicaţiei TPC-D [4.10].

În [4.7] se introduce o metodă de evaluare a caracteristicilor în care o anumită aplicaţie, cu precădere din domeniul ştiinţific, accesează memoria. Metoda de analiză se doreşte a fi independentă de structura arhitecturală a sistemului unde este rulată aplicaţia. Aceste concepte sunt combinate cu metodele de mapare implementate în APEX (proiect de studiu al caracteristicilor de performnţă ale aplicaţiilor) descris în [4.11]. (Scopul APEX este acela de a crea aplicaţii sintetice, bazat în principal pe o serie de parametri rezultaţi în urma evaluării proprietăţilor de accesare a datelor, de către aplicaţiile reale studiate, care reflectă întocmai nivelul de performanţă în execuţie ale acestora din urmă pe anumite platforme *hardware*).

Această metodologie permite analiza localizării temporale şi spaţiale a accesărilor memoriei de către o aplicaţie independent de platforma *hardware* pe care această este rulată, aşa cum este prezentat [4.7].

✓ **Interacţiunea cu memoria *cache*.**

Alt domeniu cercetat intens este cel al modului în care aplicaţiile din diversele domenii software interacţionează cu ierarhia de memorie *cache*. In [4.12] sunt prezentate o serie de aspecte legate de memoriile cache; rata de miss, efectele creşterii memoriei *cache* asupra ratei de *miss*, algoritmi de scriere şi de actualizare a datelor, menţinerea corectitudinii stocării datelor în sistemele multiprocesor, memorii *cache* distribuite de date şi instrucţiuni etc.. Sunt prezentate totodată o serie de date experimentale legate de aceste aspecte obţinute în urma unor simulări *trace driven*. Rezultate de genul celor prezentate în [4.12] pot fi generate în urma utilizării unor simulatoare a ierarhiei de memorie *cache*, ce rulează atât *trace driven* cât şi *execution driven*.

Pe lângă cele menţionate, literatura de specialitate oferă o serie de alte metode analitice de analiză, cum ar fi *Cache Miss Equations* [4.13], care se referă la analiza în faza de compilare a modului în care aplicaţia va interacţiona cu memoriile *cache*. Autorii identifică o serie de relaţii între indecşii *loop*-urilor încuibate, dimensiunile masivelor, localizarea datelor, parametrii memoriilor cache si *miss*-urile în memoria cache, cauzate de rularea *loop*-ului respectiv în vederea generării unui set de ecuaţii matematice ce sunt reprezentative pentru toate *miss*-urile dintr-un *loop*. Rezolvarea acestor ecuaţii poate ajuta compilatoarele să genereze executabile ce utilizează mai eficient memoriile *cache* sau poate reduce marja de eroare în evaluările de performanţă.

✓ **Nivelul de utilizare a memoriei cache.**

O metodologie, similară cu cea a caracterizării accesării memoriei, este analiza dinamică a memoriei fizice, accesată de o aplicaţie aflată în execuţie. Această metodă a fost introdusă în [4.14], [4.15] şi se referă atât la analiza utilizării memoriei *cache,* cât şi la analiza algoritmilor de paginare a memoriei. În [4.16] sunt propuse metode de observare în timp real a comportamentului unei aplicaţii în

scopul balansării utilizării resurselor (memoria) unui sistem. Sunt scoase în evidenţă metode de management a memoriei în funcţie de numărul minim de pagini ce trebuiesc a fi stocate în memorie la un moment dat (*working set*) pentru a asigura funcţionarea eficientă a unei aplicaţii.

✓ **Precitirea datelor**

În multe cazuri efectul negativ al *miss*-urilor în memoria *cache,* poate fi combătut prin mecanisme de precitire a datelor. Această metodă se dovedeşte foarte eficientă în cazul aplicaţilor ce accesează datele în mod regulat, având un efect mai scăzut, sau nul, în aplicaţiile ce au caracteristici neregulate, sau impredictibile, de accesare a memoriei. Eficienţa mecanismelor de precitire a datelor, coordonate de compilatoare este prezentată în [4.17]. Aici este introdusă o metodă de identificare a momentului de timp cel mai propice în execuţia unui program, în care datele pot fi precitite şi sunt evaluate beneficiile mecanismelor de precitire, asupra vitezei de execuţie a unui număr larg de subrutine componente ale unor aplicaţii ştiinţifice.

Alte studii legate mai ales de impactul precitirii instrucţiunilor asupra creşterii vitezei de execuţie, în special a aplicaţiilor comerciale, au fost prezentate în [4.18]. Aici sunt evaluate efectele asupra creşterii performanţei a unor mecanisme de precitire hardware tradiţionale; este introdus un mecanism de precitire nou destinat rezolvării *miss*-urilor cauzate de instrucţiuni nesecvenţiale şi este studiat impactul unui mecanism hibrid de precitire, secvenţial plus nesecvenţial, asupra aplicaţiilor comerciale.

✓ **Utilizarea aceluiaşi set de date într-un mediu multiprocesor.**

Pentru aplicaţiile ce rulează în paralel, cantitatea şi tipurile de reutilizări de date poate avea implicaţii semnificative asupra *coherency trafic* şi totodată un impact puternic asupra performanţei în execuţie a sistemelor multiprocesor. O serie de rezultate legate de această problemă au fost prezentate în [4.19] şi [4.20]. În [4.19] este studiat modul în care aplicaţiile paralele accesează locaţiile de memorie comune în sistemele multiprocesor fiind totodată evaluate efectele acestor tipuri de accesări asupra ratelor de *miss* şi a traficului pe magistrală.

În [4.20] este prezentat un studiu al efectului diferitelor mecanisme *hardware* componente ale unui microprocesor *out-of-order* din familia Pentium Pro asupra performanţei în execuţie a unor aplicaţii comerciale de tip OLTP (*Online*

Transaction Processing). Evaluările sunt bazate pe interpretarea datelor contorizate de regiştrii *hardware* de evaluare a performanţei. Parte din studiu este prezentat modul în care ratele de *miss* în memoria *cache* sunt influenţate de structura şi modul de organizare al acesteia.

4.3. Aplicaţii analizate.

Realizarea studiului propus s-a făcut luând în considerare atât programe standard, general valabile, utilizate în stabilirea performanţei, cât şi aplicaţii numerice reale, de calcul intensiv. Comportamentul diferit al aplicaţiilor luate în considerare la efectuarea studiului, a permis examinarea şi compararea, pe de o parte a unor aplicaţii ce solicită doar anumite parţi ale sistemului computaţional cum ar fi de exemplu memoria, sau unităţile de execuţie, iar pe de altă parte au fost studiate aplicaţii ce solicită în mod echilibrat toate componentele sistemului.

Cu scopul de a face o analiză cât mai completă, s-a urmărit comportarea şi s-au comparat aplicaţii din următoarele domenii:

a) HPC Challenge Benchmarks [4.21]. Este o suită de *kernel*-uri standard de testare destinate evaluării performanţei arhitecturilor HPC. Acestea aplicaţii sunt aplicaţii simple de testare, din domeniul, ştiinţific ce solicită doar anumite părţi componente ale sistemului computaţional. Scopul acestora a fost de a extinde aplicaţia *High Performance Linpack* utilizată în mod tradiţional pentru clasificarea sistemelor ce fac parte din lista celor mai mari 500 sisteme computaţionale din lume (Top 500 List) [4.22], cu o serie de alte aplicaţii *kernels* proiectate pentru a testa sistemul de memorie. Dintre acestea a fost studiat comportamentul aplicaţiilor:

- ***High Performance Linpack*** [4.23], [4.24] o aplicaţie pentru rezolvarea sistemelor de ecuaţii liniare. Din acest *benchmark* a fost analizată componenta în care se execută descompunerea în factori pentru o problema de 4K.
- ***RandomAccess*** [4.23] cunoscut şi sub numele de GUPS *(Giga Updates Per Second).* Este o aplicaţie a cărui scop în principal este evaluarea performanţei sistemului de memorie prin măsurarea ratei de actualizare aleatoare a locaţiilor acesteia. Dimensiunea datelor de intrare analizate a fost de 1024 MB.

- ***STREAM*** [4.24] aplicaţie utilizată pentru a determina lărgimea maximă de bandă (bandwidth) ce poate fi susţinută de sistemul de memorie al unui anumit sistem. S-au luat în studiu $16*10^6$ elemente.
- ***FFT*** [4.24] program ce calculează transformata Fourier rapidă. A fost folosită o transformată Fourier complexă bidimensională. Datele de intrare au fost de 1K. Au fost examinate atât faza de calcul cât şi cea de comunicare.

b) NAS Parallel Benchmarks [4.25]. Aceste *benchmark*-uri au fost create de NASA în anii `90 în scopul analizei creşterii performanţei odată cu creşterea numărului de unităţi de calcul dintr-un sistem. Din acest set de aplicaţii au fost luate în considerare:

- ***NAS BT*** [4.25] aplicaţie ce rezolvă un sistem de ecuaţii bloc-tridiagonal pe o grilă 3D. În studiul efectuat, pentru a face posibilă colectarea fişierelor *trace*, am utilizat o variantă a datelor de intrare relativ mică (*Class A*).
- ***NAS CG*** [4.26] o aplicaţie ce rezolvă conjugatul gradient, utilizând o matrice simetrică *sparse*. Asemeni aplicaţiei *NAS BT* această aplicaţie a fost analizată pentru o variantă *Class A* a datelor de intrare.

c). HPC Aplicaţii ştiinţifice reale. Spre deosebire de aplicaţile simple de testare prezentate anterior care în mod obişnuit au un comportament omogen, au fost studiate o serie de aplicaţii reale din domeniul ştiinţific, cu comportament neregulat, a căror performanţă nu poate fi evaluată intuitiv. Dintre acestea s-au selectat aplicaţii din domeniile fizicii particulelor, a chimiei computaţionale şi al analizei structurale, după cum urmează:

- ***GTC*** [4.9][4.33] (Gyro-Kinetic Toroidal Code) aplicaţie ce simulează interacţiunea dintre particule într-un reactor de fuziune nucleară de tip Tokamak. Pentru analiză a fost utilizată o problema cu $3*10^6$ particule.
- ***NAB*** [4.27] (Nucleic Acid Bilder) aplicaţie ce permite construirea modelelor de biomolecule, în mod special acizi nucleici si proteine. Această aplicaţie se foloseşte pentru măsurarea forţelor ce

încearcă să aducă atomii în poziţia de echilibru, după ce particulelor le-a fost imprimată o viteză iniţială aleatoare. Pentru analiză a fost studiată o molecula cu 1957 de atomi.

- ***LSDyna*** [4.28] aplicaţie ce rezolvă elementele finit, utilizată în simularea şi analizarea fenomenelor fizice neliniare. Este utilizată de exemplu în stabilirea modului de deformare al diferitelor părţi componente ale unei maşini în urma unui accident. Pentru analiză a fost aleasă ca problemă, studiul modificărilor ce apar la o elice de avion în timpul zborului, luâd în considerare 1024 de puncte de pe această.

d). Aplicaţii comerciale. Dintre aplicaţiile comerciale au fost alese pentru a fi studiate:

- ***TPC-C*** [4.10] aplicaţie ce simulează procesul de tranzacţionare *online* şi este proiectată pentru a testa milioane de astfel de tranzacţii. Este centrată în jurul unor utilizatori de depozite ce execută tranzacţii cu o bază de date. În cazul studiat au fost simulate 1600 astfel de depozite.
- ***SAP SD*** [4.29] aplicaţie ce simulează procesarea comenzilor şi plăţilor. Permite studiul rulării atât a aplicaţiei cât şi a bazei de date pe acelaşi sistem *hardware* fizic.

4.4. Metodologia şi infrastructura experimentală.

Un mijloc general acceptat pentru evaluarea modului de rulare a unei aplicaţii sunt fişierele *trace*. Fişierele *trace* oferă o modalitate de caracterizare a performanţei unei aplicaţii oferind o modalitate de explorare a interacţiunii între aplicaţie şi platforma *hardware* pe care este rulată. În general aceste fişiere sunt colectate folosind un simulator funcţional ce rulează atât aplicaţia testată, cât şi sistemul de operare. În felul acesta fişierele trace nu conţin doar instrucţiunile executate, ca parte a aplicaţiei utilizator, ci conţin şi informaţii despre modul în care această aplicaţie interacţionează cu sistemul de operare. Fişierele de urmărire a execuţiei reflectă în mod precis instrucţiunile executate de sistem, ca urmarea a

rularii aplicaţiei, precum şi o serie de evenimente cum ar fi *trap*, I/O şi activitatea DMA.

Metodologia de colectare şi validare a unor astfel de fişiere este prezentată schematic în figura 4.1. şi presupune efectuarea următoarelor acţiuni:

- Aplicaţia de evaluat este instalată pe un sistem *hardware* existent, numit sistem de referinţă.

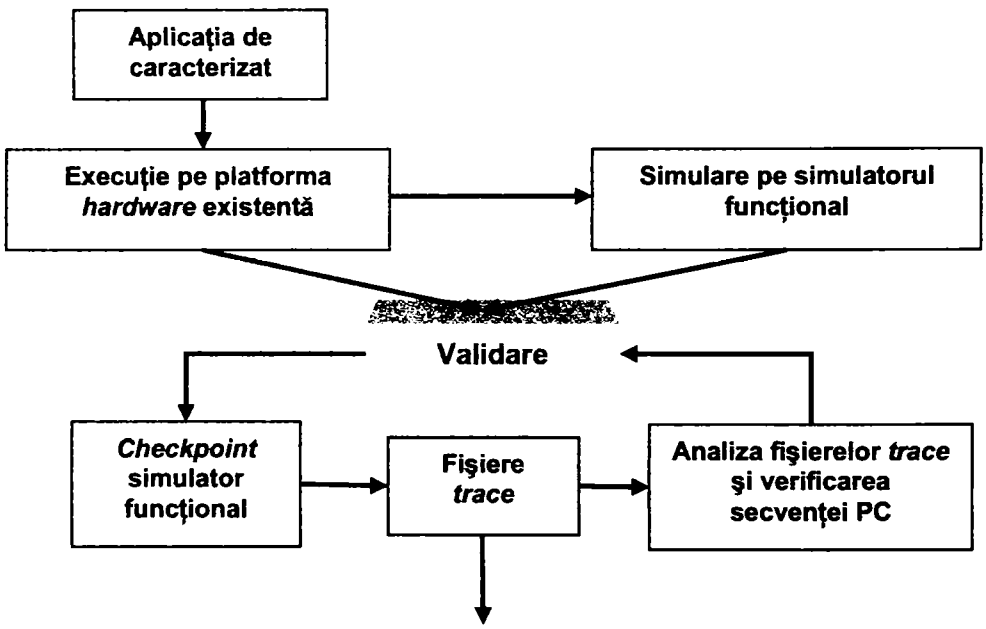


Figura 4.1. Etape parcurse în colectarea şi validarea fişierelor .

- Pentru a asigura corectitudinea execuţiei, după instalare aplicaţiei, aceasta este rulată de câteva ori. În timpul acestui proces sunt identificate regiunile reprezentative pentru execuţia în cauză, adică scurte zone care reflectă întocmai comportamentul aplicaţiei respective. Aceste regiuni, sunt izolate cu marcatori, pentru indicarea precisă a locului de începere şi încheiere a colectării fişierelor *trace*. Totodată sunt colectate o serie de date statistice de performanţă, utilizând utilitarele puse la dispoziţie de sistemul de operare, date ce vor fi ulterior folosite pentru validare.

- Odată ce aplicaţia rulează corect se colectează un fişier, care este o copie fidelă a partiţiei *root* a sistemului, precum şi o copie a întregului sistem de fişiere al sistemului *hardware* de referinţă. Acest fişier este utilizat pentru a reproduce ulterior execuţia aplicaţiei în simulatorul funcţional.
- Simulatorul funcţional este configurat astfel încât să reproducă întocmai structura *hardware* şi *software* a sistemului de referinţă. Prin această se asigură acelaşi comportament al aplicaţiei în cele două medii.
- În momentul în care execuţia atinge unul din punctele de interes, marcate prin identificatori, se colectează un aşa numit *checkpoint,* care reprezintă întocmai starea sistemului la momentul respectiv.
- Simularea este reluată de la *checkpoint* şi se începe scrierea fişierului *trace.*

Un pas important în colectarea fişierelor *trace* este validarea. Acest proces ne dă siguranţa că modul de comportare reflectat de către fişier, este identic cu comportamentul *hardware*-ului de referinţă, dacă acesta ar executa aceeaşi porţiune a aplicaţiei. Pentru validare atât din sistemul de referinţă, cât şi din simulatorul funcţional, sunt colectate o serie de date statistice generate de sistemul de operare, cum ar fi apelări de sistem, întreruperi, *traps* etc. Pe lângă acestea se colectează rezultatul programului în cauză. În faza de validare aceste date colectate din *hardware*-ul de referinţă, sunt comparate cu datele colectate din simulatorul funcţional.

Un alt pas în procesul de validare este introducerea în punctele de interes a unor apeluri către un set de librării, livrate împreună cu sistemul de operare, care permit resetarea regiştrilor *hardware* de evaluare a performanţei cu care este echipat procesorul. În felul acesta, numărătoarele pot colecta date statistice, ca de exemplu numărul de instrucţiuni sau *miss*-uri în memoriile *cache*, din exact aceiaşi porţiune a aplicaţiei, din care a fost colectat fişierul *trace*. Aceste date sunt ulterior comparate cu cele generate de simulatoarele ierarhiei de memorie, sau cele de performanţă, ce rulează acest fişier *trace* [4.2].

Un fişier *trace* este declarat valid dacă atât datele statistice generate de sistemul de operare cât şi datele colectate de numărătoarele de performanţă ale procesorului sunt identice, sau diferă nesemnificativ, de aceleaşi date colectate din simulatoarele funcţionale, sau cele de performanţă.

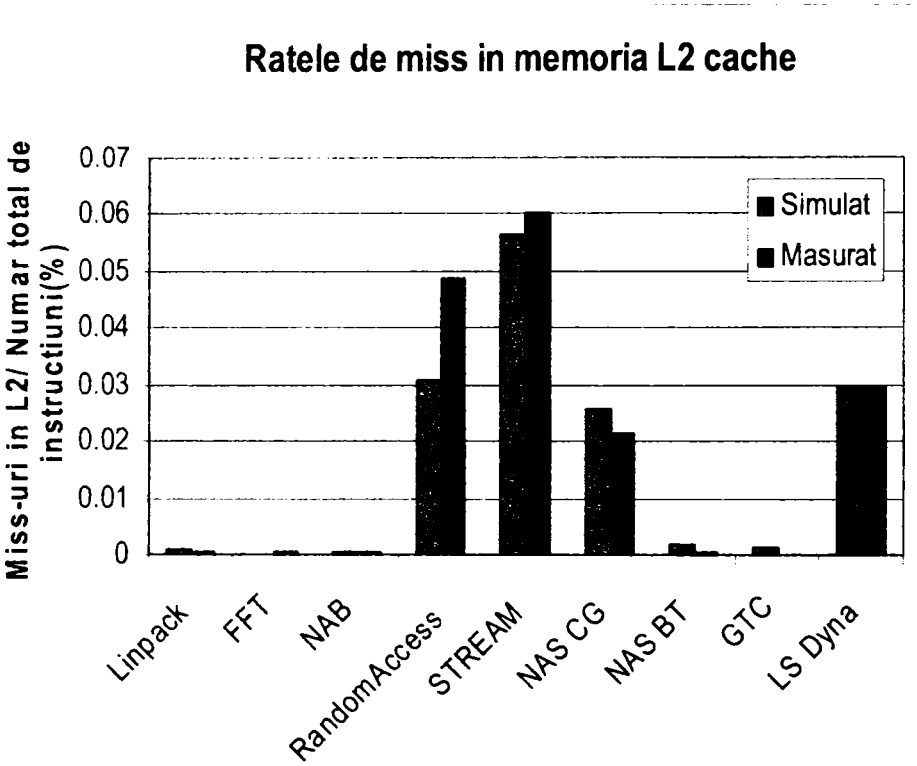


Figura. 4.2. Validarea ratei de *miss* în memoria L2 *cache*. Figura prezintă o comparaţie între rata de miss în memoria *cache* L2, măsurată de numărătoarele *hardware* ale procesorului Sun Fire 6800 UltraSparc-III, cu cea colectată din simulatorul de performanţă rulând fişierele de *trace*.

Platforma de referinţă folosită pentru efectuarea studiului a fost o platformă Sun Fire 6800 UltraSparc-III+ [4.31], 900 MHZ; rulând sistemul de operare Solaris 10. Pentru a izola comportamentul aplicaţiei studiate de activităţile suplimentare ce le implică rularea în paralel, în majoritatea cazurilor prezentate, s-au utilizat variantele uniprocesor a programelor de testare. Totuşi analiza caracteristicilor de accesare a memoriei, s-a făcut folosind o variantă paralelă a acestor aplicaţii, varianta ce rulează pe 16 procesoare, cu două excepţii: În cazul aplicaţiei TPC-C s-a folosit o variantă ce rulează pe numai 8 procesoare datorită dificultăţii generării fişierelor *trace,* pentru o versiune destinată rulării pe un număr mai mare de

microprocesoare. De asemenea în cazul aplicaţiei LS Dyna s-a folosit o variantă ce rulează pe 8 procesoare datorită limitărilor legate de scalabilitatea versiunilor ce rulează pe un număr mai mare de microprocesoare.

În toate cazurile, fară excepţie, pentru a izola comportamentul efectiv al aplicaţiilor utilizator, de acţiunile sistemului de operare s-au urmărit şi contorizat numai rezultate legate de aplicaţia utilizator.

Colectarea rezultatelor s-a făcut utilizând cu preponderenţă FAF (*Flexible Analysis Framework*), noua structură de analiză ce a fost prezentată pe larg în capitolul anterior. Numai în cazul studiului efectelor instrucţiunilor de precitire (*prefetch*) au fost folosiţi regiştrii procesorului destinaţi înregistrării datelor de performanţă.

Rezultatele prezentate au fost obţinute folosind versiuni nemodificate ale aplicaţilor amintite, compilarea lor făcându-se folosind optimizări standard. În acest fel executabilele studiate sunt reprezentative pentru modul în care ele sunt utilizate în practică. Pe parcursul prezentării rezultatelor au fost evidenţiate cele mai importante concluzii.

4.5. Rezultate.

4.5.1. Distribuţia instrucţiunilor.

Pentru a înţelege în detaliu nivelul de performanţă cu care o aplicaţie rulează pe o anumită platformă *hardware* se impune cunoaşterea distribuţiei instrucţiunilor, mai precis procentajul fiecărui tip de instrucţiune, în totalul instrucţiunilor ce alcătuiesc aplicaţia respectivă.

În figura 4.3. se prezintă distribuţia procentuală a diferitelor tipurilor de instrucţiuni, pentru fiecare dintre aplicaţiile luate în studiu. Categoriile de instrucţiuni reprezentate includ: instrucţiunile în virgulă mobilă (adunări, înmulţiri, citiri/scrieri din memorie); instrucţiuni cu întregi (instrucţiuni aritmetice şi logice, citiri/scrieri din memorie); instrucţiunile de salt şi instrucţiunile de precitire soft.

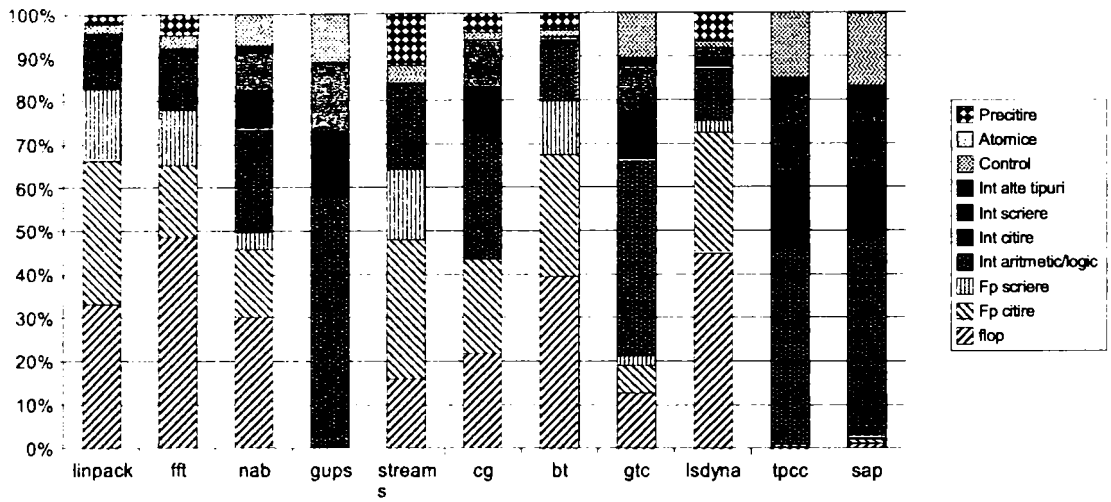


Figura.4.3. Distribuţia instrucţiunilor. Procentajul celor mai importante categorii de instrucţiuni.

Analizând datele prezentate, se pot face următoarele observaţii şi trage câteva concluzii importante:

- Cele două aplicaţii comerciale (*TPC-C* şi *SAP SD*) prezintă o distribuţie a instrucţiunilor aproximativ identică, comparativ cu aplicaţiile stinţifice din suita HPC, care prezintă, de la o aplicaţie la alta, o mare diversitate în distribuţia diferitelor categorii de instrucţiuni.
- Programele ştiinţifice, care prin definiţie efectuează operaţii matematice cu numere în virgulă mobilă, prezintă un număr destul de redus de instrucţiuni de acest tip; în jurul a doar 20%; maximul nedepăşind 50%. O serie de aplicaţii *(RandomAccess, STREAM, NAS CG şi GTC)* având chiar mai multe instrucţiuni ce operează cu întregi, decât cele ce operează cu virgulă mobilă. Această observaţie contrazice viziunea tradiţională, aceea că performanţa aplicaţiilor ştiinţifice poate fi estimată doar prin măsurarea componentei operaţiilor în virgulă mobilă (*FLOPS*) ignorând complet componenta ce operează cu întregi.

Numărul mare de instrucţiuni cu întregi în aplicaţiile de algebră liniară, se explică prin faptul că în aplicaţiile ştiinţifice ce efectuează calcule mai ales cu date în virgulă mobilă, există un număr foarte mare de instrucţiuni de indexare în masive şi matrici, ce operează cu date întregi.

- Datele prezentate în figura 4.3. permit de asemenea analiza echilibrului între instrucţiunile de citire şi scriere în memorie. Dacă privim doar din punctul de vedere al instrucţiunilor de citire/scriere de date întregi, observăm că acestea sunt echilibrate unele faţă de altele, atât în aplicaţiile stiinţifice cât şi în cele comerciale. Pe când, comparând frecvenţa instrucţiunilor de citire/scriere a datelor în virgulă mobilă, realizăm că instrucţiunile de citire sunt predominante. Acest lucru este explicabil prin faptul că pentru a efectua o operaţie matematică în virgulă mobilă este necesar a se citi din memorie valoarea a doi operanzi şi a se scrie doar un singur rezultat.

In urma observaţiilor făcute trebuie reţinute următoarele:

- ***Variabilitatea distribuţiei instrucţiunilor în cazul aplicaţiilor ştiinţifice, face imposibilă caracterizarea de grup a acestora şi arată că nu se pot trage concluzii comune tuturor aplicaţilor de acest gen, pe baza rezultatelor obţinute analizând numai comportamentul unora dintre ele.***
- ***Datele obţinute relevă faptul că aplicaţiile stinţifice trebuie analizate printr-o metodologie mai generală decât doar prin analiza componentei FLOP.***

4.5.2. Analiza accesării locaţiilor de memorie.

Este cunoscut faptul că aplicaţiile stiinţifice sunt cele care solicită în cea mai mare măsură sistemul computaţional şi de asemenea este cunoscut faptul că performanţa aplicaţiilor rulate pe un procesor, este influenţată în mod decisiv de felul în care acestea accesează memoria. Din acest motiv în prezenta secţiune se vor studia: aspectele legate de localizarea temporală şi spaţială a accesării memoriei.

4.5.2.1. Localizarea temporală.

Prin localizare temporală se înţelege tendinţa unei aplicaţii de a reaccesa aceleaşi adrese de memorie, pe care le-a accesat anterior într-un moment apropiat

în timp [2.2]. În mod uzual această tendinţă este evaluată prin măsurarea „distanţei de reutilizare".

Distanţa de reutilizare reprezintă numărul total de accesări ale memoriei, efectuate între două accesări succesive, ale aceleiaşi locaţii. In studiul efectuat am prezentat distanţa de reutilizare a datelor din memorie folosind o metodă similară cu cea folosită de Weinberg şi alţii [4.7].

O serie de rezultate obţinute în cadrul acestui studiu, ce reprezintă localizarea temporală în funcţie de distanţa de reutilizare, pentru setul de aplicaţii luat în considerare sunt prezentate în figura 2.4.

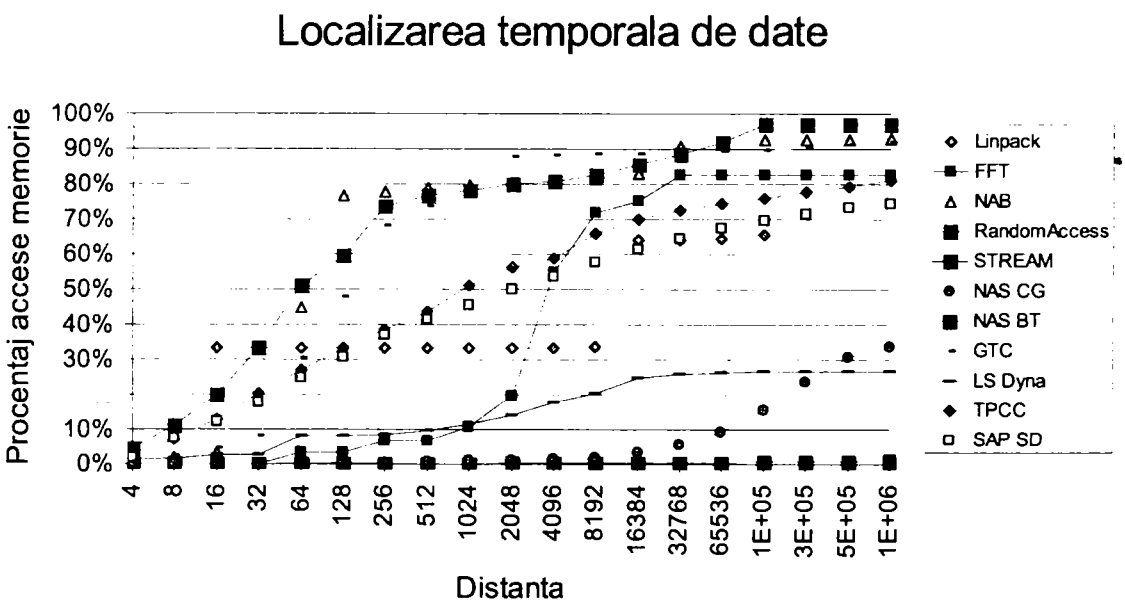


Figura 4.4. Localizarea temporală date. În această figură sunt prezentate procentajele cumulate a accesărilor memoriei ce au o distanţă temporală mai mică decât pragurile reprezentate pe abscisă. Distanţa temporală este definită ca fiind numărul total de accesări unice ale memoriei, dintre două accesări consecutive ale aceleaşi locaţii de memorie.

Acest tip de rezultate este util în analiza profilului aplicaţiilor [4.32]. Fiecare punct din grafic reprezintă procentele cumulate ale cazurilor în care numărul total de accesări unice ale memoriei, dintre două accesări consecutive ale aceleaşi locaţii de memorie, este mai mic sau egal cu valorile reprezentate pe abscisă. Deoarece

aceste distanţe sunt cuantificate ca număr total de accesări unice între două accesări succesive ale aceleaşi locaţii a memoriei fizice, analiza poate fi efectuată fară a ţine seama de particularităţile arhitecturale ale memoriei şi din această cauză poate oferi informaţii importante asupra dimensiunilor optime ale diferitelor niveluri ale acesteia.

Cu referire la datele prezentate în figura 4.4. sunt necesare următoarele clarificări:

- Distanţele de reutilizare între două accesări ale aceleaşi locaţii de memorie ce sunt mai mici decât „**4**" nu sunt reprezentate, dat fiind faptul că este improbabil ca datele corespunzătoare acestor adrese, să fie eliminate dintr-o memorie *cache* de dimensiuni rezonabile, într-un interval de timp foarte scurt.
- distanţele între două accesări ale aceleaşi locaţii de memorie ce sunt mai mari decit $\mathbf{10^6}$, au fost ingnorate datorită numărului nesemnificativ al acestora, fapt ce face ca valorile de la dreapta extremă a graficului să aibă o influenţă nesemnificativă asupra scorului final.

Prin stabilirea acestor limitări este posibil studiul unui număr important de diverse dimensiuni ale memoriei *cache*, fară a introduce o complexitate deranjantă în simulare.

Având în vedere precizările făcute, formula de calcul a scorului localizării temporale este dată de relaţia (1).

$$scor = \frac{\sum_{i=0}^{\log(N)-1}\left(\left(reuse_{2^{i+1}} - reuse_{2^i}\right)*\left(\log_2(N) - i\right)\right)}{\log_2(N)}, \; 0 < scor < 1 \qquad (8)$$

în care:

- ***reuse*** sau ***reuse distance*** - al unei adrese de memorie **A**, este reprezentată de distanţa măsurată în numărul de adrese de memorie unice, accesate de la ultima accesare a adresei **A**.

- ***reuse$_i$*** - este fracţiunea accesărilor memoriei a cărei *reuse distance* este mai mică sau egală cu o anumită valoare **i.**
- **N -** reprezintă *reuse distance* maxim.
- ***scor* = 0** indică o lipsă totală de localizare temporală a accesărilor memoriei;
- ***scor* = 1** indică faptul că toate accesările memoriei sunt egale cu cel mai mic *reuse distance* măsurat.

Relaţia (1) este de fapt o variantă a relaţiei (2), relaţie utilizată în [4.7], al cărei numărător a fost modificat, în aşa fel încât scorul total să fie cuprins între zero şi unu.

$$scor = \frac{\sum_{i=0}^{\log(N)-1} ((reuse_{2^{i+1}} - reuse_{2^i}) * \log_2(N) - i)}{\log_2(N)} \qquad (9)$$

Cu aceste ajustări termenii au reprezentarea adecvată; chiar şi distanţele mici au un impact semnificativ asupra scorului final. Asemeni celor prezentate în [4.7], valorile aflate aproape de originea axelor, indică o localizare temporală mică, pe când o valoare a scorului egală cu „**1**" indică un număr minim de accesări unice, între două accesări succesive, ale aceleaşi locaţii de memorie. Rezultatele obţinute în urma analizei localizării temporale vor fi discutate, împreună cu cele obţinute în urma analizei localizării spaţiale, în subcapitolul 4.5.2.3.

Într-o primă apreciere însă, se pot face următoarele remarci:

❖ Aplicaţiile *STREAM* şi *RandomAccess* se comportă conform aşteptărilor. Aplicaţia *RandomAccess* foloseşte adrese generate aleatoriu pentru a citi şi scrie datele din memorie, iar aplicaţia *STREAM* a fost creată pentru testarea lărgimii de bandă de acesare a memoriei. Datorită modului în care sunt proiectate, în execuţie ambele aplicaţii accesează locaţii de memorie noi, fară a reutiliza datele accesate anterior. *RandomAccess* de fapt execută o instrucţiune de citire din memorie de la o anumită adresă, urmată de o instrucţiune de scriere în memorie la aceeaşi adresă. Astfel de operaţii nu sunt însă contorizate de algoritmul prezentat în lucrare, datorită restricţiilor impuse ce au fost prezentate mai sus.

❖ *NAS CG* şi *LS Dyna* prezintă un scor redus ale localizării temporale (max. 30%) deoarece accesează date distribuite. În cazul aplicaţiei *LS Dyna* s-a studiat doar porţiunea structurii de tip arbore, parcursă în ambele direcţii, pentru identificarea şi eliminarea soluţiilor greşite ce reprezintă doar o parte a algoritmului de rezolvare. Din această cauză este rezonabilă existenţa unui scor mic al localizării temporale, deoarece parcurgerea structurii de date sub formă de arbore creată de această aplicaţie se face ramură cu ramură. În această porţiune a algoritmului sunt accesate permanent date noi. Pe măsură ce algoritmul avansează în arbore, nivelul de localizare temporală creşte, dar cu toate acestea majoritatea calculului se face la nivelurile iniţiale. *NAS CG* este o aplicaţie de testare, care verifică un algoritm de calcul al conjugatei unei matrici *sparse* (a cărei elemente sunt în majoritate zero). Într-un astfel de calcul fiecare element al matricii va fi accesat relativ rar.

❖ Celelalte aplicaţii studiate au o tendinţă de a accesa date din locaţiile de memorie accesate anterior, ce depăşeşte în general nivelul de 70%. Acest lucru este reflectat de datele prezentate în zona îndepărtată de origine a abscisei (figura 2.4)

Aplicaţiile ştiinţifice prezintă porţiuni în care concentraţia de instrucţiuni de accesare a memoriei creşte semnificativ. De exemplu aplicaţia Linpack prezintă trei astfel de zone distincte. În schimb aplicaţiile comerciale *SAP SD* si *TPC-C* ambele prezintă o creştere graduală a acestui tip de instrucţiuni. Aplicaţiile ştiinţifice au fost proiectate pentru a rezolva un anumit *task* şi din această cauză algoritmul tinde să acceseze elementele asupra cărora acţionează repetitiv într-un *loop*. Linpack reprezintă un exemplu interesant având trei astfel de zone, sub formă de treaptă, ceea ce indică existenţa a tot atâtea *nested loops*. Acest rezultat dovedeşte atenţia acordată în proiectarea acestei aplicaţii legată de organizarea accesării datelor, pentru a obţine un avantaj maxim din stocarea lor în memoria *cache* rapidă.

4.5.2.2. Localizarea spaţială.

Localizarea spaţială este definită ca o proprietate a unui program de a accesa adrese de memorie apropiate unele de altele [2.2]. Localizarea spaţială poate fi studiată prin efectuarea diferenţei între adresele de memorie ce sunt accesate de două instrucţiuni apropiate în timp.

Asemeni definiţiei localizării temporale, Weinberg şi alţii [4.7] au definit şi o unitate de măsură pentru măsurarea localizării spaţiale a unui program. Această unitate de măsură este dată de relaţia (3),

$$scor = \sum_{i=1}^{\infty} \frac{stride_i}{i}, \quad 0 < scor < 1 \tag{10}$$

în care:

- ***stride$_i$*** - reprezintă fracţiunea accesărilor locaţiilor de memorie pentru care diferenţa între adresa accesării curente şi cea a accesărilor aflate într-o fereastră de dimensiune „**W**" a fost egala cu „***i***".

Utilizând această unitate de măsură se poate afirma că un program care accesează locaţii de memorie consecutive va avea scorul localizării spaţiale „**1**" pe când un program care accesează fiecare a doua locaţie de memorie va avea scorul localizării spaţiale „**0.5**". Un program ce accesează locaţii de memorie distribuite uniform între „**1**" şi „**2**" va avea un scor de **„0.75",** iar un program ce accesează locaţii de memorie aleatoriu va avea un scor **„0"**.

În [4.7] Weinberg şi alţii prezintă rezultate referitoare la localizarea spaţială doar pentru ferestre de dimensiune maximă **W=32**, asumându-şi faptul că o fereastră de asemenea mărime va cuprinde toate "*loop*-urile" de dimensiuni rezonabile şi va fi astfel reprezentativă pentru a produce scorul localizării spaţiale a tuturor aplicaţiilor ştiinţifice.

Datele experimentale obţinute în urma prezentului studiu contrazic această ipoteză. Multe dintre aplicaţiile evaluate nu se stabilizează din punctul de vedere al localizării spaţiale pentru ferestre de dimensiuni reduse. Din această cauză s-a apreciat că, pentru rigurozitate, este mai completă şi concludentă prezentarea scorului localizării spaţiale pentru o gamă variată ale dimensiunilor ferestrei **W.**

În figura 4.5. este prezentat scorul localizării spaţiale pentru fiecare aplicaţie sudiată pentru o fereastra **W**, a cărei dimensiune variază între **1** şi **1024** accesări anterioare ale memoriei.

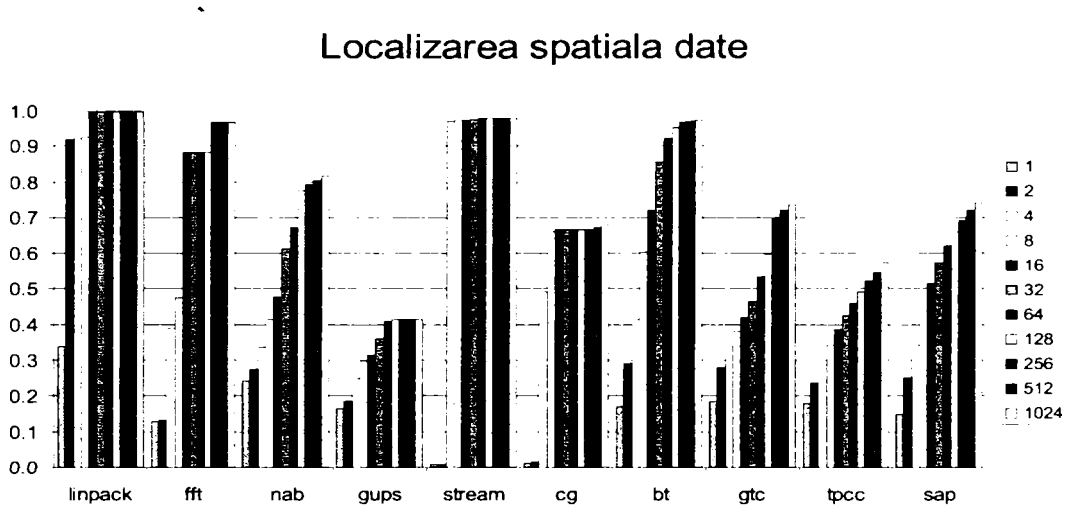


Figura 4.5. Localizarea spaţială date: Figura prezintă scorul localizării spaţiale rezultat după modelul relaţiei (10) pentru diferite dimensiuni a ferestrei de analiză.

Analizând figura se observă că:

❖ *STREAM* are un scor al localizării spaţiale de aproape „**1**", iar *RandomAccess* (GUPS) nu prezintă nici un fel de localizare spaţială. Având în vedere caracteristicile acestor aplicaţii această comportare este previzibilă.

❖ Aplicaţiile ştiinţifice reale (*NAS, BT, GTC, NAB*) prezintă caracteristici ale localizării spaţiale apropiate mai mult de aplicaţiile comerciale, decât *benchmark*-urile ce fac parte din *HPC* Challenge.

❖ Aplicaţiile care presupun efectuarea unor calcule diverse de algebră liniară, cum sunt (*LSDyna, NAS CG si Linpack*), au un comportament unitar, fiecare program manifestând întreaga sa localizare spaţială într-o fereastră **W=32**. Acest fapt se datorează modului în care programatorii au proiectat accesul la date pentru aceste aplicaţii, în scopul obţinerii algortimului cel mai eficient, pentru o anumită dimensiune a memoriei *cache*.

Similar cu datele prezentate anterior, în figura. 4.6. se prezintă, de această dată, localizarea spaţială a instrucţiunilor aplicaţiilor studiate, pentru o fereastră **W** a cărei dimensiuni variază între 1 şi 1024.

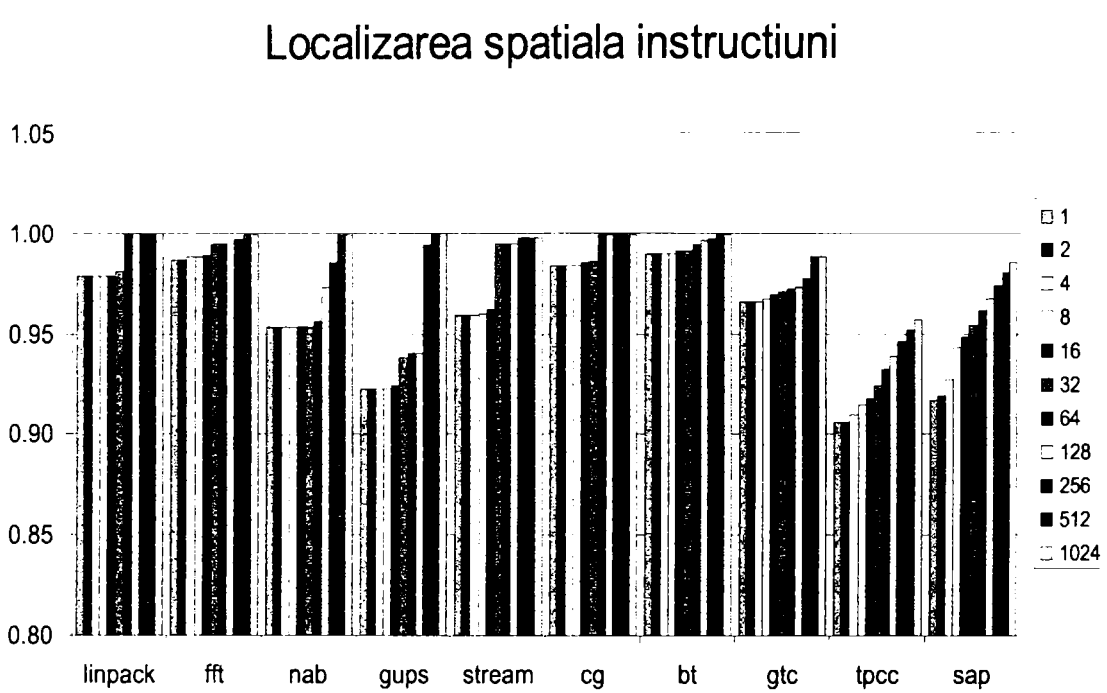


Figura 4.6. Localizarea spaţiala a instrucţiunilor. Figura prezintă scorul localizării spaţiale rezultat după modelul relaţiei (10) pentru o varietate de ferestre de analiză.

Observaţiile ce pot fi făcute în urma acestei analize sunt următoarele:

- Spre deosebire de localizarea spaţială a datelor, localizarea spaţială a instrucţiunilor nu este atât de variată, aplicaţiile în general având tendinţa să acceseze instrucţiuni consecutive.
- Pentru dimensiuni ale ferestrei **W ≥ 512**, toate aplicaţiile ştiinţifice cu excepţia la *GTC* au scorul localizării spaţiale **1**, multe dintre aplicaţii apropiindu-se de acest scor chiar pentru ferestre **W ≥64**.
- O serie de aplicaţii ştiinţifice (*Linpack, NAS CG şi BT*) prezintă caracteristici ale localizării spaţiale foarte apropiate unele faţă de altele.
- Asemănător cu aplicaţiile ştiinţifice şi cele două aplicaţii comerciale *SAP* şi *TPC-C*, se comportă unitar, dar acestea prezintă cele mai mici scoruri ale localizării spaţiale, chiar pentru ferestre de dimensiuni W=1024.

În figură 4.7. este prezentată cumulativ fracţiunea accesărilor memoriei aflate într-o fereastră **W=32** ce au o distanţă mai mică decât un anumit prag reprezentat de valorile de pe abscisă

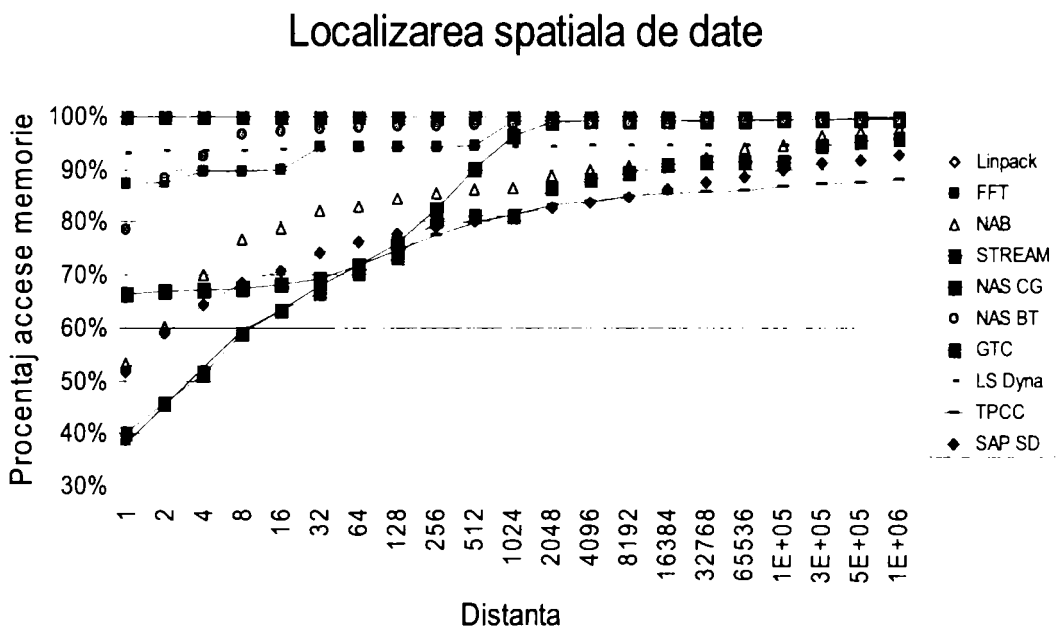


Figura 4.7. Localizarea spaţială a datelor. Această figură prezintă cumulativ fracţiunea accesărilor memoriei ce au o distanţă mai mică decât un anumit prag considerând o fereastră W=32. Pentru a face mai clară reprezentarea datelor în grafic axa verticală începe la valoarea de 30% iar curba ce reprezintă aplicaţia *RandomAccess* nu este prezentată.

Analizând datele obţinute se pot face câteva remarci:

❖ Aşa cum este de aşteptat cu excepţia lui *RandomAccess* toate celelalte aplicaţii accesează repetat aceeaşi locaţie de memorie în fereastra considerată.

❖ Curba aferentă aplicaţiei *FFT* prezintă o treaptă care poate fi explicată prin natura multi-scalară a acestui program de testare.

❖ Un caz interesant este cel al aplicaţiei *NAS CG* ce prezintă o curbă a localizării spaţiale în formă de „**S**", având majoritatea accesărilor memoriei la distanţe relativ mici unele de altele. Această se întâmplă deoarece aplicaţia parcurge elementele unei matrici *sparse* (distribuite) iar *offset*-urile între accesărilor consecutive au în mod predominant, acelaşi ordin de mărime.

4.5.2.3. Consideraţiuni globale asupra localizării datelor.

Localizarea accesărilor memoriei, pentru fiecare din aplicaţiile studiate, atât din punct de vedere temporal, cât şi din punct de vedere spaţial, este prezentată în figura 2.8. În această figură pentru calculul scorurilor localizării spaţiale s-a ales o fereastră de dimensiune W=32. Alegerea acestei dimensiuni s-a făcut pentru a fi posibilă compararea rezultatelor prezentate în teză, cu cele obţinute de J. Weinberg, ş.a. (figura 2.9.), prezentate în [4.7].

Comparând cele două figuri se remarcă următoarele:

- Scorurile aplicaţiilor *RandomAccess, STREAM* şi *FFT* sunt echivalente şi sunt conforme cu aşteptările. Spre exemplu datorită naturii aleatorii a aplicaţiei *RandomAccess*, atât scorurile localizării temporale, cât şi a celei spaţiale sunt aproximativ zero.

- În ambele situaţii scorul localizării spaţiale a aplicaţiei *STREAM* este foarte ridicat, pe când cel al localizării temporale este aproape **„0".** Acest fapt dovedeşte faptul că aplicaţia nu refoloseşte datele pe care le accesează.

- Aplicaţiile *Linpack* si *NAS CG* prezintă unele diferenţe între studiul prezentat în această lucrare şi cel prezentat în [4.7]. Versiunea aplicaţiei *NAS CG* studiată în această lucrare prezintă o localizare temporală mai redusă, pe când scorurile localizării spaţiale sunt aproximativ egale. Aceste diferenţe pot fi datorate dimensiunilor diferite ale fişierelor de date de intrare, a densităţii diferite a acestora, precum şi a diferenţelor între versiunile compilatoarelor şi a opţiunilor folosite în compilare.

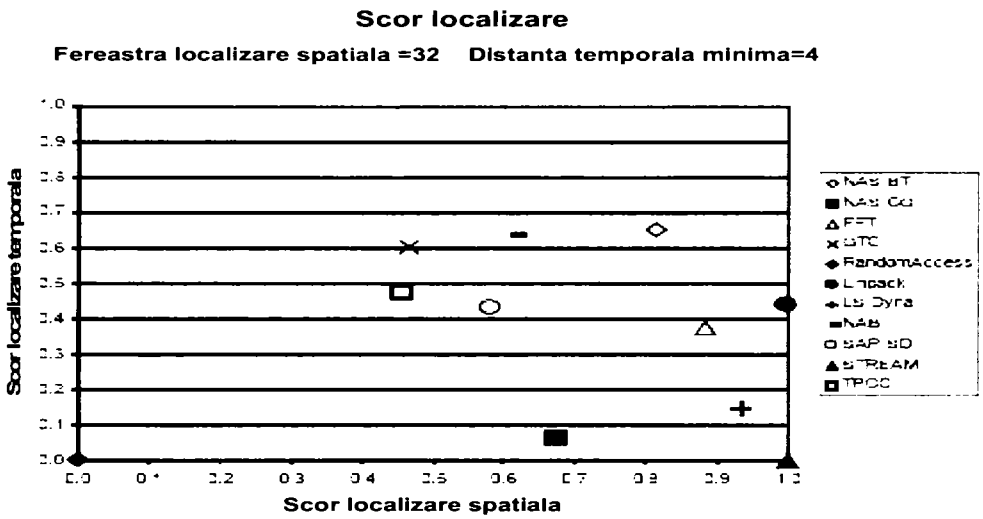


Figura 4.8. Localizarea spaţială şi temporală de date: Figura prezintă pentru fiecare aplicaţie atât scorurile localizării temporale cât şi cele ale localizării spaţiale.

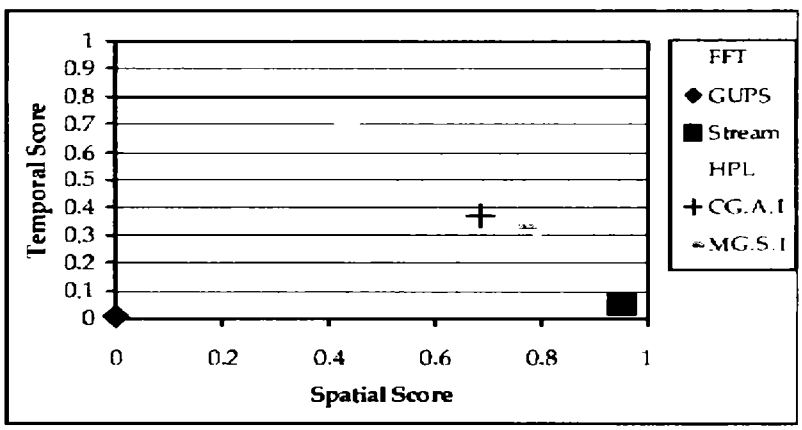


Figura 4.9. Reprezintă datele obţinute de J.Weinberg [4.7] şi este prezentată pentru a permite corelarea cu datele prezentate în Figura 2.8.

Din datele prezentate în figura 4.8. trebuie reţinute următoarele:

- ***O serie de benchmark-uri din suita HPC sunt modificate de proiectanţii lor pentru a limita localizarea datelor în memorie atât în dimensiunea spaţială cât şi în cea temporală.***

➢ ***În general aplicaţiile de calcul intensiv reale se comportă mai asemănător cu aplicaţiile comerciale în ceea ce priveşte localizarea datelor în memorie, decât cu benchmark-urile din suita HPC.***

4.5.3. Analiza memoriei cache.

În cele ce urmează se va extinde studiul localizărilor, spaţiale şi temporale, (care sunt independente de detaliile de configurare a ierarhiei de memorie) şi se va examina comportarea diferitelor aplicaţii în corelaţie cu arhitectura memoriei *cache*.

Rezultatele prezentate în această secţiune au fost generate prin utilizarea unor simulatoare ale ierarhiei memoriei *cache,* ce funcţionează în mod *trace-driven.* S-au luat în considerare două aspecte:

1) Sensibilitatea aplicaţiilor la dimensiunea memoriei *cache.*

2) *Sensibilitatea* aplicaţiilor la numărul de căi a memoriei *cache*

Prin „sensibilitatea unei aplicaţii" se înţelege modul în care indicatorii de performanţă ai aplicaţiei respective, reacţionează la diferitele modificări arhitecturale; în cazul de faţă la creşterea dimensiunii memoriei *cache*.

4.5.3.1. Sensibilitatea aplicaţiilor la dimensiunile memoriei *cache*.

Studiul efectuat a avut în vedere atât memoria *cache* de date cât şi cea de instrucţiuni. În ambele cazuri a fost scoasă în evidenţă rata de *miss* într-o memorie cu mapare directă, de diferite dimensiuni, având liniile de 64B.

✓ **Memoria Cache de date.**

În Figura 4.10. este prezentată sensibilitatea fiecăreia dintre aplicaţiile studiate, la dimensiunile memoriei *cache* de date.

Din analiza graficelor prezentate în figură, se constată că:

❖ o memorie mai mare de 2 MB, nu aduce îmbunătăţiri în privinţa ratei de *miss*, în cazul nici uneia dintre aplicaţiile studiate. Mai mult decât atât, majoritatea aplicaţiilor studiate au o rată de *miss* mai mică de 6%, dacă se utilizează memorii cu dimensiunea egală, sau mai mare de 16 KB; singurele care fac

excepţie sunt aplîcaţiile *STREAM* si *NAS CG*. In cazul acestor aplicaţii pentru a aduce rata miss-urilor sub 10%, este necesară o memorie cache de date de 512 MB.

❖ aşa cum era de aşteptat rata de *miss* a aplicaţiei *RandomAccess* nu poate fi înbunătăţită oricât de mare ar fi memoria *cache* de date.

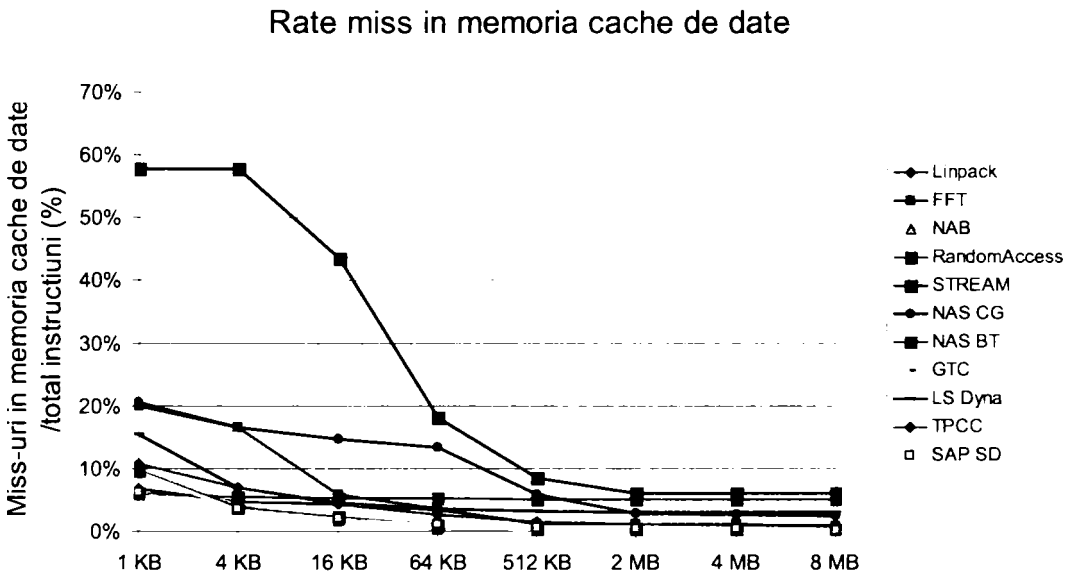


Figura 4.10. Sensibilitate aplicaţiilor la dimensiunile memoriei cache de date: În această figură este prezentată sensibilitatea fiecărei aplicaţii la dimensiunile memoriei cache şi reprezintă ratele de "miss" într-o memorie cu mapare directă de diferite dimensiuni având liniile de 64B.

✓ **Memoria cache de instrucţiuni**

În figura 4.11. este prezentată sensibilitatea fiecărei aplicaţii la dimensiunile memoriei *cache* de instrucţiuni. Datele prezentate în figură dovedesc că:

❖ ratele de *miss* pentru toate aplicaţiile studiate sunt foarte sensibile la creşterea memoriei *cache* de instrucţiuni fiind invers proporţionale cu creşterea acesteia.

❖ datorită caracteristicilor ei aplicaţia comercială *TPC-C* prezintă ratele de *miss* cele mai ridicate. Această aplicaţie prezintă rate de *miss* relativ mari chiar pentru memorii *cache* de instrucţiuni de dimensiuni de 256 KB. Pentru a obţine o rată de *miss* mai mică de 1% ar fi nevoie de o memorie *cache* de cel puţin 1MB.

❖ este interesant de asemenea de menţionat faptul că atât aplicaţiile *HPC Challenge* cât şi cele *NAS Parallel Benchmarks* nu beneficiază de dimensiuni ale

memoriei *cache* mai mari de 16KB; ratele de *miss* într-o memorie de această dimensiune fiind neglijabile. Aceasta ne indică faptul că întregul executabil al acestor aplicaţii poate fi cuprins în 16KB.

Atât datele prezentate în figura 4.10. cât şi cele prezentate în figura 4.11. oferă informaţii utile designer-ilor memoriei *cache* de date şi respectiv de instrucţiuni, legate de modul de dimensionare a acestora.

De reţinut în urma celor anterior prezentate sunt următoarele:

- ***În cazul aplicaţiilor ştiinţifice cât şi a celor comerciale nu sunt necesare memorii cache de date cu dimensiuni mai mari de 2 MB.***
- ***Cu privire la memoria cache de instrucţiuni se constată că o dimensiune de 256 KB este suficientă pentru toate aplicaţiile ştiinţifice.***
- ***Aplicaţiile comerciale, dar mai ales TPC-C, pentru a rula cu o performanţă corespunzătoare necesită o memorie cache de instrucţiuni de cel puţin 1MB.***

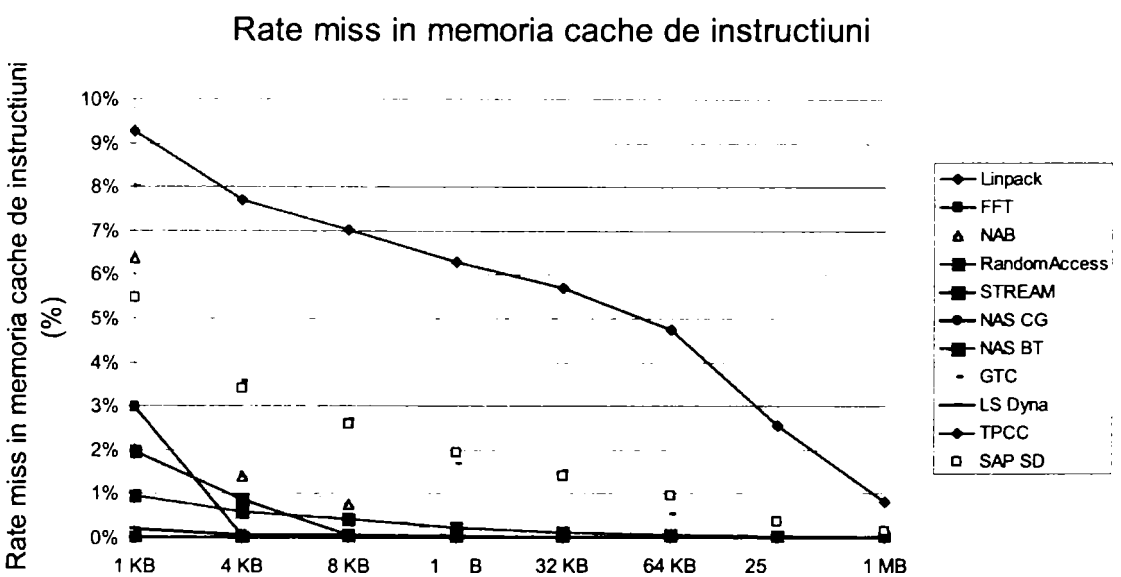


Figura 4.11. Sensibilitate aplicaţiilor la dimensiunile memoriei *cache* de instrucţiuni: În figură este prezentată sensibilitatea fiecarei aplicaţii la dimensiunile memoriei *cache* şi reprezintă ratele de *"miss"* într-o memorie cu mapare directă de diferite dimensiuni având liniile de 64B.

4.5.3.2. Sensibilitatea aplicaţiilor la creşterea numărului de căi în memoria *cache*.

În figura 4.12 este prezentată sensibilitatea aplicaţiilor la creşterea numărului de căi în memoria *cache* de date. Pentru acest studiu a fost aleasă o memorie de 64KB cu metoda de înlocuire a liniilor în *cache* de tip LRU.

Din analiza rezultatelor prezentate se poate concluziona că:

- majoritate aplicaţiilor nu beneficiază de creşterea nivelului de asociativitate al memoriei cache, deoarece mărimea setului de date accesat de aceste aplicaţii este mai mare de 64 KB.
- aplicaţia STREAM nu beneficiază de creşterea nivelului de asociativitate datorită faptului că datele accesate umplu fiecare set al memoriei cache cu mapare directă, înlocuind astfel toate variabilele ce stochează indecşii de apelare a elementelor într-un loop, care eventual ar fi putut fi re reaccesaţi în urma creşterii nivelului de asociativitate.

În afară de cazul prezentat au fost studiate şi memorii *cache* de alte dimensiuni, între 16 KB şi 128 KB fără ca rezultatele obţinute să difere semnificativ.

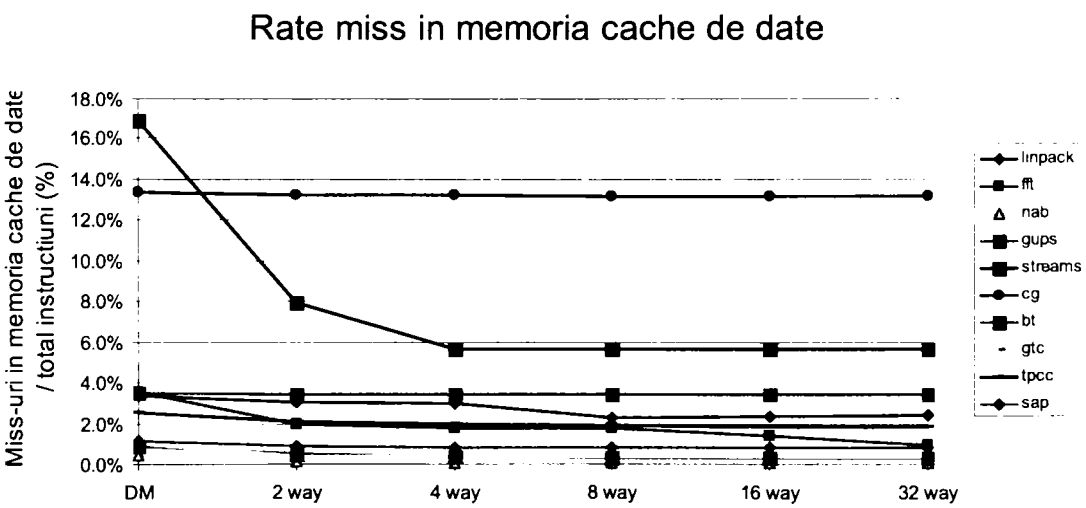


Figura 4.12. Sensibilitatea aplicaţiilor la creşterea numărului de căi în memoria *cache* de date: Această figură prezintă ratele de *miss* în memoriea *cache* de date raportate la numărul total de instrucţiuni pentru o memorie *cache* de 64 KB. Pe abscisă este reprezentat gradul de asociativitate al memoriei *cache* pornind de la DM (grad de asociativitate 0, mapare directă) până la 32way (32 de căi).

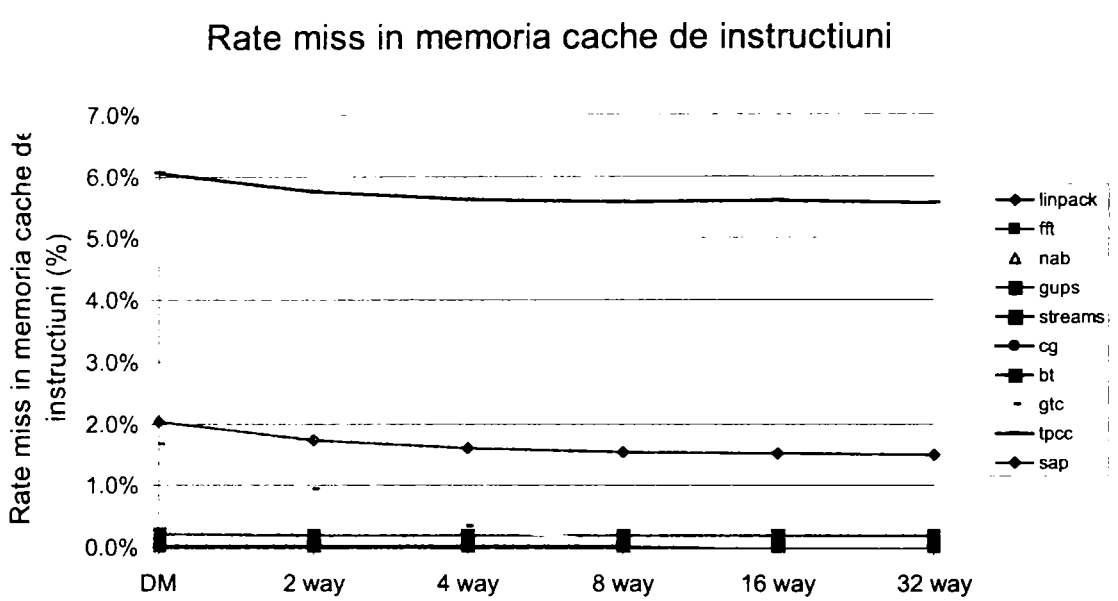


Figura 4.13. Sensibilitatea aplicaţiilor la creşterea numărului de căi în memoria *cache* de instrucţiuni: Această figură prezintă ratele de *miss* în memoriea *cache* de date raportate la numărul total de instrucţiuni pentru o memorie *cache* de 64 KB. Pe abscisă este reprezentat gradul de asociativitate al memoriei *cache* pornind de la DM (grad de asociativitate 0, mapare directă) până la 32way (32 de căi).

În figura 4.13. este prezentată sensibilitatea aplicaţiilor la creşterea numărului de căi în memoria *cache* de instrucţiuni, prezentându-se ratele de "*miss*" pentru o memorie cache de 64 KB când se creşte numărul de căi, în condiţiile prezentate deja, în cazul memoriei *cache* de date.

Deoarece din datele prezentate în figură rezultă că atât aplicaţiile *HPC Challenge* cât şi cele *NAS Parallel Benchmarks* nu beneficiază de dimensiuni ale memoriei *cache* mai mari de 16KB; ratele de *miss* în memoria de 64KB studiată sunt neglijabile, aplicaţiile fiind astfel complet insensibile la creşterea numărului de căi în memoria *cache.* Dintre celelalte aplicaţii, aplicaţiile comerciale *TPC-C* şi *SAP* prezintă o sensibilitate foarte redusă la creşterea numărului de căi. Singură aplicaţia *GTC* prezintă o sensibilitate ridicată, rata de *miss* scăzând semnificativ pentru memorii cu până la 8 căi, situaţie în care ajunge nesemnificativă.

Referitor la datele prezentate mai sus se pot trage următoarele concluzii cu caracter general:

- ***În cazul studiat atât pentru aplicaţiile ştiinţifice cât şi pentru cele comerciale complexitatea arhitecturală asociată cu creşterea***

numărului de căi în memoria cache de date nu este justificată, creşterea acestui număr pentru o memorie cache de 64KB neaducând nici un beneficiu semnificativ.

- ***Aplicaţiile comerciale şi un mare număr de aplicaţii ştiinţifice sintetice nu manifestă o sensibilitate ridicată la creşterea numărului de căi în memoria cache de instrucţiuni, singura care face excepţie este aplicaţia stiinţifică reală GTC.***

4.5.3.3. Nivelul de utilizare al aceluiaşi set de date într-un mediu multiprocesor.

În aplicaţiile paralele, efectele rezultate de traficul generat pentru a menţine operaţiunile de accesare a memoriei coerente, sunt tot atât de importante din punct de vedere al performanţei ca şi *miss*-urile *compulsory* (datele nu sunt în *cache*), *capacity* (datele au fost înlocuite datorită dimensiunii limitate ale memoriei *cache*) şi *coflict* (datele au fost înlocuite în memoria *cache* deoarece s-au suprapus cu date ce au fost accesate mai recent). Efectele acestor tipuri de *miss*-uri au fost studiate în paragrafele anterioare.

Pentru a evidenţia efectele ce apar în memoria *cache* cauzate de rularea în paralel pe mai multe procesoare a aplicaţiilor studiate, au fost rulate versiunile paralele (*OpenMP*) a fiecăreia dintre acestea, într-un simulator multi-procesor al memoriei *cache*. Configuraţia simulatorului este prezentată în Tabelul 4.1.

Parametru	**Configuraţie**
Memorie cache instrucţiuni L1 (nivelul 1)	32KB, 4 căi, 64B linii, PLRU
Memorie cache date L1 (nivelul 1)	32KB, 4 căi, 64B linii, PLRU, WT, nWA
Memorie cache L2, date şi instr. (nivelul 2)	2MB, 8 căi, 64Blinii, PLRU, WB, WA, include conţinutul memoriei de date L1
Protocol ordonare accese memorie	MESI

Tabelul 4.1. Configuraţia simulatorului ierarhiei de memorie *cache*, a sistemului multiprocesor studiat.

Figura 4.14. prezintă *miss*-urile în nivelul doi (L2) al memoriei *cache* raportat la numărul total de instrucţiuni executate pentru fiecare aplicaţie.

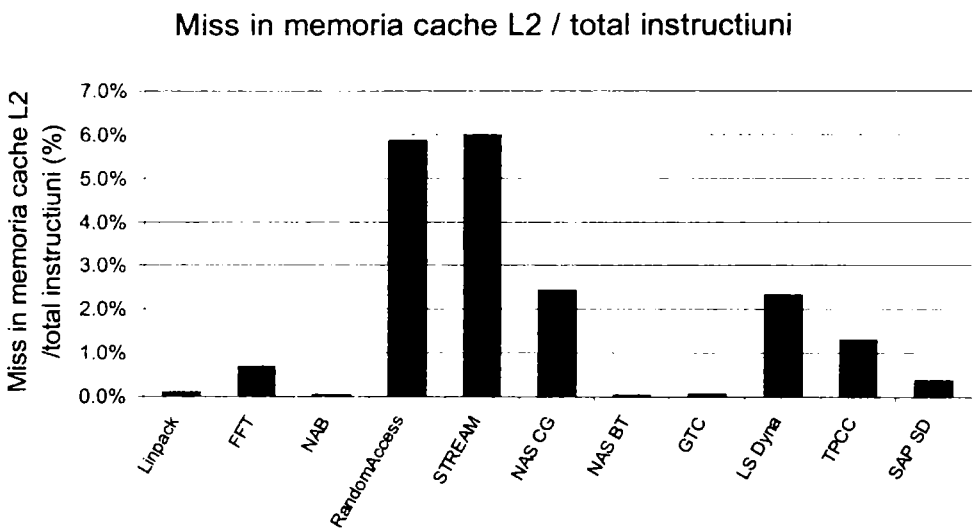


Figura 4.14. Distribuţia *miss*-urilor în memoria *cache*. Figura prezintă *miss*-urile în nivelul 2 al memoriei *cache* raportat la numărul total de instrucţiuni executate pentru fiecare aplicaţie.

Figura 4.15. prezintă aceleaşi date ca şi figura 4.14., dar *miss*-urile sunt împărţite în mai multe categorii, de sus în jos, după cum urmează:

- *miss*-uri a căror date nu sunt prezente în nici o memorie *cache* a unui alt procesor, motiv pentru care trebuie deservite de către memoria principală. Pot fi cauzate de o instrucţiune de scriere, citire sau precitire.
- *miss*-uri a căror date sunt prezente în una sau mai multe memorii *cache,* fară ca să fi fost rescrise de unul dintre procesoare. Pot fi cauzate de o instrucţiune de scriere, citire sau precitire. Aceste date, în funcţie de implementare, pot fi achiziţionate printr-un transfer între memoriile *cache,* sau pot fi deservite de către memoria principală.
- *miss*-uri a căror date sunt prezente în memoria *cache* a unui alt procesor şi pe care acesta le-a rescris. Poate fi cauzat de o instrucţiune de scriere, citire sau precitire. Aceste date pot fi achiziţionate numai printr-un transfer între memoriile *cache*.

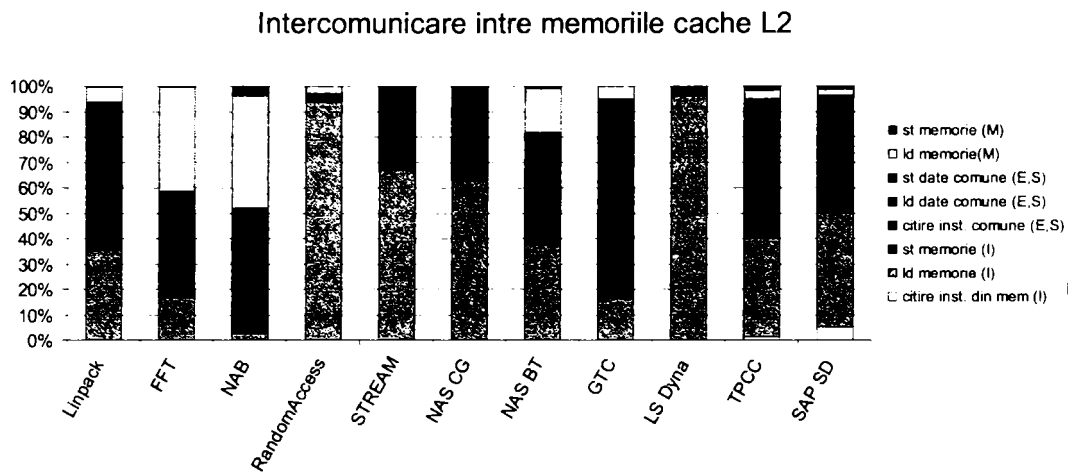


Figura 4.15. Distribuţia *miss*-urilor în memoria *cache* în procente: Această figură prezintă aceleaşi date ca şi Figura 11, diferenţa constând în faptul că fiecare categorie este prezentată ca o fracţiune a *miss*-rilor în momoria *cache* de la nivelul 2 (L2), pentru fiecare aplicaţie.

Analizând rezultatele prezentate în cele două figuri se pot face următoarele observaţii:

- Aplicaţiile *RandomAccess* şi *STREAM* având în vedere caracteristicile lor generează cel mai mare număr de *miss*-uri, deservite de către memoria principală.
- Versiunile paralele ale programelor ştiinţifice sintetice *FFT, NAB* cât şi cele reale *GTC* prezintă acelaşi comportament în ceea ce priveşte modul de accesare al datelor comune. Majoritatea datelor din memorie fiind accesate de către fiecare *thread* soft în parte.
- Aplicaţiile *FFT* şi *NAB* prezintă un nivel ridicat de comunicare între *thread*-uri atât prin instrucţiuni de citire cât şi de scriere a datelor comune. Spre deosebire de acestea, *GTC* accesează datele comune doar prin instrucţiuni de citire. Datele comune în cazul acestei aplicaţii sunt reprezentate de constante ce sunt accesate de toate *thread*-urile soft.
- Aplicaţiile *LSDyna, RandomAccess, STREAM* şi *NAS CG* prezintă acelaşi comportament, majoritatea *miss*-urilor fiind cauzate de accesări a datelor private.
- Între toate aplicaţiile studiate *SAP* cauzează numărul cel mai mare de *miss*-uri de instrucţiuni deservite de memoria principală.

❖ Cu toate că *miss*-urile cauzate de accesări a datelor comune sunt în număr mai mare decât a multor aplicaţii ştiinţifice, majoritatea *miss*-urilor cauzate de cele două aplicaţii comerciale sunt cauzate de accesări a datelor private.

4.5.4. Precitirea datelor.

Dat fiind faptul că viteza de execuţie a aplicaţiilor ştiinţifice (*HPC*) este guvernată în mare măsură de timpul necesar deservirii *miss*-urilor în memoria *cache* şi de abilitatea de a aduce datele în nivelurile rapide ale acesteia, cât mai apropiate de procesor, instrucţiunile de precitire date au un impact semnificativ asupra creşterii vitezei de execuţie.

Tabelul 4.2. prezintă creşterea performanţei fiecărei aplicaţii studiate când instrucţiunile de precitire sunt activate. Studiul a fost realizat prin activarea/dezactivarea mecanismului de precitire, într-un sistem *hardware* de referinţă.

Aplicaţia	CPI fără precitire	CPI cu precitire	Imbunătăţire [%]
Linpack	**1.07**	**0.47**	**126.25 %**
FFT	**1.15**	**1.10**	**4.94 %**
NAB	**4.99**	**1.62**	**207.39 %**
RandomAccess	**11.01**	**11.01**	**0.00 %**
STREAM	**9.58**	**3.38**	**183.37 %**
NAS CG	**6.52**	**2.45**	**166.12 %**
NAS BT	**0.92**	**0.84**	**9.41 %**
GTC	**1.15**	**1.15**	**0 %**
LS Dyna	**8.09**	**2.70**	**199.44 %**

Tabelul 4.2. Efectul precitirii datelor asupra performanţei diferitelor aplicaţii.

Datele prezentate în tabel indică faptul că în cazul utilizării mecanismelor de precitire, aplicaţiile ce accesează datele în mod regulat, predictibil, cum este de exemplu aplicaţia *STREAM,* ating niveluri de performanţă îmbunătăţite de aproximativ două ori, iar aplicaţiile ce accesează datele aleatoriu, cum este de exemplu *RandomAccess* nu prezintă nici o creştere a vitezei de execuţie.

Se pot fâce de asemenea următoarele remarci:

❖ Aplicaţia *GTC* nu beneficiază aproape de loc de instrucţiunile de precitire datorită modului neregulat, impredictibil şi suficient de complex, în care accesează datele din memorie. Din această cauză în această aplicaţie nu există aproape de loc instrucţiuni de precitire introduse de compilator.

❖ Lipsa de îmbunătăţire a vitezei de execuţie a aplicaţiei *FFT* se datorează faptului că fişierul de date de intrare a fost suficient de mic astfel încât a putut fi încadrat în întregime în memoria *cache,* făcând astfel mecanismul de precitire irelevant.

❖ Aplicaţiile *LS Dyna, Linpack, NAS CG* şi *NAB* sunt aplicaţii de algebră liniară iar rezultatele obţinute relevă faptul că acestea beneficiază din plin de pe urma mecanismului de precitire, datorită modului regulat şi predictibil în care accesează memoria.

4.6. Concluzii.

În acest capitol a fost prezentată în detaliu caracterizarea unui set semnificativ de aplicaţii ştiinţifice, ce acoperă spaţiul programelor de calcul matematic intensiv. Pe lângă acestea, pentru a extinde spaţiul de analiză au fost caracterizate şi două aplicaţii comerciale. Au fost prezentate rezultate pentru o gamă variată de studii ce includ, localizarea accesărilor memoriei, sensibilitatea aplicaţiilor la dimensiunile şi structura memoriei cache, distribuţia intercomunicărilor de date şi impactul precitirii datelor asupra performanţei.

Tehnicile utilizate pentru obţinerea datelor sunt acceptate şi folosite pe scară largă în industrie. Contribuţia principală a acestei lucrări este faptul că s-au aplicat un spectru larg de tehnici de caracterizare, atât aplicaţiilor din spectrul *HPC*, cât şi celor din spectrul aplicaţiilor comerciale. Generarea datelor obţinute necesită un efort mare şi o infrastructură extrem de elaborată, mai ales în ceea ce priveşte aplicaţiile *TPC-C* şi *SAP* pentru care este extrem de dificil studiul unei variante reprezentative a acestor aplicaţii.

Observaţiile cele mai importante sunt următoarele:

a. Marea parte a aplicaţiilor din spectrul ştiinţific conţin o componentă importantă de instrucţiuni ce operează cu întregi. Aceasta sugerează faptul că evaluarea performanţei acestui tip de aplicaţii doar pe baza eficienţei procesării instrucţiunilor în virgulă mobilă nu este reprezentativă pentru măsurarea performanţei aplicaţiilor aflate în acest spaţiu.

b. Pentru aplicaţiile de testare, localizarea accesării memoriei este conform aşteptărilor, dar este surprinzător faptul ca o serie de aplicaţii din spectrul HPC au proprietăţi similare cu aplicaţiile comerciale, atât *NAB* cât şi *GTC* numărându-se printre ele.

c. Studiile efectuate pentru evaluarea modului în care aplicaţiile ştiinţifice, utilizează memoria *cache,* studiul distribuirii intercomunicării de date, beneficiul utilizării mecanismului de precitire, relevă poate cea mai importantă contribuţie a acestui capitol şi anume faptul că caracteristicile aplicaţiilor din domeniul ştiinţific sunt foarte diverse. În concluzie nu se poate afirma că un program sau un grup de programe reprezintă pe deplin spaţiul acestui tip de aplicaţii *software*. Din această cauză pentru luarea deciziilor de design arhitectural a noilor sisteme *hardware,* este imperativă analizarea unui spectru larg al acestor aplicaţii.

Contribuţii:

1. Realizarea unui studiu substanţial şi sistematic privind colectarea de date de performanţă pentru un spectru larg de aplicaţii care inplică un regim computaţional greu, care îşi aduce contribuţia la înţelegerea modului de comportare al aplicaţiilor computaţionale şi punerea acestor date la dispoziţia comunităţii implicată în proiectarea microprocesorelor.

2. Efectuarea unei comparaţii între modul de rulare al aplicaţiilor din domeniul ştiinţific şi cele din domeniul comercial.

3. Evidenţierea faptului că contrar accepţiunii generale unitatea de măsură MFLOPS nu este suficientă pentru a evalua performanţa aplicaţiilor de calcul numeric intensiv.

4. Studiul efectelor *prefetch* (mecanismelor de precitire a datelor) asupra performanţei în execuţie a diverselor aplicaţii, lucru care pâna acum, în literatura de specialitate, nu s-a bucurat de o atenţie deosebită. În urma studiului prezent a

rezultat faptul că o serie mare de aplicaţii ştiinţifice în special cele de algebră liniară beneficiază din plin de mecanismele de precitire.

5. Evidenţierea faptului că în cazul aplicaţiilor ştiinţifice, spre deosebire de cele comerciale nu se pot trage concluzii asupra comportamentului de grup al acestor aplicaţii pe baza rezulatelor obţinute analizând numai comportamentul unora dintre ele.

Contribuţia personală în realizarea acestui studiu a constat în:

- iniţierea şi coordonarea studiul prezentat;
- generarea şi validarea fişierele *trace* care au stat la baza obţinerii tuturor datelor experimentale prezentate;
- generarea datelor şi proiectarea metodologiei de validare între *hardware*-ul existent şi datele obţinute din fişierele *trace,* având ca referinţă rata de *miss* în memoria L2 *cache* şi distribuţia instrucţiunilor.
- am contribuit conceptual la proiectarea modulelor ce execută analiza localizării temporale şi spaţiale a accesării locaţiilor de memorie.
- am proiectat şi implementat mecanismul necesar studiului efectelor structurii de precitire a datelor asupra performanţei aplicaţiilor stiinţifice şi am obţinut datele experimentale.
- am contribuit la analizarea întregului set de date experimentale şi la extragerea concluziilor.

5. Tehnici de îmbunătăţire a performanţei procesoarelor superscalar – stadiu actual.

5.1. Bazele modului de procesare *out-of-order*.

Microprocesoarele actuale au încorporat în decursul timpului un întreg şir de concepte, idei, invenţii, inovaţii şi îmbunătăţiri, fiecare nouă achiziţie reprezentând un câştig mai mare sau mai mic, din punctul de vedere al îmbunătăţirii performanţei acestora.

Din punctul de vedere al procesării instrucţiunilor se disting două moduri principale: procesarea *in-order* şi procesarea *out-of-order*. Istoric vorbind, primul mod de execuţie al instrucţiunilor, a fost cel *in-order* sau modelul secvenţial. Cu toate că din punct de vedere cronologic a fost primul mod de execuţie el este utilizat chiar şi la ora actuală de o serie de microprocesoare moderne cum ar fi microprocesorul Niagara realizat de firma Sun Microsystems [5.1]. Acest procesor face parte din familia CMT (*Chip Multi Threaded*). Această familie de procesoare se caracterizează prin încorporarea mai multor unităţi de calcul în aria microprocesorului, fiecare unitate fiind capabilă să execute în paralel unul sau mai multe *thread*-uri soft. Pentru acest tip de procesoare performanţa se reflectă în volumul de muncă efectuat în fiecare unitate de timp şi nu în viteza de execuţie a unui singur *thread* soft.

Modul secvenţial de execuţie al unui program, implică existenţa unui *program counter* (PC), pe care procesorul îl foloseşte pentru a citi instrucţiunile din memorie şi a le lansa în execuţie. În timpul procesului de execuţie sunt efectuate operaţii de citire sau scriere date din/sau în memorie, sau se operează asupra regiştrilor microprocesorului. După finalizarea execuţiei şi modificarea stării logice, sau arhitecturale a procesorului, cu rezultatul instrucţiunii procesate, urmează lansarea în execuţie a instrucţiunii următoare.

În principiu un microprocesor *in-order* execută deci rând pe rând instrucţiunile în ordinea secvenţială în care acestea se află în program. Acest mod de execuţie poate fi modificat însă de instrucţunile de salt, *branch* sau *jump,* care direcţionează execuţia programului către o instrucţiune, alta decât cea care urmează în ordinea secvenţială.

În cazul apariţiei acestui gen de instrucţiuni, sau ori de câte ori execuţia programului este perturbată datorită unui eveniment extern sau intern, ca de exemplu o întrerupere sau un *overflow*, pentru păstrarea unei execuţii corecte, trebuie menţinută starea logică a procesorului, din momentul apariţiei acelui eveniment. Prin starea logică a procesorului se înţelege setul de valori existente la un moment dat în: regiştrii de uz general, registrul care stochează PC-ul şi alţi regiştrii cu funcţii speciale. Păstrarea stării logice a procesorului în momentul apariţiei întreruperii este importantă pentru că, după execuţia rutinei de rezolvare a întreruperii, procesorul să poată relua procesarea cu instrucţiunea a cărei execuţie a fost întreruptă, pornind exact din starea în care a fost înainte de apariţia perturbării.

Una dintre cele mai importante achiziţii în domeniul construcţiei de microprocesoare a fost tehnologia *pipeline* [5.2];[5.3]. Un *pipeline* poate fi comparat cu o linie de asamblare, instrucţiunile fiind procesate în etape în fiecare etapă executând-se doar o mică parte din total. Rezultatele obţinute în urma procesării într-o anumită etapă a *pipeline*-ului servesc ca date de intrare pentru etapa imediat următoare. La un moment dat în *pipeline* sunt prezente mai multe instrucţiuni în diverse faze de execuţie. În tehnologia originală de implementare a *pipeline*-lui, doar o singură instrucţiune este citită într-un interval de *clock*.

Tehnologia *pipeline* într-o formă îmbunătăţită este implementată şi în prezent în procesoarele moderne. La sfirşitul anilor `80 au început să apară microprocesoarele superscalar care se caracterizează prin capacitatea de a iniţia instrucţiuni multiple, pe durata unei perioade de *clock* şi să le execute în paralel, folosind multiple *pipeline*-uri [5.4], [5.5].

Bariera iniţierii unei singure instrucţiuni într-un ciclu de ceas a fost astfel depăşită odată cu implementarea noii tehnologii, iar începând cu anii 90 metoda superscalar a ramas standardul în proiectarea microprocesoarelor performante.

Odată cu apariţia microprocesoarelor superscalar s-a renunţat la modul secvenţial de procesare a instrucţiunilor şi s-a adoptat aşa numitul mod de execuţie *out-of-order*, cu deosebite avantaje în privinţa performanţei.

Pentru a obţine o performanţă înaltă procesoarele superscalar (deoarece mai multe instrucţiuni sunt executate în paralel), deviază puternic de la modul secvenţial de execuţie. Astfel, programul în sine oferă informaţie doar asupra instrucţiunilor ce trebuiesc executate şi nu asupra modului în care acestea urmează a fi de fapt executate. În execuţia unui program procesorul superscalar porneşte de la varianta secvenţială a acestuia, elimină părţile inutile de secvenţialitate, transformându-l într-un model paralel, pe care îl execută păstrând totuşi în exterior aparenţa a unei execuţii secvenţiale.

La baza execuţiei în paralel a instrucţiunilor stă identificarea şi exploatarea aşa numitului *Instruction Level Parallelism* (ILP) care este de fapt o măsură a numărului de instrucţiuni dintr-un program ce pot fi executate în paralel [5.6].

În vederea exploatării eficiente a ILP-ului, procesoarele superscalar conţin mai multe unităţi de execuţie şi încearcă să lanseze în execuţie câte o nouă instrucţiune în fiecare interval de *clock*. Una din problemele care apar în acest caz este aceea că execuţia corectă a programului cere ca fiecare instrucţiune să aştepte până ce instrucţiunea premergătoare şi-a încheiat execuţia. Din aceasta cauză pentru a putea executa instrucţiuni în paralel, microprocesorul superscalar în momentul în care lansează în execuţie o instrucţiune trebuie să cunoască instrucţiunile de care aceasta este dependentă. Mai precis trebuie să cunoască instrucţiunile ce urmează să producă operanzii ce vor fi citiţi de instrucţiunea în cauză, instrucţiuni care se pot afla în una din următoarele situaţii: sunt deja în faza de execuţie, sau au fost distribuite unităţilor de execuţie, dar aşteaptă finalizarea execuţiei unor instrucţiuni premergătoare.

Instrucţiunile ce sunt independente de instrucţiuni premergătoare pot fi lansate în execuţie în orice ordine urmând a fi retrase din execuţie în ordinea secvenţială în care apar în program. Microprocesoarele şi compilatoarele moderne utilizează o serie de tehnici sofisticate, cu scopul de a mări şi exploata la cel mai înalt nivel gradul de paralelism al instrucţiunilor. Din categoria instrucţiunilor ce pot fi executate în paralel fac parte instrucţiunile de citire sau scriere din memorie, instrucţiunile ce operează cu întregi precum şi instrucţiunile ce operează cu operanzi reali. Dificultăţile în procesare apar în cazul instrucţiunilor condiţionale *branch* sau *jump,* instrucţiuni ce nu pot fi executate în paralel, datorită dependenţelor dintre acestea şi instrucţiunile ce depind de rezultatul acestora.

În principiu pentru procesarea în paralel a instrucţiunilor un microprocesor *out-of-order* trebuie să aibă capacitatea de a citi şi/sau a decodifica mai multe instrucţiuni în acelaşi interval de ceas. După ce au fost citite din memoria *cache* de instrucţiuni, instrucţiunile sunt separate; singurele conexiuni între ele ramânând dependenţele de date. Execuţia lor se face cu un minimum de restricţii ce ţin mai ales de ordinea în care au fost citite. În timpul procesării instrucţiunilor, procesorul menţine suficientă informaţie despre ordinea secvenţială a acestora, încât să poată fi reaşezate în această ordine înainte de a modifica starea logică a *chip*-ului, în vederea implementării întreruperilor precise [5.7].

Principalii factori limitatori ai ILP-ului, cu efect negativ asupra performanţei microprocesorului, sunt dependenţele de date şi cele de control, precum şi *miss*-urile în memoria *cache*. Pentru execuţia instrucţiunilor în paralel arhitectura microprocesorului trebuie să fie dotată cu mijloace *hardware* care să aibă disponibilităţi pentru:

- determinarea şi rezolvarea dependenţelor între instrucţiuni;
- strategii de determinare a momentului în care o instrucţiune este pregătită pentru a fi executată;
- tehnici de a transmite date de la o operaţie la alta.

5.2. Dependenţele de date şi control. [5.8]

a) Dependenţele de control identifică relaţiile ce se stabilesc între instrucţiunile condiţionale de salt (*branch*, *jump*, etc.) şi instrucţiunile a căror execuţie depinde de direcţia în care se va îndrepta execuţia, ca urmare a rezolvării instrucţiunilor de control.

Instrucţiunile existente în blocurile primare conferă un anumit nivel de paralelism, deoarece în limitele acestor blocuri nu există dependenţe de control. Pentru obţinerea unui nivel şi mai ridicat al ILP-ului, trebuie rezolvate însă problemele care sunt determinate de instrucţiunile condiţionale *branch* sau *jump*.

Una dintre metodele folosite pentru a soluţiona această problemă este utilizarea predictoarelor [5.9], [5.10], al căror rol este prezicerea rezultatului unei instrucţiuni condiţionale înainte ca acesta să fie disponibil, ca urmare a execuţiei acestei instrucţiuni. În acest fel procesorul este direcţionat, în mod speculativ, spre

a citi şi executa instrucţiunile din direcţia prezisă. În cazul în care predicţia se dovedeşte a fi corectă, instrucţiunile devin nespeculative şi pot să modifice starea logică a procesorului, ca oricare altă instrucţiune obişnuită. În caz contrar trebuie luate măsuri pentru anularea efectului produs de instrucţiunile executate speculativ şi de a nu permite ca starea procesorului să fie modificată de rezultatul acestora.

b). Dependenţele de date reprezintă relaţiile ce se stabilesc între instrucţiunile a căror execuţie depinde de disponibilizarea rezultatelor unor instrucţiuni anterioare. La fel ca şi dependenţele de control şi cele de date reprezintă un factor ce limitează posibilitatea de procesare în paralel a instrucţiunilor. În cazul în care multiple instrucţiuni accesează acelaşi mediu de stocare, acţiune numită **hazard**, este necesară parcurgerea unor etape menite să dea siguranţă că accesarea mediul de stocare este făcută în ordinea corectă. Asemeni soluţiei adoptate în cazul rezolvării dependenţelor de control şi pentru rezolvarea dependenţelor de date se poate recurge la predictoare de valori asemeni celui prezentat în [5.11].

Se disting două tipuri de dependenţe de date [5.3]:

b.1. naturale,

b.2. artificiale.

b.1. Dependenţele naturale, sunt cele existente în versiunea secvenţială de execuţie a programului şi sunt numite *true dependencies* sau citit după scris (**RAW**). Pentru rezolvarea acestui tip de dependenţă trebuie ca instrucţiunea consumator (aflată la un moment de timp mai îndepărtat) să citească mediul de stocare comun, numai după ce instrucţiunea producător (aflată la un moment de timp mai apropiat) şi-a scris rezultatul în mediul de stocare.

b.2. Dependenţe artificiale (hazarduri), acest tip de dependenţe rezultă ca urmare a execuţiei nesecvenţiale a programului, de către procesorul *out-of-order*. Sunt două tipuri de asemenea hazarduri: scris după citit (**WAR**) sau scris după scris (**WAW**).

- Hazardul **WAR** apare când o instrucţiune trebuie să-şi scrie rezultatul într-un mediu de stocare, dar trebuie să aştepte ca instrucţiunile precedente, care trebuie să citească valoarea existentă

în mediul de stocare la un moment de timp anterior, să facă acest lucru.

- Hazardul **WAW** apare când multiple instrucţiuni modifică acelaşi mediu de stocare, condiţia unei procesări corecte cere ca această modificare să se facă, în ordine cronologică.

Cele trei tipuri de hazarduri sunt exemplificate cu ajutorul setului de instrucţiuni prezentat în figura 5.1.

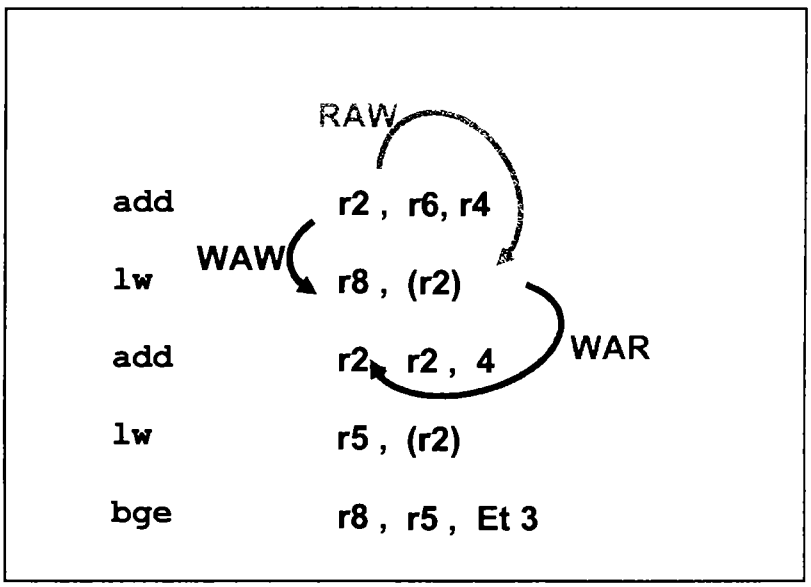


Figura 5.1 Exemplu de hazarduri de date care implică regiştrii.

Prima instrucţiune ***add*** produce o anumită valoare în registrul „r2" care apoi este folosită atât de instrucţiunea ***lw*** cât şi de cel de-al doilea ***add***. Aceasta constituie un hazard de tip RAW, deoarece atât ***lw***-ul cât şi al doilea ***add*** trebuie să aştepte până când prima instrucţiune ***add*** scrie registrul „r2".

Între prima şi a doua instrucţiune ***add*** se crează un hazard de tip WAW, deoarece ambele instrucţiuni scriu acelaşi registru „r2". O execuţie dinamică, corectă a acestui set de instrucţiuni, trebuie să asigure ca primul ***add*** să fie executat după cel de-al doilea.

Între primul ***lw*** şi ***add***-ul care îl urmează, se creează un *hazard* de tip WAR deoarece ***lw*** trebuie să citească valoarea stocată în memorie la adresa specificată de registrul „r2", înainte ca ***add***-ul să-şi salveze rezultatul în „r2".

5.3. Unele măsuri pentru îmbunătăţirea performanţei microprocesorului, specifice diverselor faze de procesare a instrucţiunilor.

Etapele principale de procesare a unei instrucţiuni sunt [5.3]:

a. citirea instrucţiunilor;
b. decodificarea;
c. distribuirea şi execuţia;
d. operaţiile cu memoria;
e. retragerea din execuţie.

a. Etapa de citire a instrucţiunilor; are rolul de a furniza un şir continuu de instrucţiuni etapelor următoare ale *pipeline*-ului. Pentru aceasta, majoritatea microprocesoarelor conţin o memorie *cache*, în care sunt stocate instrucţiunile cele mai recent folosite [5.12]. Procesorul foloseşte PC-ul pentru a cauta dacă instrucţiunea care urmează să o execute, este prezentă în una dintre liniile aceastei memorii rapide. În cazul în care ea este prezentă, spunem că avem un *hit* în *cache*, în caz contrar spunem ca avem un *miss* şi ca urmare linia care conţine instrucţiunea este adusă în memoria *cache* din memoria principală.

În cazul tipic de funcţionare, instrucţiunile sunt citite din memoria *cache* de la adresa rezultată din PC-ul precedent, plus numărul de instrucţiuni citite în perioada de *clock* precedentă. În cazul unei instrucţiuni de control (*branch)* însă, în urma unui proces de redirecţionare, instrucţiunea ce urmează a fi executată este furnizată de ţinta acesteia.

Procesul de rezolvare a unei instrucţiuni de control (*branch)* poate fi separat în următoarele etape:

- identificarea instrucţiunilor de control;
- calcularea ţintei instrucţiunii de control;

- determinarea direcţiei instrucţiunii de control (adevărat sau fals);
- transferarea controlului prin redirecţionarea citirii instrucţiunilor în cazul în care condiţia instrucţiunii de control se dovedeşte a fi adevarată.

Datorită întirzierilor ce apar în timpul acestui proces de redirecţionare, existenţa unui mecanism de rezolvare rapidă a instrucţiunilor de control este critică pentru obţinerea unei execuţii peformante. Din acest motiv procesoarele moderne utilizează mecanisme speculative denumite predictoare pentru prezicerea direcţiei şi ţintei instrucţiunilor de *branch*. Pentru a asigura un flux continuu de instrucţiuni în *pipeline*, aceste predictoare acţionează încă din etapa de citire din memorie a instrucţiunilor.

Din păcate utilizarea unor astfel de mecanisme poate avea efecte negative asupra performanţei deoarece de multe ori speculaţiile se dovedesc a fi false. În aceste condiţii pentru a asigura o execuţie corectă a programului trebuie să se prevadă mecanisme de revenire, care în cele mai multe cazuri afectează negativ viteza de execuţie a procesorului.

În ceea ce priveşte instrucţiunile de control se identifică două direcţii ce pot fi atacate pentru a îmbunătăţi performanţa în execuţie a acestora:

1. Reducerea timpului pierdut în cazul unei predicţii greşite.

Pentru reducerea timpului pierdut în cazul unei predicţii greşite, procesoarele moderne utilizează o serie de tehnici ca de exemplu:

- utilizarea unui *pipeline* mai scurt,
- utilizarea unei memorii *trace cache* [5.13] în care sunt stocate informaţii dinamice despre calea urmată în urma execuţiei programului,
- adăugarea unor blocuri *hardware* ce permit executarea atât a instrucţiunilor de pe direcţia FALS, cât şi a celor de pe direcţia ADEVARAT a instrucţiunii de *branch*.

2. Reducerea numărului de predicţii greşite.

În cazul procesoarelor actuale reducerea numărului de predicţii greşite ale instrucţiunilor de control, se realizează prin mecanisme sofisticate de creştere a acurateţii predictoarelor, precum şi prin găsirea unor metode de reducere a

"timpului de învăţare" a acestuia. (Prin "timp de învăţare" înţelegând timpul în care odată cu efectuarea unui număr de instrucţiuni condiţionale predicţiile devin tot mai precise.) În acest sens una din soluţii este utilizarea unui predictor hibrid, cu două niveluri [5.10]. Durata de învăţarea a nivelului unu este mai scurtă, el fiind capabil să genereze o rată de succes destul de bună într-un timp foarte scurt. Pentru nivelul doi "durata de învăţare" este mai mare, dar şi acurateţea predicţiilor este mai bună).

b. Etapa de decodificare; are rolul de a identifica instrucţiunile în funcţie de codul fiecăreia. Pe parcursul acestei etape instrucţiunile sunt extrase din *buffer*-ele de citire, sunt analizate şi li se stabilesc legăturile de dependenţă reale de tip (RAW), după care sunt rezolvate dependenţele artificiale de tip (WAR sau WAW), cauzate de refolosirea regiştrilor generali. După efectuarea acestei operaţii, instrucţiunile sunt distribuite în *buffer*-ele asociate cu unităţile de execuţie pentru a fi procesate la un moment ulterior, în funcţie de tipul fiecărei instrucţiuni.

Rezultatul decodificării instrucţiunilor este un set de biţi, care specifică următoarele informaţii despre instrucţiunea în cauză:

- operaţia ce trebuie executată,
- identificatorii mediilor de stocare unde se găsesc operanzii ce trebuiesc citiţi,
- locul unde va fi stocat rezultatul instrucţiunii.

c. Etapa de distribuire şi execuţie; are rolul de a determina, care dintre instrucţiuni sunt pregătite pentru a fi lansate în execuţie. Ideal ar fi ca o instrucţiune să fie gata de execuţie imediat ce operanzii sursă devin disponibili. Lucrul aceasta însă nu este întodeauna posibil, deoarece lansarea în execuţie depinde în mare parte şi de disponibilitatea unităţilor de execuţie, de porturile fişierului de regiştrii şi de interconexiuni. Odată ce toate interdependenţele de date au fost rezolvate şi unitatea de execuţie destinată acestui tip de instrucţiune este liberă, instrucţiunea este lansată în execuţie. Execuţia în sine se efectuează în funcţie de disponibilizarea valorilor operanzilor citiţi de fiecare instrucţiune în parte şi nu de ordinea în care ele au fost citite din memorie. Această caracteristică

importantă, prezentă în cele mai multe implementări superscalar, este cunoscută ca *dynamic instruction scheduling* [5.14], [5.15].

d. Etapa de accesare a locaţiilor de memorie; este o etapă importantă deoarece viteza cu care instrucţiunile de accesare a memoriei sunt procesate, în cazul procesoarelor *out-of-order* are o influenţă decisivă asupra performanţei, motiv pentru care trebuie să i se acorde o atenţie deosebită.

Pentru a reduce timpul de accesare a memoriei, aceasta este organizată în ierarhii. În acest fel majoritatea cererilor de date sunt deservite de blocurile de memorie aflate la nivelele inferioare ale ierarhiei, care sunt mai rapide, doar o mică parte urmând a fi deservite de memoria principală care este mai lentă. La ora actuală aproape toate procesoarele moderne conţin mai multe niveluri de memorie rapidă (*cache*) pentru stocarea datelor, de exemplu: o memorie primară foarte rapidă şi o memorie secundară ceva mai lentă. Memoriile *cache* în cazul procesoarelor moderne fac parte integrantă din *chip,* reducându-se astfel timpul de access al acestora.

Spre deosebire de instrucţiunile aritmetice (ALU), pentru care este posibilă identificarea încă în faza de decodificare a regiştrilor ce vor fi accesaţi de o anumită instrucţiune, pentru instrucţiunile de accesare a memoriei este posibilă identificarea locaţiilor de memorie ce vor fi accesate, numai după parcurgerea fazei de execuţie.

Determinarea adresei locaţiei de memorie ce urmează a fi accesată, necesită o operaţie de adunare ce este de obicei efectuată între doi operanzi întregi. Pentru aceasta, instrucţiunile de accesare a memoriei sunt distribuite, în faza de execuţie a *pipeline*-ului, unităţilor aritmetice ce efectuează operaţia de adunare necesară determinării adresei locaţiei de memorie, ce urmează a fi citită sau scrisă. Urmează apoi faza de translaţie a adresei, din adresa virtuală (generată de procesor) în adresa fizică, utilizată pentru accesarea memoriei. Pentru urgentarea efectuării acestei operaţii procesoarele moderne utilizează un fel de memorie *cache* denumită TLB (*translation lookaside buffer)* [5.16]. Odată ce adresa fizică a locaţiei de memorie ce urmează a fi accesată a fost obţinută în urma translaţiei, instrucţiunea poate fi trimisă spre procesare memoriei.

Trebuie subliniat faptul că deşi fazele de translaţie şi cea de accesare a memoriei par a fi parcurse în serie, în multe tipuri de procesoare superscalar aceste operaţii sunt efectuate de fapt în paralel. Accesarea primului nivel de memorie *cache*

este efectuat în paralel cu accesarea TLB-ului, iar adresa rezultată în urma translaţiei este folosită pentru a o compara cu *tag*-urile stocate în *cache*, determinându-se în felul acesta dacă accesul a fost *hit* sau *miss*.

Mai multe instrucţiuni de accesare a memoriei pot fi executate în paralel şi procesate nesecvenţial, dacă hazardurile de date au fost eliminate şi se păstrează semantica execuţiei. Procesoarele moderne dispun de o serie de mecanisme ce permit executarea instrucţiunilor de citire si scriere în memorie, în afara ordinii în care acestea apar în program. Pentru aceasta se utilizează *buffer*-ul instrucţiunilor de scriere în memorie. Acest *buffer* stochează temporar adresele locaţiilor de memorie ce urmează a fi accesate, împreună cu datele ce vor fi scrise la adresele respective de instrucţiunile în cauză, până când instrucţiunile de scriere sunt retrase din execuţie. Un astfel de mecanism asigură posibilităţile de transfer a datelor din *buffer* între o instrucţiune de scriere ce nu a fost încă retrasă din execuţie şi o instrucţiune de citire executată cronologic mai tirziu, instrucţiune care urmează să acceseze aceeaşi locaţie de memorie. În acest fel instrucţiunea de citire urmează să citească din *buffer* datele corecte şi nu cele neactualizate încă din memorie.

Pentru o procesare performantă este imperios necesar ca instrucţiunile de citire să fie executate cât mai rapid, chiar în afara ordinii secvenţiale. O execuţie de asemenea manieră poate conduce la rezultate incorecte atunci când există instrucţiuni de scriere ce au fost executate la momente cronologic anterioare ce accesează aceeaşi locaţie de memorie ca şi instrucţiunea de citire şi care nu şi-au scris încă datele în memorie. Pentru a evita astfel de situaţii, înainte ca o instrucţiune de citire să fie lasată să accesese memoria, *buffer*-ul de instrucţiuni de scriere este scanat pentru a verifica dacă conţine instrucţiuni nerezolvate, adică fie că nu au calculată adresa locaţiei de memorie ce o vor accesa, fie urmează să acceseze aceeaşi adresă cu instrucţiunea de citire.

Unele procesoare *out-of-order* moderne implementează în plus şi un mecanism de prezicere al cărui scop este acela de a preciza faptul că o instrucţiune de citire accesează aceeaşi locaţie de memorie cu o instrucţiune de scriere, înainte ca adresele celor două instrucţiuni să fie cunoscute. Prezicerea se face în mod speculativ şi din această cauză este nevoie de un mecanism de revenire în cazul unei erori [5.17].

În figura 5.2 este prezentată logica de detectare a hazardurilor de memorie.

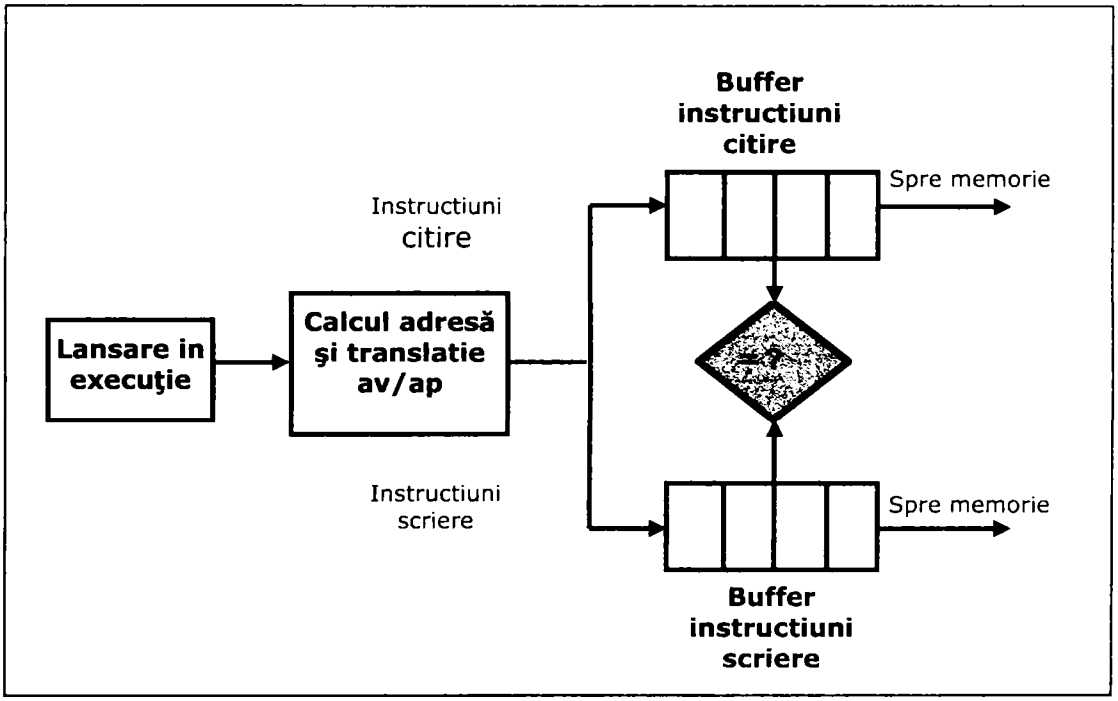


Figura 5.2. Logica de detectare a hazardurilor de memorie. Adresele instrucţiunilor de citire sunt comparate cu cele ale instrucţiunilor de scriere premergătoare pentru a se valida dacă acestea pot accesa memoria.

Adresele instrucţiunilor de scriere sunt stocate într-un *queue*. Aceste adrese trebuie să rămână în *buffer* până când instrucţiunile sunt gata pentru a fi retrase din execuţie. Adresele noilor instrucţiuni de citire sunt comparate cu cele existente în *buffer*-ul instrucţiunilor de scriere. În cazul în care sunt identice instrucţiunea de scriere trebuie să aştepte până când instrucţiunea de scriere în cauză îşi termină execuţia.

Asemeni instrucţiunilor de control şi pentru instrucţiunile de accesare a memoriei se identifică două direcţii ce pot fi atacate pentru a îmbunătăţi performanţa în execuţie a acestora:

1. Reducerea timpului pierdut datorită *miss*-urilor în memoria *cache.*
2. Reducerea numărului de *miss*-uri în memoria *cache.*

1. Pentru reducerea timpului pierdut în cazul unui *miss* în memoria *cache*, procesoarele moderne utilizează o ierarhie de memorie pe mai multe niveluri aşa cum este prezentat în figura 5.3. Pentru reducerea timpului în care datele sunt returnate, în cazul unui *miss* este bine să se adopte o soluţie în care nivelurile „1" si „2" de *cache* sunt accesate în paralel, urmând ca accesul nivelului „2" să fie abandonat, în cazul unui *hit* in nivelul „1".

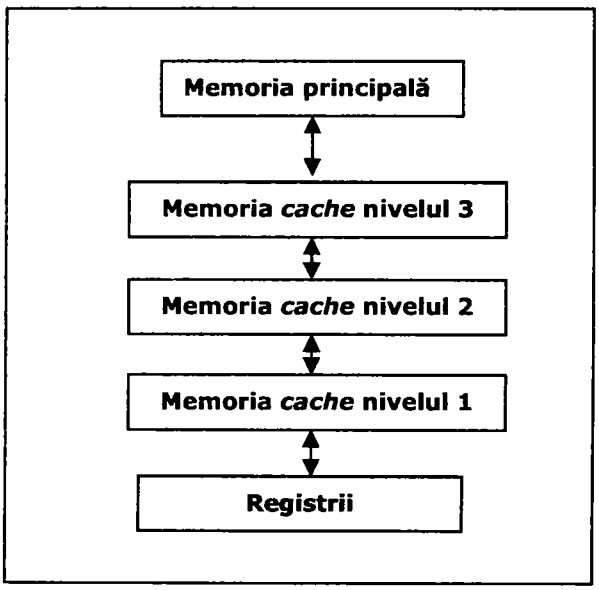


Figura 5.3. Organizarea ierarhiei de memorie.

2. Pentru reducerea numărului de *miss*-uri în memoria *cache* sunt utilizate în principal două metode:

- mecanisme de precitire (*prefetch)* ce pot fi realizate atât la nivelul *software* utilizând unele instrucţiuni speciale, cât şi la nivelul *hardware*-ului folosind structuri specializate. Aceste mecanisme dată fiind o anumită adresă de accesare a memoriei, citesc în avans datele ce se află la adresele următoare, şi le stochează în memoria *cache* sau în *buffer*-e de precitire predestinate.
- structuri de memorie numite *victim cache* [5.18]. Aceste structuri sunt poziţionate lângă memoria *cache* şi au rolul de stocare a datelelor ce sunt eliminate din aceasta. În cazul unui *miss* în *cache* şi a unui *hit* în *victim cache* datele din acestea sunt reinstalate înapoi în memoria *cache*.

e. Ètapa de retragere din execuţie; este etapa finală în execuţia unei instrucţiuni, etapă în care rezultatele instrucţiunii în cauză modifică starea logică a procesorului, păstrând aparenţa unei execuţii secvenţiale. Existenţa unui mecanism pentru reaşezarea instrucţiunilor în ordinea în care au fost citite este important pentru evitarea unei execuţii greşite, mai ales în cazul apariţiei unor condiţii, interne (excepţie) sau externe (întrerupere) ce pot conduce la întreruperea execuţiei. Mecanismul în cauză utilizează un *buffer* de reordonare [5.7]. Locaţiile acestuia stochează valorile rezultatelor pe măsură ce acestea devin disponibile, înainte ca instrucţiunile ce le produc să fie retrase din execuţie. Tot aici este stocat şi PC-ul instrucţiunilor, precum şi toate condiţiile întreruperilor ce pot apărea, ca urmare a execuţiei unei instrucţiuni. Aceste date sunt folosite pentru a informa structurile *hardware* implicate în faza de retragere din execuţie, când trebuie iniţiată o excepţie şi care este PC-ul instrucţiunii ce o cauzează, elemente necesare pentru a păstra starea precisă a întreruperilor.

Mecanismele prezentate destinate îmbunătăţiri performanţei, utilizate în diversele faze de procesare a unei instrucţiuni, au o pondere diferită în nivelul global de performanţă al procesorului. Trebuie menţionat faptul că efectele utilizării fiecăruia dintre mecanismele de mai sus asupra performanţei este influenţată de configurarea arhitecturală a procesorului studiat, sau dupa caz a simulatorului de performanţă ce-l descrie pe acesta, precum şi de structura aplicaţiei rulate.

Pentru a avea o imagine asupra contribuţiei câtorva dintre mecanismele consacrate utilizate la creşterea performanţei microprocesorului, în figura 5.4. se prezintă rezultatele obţinute de autorul tezei, în urma unui studiu asupra nivelului de performanţă exprimat în CPI, pentru cuantificarea beneficiilor obţinute în cazul în care se utilizează:

- nivelului doi al ierarhiei de memorie *cache*;
- predictorul hibrid de instrucţiuni condiţionale;
- mecanismul de dezambiguare a accesărilor memoriei.

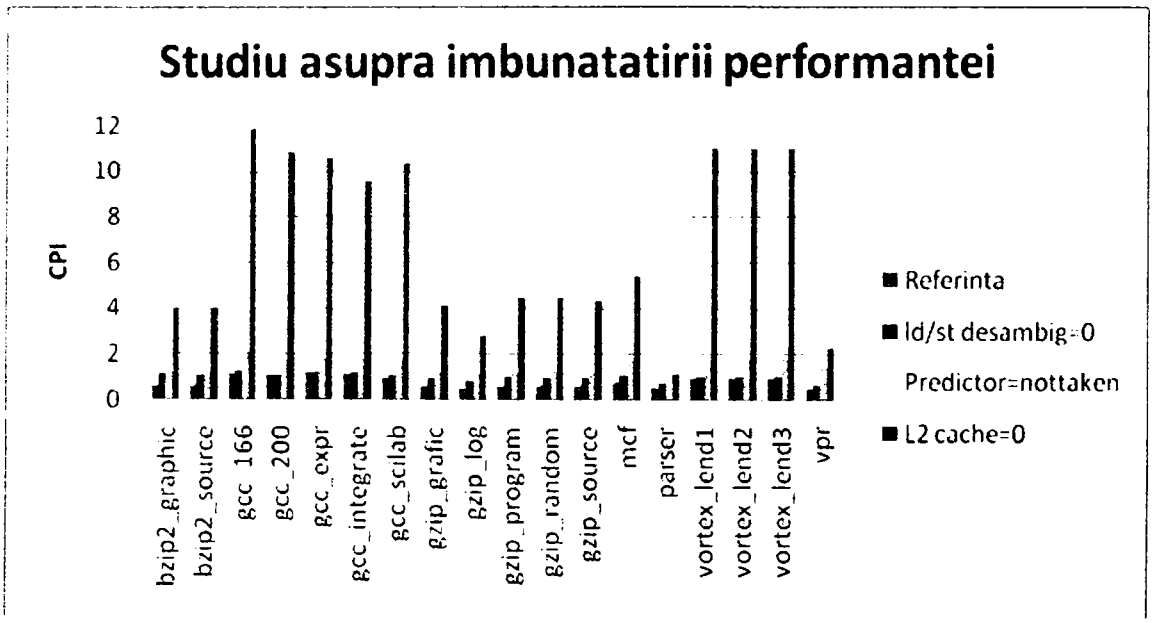


Figura 5.4. Îmbunătăţirea performanţei rezultate în urma utilizării unor structuri *hardware* larg răspândite.

Datele prezentate în figură au fost generate utilizând simulatorul *sim-outorder* ce face parte din suita SimpleScalar [5.19] şi sunt specifice arhitecturii simulate de *sim-outorder* în configurarea prezentată în tabelul 7.1. Studiul a fost efectuat asupra unui subset al aplicaţiilor SPEC CPU2000 prezentat în Anexa 2.

În figura 5.4. sunt prezentate următoarele cazuri:

a) cazul de referinţă (culoare albastră) în care sunt activate concomitent următoarele mecanisme:

- nivelul doi al ierarhiei de memorie *cache*,
- un predictor hibrid al instrucţiunilor condiţionale,
- mecanismul de dezambiguare al accesărilor memoriei.

b) cazul în care nivelul doi al ierarhiei de memorie lipseşte (culoare violet)

c) cazul în care s-a folosit în locul predictorului hibrid, un predictor primitiv al instrucţiunilor condiţionale, predictorul *nottaken,* care acţionează

d) ca şi când toate instrucţiunile de control condiţionale ar lua direcţia FALS. (culoare verde)

e) cazul în care mecanismul de dezambiguare al accesărilor memoriei lipseşte. Astfel o instrucţiune de citire din memorie trebuie să aştepte până când instrucţiunile de scriere, cronologic anterioare şi-au terminat execuţia (culoare roşie).

Datele prezentate în figura 5.4. scot în evidenţă următoarele:

- Cea mai mare influenţă asupra creşterii nivelului de performanţă îl are existenţa nivelului doi al ierarhiei de memorie *cache*. Întreaga suită de aplicaţii testate beneficiază de existenţa acestui nivel, cele mai mari îmbunătăţiri fiind constatate în cazul aplicaţiilor ce accesează repetitiv aceleaşi date din memorie. Astfel performanţa aplicaţiei *vortex* este de 11 ori mai mare în cazul de referinţă comparativ cu cazul în care memoria cache de la nivelul doi lipseşte.

- Un alt factor important asupra creşterii vitezei de execuţie este existenţa unui predictor performant al instrucţiunilor condiţionale. Existenţa unui predictor hibrid determină o creştere a performanţei în medie de două ori, pentru întregul set de aplicaţii studiat. Cea mai mare îmbunătaţire se remarcă în cazul aplicaţiei *vpr* pentru care performanţa în cazul de referinţă este de 2.8 ori mai mare decât în cazul în care a fost utilizat predictorul *nottaken.*

- În legătură cu mecanismul de dezambiguare al acesărilor memoriei se remarcă faptul că acest mecanism influenţează cel mai puţin creşterea vitezei de execuţie. Aplicaţiile ce accesează seturi largi de date în memorie beneficiază cel mai mult de existenţa lui. Viteza de execuţie a celor două aplicaţii ce execută comprimări/decomprimări date *bzip2* şi *gzip* este cu aproximativ 1.8 ori, respectiv 1.4 ori mai mare în cazul de referinţă. Pe de altă parte creşterile performanţei aplicaţiilor *gcc* si *vortex* sunt nesemnificative.

In tabelul 5.1 se face o sinteză a datelor prezentate în figura 5.4.

Tipul aplicaţiei	CPI			
	Toate mecanismele activate	Dezactivat		
		Nivelul doi cache	Predictor hibrid	Dezambiguare accesări memorie
bzip 2	0,8	4	1,3	1,3
gcc	1,1	10,7	1,9	1,1
gzip	0,25	3,8	1,2	1
mcf	0.8	5,5	1,8	1
parser	0,5	1,2	1,6	0,7
vortex	1	11	1,9	1
vpr	0,5	2,2	1,7	0,7
CPI mediu pt. aplicaţiile studiate	0.83	4,48	1,62	0,97

Tabelul 5.1. Valorile medii ale CPI pentru fiecare aplicaţie, precum şi CPI mediu pentru aplicaţiile luate în studiu.

5.4. Direcţii actuale în proiectarea microprocesoarelor.

Aşa cum rezultă din cele prezentate anterior îmbunătăţiri semnificative în creşterea vitezei de execuţie a unui procesor rezultă din posibilitatea de a executa mai multe instrucţiuni în paralel. Microprocesorul superscalar, este echipat cu mecanismele necesare pentru a exploata ILP existent într-un program secvenţial având capacitatea de procesare în paralel, (în afara ordinii secvenţiale) a instrucţiunilor independente. Deşi în cazul acestui procesor execuţia în paralel este limitată de dependenţele între instrucţiuni, această soluţie arhitecturală a satisfăcut o perioadă îndelungată parametrii de performanţă impuşi de majoritatea aplicaţiilor software.

În prezent utilizatorii de *software* solicită efectuarea unui număr tot mai mare de aplicaţii ce pot fi rulate în paralel, asemeni serverelor web, ce conţin un număr ridicat de *thread*-uri soft, sau cer rularea mai multor aplicaţii simultan, ca de exemplu navigarea pe internet, ascultarea de muzică şi codificarea/decodificarea imaginilor video.

Pentru sàtisfacerea solicitărilor tot mai mari ale utilizatorilor, cu atât mai mult cu cât tehnologia actuală de fabricare a *chip*-urilor are resursele pentru integrarea unui număr din ce în ce mai mare de tranzistori în aria acestuia, proiectanţii de procesoare studiază noi soluţii arhitecturale, dintre care unele au fost deja relizate practic. Trei dintre aceste soluţii implementate în noile procesoare sunt prezentate în continuare.

- **Simultaneous Multithreading (SMT).**

SMT este o technică prin care se eficientizează utilizarea resursele *hardware* ale microprocesorului, acesta fiind capabil să citească şi să execute simultan instrucţiuni componente a două *thread*-uri soft. În cazul cel mai general legătura dintre un *thread* şi o unitate de execuţie este pur dinamică. În principiu în cazul SMT un procesor fizic este făcut să apară sistemului de operare şi aplicaţiilor *software* ca fiind alcătuit din mai multe procesoare logice. În această implementare fiecare procesor logic are starea lui arhitecturală proprie şi acces la resursele procesorului fizic unic. Din punct de vedere al sistemelor de operare acestea pot lansa în execuţie programe către procesoarele logice, aşa cum le-ar lansa în sistemele multiprocesor. Din punct de vedere microarhitectural aceasta înseamnă că instrucţiunile rulate de procesoarele logice vor utiliza resursele fizice comune, (incluzând memoriile *cache*, unităţile de execuţie, predictoarele instrucţiunilor de control şi magistralele), înbunătăţindu-se astfel semnificativ nivelul de utilizare al acestor resurse, fapt ce permite obţinerea unei performanţe ridicate cu costuri minime. Starea arhitecturală a fiecărui procesor logic cuprinde regiştrii de uz general şi cei de control, controlerul de întreruperi şi o serie de alte resurse ce menţin starea procesorului.

Intel a implementat această tehnică pentru prima dată în procesoarele Xeon, destinate serverelor, în anii 2002 şi a numit-o *hyperthreading* [5.20]. Prin folosirea acestei metode Intel a obţinut o creştere în performanţă de 25%, crescând numărul de structuri *hardware* cu doar 5%.

În prezent o serie de procesoare moderne implementează această tehnică ca de exemplu Ultra SPARC T1 [5.1] si Power 5 [5.21].

➢ **Chip Multiprocessors (CMP).**

Sistemul presupune existenţa a mai multe unităţi de procesare integrate în acelasi chip. Acest sistem a fost introdus în [5.22] şi a devenit o realitate a zilelor noastre fiind produs de majoritatea fabricanţilor de microprocesoare.

În aceste arhitecturi se utilizează procesoare relativ simple *in-order* cum este cazul microprocesorului Niagara proiectat de firma Sun Microsystems, sau complexe *out-of-order* cum este cazul familiei de procesoare Power (începând cu Power4) produse de IBM [5.23].

Avantajul acestor arhitecturi este capacitatea de a executa mai multe *thread*-uri în paralel, putând astfel să efectueze un număr mult mai ridicat de instrucţiuni.

Dezavantajul acestor procesoare apare în condiţiile în care aplicaţia nu poate fi descompusă în *thread*-uri, situaţie în care resursele unui astfel de procesor nu ar fi utilizate într-un mod eficient. Acesta nu este însă cazul aplicaţiilor comerciale moderne, ca de exemplu procesarea *online* a tranzacţiilor, bazele de date şi serverele de web, ce sunt caracterizate printr-o abundenţă de *thread*-uri paralele. Pe lângă nivelul de performanţă ridicat în procesarea aplicaţiilor paralele un alt avantaj al arhitecturii CMP este simplitatea design-ului ce se bazează pe utilizarea unui set de procesoare identice, precum şi uşurinţa verificării produsului final.

➢ **Simultaneous Speculative Threading (SST).**

Unele dintre procesoarele moderne renunţă la metoda tradiţională de a obţine performanţă ridicată, doar prin mărirea vitezei de procesare a unui singur *thread* de execuţie şi se îndreptă spre metode alternative ca de exemplu creşterea volumului total de muncă rezultat în urma execuţiei în paralel a mai multor astfel de *thread*-uri. De cele mai multe ori procesoarele din această gamă sacrifică viteza de procesare a unui singur *thread* în schimbul implementării suportului *hardware* pentru rularea în paralel a unui număr ridicat de *thread*-uri (64 în cazul procesorului Niagara2 produs de Sun Microsystems).

Deoarece numărul de tranzistori ce pot fi integraţi într-un chip este limitat iar complexitatea unităţilor de execuţie şi control a *pipeline*-urilor *out-of-order* este extrem de ridicată, la care se adaugă consumul mare de energie al acestora

determină firmele producătoare de astfel de microprocesoare să recurgă la implementarea unor *pipeline*-uri simple *in-order*. În acest caz performanţa în execuţie are însă de suferit datorită modului de soluţionare în *pipeline*-urile *in-order* a unui eveniment de lungă durată, ca de exemplu un *miss* în memoria *cache,* care conduce la întreruperea execuţiei instrucţiunilor.

Pentru a contracara efectele dăunătoare asupra performanţei în scenariile mai sus menţionate şi pentru a accelera performanţa/*thread* proiectanţii de microprocesoare au inventat o serie de tehnici noi, printre care şi SST [5.24].

Procesoare cum este de exemplu procesorul ROCK produs de firma Sun Microsystems [5.25], oferă deja suport *hardware* pentru SST. Acest mod de operare este oarecum similar cu SMT, doar că în acest caz resursele *hardware* dedicate pentru a executa cele două *thread*-uri soft din cazul SMT sunt concentrate pentru a executa un singur *thread* soft, la un nivel de performanţă foarte ridicat. Unităţile *hardware* ale SST extrag două *thread*-uri de execuţie dintr-un singur program secvenţial. În cele mai multe cazuri unul dintre *thread*-uri, cel pe care îl numim „din urmă" conţine o instrucţiune de citire din memorie ce a cauzat un *miss* în memoria *cache* şi instrucţiunile ce depind de ea, iar celălalt *thread* pe care-l numim „dinainte" conţine instrucţiuni independente. Astfel, în timpul în care *thread*-ul din urmă şi-a oprit execuţia aşteptând rezolvarea instrucţiunii de citire ce a cauzat *miss*-ul în *cache*, *thread*-ul „dinainte" continuă să execute instrucţiuni independente. Odată ce datele instrucţiunii de citire sunt returnate de către ierarhia de memorie, *thread*-ul „din urmă" începe să execute instrucţiunile dependente. În cazul în care o instrucţiune ce urmează a fi executată de *thread*-ul „din urmă" depinde de rezultatul unei instrucţiuni ce urmează a fi executată de *thread*-ul „dinainte", execuţia instrucţiunilor *thread*-ului „din urmă" nu începe până când acea instrucţiune şi-a încheiat execuţia.

Toate cele trei sisteme prezentate au unele avantaje dar şi unele limitări. Fiecare sistem este analizat minuţios de către proiectanţii platformelor *hardware* noi şi din implementarea lui se încearcă obţinerea performanţei maxime. În prezent există o serie de microprocesoare ce îmbină două dintre soluţiile prezentate mai sus. Singurul procesorul ROCK le îmbină pe toate trei.

5.5 Concluzii.

Modul de procesare *out-of-order* a reprezentat una dintre cele mai importante achiziţii în domeniul arhitecturilor de calculatoare. Procesoarele superscalar ce îl implementează au reprezentat standardul de performanţă pe parcursul mai multor decenii şi reprezintă încă baza de la care pornesc noile concepte arhitecturale.

În capitolul de faţă se face o sinteză a bazelor modului de procesare *out-of-order*. Este prezentat modul în care procesoarele superscalar exploatează nivelul de paralelism între instrucţiuni (ILP-ul) şi procesarea lor în paralel, în scopul creşterii nivelului de performanţă. Sunt de asemenea prezentaţi o serie de factori limitatori ai ILP ca de exemplu interdependenţele de date şi control. Sunt prezentate pe scurt operaţiile ce sunt efectuate în fiecare etapă a unui pipeline *out-of-order* clasic scoţându-se în evidenţă o serie de „evenimente" cu conotaţie negativă asupra performanţei precum şi principalele mecanisme *hardware* utilizate în diversele faze ale procesării unei instrucţiuni, destinate rezolvării acestor evenimente.

Folosind un simulator de performanţă al unui procesor superscalar generic, s-a efectuat un studiu şi au fost prezentate date de evaluare referitoare la infuenţa benefică a unor mecanisme de creştere a vitezei de execuţie utilizate în mod uzual în procesoarele superscalar.

Studiul efectuat evidenţiază faptul că cel mai mare efect asupra îmunătăţirii performanţei îl are existenţa nivelului doi al memoriei cache, fiind urmat de predictorul hibrid al instrucţiunilor de control şi în cele din urmă de mecanismul de dezambiguare al accesărilor memoriei.

În final au fost prezentate o serie de mecanisme microarhitecturale de ultimă oră folosite în microprocesoarele actuale în vederea creşterii performanţei. Implementarea lor asigură creşterea volumului de muncă efectuat prin utilizarea a mai multor thread-uri de execuţie aşa cum este cazul CMP şi SMT cât şi a performanţei/*thread* aşa cum este cazul SST.

Sinteza prezentată asupra modului de execuţie *out-of-order,* a principalelor probleme de performanţă legate de acest mod de execuţie, precum şi a principalelor mecanisme microarhitecturale utilizate în soluţionarea lor, reprezintă principala contribuţie a acestui capitol.

În prezentul capitol s-a făcut o prezentare succintă a problemelor esenţiale ce apar în cazul procesării *out-of-order* precum şi a principalelor mecanisme *hardware* utilizate în diversele faze ale procesării unei instrucţiuni, destinate rezolvării acestor probleme, precum şi principalele soluţii utilizate la ora actuală în producţia de microprocesoare.

Din datele prezentate mai sus este evident faptul că o metodă rapidă de dezambiguare în cazul dependenţelor între instrucţiunile de citire/scriere, realizarea unui mecanism nespeculativ care ar corecta predicţiile greşite şi ar furniza direcţia corectă în cazul dependenţelor de control, cât şi detectarea rapidă a unui *hit* sau *miss* în memoria *cache*, ar aduce o contribuţie importantă la îmbunătăţirea performanţei.

În legătură cu cele afirmate mai sus, în continuare voi propune un studiu a cărui scop este identificarea şi evaluarea unor modalităţi de îmbunătăţire a performanţei unui microprocesor ce implementează modul de execuţie *out-of-order*.

6. „Implication Engine" un nou mecanism de îmbunătăţire a performanţei microprocesoarelor *out-of-order.*

Din cele prezentate în capitolul anterior rezultă că microprocesoarele actuale operează în mare parte cu operanzi întregi, iar pentru îmbunătăţirea performanţei se recurge şi la o serie de tehnici speculative, care fac posibil ca instrucţiunile a căror execuţie este dependentă de instrucţiunile precedente, să-şi poată totuşi începe execuţia înainte ca rezultatele acestora să devină disponibile, permiţând în acest fel utilizarea mai eficientă a unităţilor funcţionale. Deşi în acest mod se conferă execuţiei un nivel de acurateţe mai mult sau mai puţin ridicat. Nu în rare cazuri însă, speculaţiile se dovedesc a fi false, situaţie în care sunt necesare mecanisme de revenire şi structuri *hardware* noi, care toate cresc complexitatea procesorului şi au influenţă negativă asupra performanţei acestuia.

Din acest motiv este evident faptul că realizarea unui mecanism nespeculativ care, dacă nu total, cel puţin în parte, să elimine predicţiile greşite, ar aduce o contribuţie importantă la îmbunătăţirea performanţei.

In legătură cu cele afirmate mai sus, voi propune un concept nou, bazat pe o soluţie **nespeculativă**, a cărui scop este identificarea şi evaluarea unei modalităţi de îmbunătăţire a performanţei unui microprocesor ce implementează modul de execuţie *out-of-order*. Soluţia pentru îmbunătăţirea performanţei microprocesoarelor pe care o voi dezvolta în continuare şi pe care am denumit-o ***Implication Engine* (IE),** este o nouă structură *hardware,* care elimină neajunsurile create de caracterul speculativ al mecanismelor de predicţie actuale, în special cele legate de complexitatea structurilor *hardware,* ce execută revenirea în cazul în care speculaţia se dovedeşte a fi falsă.

- ***Principial soluţia propusă se bazează pe identificarea şi apoi exploatarea şi valorificarea potenţialului de utilizare doar a unui subset al setului de valori conţinute de regiştrii operanzi.***

Implication Engine oferă o modalitate rapidă pentru dezambiguarea instrucţiunilor de citire/scriere din memorie, determinarea direcţiei fals/adevărat a instrucţiunilor de control condiţionale, detectarea unui *hit* sau *miss* în memoria *cache*, detectarea operanzilor scurţi (în care doar un subset al biţilor sunt semnificativi), toate acestea beneficiind de atributul de a fi **un mecanism nespeculativ.**

6.1. Premisele edificării IE.

Edificarea noului concept a fost posibilă prin coroborarea unor tehnici prezentate în literatura de specialitate, cu unele observaţii ale autorului prezentei teze, asupra perdictoarelor actuale şi a unor similitudini între unele porţi logice şi unele instrucţiuni executate de microprocesoare şi a faptului că valorile parţiale a regiştrilor operanzi pot fi determinate în urma aplicării unui principiu similar cu algoritmul de testare PODEM.

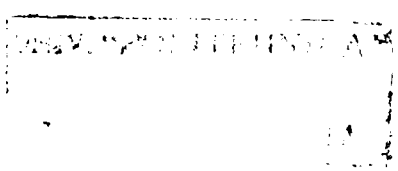
Cu alte cuvinte conceptul IE se bazează pe următoarele elemente:

a) Predicţia valorilor, un mecanism ce permite folosirea speculativă a datelor înainte ca acestea să devină disponibile în urma execuţiei regulate.

b) Exploatarea potenţialului de utilizare doar a unui subset al valorilor regiştrilor operanzi.

c) Similitudini între porţile logice şi instrucţiunile executate de microprocesoare.

d) Strategia de lucru asemănătoare cu cea adoptată în cazul algoritmului PODEM, cunoscut în domeniul circuitelor logice, utilizat pentru testarea calităţii acestora.



6.1.1.Predicţia valorilor.

În literatura de specialitate se propune o tehnică prin care se urmăreşte depăşirea evenimentelor nedorite, mai precis a dependenţelor de date, ce apar în cazul procesării *out-of-order,* denumită „predicţia valorilor" [6.1]. Asemănător mecanismelor implementate deja, şi acestă tehnică face parte din setul de tehnici speculative propuse pentru îmbunătăţirea performanţei procesoarelor actuale. Particularitatea acestui mecanism se bazează pe conceptul prin care o anumită valoare poate fi utilizată de mai multe ori în decursul execuţiei unui anumit program şi exploatează legătura dintre adresele instrucţiunilor de citire din memorie şi datele pe care acestea le returnează, utilizând întregul set de valori conţinute de operanzi.

Mecanismul în cauză permite predicţia valorilor din regiştrii pe 32 sau 64 de biţi, pe baza valorilor întâlnite în momente de timp anterioare.

Pentru implementarea acestei tehnici autorii utilizează un mecanism de predicţie pe două niveluri. Ambele niveluri sunt implementate în formă de tabel cu *map*-are directă, indexate cu PC-ul instrucţiunii ce urmează a fi prezisă.

- Primul nivel verifică dacă predicţia are şanse să fie corectă. Pentru acesta este folosit un mecanism tradiţional de predicţie bazat pe numărătoare pe 2 biţi, ce stochează date despre istoricul direcţiei unei anumite valori.
- Nivelul doi generează valoarea prezisă. Această valoare este iniţializată cu valoarea rezultatului unei instrucţiuni, când aceasta este executată pentru prima dată şi este reactualizată cu noua valoare, în cazul unei predicţii greşite.

Utilizând acestă tehnică s-au obţinut imbunătăţiri substanţiale ale performanţei [6.1], autorii raportând îmbunătăţiri între 22.7% şi 68,9% în funcţie de configuraţia unităţii de predicţie. Cu toate acestea, fiind un mecanism speculativ, implementarea acestei soliţii, în situaţia în care predicţia se dovedeşte a fi falsă, presupune existenţa unor mecanisme de revenire şi structuri *hardware* noi, care toate cresc complexitatea procesorului şi au influenţă negativă asupra performanţei acestuia.

În conexiunea „predicţia valorilor – Implication Engine", de reţinut este faptul că pentru a precalcula rezultatul unei instrucţiuni, ambele mecanisme au scopul de a determina valorile datelor conţinute în regiştrii

operanzi înainte ca acestea să devină disponibile în urma procesării regulate. Spre deosebire de predicţia valorilor IE este un mecanism nespeculativ.

6.1.2.Evaluarea potenţialului utilizării unui subset al valorilor regiştrilor operanzi.

- Pentru obţinerea unui nivel ridicat de performanţă în execuţie, instrucţiunile de citire trebuie să fie executate înaintea instrucţiunilor premergătoare de scriere, atâta timp cât nu încearcă să acceseze aceleaşi locaţii de memorie cu acestea. Pentru rezolvarea situaţiei în care o instrucţiune de citire încearcă să acceseze aceeaşi locaţie de memorie cu o instrucţiune de scriere, metoda tradiţională impune ca instrucţiunile de citire să aştepte, până când adresele datelor ce urmează a fi accesate de instrucţiunile de scriere au fost calculate.

Observaţie: *Faptul că adresa unei instrucţiuni de scriere este diferită de adresa unei instrucţiuni de citire, poate fi detectat însă, prin cunoaşterea doar a unui subset al biţiilor ce le alcătuiesc.*

- Deoarece *pipeline*-ul procesoarelor moderne cuprinde din ce în ce mai multe etape, timpii morţi asociaţi procesării unor predicţii greşite, ale instrucţiunilor de control, sunt un factor cu influentă majoră asupra încetinirii vitezei de execuţie. Calculareа direcţiei instrucţiunilor de control necesită câteva cicluri de *clock*, şi anularea rezultatelor, în cazul procesării unei predicţii greşite, înseamnă pierdere de timp.

Observaţie: *In multe cazuri direcţia unei astfel de instrucţiuni condiţionale poate fi determinată prin cunoaşterea doar a unui subset din biţii ce alcătuiesc regiştrii operanzilor ce o determină. De exemplu prin folosirea informaţiei parţiale putem concluziona nespeculativ că o instrucţiune* ***beq*** *(branch equal) se va îndrepta spre direcţia fals în cazul în care doar un bit din operanzii ce urmează a fi comparaţi pentru determinarea direcţiei este diferit.*

- Pentru a accesa memoria *cache* este necesară cunoaşterea întregii valori a adresei locaţiei de memorie ce urmează a fi accesată. Această adresă este determinată în urma unei operaţiuni de adunare ce are loc în etapa de execuţie a *pipeline*-ului. Posibilitatea de a determina rapid un *hit* sau un *miss* în memoria *cache* ar contribui semnificativ la creşterea performanţei.

Observaţie: *Un hit sau un miss în memoria cache se decide în urma comparării tag-urilor entităţilor stocate în aceasta. Egalitatea sau inegalitatea între tag-uri poate fi determinată cunoscând valorile parţiale a biţilor adresei de memorie. În cazul în care se pot determina suficienţi dintre cei mai puţin semnificativi biţi ai adresei, se poate lansa operaţiunea de indexare în cache, înainte ca această operaţiune să poată fi efectuată în mod normal. Numărul de biţi necesari pentru indexare se află într-o legătură directă cu arhitectura memorie cache. Pe baza informaţiei parţiale se pot detecta miss-uri în nivelurile inferioare ale memoriei cache şi se poate lansa accesul nivelurilor superioare, sau chiar a memoriei principale, înainte ca întreaga adresă accesată de instrucţiunea în cauză să fie efectiv calculată. De asemenea hit-urile în memoria cache pot fi detectate pe baza informaţiei parţiale, datele urmând a fi returnate instrucţiunilor dependente într-un timp mai scurt decât în mod normal.*

Din prezentarea acestor exemple se poate desprinde ideea pe care este fundamentat conceptual mecanismul IE, aceea că pentru dezambiguarea instrucțiunilor de citire/scriere din memorie, determinarea direcției fals/adevărat a instrucțiunilor de control condiționale, detectarea unui hit sau miss în memoria cache, detectarea operanzilor scurți, se poate utiliza doar informația parțială continută în regiştrii operanzi.

6.1.3 Algoritmul PODEM.

În fabricarea circuitelor logice, una din problemele importante ce trebuie avută în vedere este identificarea şi eliminarea produselor defecte rezultate în urma procesului de fabricaţie. Verificarea unui circuit logic este posibilă prin proiectarea unui mecanism care să aplice valorilor de intrare, toate combinaţiile posibile şi în

funcţie de acestèa să se urmărească valorile rezultante la ieşire. Lucrând în acest fel pentru testarea unui circuit cu „n" intrări, ar trebui generate „2^n" combinaţii. In cazul unui circuit cu valoarea „n" de ordinul sutelor, chiar dacă frecvenţa de generare a combinaţiilor ar fi de ordinul GHz-ilor, ar fi necesari ani de zile pentru verificarea unui singur circuit.

Rezolvarea problemei s-a făcut prin adoptarea unor ipoteze simplificatoare ce au condus la conceperea unor algoritmi de testare, a căror denumire generică este de *Automatic Test Pattern Generation* (ATPG) care a redus semnificativ numărul necesar de combinaţii al valorilor de intrare, reducând astfel timpul de testare.

Un algoritm eficient de testare, utilizat la ora actuală este *Path Oriented Decision Making* (PODEM) introdus de Goel [6.3] şi reluat în [6.2].

Esenţa algoritmului constă în generarea unor vectori de testare ce se aplică intrărilor primare (intrările circuitului combinaţional ce pot fi controlate din exterior), urmărind la ieşirile primare (ieşirile ce pot fi observate din exterior) efectul aplicării lor.

- **Vector de testare;** este un set de valori aplicat intrărilor primare ale unui circuit combinaţional, care prin felul în care au fost alese, va produce la ieşire o valoare logică univocă, în baza căreia se poate face separarea circuitele valide de cele defecte.

Utilizând algoritmul PODEM se reduce substanţial numărul de vectori de testare necesari pentru detectarea defectelor posibile ale unui circuit combinaţional. Nivelul de reuşită al algoritmului este apreciat prin factorul de eficienţă. Un factor de eficienţă de 100% înseamnă că pentru fiecare defect se găseşte un vector de testare care îl face să se manifeste şi să fie detectat, sau în caz contrar defectul este declarat nedetectabil.

În lucrarea [6.4] autorii prezintă o evaluare a performanţei aplicării algoritmului PODEM pentru o serie de circuite combinaţionale ce fac parte din programul de testare ISCAS-85 [6.5] Pentru cele zece circuite luate în considerare de ISCAS-85, care au numărul de intrări cuprins între 36 si 207, factorul de eficienţă al algoritmului PODEM variază între 88,25% si 100%.

Conform celor prezentate în lucrarea [6.4] pentru circuitul C880 care are 60 de intrări şi pentru care în mod obişnuit ar trebui generate 2^{60} combinaţii ale valorilor de intrare, utilizând algoritmul PODEM este necesară generarea doar a 63 vectori de testare, la un factorul de eficienţă de 100%, ceea ce reprezintă o reducere mai mult decât substanţială ($\sim 2^6$ vectori de testare, faţă de 2^{60} combinaţii posibile ale valorilor de intrare). Un alt circuit spre exemplu circuitul C7552 care are 207 intrări primare poate fi testat aplicând algoritmul PODEM cu doar 251 de vectori de testare, la un factor de eficienţă de 98.25%.

În generarea vectorilor de testare se porneşte de la următoarele supoziţii:

a) circuitul combinaţional generează valori incorecte doar datorită defectelor de tip *stuck-at-0* (în care valoarea liniei în cauză este întodeauna „0"), sau *stuck-at-1* (în care valoarea liniei în cauză este întodeauna „1").

b) singurele defecte ce apar în circuit sunt de tipul *stuck-at.*

Pentru generarea setului de valori valide ale unui vector de testare sunt parcurse trei etape, al căror scop este producerea următoarelor efecte:

- manifestarea defectului;
- propagarea defectului spre una din ieşirile primare;
- validarea efectului vectorului de testare.

Valorile aferente vectorului de testare se obţin efectuându-se recursiv două operaţii:

- **implicare înainte;** operaţia prin care sunt determinate valorile de ieşire ale unei porţi logice, atunci când se cunosc valorile de intrare.

- **implicare înapoi;** operaţia prin care cunoscând valoarea de ieşire, se determină valorile de intrare.

Modul în care este generat setul de valori aferente unui vector de testare sunt prezentate în exemplul ce urmează:

Pentru uşurarea demonstraţiei considerăm un circuit combinaţional simplu, ca cel prezentat în figura 6.1.

Presupunem ca pe linia de ieşire a porţii 2 ar exista un defect de tip *stuck-at-0*. Pentru generarea vectorului de testare conform celor trei etape prezentate anterior, se vor face următoarele raţionamente:

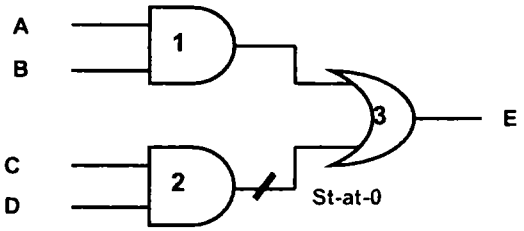


Figura 6.1. Exemplu de circuit combinaţional.

- **Manifestarea defectului.**

Pentru a face ca defectul să se manifeste, trebuie să aplicăm un vector de testare (adică acele valori de intrare în poarta „2" care vor produce la ieşire (pe linia 2-3) o valoare logică diferită decât cea defectă. În exemplul de faţă valoarea diferită de valoarea defectă este „1". Pentru a produce valoarea „1" pe această linie prin **implicare înapoi** se determină că ambele intrări în poarta „2" trebuie să aibă valoarea „1" (C=1 si D=1). Dacă această condiţie este îndeplinită putem spune că defectul a fost făcut să se manifeste deoarece valoarea la locul defectului în circuitul defect va fi diferită de cea a circuitul corect.

- **Propagarea defectului.**

În urma pasului anterior defectul a fost făcut să se manifeste. Pentru a-l face observabil trebuie să fie propagat spre una din ieşirile primare, unde poate fi cuantificat. În cazul prezentat în figură, ieşirea din poarta „3" este o ieşire primară. Pentru ca defectul să poată fi observat, valoarea logică de ieşire a acestei porţi în circuitul defect va fi „0" iar în circuitul corect va fi „1". Dacă valoarea logică de ieşire din poarta „2" este „0", conform presupunerii facută iniţial, pentru ca valoarea de ieşire din poarta „3" să fie „0", şi defectul să fie propagat, prin **implicarea înapoi** se stabileşte că linia „1-3" (de intrare în poarta „3") trebuie să fie setată la valoarea „0". Pentru ca acest lucru să fie posibil prin **implicare înapoi** se stabileşte că ambele intrări în poarta „1" sunt „0" (A=0 si B=0).

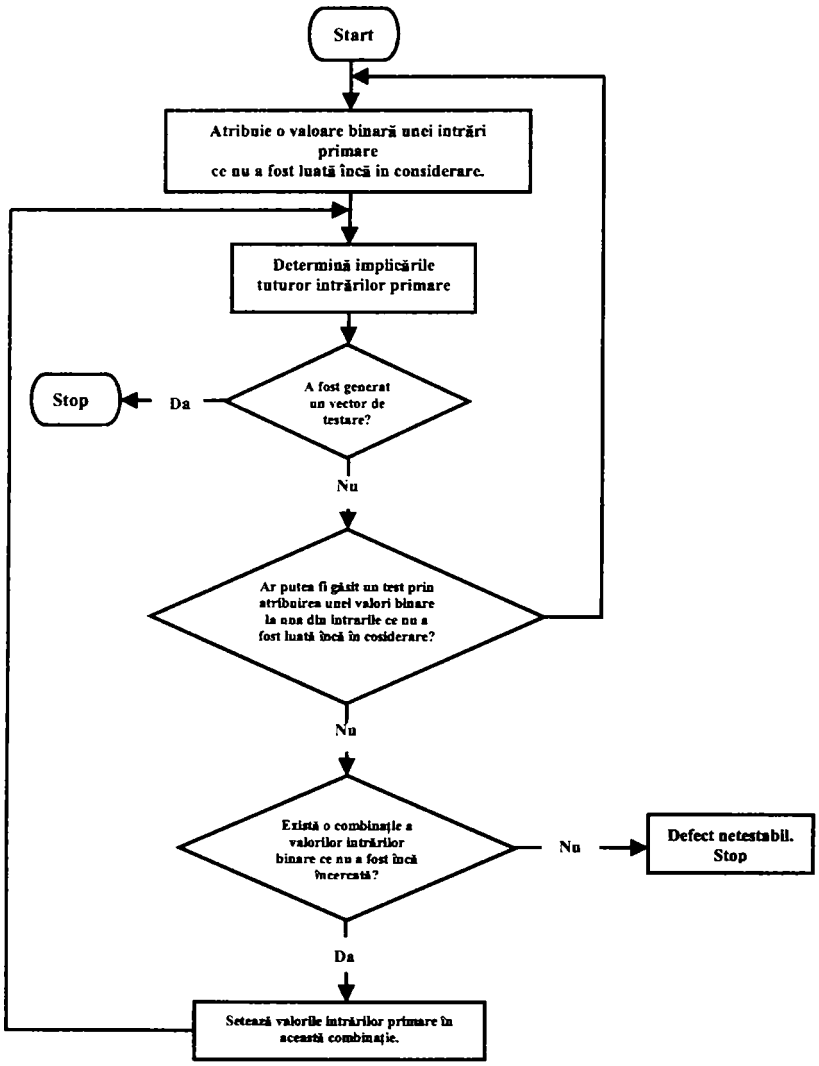


Figura 6.2. Algoritmul PODEM. Schema logică.

- **Validarea efectului vectorului de testare.**

Conform raţionamentului făcut în cei doi paşi anteriori, vectorul de testare pentru circuitul din figura 6.2 este {A=0, B=0, C=1, D=1}. Pentru ca acest set de valori să reprezinte un vector de testare valid, pentru exemplul dat, trebuie să facă ca defectul *stuck-at-0,* de pe linia 2-3, să se manifeste şi să fie propagat la ieşire. În urma aplicării vectorului de testare prin implicarea înainte se determină că în cazul

unui circuit corect, valoarea de ieşire a porţii 3 este E=1; iar în cazul unui circuit defect valoarea de ieşire a porţii 3 este E=0. Se poate deci concluziona că valorile menţionate reprezintă un vector de testare valid.

Modul general de funcţionare al algoritmului podem poate fi reprezentat prin schema logică din figura 6.2 [6.2]

Relaţia care se stabileşte între algoritmul PODEM şi Implication Engine este aceea că, IE pentru a stabili valorile parţiale din operanzi şi rezultatul unei instrucţiuni, bazat pe similitudinile ce există între circuitele cu porţi logice şi instrucţiunile executate de calculator, utilizează un principu asemănător cu PODEM.

6.1.4. Similitudini între porţile logice şi instrucţiunile executate de microprocesoare. Exemple.

a) Poarta logică *"ŞI".* (Tabelul 6.1.)

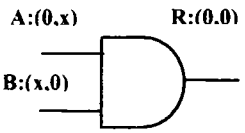
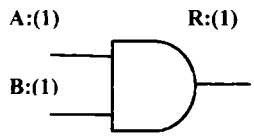
Dacă una din valorile intrărilor primare în poarta **ŞI** este **0**, prin aplicarea implicării înainte se poate determina că valoarea de ieşire va fi tot **0**.

Dacă valoarea ieşirii unei porţi logice **ŞI** este **1** se poate determina prin implicare înapoi că pentru ca acest lucru să fie posibil valorile intrărilor în această poartă trebuie să fie toate egale cu **1**.

- **Instrucţiune *"and "*.**

Dacă unul dintre operanzii unei instrucţiuni and are unul sau mai multi biţi cu valoarea **0**, prin aplicarea implicării înainte se poate determina că biţii corespunzători din rezultat vor fi **0**, indiferent de valorile biţilor corespunzători din cel de-al doilea operand.

De asemenea dacă rezultatul unei instrucţiuni *and* are unul sau mai mulţi biţi cu valoarea **1**, prin aplicarea implicării înapoi se poate determina în mod neechivoc că biţii corespunzători din operanzii de intrare vor avea de asemenea valoarea **1**.

Poarta logică "ŞI"		Instrucţiunea *"and"*	
		Val. Cunoscute	**Val. posibile**
Implicare înainte		**A: 00xx1xox** **B: xx10x0xx**	**R: 00x0x00x**
Implicare înapoi		**R: 10xx101xx**	**A: 1xxx1x1xx** **B: 1xxx1x1xx**

Tabelul 6.1. Similtudinea între poarta logică *"SI"* şi instrucţiunea *"and"*.

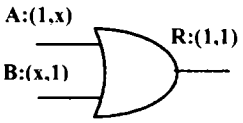
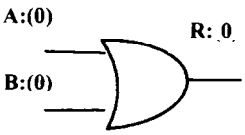
b) Poarta logică *„SAU"*. (Tabelul 6.2.)

Dacă una din valorile intrărilor primare în poarta **„SAU"** este **„1",** prin aplicarea implicării înainte, se poate determina că valoarea de ieşire va fi tot **„1".**

Dacă valoarea ieşirii unei porţi logice „SAU" este „1" se poate determina prin implicare înapoi că pentru ca acest lucru să fie posibil valorile intrărilor în această poartă trebuie să fie toate egale cu **„1".**

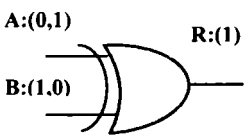
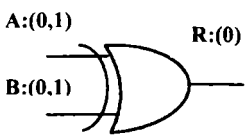
- **Instrucţiune *"or "*.**

Dacă unul dintre operanzii unei instrucţiuni ***or*** are unul sau mai multi biţi cu valoarea **„1",** prin aplicarea implicării înainte se poate determina că biţii corespunzători din rezultat vor fi **„1",** indiferent de valorile biţilor corespunzători din cel de-al doilea operand. De asemenea dacă rezultatul unei instrucţiuni **or** are unul sau mai mulţi biţi cu valoarea **„0"**, prin aplicarea implicării înapoi se poate determina în mod neechivoc că biţii corespunzători din operanzii de intrare vor avea de asemenea valoarea **„0".**

Poarta logică *"SAU"*		Instrucţiunea *"or"*	
		Val. cunoscute	Val. Posibile
Implicare înainte		**A: 00xx1x0x** **B: xx10x0xx**	**R: ----1--1------**
Implicare înapoi		**R: 10xx101xx**	**A: 1xxx1x1xx** **B: 1xxx1x1xx**

Tabelul 6.2. Similtudinea între poarta logică *"SAU"* şi instrucţiunea *"or"*.

c) Poarta logică *„XOR"*. (Tabelul 6.3.)

Poarta logică *"XOR"*		Instrucţiunea *"xor"*	
		Val. cunoscute	Val. posibile
Implicare înapoi	 	**R: 1xxx001xx**	**A: 1xxx100xx** **B: 0xxx101xx**

Tabelul 6.3. Similitudinea între poarta logică "XOR" şi instrucţiunea "xor"

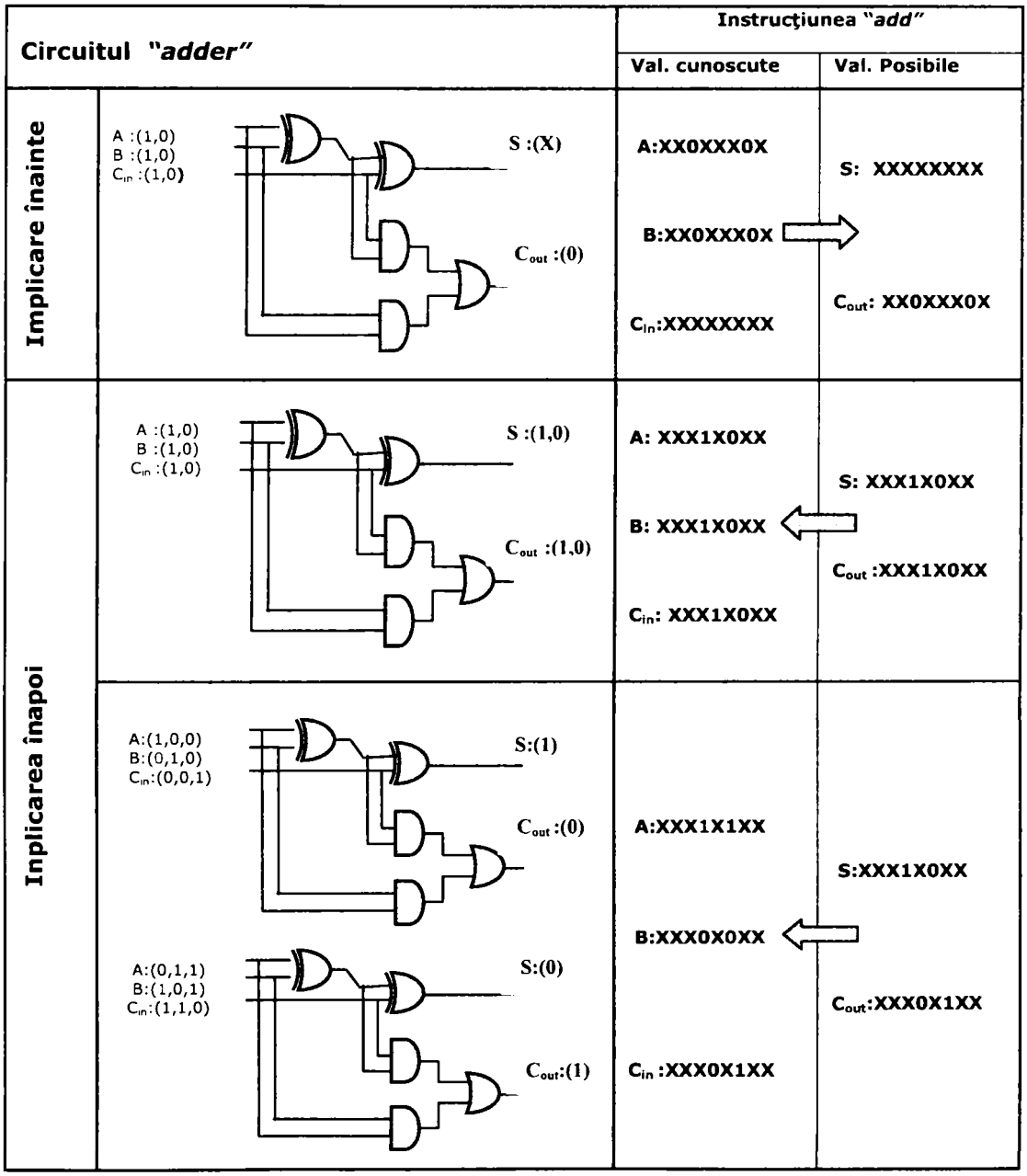
În cazul acestei porţi logice nu se poate determina în mod neechivoc prin implicare înainte valoarea ieşirii, bazat doar pe valoarea uneia dintre intrări. Pentru a determina valoarea de ieşire a porţii **„XOR"** este necesară cunoaşterea ambelor valori de intrare. În schimb cunoscând valoarea ieşirii unei porţi logice **„XOR"** şi valoarea uneia dintre intrări, prin aplicarea implicării înapoi se poate determina, în mod neechivoc, valoarea celeilalte intrări. Astfel dacă valoarea ieşirii unei porţi logice **„XOR"** este **„1"** iar valoarea uneia dintre intrări este **„0"** se poate determina prin implicare înapoi că valoarea celeilalte intrări este **„1"** şi reciproc. De asemenea dacă valoarea ieşirii unei porţi logice **„XOR"** este **„0"** iar valoarea uneia dintre intrări este **„0"** se poate determina prin implicare înapoi că valoarea celeilalte intrări este **„0".** Similar dacă valoarea ieşirii este **„0"** iar valoarea uneia dintre intrări este **„1"** se poate determina că valoarea celeilalte intrări este **„1".**

- **Instrucţiune *„xor "*.**

Dacă rezultatul unei instrucţiuni ***„xor"*** are unul sau mai mulţi biţi cu valoarea **„1"**, iar biţii corespunzători din unul din operanzii de intrare **„0"** prin aplicarea implicării înapoi se poate determina în mod neechivoc că biţii corespunzători din celalalt operand de intrare vor avea valoarea **„1"** şi reciproc. De asemenea dacă rezultatul unei instrucţiuni ***xor*** are unul sau mai multi biţi cu valoarea **„0"** iar biţii corespunzători din unul din operanzii de intrare **„0"** prin aplicarea propagării înapoi se poate determina în mod neechivoc că biţii corespunzători din celălalt operand de intrare vor avea valoarea **„0".** Similar dacă valoarea unor biţi din rezultat este **„0"** iar valoarea biţilor corespunzători din unul din operanzii de intrare **„1"** prin aplicarea propagării înapoi se poate determina că biţii corespunzători din celălalt operand de intrare vor avea valoarea **„1".**

d) Unitate *„adder"* pe un bit (Tabelul 6.4.)

Pentru o ilustrare mai completă a similitudinilor între porţile logice şi instrucţiunile executate de un microprocesor, precum şi pentru exemplificarea modului de generare a informaţiei parţiale pentru o instrucţiune complexă, în cele ce urmează voi prezenta corelaţia între un bloc logic ce efectuează adunarea pe un bit, şi o instrucţiune ***add***.



Tabelul 6.4 Similitudinea intre unitatea *„adder"* pe un bit şi instrucţiunea *„add"*.

Intrările sunt reprezentate prin **A**, **B**, iar *carry-in* dintr-un nivel mai puţin semnificativ este reprezentat prin $\mathbf{C_{in}}$. Ieşirile din blocul de adunare sunt rezultatul **„S"** şi *carry-out* **„$\mathbf{C_{out}}$"**. Pentru estimarea neechivocă a valorii unei ieşiri prin

aplicarea implicării înainte, este necesară cunoaşterea a cel puţin două dintre cele trei valori de intrare. Astfel dacă două dintre valorile de intrare sunt egale cu **„0"** putem determina prin implicare înainte că valoarea lui **„C_{out}"** din această poziţie este egală cu **„0".** În toate celelalte cazuri pentru determinarea valorilor **„S"** şi **„C_{out}"** este necesară cunoaşterea întregului set de valori de intrare **A, B** si **C_{In}.** Implicarea înapoi oferă de asemenea posibilităţi de determinare a unora dintre valorile de intrare, cunoscând valorile de ieşire. In cel mai favorabil caz pentru determinarea precisă a tuturor valorilor de intrare este necesară cunoaşterea valorilor doar a celor două ieşiri. Asfel dacă **„S"** şi **„C_{out}"** au ambele valoarea **„0"** valorile celor trei intrări sunt egale cu **„0".** Similar dacă **„S"** şi **„C_{out}"** au ambele valoarea **„1"** valorile celor trei intrări sunt egale cu **„1".** In cazuri mai nefavorabile este necesară cunoaşterea a celor două valori de ieşire, precum şi a uneia dintre valorile de intrare, pentru a determina prin implicare înapoi valoarea celorlalte două intrări. Astfel dacă valoare lui **„S"** este **„1"** şi valoarea lui **„C_{out}"** este **„0"** cunoscând că valoarea uneia dintre cele trei intrări este egală cu **„1"** putem determina prin implicare înapoi că valorile celorlalte două intrări sunt egale cu **„0".** Similar dacă valoarea lui **„S"** este **„0"** iar valoarea lui **„C_{out}"** este **„1"** cunscând că valoarea uneia dintre cele trei intrări este egală cu **„0"** putem determina prin implicare înapoi că valorile celorlalte două intrări sunt egale cu **„1".** În tabelul următor cu roşu sunt marcate valorile intrării ce trebuie cunoscută alături de valorile celor două ieşiri pentru a putea determina valorile celorlalte două intrări atunci când se aplică implicarea înapoi.

- **Instrucţiunea *„add".***

Acelaşi algoritm se poate aplica şi în cazul instrucţiunii ***„add"***. Dacă cunoaştem că valorile unor biţi a doi dintre cei trei operanzi de intrare sunt egali cu **„0"** aplicând implicarea înainte putem concluziona că valoarea lui **„C_{out}"** din acea poziţie este egală cu **„0".** În ceea ce priveşte implicarea înapoi dacă ştim că valorile unui bit din rezultatul **„S"** şi **„C_{out}"** din acea poziţie sunt **„0"** putem afirma cu certitudine aplicând implicarea înapoi că valorile corespunzătoare din operanzii de intrare şi **„C_{In}"** în acea poziţie sunt egali cu **„0".** Similar dacă ştim că valorile unui bit din **„S"** şi **„C_{out}"** din acea poziţie sunt **„1"** putem afirma aplicând implicarea înapoi ca valorile corespunzătoare ale operanzilor de intrare şi **„C_{In}"** în această poziţie sunt **„1".** Pentru cazurile în care se cunosc ambele valori de ieşire şi una

dintre valorile de intrare se poate aplica un raţionament similar cu cel prezentat în cazul blocului de adunare pe un bit.

De reţinut este faptul că exemplele prezentate relevă similitudinea care există între porţile logice şi instrucţiunile executate de calculator. Acestă similitudine stă la baza modului în care lucrează IE, certificând posibilitatea utilizării unui algoritm asemănător algoritmului PODEM, pentru determinarea valorilor parţiale ai regiştrilor operanzi şi ai rezultatului unei instrucţiuni înainte ca acesta să fie executată.

6.2. Implication Engine.

Pornind de la un concept de predicţie a valorilor, făcând observaţia că pentru a determina rezultatul produs de o anumită instrucţiune este suficientă cunoaşterea parţială a valorilor conţinute în regiştrii operanzi, luând în considerare unele similitudini între porţile logice şi instrucţiunile utilizate în limbajul de asamblare ce fac parte din ISA-ul (*Instruction Set Architecture*) microprocesoarelor şi utilizând ca strategie de lucru un principiu asemănător algoritmului de testare al circuitelor combinaţionale, a fost edificat noul concept *Implication Engine*, care are calitatea că este nespeculativ.

- ***Implication Engine este un mecanism care folosind informaţia parţială extrasă din regiştri operanzi, aplică implicarea înainte în graful de instrucţiuni dependente, pentru a pre-calcula valoarea rezultatului bazându-se pe valorile regiştrilor operanzi; sau implicarea înapoi pentu a precalcula valorilor operanzilor de intrare a instrucţiunilor aflate în acest graf, pornind de la rezultatul acestora. Aceeaşi tehnică poate fi aplicată şi pentru identificarea doar a unui subset al biţilor constituienţi ai operanzilor sau ai rezultatului.***

Funcţionând în acest fel informaţia parţială obţinută de IE poate contribui la dezambiguarea instrucţiunilor de citire/scriere din memorie, determinarea direcţiei fals/adevarat a instrucţiunilor de control condiţionale, detectarea rapidă a unui *hit* sau *miss* în memoria *cache*, detectarea nespeculativă a operanzilor scurţi (în care doar un subset al biţilor sunt semnificativi).

Modul de operare al IE este reprezentat prin schema logică din figura 6.3.

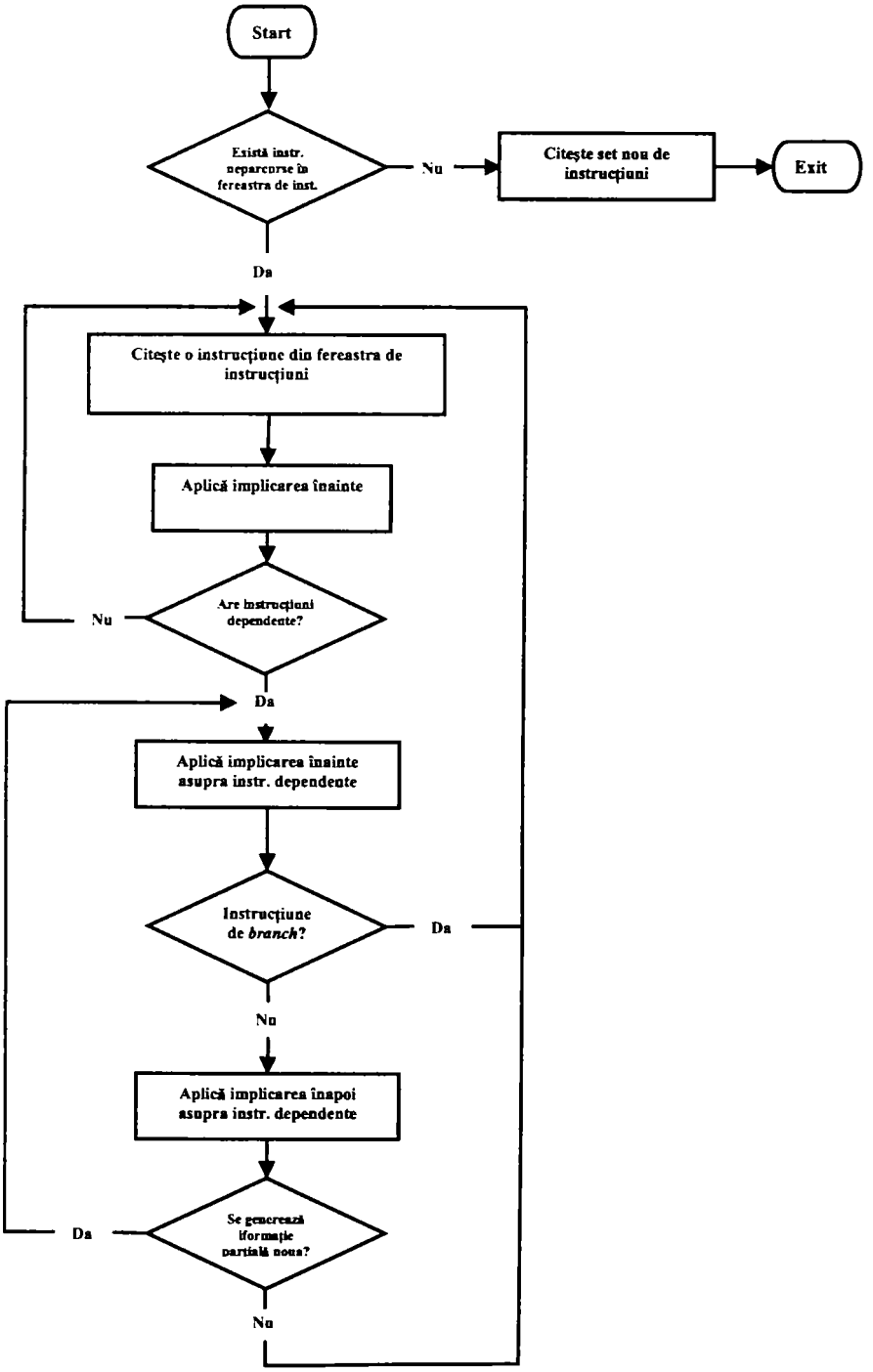


Fig. 6.3. Algoritmul IE. Schemă logică.

6.2.1. Surse de informaţie parţială.

➢ **Implicarea înainte.** Cele mai importante surse de valori parţiale pentru implicarea înainte sunt, operanzii *immediate* (operanzi ai căror valori sunt cunoscute în totalitate încă din etapa de decodificare) şi *shift* (deplasări la stânga sau la dreapta a biţilor unui operand).

Aşa cum s-a precizat în subcapitolul anterior dacă o instrucţiune execută o operaţie aritmetică, sau logică, având ca unul dintre operanzi un *immediate*, o mare parte din biţii rezultatului pot fi determinaţi aplicând implicarea înainte cunoscând doar valoarea operandului *immediate*. Doar într-un număr limitat de cazuri pentru a putea determina informaţie parţială despre valoarea rezultatului este necesară conoaşterea valorii parţiale a celuilalt operand. De multe ori această informaţie devine oricum disponibilă în urma modului iterativ în care algoritmul operează asupra lanţului de instrucţiuni dependente şi a modului în care informaţia este propagată de la o instrucţiune la alta.

Să luăm ca exemplu instrucţiunile ***andi*** (*and immediate*) şi ***sll*** (*shift left logic*) prezentate mai jos:

- ***andi R2, R1, 1 ;***

În cazul acestei instrucţiuni cunoscând că valoarea operandului *immediate* este 00000001 şi presupunând că valoarea din registrul R1=XXXXXXXX, IE determină că valoarea din registrul R2=0000000X adică, cu excepţia celui mai puţin semnificativ bit, toţi ceilalti biţii vor fi egali cu 0.

- ***sll R4, R3, 4 ;***

În cazul acestei instrucţiuni cunoscând faptul că valoarea de *shift* (numărul de biţi cu care urmează a fi deplasat spre stânga conţinutul registrului R3 este 4) şi presupunând că valoarea din registrul R3=xxxxxxxx prin aplicarea algoritmului IE de implicare înainte se poate determina că valoarea din registrul R4=xxxx0000 adică cei mai puţin semnificativi 4 biţi ai rezultatului stocat în registrul R4 sunt egali cu 0.

➢ **Implicarea înapoi.** Surse potenţiale pentru aplicarea algoritmului de implicare înapoi sunt predicţiile instrucţiunilor de control condiţionale şi predicţiile store set [6.6].

Să luăm în considerare instrucţiunea ***beq*** *(branch equal)*

- ***beq* R2, R1, offset;**

Dacă pentru o instrucţiune de control condiţională ***beq*** înainte ca direcţia acestei instrucţiuni să fie calculată nespeculativ, este prezisă direcţia adevărat, se poate concluziona că cei doi operanzi utilizaţi în calcularea direcţiei sunt egali, R1=R2. În consecinţă dacă sunt cunoscute valorile a doar unor biţi ce alcătuiesc unul dintre operanzi concluzia evidentă este că biţii poziţionaţi identic din celalalt operand au aceeaşi valoare.

Acelaşi raţionament poate fi aplicat asupra întregului spectru de instrucţiuni condiţionale şi ca urmare se pot determina informaţii valoroase despre operanzii de intrare utilizati în calculul direcţiei acestora cunoscând informaţii despre rezultat.

Raţionamentul poate fi aplicat şi în cazul predictorului ***"store set"*** , figura 6.4. Dacă un mecanism de predicţie *store set* prezice ca o instrucţiune de citire şi una de scriere, aflate în coada instrucţiunilor de citire-scriere din memorie accesează aceleaşi date, prin aplicarea mecanismului implicării înapoi se poate concluziona că operanzii utilizaţi la calculul adresei de memorie sunt egali.

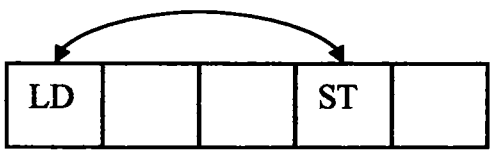


Figura 6.4. Coadă de instrucţiuni de accesare a memoriei. Exemplu de predicţie *store set*.

Să luăm ca exemplu cele două instrucţiuni de accesare a memoriei din figura 6.4.

- ***ld* R1, (R2)**
- ***st* (R3), R4**

Instrucţiunea de citire citeşte conţinutul locaţiei de memorie a cărei adresă este specificată de valoarea conţinută în registrul R2 şi o salvează în registrul R1. Instrucţiunea de scriere scrie locaţia de memorie a cărei adresă este specificată de valoarea conţinută în registrul R3 cu datele conţinute în registrul R4. Dacă predictorul *store set* prezice un *alias* între cele două instrucţiuni, pe baza acestei informaţii IE poate determina că valorile din R2 si R3 sunt egale.

6.2.2. Modalităţi de integrare IE într-un microprocesor.

Pentru implementarea mecanismului propus într-un microprocesor, am identificat două posibilităţi:

1. În prima variantă, aşa cum este ilustrat în figura 6.5, IE poate rula ca o etapă a pipeline-ului. Instrucţiunile sunt citite şi decodificate după care sunt transmise IE, acesta execută implicarea înainte şi înapoi asupra instrucţiunilor ce se află în coada de instrucţiuni decodificate atâta timp cât încă se determină informaţie parţială nouă.

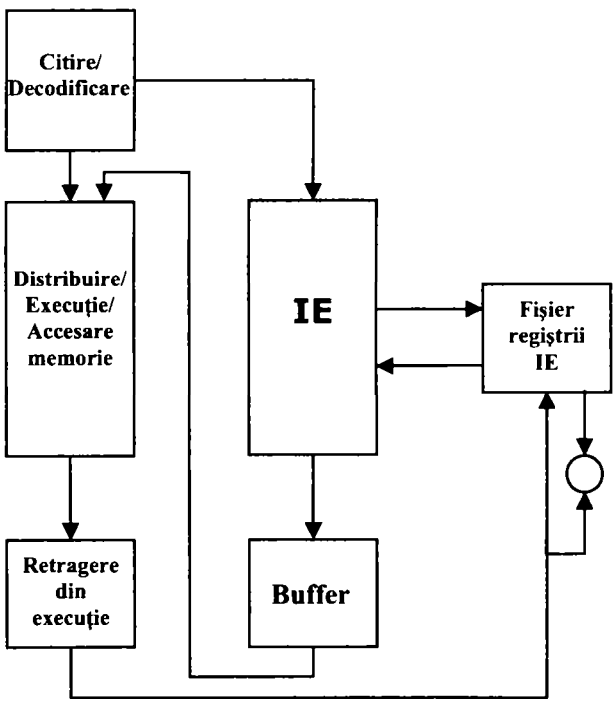


Figura 6.5. Implementare IE - schemă bloc.

IE determină această informaţie în funcţie de codul instrucţiunii în cauză şi de eventuala informaţie disponibilă despre datele conţinute în regiştrii operanzi. După ce IE şi-a încheiat execuţia, instrucţiunile împreună cu informaţia parţială sunt

distribuite spre unităţile de execuţie urmând ca aceasta să fie folosită în vederea îmbunătăţirii performanţei în execuţie prin dezambiguarea instrucţiunilor de citire/scriere din memorie, determinarea direcţiei instrucţiunilor de control etc. Odată ce valoarea unui operand devine disponibilă ca urmare a modului normal de execuţie, pentru validare ea este comparată cu valoarea aceluiaşi operand din fişierul de regiştrii al IE. În cazul în care cele două valori sunt diferite este necesară utilizarea unui mecanism de revenire.

2. A doua variantă de implementare, încearcă rezolvarea unor probleme de performanţă ce apar în arhitecturile CMT a procesoarelor de ultimă generaţie. Numărul de *core*-uri (unităţi de procesare) a acestora şi deci implicit volumul de muncă ce poate fi efectuat de microprocesor este constrâns de suprafaţa şi consumul de putere al fiecăruia dintre acestea. Din acest motiv *core*-urile implementărilor CMT trebuie să fie foarte eficiente în ceea ce priveşte suprafaţa ocupată şi consumul de putere consumată. Pentru a asigura aceste două deziderate proiectanţii procesoarelor de acest fel adoptă nişte implementări *in-order*. Aceste *core*-uri însă pentru a asigura un volum de muncă global, ridicat sacrifică performaţa unui singur *thread* de execuţie. Pentru a îmbunătăţi performanţa/*thread* se adoptă soluţii de genul SST.

Informaţia parţială obţinută de IE poate fi folosită pentru a reduce efectele negative datorate evenimentelor a căror rezolvare necesită un timp îndelungat şi pentru care este necesară întreruperea execuţiei în *core*-urile *in-order*. În felul acesta IE poate îmbunătăţi perfomanţa/*thread* fără a adăuga structurilor hardware complexitatea reclamată de SST.

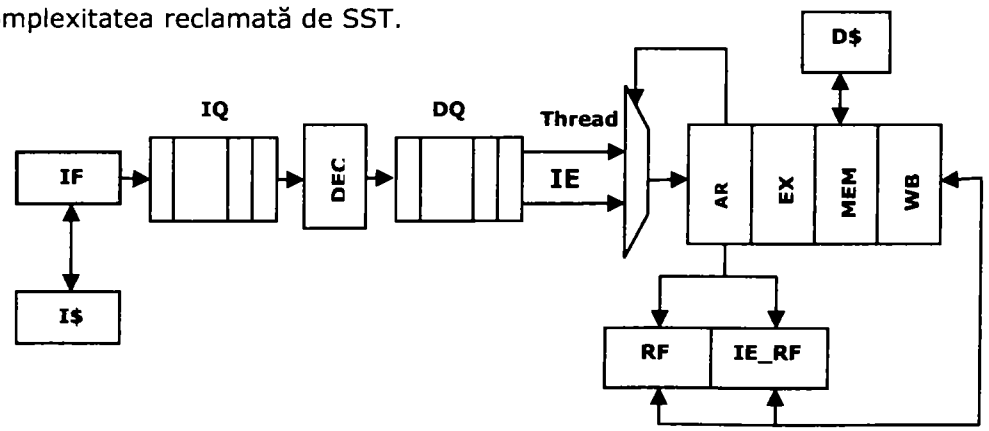


Figura 6.6 Schemă bloc a implementării IE într-un *core* al unui procesor CMT.

În figura 6.6 este reprezentat schematic modul în care IE ar urma să fie implementată în *hardware* într-un procesor CMT cu *core*-uri *in-order*. Pentru simplificarea schemei s-a ilustrat un *pipeline* clasic cu 6 etape după cum urmează:

- **IF** etapa de citire instrucţiuni. În acestă etapă instrucţiunile sunt citite din memoria *cache* de instrucţiuni şi sunt stocate în coada de instrucţiuni IQ. Tot aici sunt semnalate *miss*-urile în memoria *cache* de instrucţiuni.
- **DEC** etapa de decodificare. În această etapă instrucţiunile sunt decodificate şi sunt stocate în coada de instrucţiuni DQ de unde urmează a fi distribuite unităţilor de execuţie.
- **AR** etapa de accesare fişier de regiştri. În această fază în funcţie de modul de execuţie sunt accesaţi sau regiştri din fişierul principal de regiştri FR sau din fişierul de regiştri IE, FR_IE. Tot aici, în situaţia în care valoarea ce urmează a fi citită de o instrucţiune nu este disponibilă în fişierul principal de regiştri, este semnalată întreruperea execuţiei începând cu instrucţiunea în cauză, urmând ca aceasta să fie reluată odată ce datele devin disponibile. În momentul în care execuţia *thread*-ului principal este stopată *core*-ul comută în modul IE în care execută implicarea înainte şi înapoi asupra instrucţiunilor stocate în DQ.
- **MEM** etapa de accesare a memoriei *cache* de date. În această etapă instrucţiunile de citire/scriere accesează memoria cache. Tot aici sunt semnalate *miss*-urile în memoria *cache* de date.
- **WB** etapa de scriere a rezultatelor în fişierul de regiştri. În funcţie de modul de execuţie rezultatele instrucţiunilor sunt scrise fie în FR sau în IE_FR.

În această variantă de implementare IE permite atenuarea efectelor ce influenţează negativ performanţa, asociate unui eveniment a cărui soluţionare necesită o perioadă îndelungată, prin generarea de informaţii utile în timpul mort necesar rezolvării acestuia. Spre exemplu în momentul întâlnirii unui *miss* în memoria cache, *miss* în TLB, sau o instrucţiune de împărţire etc., procesorul în loc să-şi întrerupă execuţia, continuă citirea şi decodificarea instrucţiunilor şi comută în modul IE. Rezultatele produse de IE urmând a fi salvate în fişierul de regiştrii ai acestuia (FR_IE). Sunt generate astfel date parţiale despre operanzii şi rezultatul atât a instrucţiunilor dependente de instrucţiunea care a provocat întreruperea execuţiei, cât şi despre alte instrucţiuni ce urmează în ordine cronologică. După

epuizarea timpului necesar rezolvării evenimentului în cauză şi reluarea execuţiei, microprocesorul se foloseşte de datele parţiale generate în avans de către IE pentru a lua o serie de decizii legate de îmbunătăţirea perfomanţei (dezambiguarea instrucţiunilor de citire/scriere din memorie, determinarea direcţiei instrucţiunilor de control, etc.).

Pe lângă acestea există cazuri când IE determină întreg setul de biţi ai rezultatului unor intrucţiuni caz în care odată cu comutarea în modul normal de execuţie microprocesorul nu mai trebuie să execute instrucţiunile respective. Implementarea algoritmului IE în felul acesta necesită o creştere minimă a complexităţii structurilor *hardware* a microprocesorului. Singura structură adiţională necesară este fişierul de regiştrii ai IE. Multe dintre arhitecturile ce oferă suport pentru SMT au deja implementate fişiere de regiştrii pentru fiecare *thread* de execuţie, fişiere ce pot fi utilizate de IE fără modificări. În felul acesta beneficiile implementării IE vin cu un cost minim în ceea ce priveşte suportul *hardware* adiţional.

Acest din urmă mod de implementare al IE nu este obiectul prezentei teze, detaliile de implementare, caracterizarea precum şi evaluările de performanţă vor face subiectul unor studii viitoare.

6.3. Concluzii

În acest capitol a fost introdus un mecanism complet nou, nespeculativ, numit Implication Engine, care are capacitatea de generare a valorilor parţiale a regiştrilor operanzi şi ai rezultatului instrucţiunilor. IE poate servi cu succes îmbunătăţirii performanţei *thread*-urilor de execuţie atât în cazul microprocesoarelor *out-of-order* cât şi a microprocesoarelor CMT.

Au fost prezentate premizele care au stat la baza ideii creării unui astfel de mecanism şi anume predicţia valorilor, identificarea şi valorificarea parţială a informaţiei conţinută în operanzi, similitudinile de comportare între porţile logice şi instrucţiunile în limbaj de asamblare ce sunt executate de microprocesor şi algoritmul PODEM de testare a circuitelor combinaţionale. IE s-a născut din observarea modului de funcţionare a algoritmului PODEM şi identificarea

potenţialelor beneficii în creşterea performanţei ce ar rezulta din aplicarea unui concept similar în cazul unui microprocesor.

Au fost prezentate principial modul de operare al algoritmul PODEM şi imbunătăţirile semnificative ce le aduce în domeniul verificării circuitelor combinaţionale. Au fost de asemenea prezentate exemple de funcţionare şi au fost scoase în evidenţă similitudinile între PODEM şi IE.

Au fost identificate sursele de informaţie parţială utilizate de IE şi au fost prezentate exemple din care rezultă modul în care IE generează informaţie parţială prin executarea implicării înainte şi înapoi în graful de instrucţiuni dependente. Datele IE pot fi folosite cu succes în rezolvarea unor evenimente cu conotaţie negativă asupra performanţei cum ar fi dezambiguarea instrucţiunilor de accesare a memoriei, predicţia instrucţiunilor de control etc.

În final au fost descrise două posibilităţi în care IE poate fi implementat în *hardware.* Unul din modurile de implementare se referă la integrarea acestuia într-un microprocesor *out-of-order*, şi unul CMT care reprezintă de asemenea o altă contribuţie importantă a acestui capitol.

Conceptul IE, descrierea acestuia, modul de operare şi identificarea zonelor din execuţia instrucţiunilor, unde datele parţiale generate de IE pot fi folosite cu succes, în scopul creşterii performanţei, reprezintă contribuţia principală a acestui capitol.

7. Evaluarea potenţialului IE implementat într-un procesor superscalar.

Pentru evaluarea mecanismului IE s-a recurs la o metodă consacrată, utilizată de proiectanţii sistemelor cu microprocesor în etapa în care se iau deciziile microarhitecturale referitoare la componenţa şi organizarea structurilor *hardware,* pentru obţinerea unei arhitecturi optimizate din punctul de vedere al performanţei. Astfel mecanismul IE a fost implementat în *software* şi integrat într-un simulator al unui procesor superscalar. S-au ales o serie de aplicaţii de testare şi s-au conceput metode de evaluare a potenţialului mecanismului. Au fost generate date de caracterizare pe baza cărora să se poată studia, pe de-o parte capacitatea IE de a genera informaţie parţială despre regiştrii operanzi, iar pe de altă parte posibilitatea de a folosi informaţia parţială în scopul îmbunătăţirii performanţei.

7.1. Descrierea mediului de simulare.

Datele experimentale legate de evaluarea potenţialului de utilizare şi stabilirea limitelor IE, au fost obţinute folosind ca mijloace principale de lucru:

- programul de simulare a microprocesoarelor superscalar *sim-outorder* ce face parte din suita SimpleScalar [5.19];
- părţi din programul *sim-profile* destinat caracterizării aplicaţiilor de testare, conţinut de asemenea în suita SimpleScalar;
- o serie de executabile precompilate a unor diferite *benchmark*-uri.

Pentru implementarea mecanismul IE în simulatorul *sim-outorder*, simularea interacţiunii acestui mecanism cu celelalte blocuri componente ale simulatorului şi crearea unui mediu de simulare adecvat scopului propus, au fost necesare următoarele modificări şi adaptări:

a) Modificarea *sim-profile* într-o structură de analiză asemănătoare FAF (aşa cum este descrisă în subcap. 3.4.1.) şi integrarea acesteia în *sim-outorder* pentru a servi analizei IE.

b) Modificarea codului sursă al *sim-outorder* pentru integrarea IE în *pipeline*-ul simulat de acesta.

c) Conceperea şi integrarea în *sim-outorder* a codului sursă ce permite obţinerea datelor de caracterizare IE.

➢ **Caracteristicile simulatorului *sim-outorder*** [5.19]. Simulatorul *sim-outorder* este un model detaliat al unui procesor superscalar, fiind cel mai complex simulator, ce face parte componentă din setul de programe *SimpleScalar*. Schema implementată foloseşte un buffer de reordonare pentru redenumirea automată a regiştrilor logici şi pentru stocarea rezultatelor instrucţiunilor ce se află în execuţie. În fiecare ciclu de ceas bufferul de reordonare retrage din execuţie instrucţiunile ce au fost finalizate, în ordinea secvenţială în care acestea apar în program şi modifică starea arhitecturală a fişierului de regiştrii.

Sistemul de memorie al simulatorului implementează o coadă de instrucţiuni de citire/scriere. Valorile instrucţiunilor de scriere sunt stocate în această coadă, în cazul în care instrucţiunea de scriere este executată speculativ. Instrucţiunile de citire sunt transmise ierarhiei de memorie în momentul în care adresele tuturor instrucţiunilor de scriere precedente devin cunoscute. Instrucţiunile de citire pot fi satisfăcute de către ierarhia de memorie, sau de către instrucţiuni de scriere anterioare, în cazul în care cele două instrucţiuni, citire/scriere, accesează aceeaşi locaţie de memorie.

➢ **Înserarea mecanismului IE în *pipeline*-ul simulatorului *sim-outorder*.**

Pipeline-ul simulat de *sim-outorder* împreună cu modificările necesare pentru implementarea mecanismului IE şi pentru simularea interacţiunii acestuia cu celelalte blocuri componente ale simulatorului, este prezentat în figura 7.1.

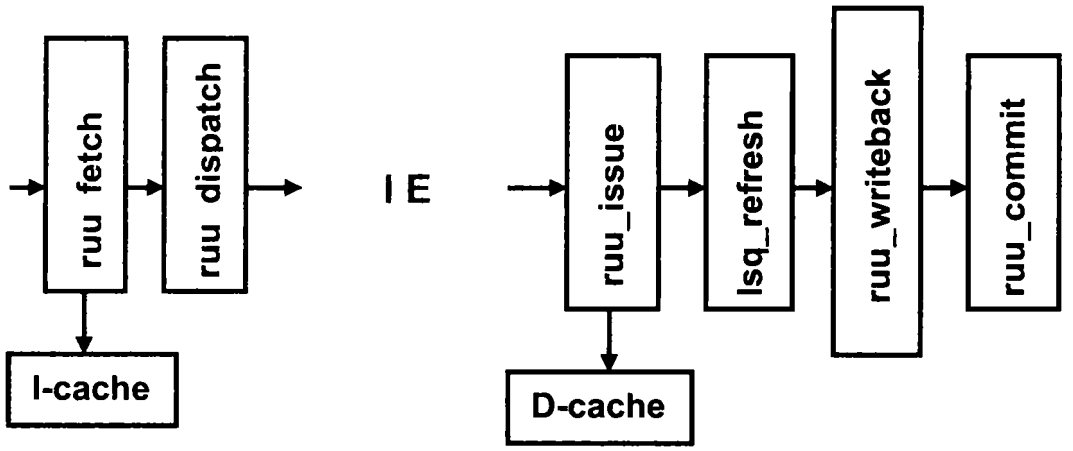


Figura 7.1 *Pipeline*-ul simulatorului sim-outorder in care este inserat mecanismul IE

Bucla principală de simulare reproduce un *pipeline* cu şapte etape, care este structurată după cum urmează:

- ***ruu_fetch();*** etapa de citire a instrucţiunilor. Unitatea de citire instrucţiuni utilizează PC-ul, starea predictorului instrucţiunilor de control şi informaţia asupra unei predicţii greşite, furnizată de unităţile de execuţie a instrucţiunilor de control, pentru a citi instrucţiunile ce urmează a fi executate.
- ***ruu_dispatch();*** decodificarea instrucţiunilor. Această funcţie operează asupra instrucţiunilor prezente în coadă de instrucţiuni, ce este alimentată în etapa de citire. În fiecare interval de ceas, distribuitorul citeşte instrucţiuni din coada de instrucţiuni şi le plasează în coada unităţii de lansare în execuţie. Totodată în această fază, sunt detectate predicţiile greşite ale instrucţiunilor de control. În cazul în care se detectează o predicţie greşită, simulatorul utilizează buffere speciale pentru menţinerea stării speculative, diferite de cele în care este menţinută starea arhitecturală corectă. În această etapă instrucţiunile sunt interconectate în funcţie de interdependenţele dintre ele.
- ***IE();*** etapa în care IE primeşte informaţii legate de codul instrucţiunii, valorile *immediate*, *shift* etc. de la etapa de decodificare, *ruu_dispatch*. În funcţie de modul în care este folosit IE, poate recepţiona rezultatele instrucţiunilor ce şi-au încheiat execuţia de la etapa *ruu_commit* şi execută propagarea înainte şi înapoi asupra instrucţiunilor citite, atâta timp cât încă se detectează informaţie parţială nouă. După ce IE şi-a încheiat execuţia, instrucţiunile

împreună cu informaţia parţială sunt transmise etapei *ruu_issue()* şi *lsq_refresh()* spre a fi distribuite şi lansate în execuţie, datele parţiale fiind folosite pentru dezambiguarea accesărilor memoriei, determinarea direcţiei fals/adevarat a instrucţiunilor de control condiţionale, detectarea rapidă a unui *hit* sau *miss* în memoria *cache*, detectarea nespeculativă a operanzilor scurţi. În cazul în care IE poate determina valorile tuturor biţilor rezultatului, instrucţiunea în cauză poate fi direct retrasă din execuţie.

- ***ruu_issue()*** şi ***lsq_refresh();*** etapa de distribuire şi lansare în execuţie. Aceste proceduri monitorizează interdependenţele între regiştrii şi între locaţiile de memorie şi modelează activarea instrucţiunilor şi lansarea lor în execuţie, către unităţile funcţionale, odată ce dependenţele de date le-au fost satisfăcute. În fiecare interval de *clock,* unitatea de lansare în execuţie identifică instrucţiunile a căror operanzi sunt disponibili. Lansarea în execuţie a instrucţiunilor de citire din memorie ce pot fi executate, este stopată dacă în coada instrucţiunilor de citire/scriere există o instrucţiune precedentă de scriere a cărei adresă nu este încă disponibilă. Dacă adresa unei instrucţiuni de scriere ce asteaptă să acceseze memoria, este identică cu cea a unei instrucţiuni de citire, valoarea acesteia este transmisă instrucţiunii de citire în cauză, evitându-se astfel un acces al memoriei ce nu este necesar. Cele două adrese pot fi declarate identice doar dacă sunt cunoscute în întregime în urma execuţiei regulate. Deoarece există cazuri în care IE poate determina întreg setul de biţi ai adreselor în cauză într-o etapă anterioară a pipeline-ului, decizia de egalitate poate fi luată pe baza informaţiei IE eliminându-se astfel timpul necesar obţinerii celor două adrese în execuţia regulată. În caz contrar instrucţiunea de citire este transmisă pentru a fi deservită de ierarhia de memorie.

Etapa de execuţie este implementată de asemenea în ***ruu_issue().*** În fiecare ciclu de ceas în această subrutină sunt procesate instrucţiunile care sunt furnizate de către etapa de lansare în execuţie a pipeline-ului. Tot în acestă etapă se verifică disponibilitatea unităţilor de execuţie, iar în cazul în care acestea au porturi libere, instrucţiunile sunt lansate în execuţie spre unităţile funcţionale respective. În final această procedură stabileşte momentul când instrucţiunile îşi vor scrie rezultatele în fişierul de regiştrii şi în memorie, ţinând cont de timpii de acces ai fiecărei unităţi în parte.

- ***ruu_writeback();*** etapa de returnare a rezultatelor. În fiecare ciclu de ceas în această procedură sunt scanate cozile de evenimente, pentru a se

identifica instrucţiunile ce au terminat execuţia. După ce o instrucţiune este identificată, subrutina parcurge şi marchează şirul de instrucţiuni ce depind de rezultatul instrucţiunii în cauză. În cazul în care una din instrucţiunile dependente aşteaptă doar rezultatul instrucţiunii ce şi-a încheiat execuţia, este marcată gata pentru execuţie.

În această etapă sunt detectate de asemenea predicţiile greşite ale instrucţiunilor de control. Când o astfel de predicţie greşită este detectată, rezultatele instrucţiunilor speculative ce au fost executate până în acel moment sunt anulate, *pipeline*-ul este curăţit de instrucţiunile speculative aflate în procesare, iar execuţia simulatorului este relansată dintr-o stare anterioară corectă cunoscută, salvată într-un *checkpoint*.

- ***ruu_commit();*** etapa în care sunt procesate instrucţiunile din etapa *ruu_writeback()* a pipeline-ului, ce sunt gata pentru a fi retrase din execuţie. Această procedură execută retragerea din execuţie a instrucţiunilor în ordinea secvenţială în care acestea apar în program şi modifică conţinutul memoriei *cache,* sau a memoriei principale, cu valorile instrucţiunilor de scriere în memorie. În urma etapei de retragere din execuţie rezultatele instrucţiunilor sunt plasate în fişierul de regiştrii.

Configurarea simulatorului de performanţă utilizat pentru generarea datelor, este prezentată în tabelul 7.1.

Caracteristici	Valoare
Lungime *pipeline*	7 etape; 16/16/16 fetch/issue/commit; 256 intrari in RUU; 128 intrari in LSQ
Predictor instructiuni conditionale	Combined (bimodal - 2K intrari; 2level -1K intrari); RAS 8 intrari; BTB 512 intrari, 4-cai
Sistemul de memorie	I-cache: 8KB (mapare directa, 32B dimensiunea liniei); durata acces 1 ciclu de clock D-cache: 16KB(4-cai asociativa, 64B dimensiunea liniei); durata de acces 1 ciclu de clock L2: 64KB(4-cai asociativa, 64B dimensiunea liniei); durata de acces 6 ciclii de clock Memoria principala: durata de acces 18 ciclii de clock
Unitati functionale	4 unitati procesare operanzi intregi; 1 unitate inmultiri/impartiri intregi 4 unitati procesare operanzi in virgula mobila; 1 unitate inmultiri/impartiri/radical operanzi in virgula mobila

Tabelul 7.1. Configurarea simulatorului de performanţă

7.2. Evaluarea potenţialului IE de a extrage valori parţiale din regiştrii operanzi şi de a le propaga în graful instrucţiunilor dependente.

Pentru o înţelegere mai bună a datelor prezentate în cele ce urmează voi face următoarele precizări:

1). Datele prezentate au fost obţinute în următorele condiţii:

- Sunt luate în considerare instrucţiuni care pot influenţa negativ performanţa în rulare, aparţinând următoarelor cinci categori:
  a. instrucţiuni aritmetice/logice,
  b. instrucţiuni acces memorie,
  c. instrucţiuni de control,
  d. instrucţiuni în virgulă mobilă,
  e. instrucţiuni eterogene cu destinaţie specială (*syscall, break*, etc).

Codurile instrucţiunilor ce fac parte din categoriile de mai sus sunt prezentate în Anexa 4 [5.19].

- O instrucţiune este considerată a avea informaţie IE dacă cel puţin una dintre valorile biţilor din cei doi regiştrii operanzi, sau din rezultat, a fost determinată în urma utilizării mecanismului IE.
- Numărul total al instrucţiunilor ce conţin date obţinute cu ajutorul IE, este raportat la numărul total de intrucţiuni executate.
- Rezultatele obţinute prezintă cazul când mecanismul IE este activat în comparaţie cu cazul de referinţă, caz în care simulatorul execută programul de testare fară interferenţa IE.

2). Pentru o estimare cât mai completă a potenţialului IE de a genera date utile s-a considerat util studiul comportamentului acestuia în două cazuri diferite şi anume:

- Mecanismul IE funcţionează de sine stătător.
- Mecanismului IE îi sunt returnate rezultatele tuturor instrucţiunilor care şi-au terminat execuţia.

3). S-a considerat semnificativ pentru aprecierea capacităţii mecanismului IE de a genera date utile pentru îmbunătăţirea performanţelor procesorului, studiul următorilor indicatori:

a. Distribuţia instrucţiunilor ce conţin informaţie parţială generată de IE.

Datele obţinute în urma acestei analize permit estimarea numărului total de instrucţiuni ce conţin informaţie parţială, raportat la numărul total de instrucţiuni executate într-o anumită aplicaţie.

b. Procentajul de instrucţiuni din fiecare categorie ce conţin date obţinute cu ajutorul IE, raportat la numărul de instrucţiuni executate, din categoria respectivă.

În acest fel se face o evaluarea a numărului total de instrucţiuni ce conţin date IE, din fiecare categorie de instrucţiuni luată în considerare.

c. Numărul total de biţi depistaţi cu ajutorul IE, conţinuţi în cei doi regiştrii operanzi citiţi de o instrucţiune, precum şi în rezultatul produs de aceasta.

Datele de acest fel sunt importante pentru evaluarea cantităţii de informaţie parţială ce poate fi generată pentru fiecare instrucţiune. Pentru cele mai multe utilizări ale IE este suficientă cunoaşterea unui singur bit din regiştrii operanzi. Cu toate acestea probabilitatea de rezolvare a conflictelor de date şi control creşte odată cu creşterea numărului de biţi ai căror valoare este cunoscută.

d. Numărul total de instrucţiuni şi numărul total de biţi IE, aflaţi la fiecare "nivel de dependenţă", pentru setul de aplicaţii studiate. (Acest indicator va fi cuantificat numai in situaţia in care mecanismului IE i-au fost retransmise rezultatele tuturor instrucţiunilor care şi-au terminat execuţia).

"Nivelul de dependenţă" este o noţiune, pe care am introdus-o pentru a putea cuantifica nivelul (distanţa) până la care IE poate transmite informaţie parţială, în graful de instrucţiuni dependente.

4). Datele experimentale au fost obţinute în urma studiului efectuat asupra unui subset al programelor SPEC CPU2000 şi anume a *164.gzip, 175.vpr, 176.gcc, 181.mcf, 197.parser, 255.vortex, 256.bzip2.*

5). Evaluarea potenţialului IE s-a efectuat prin simularea a $5*10^8$ instrucţiuni, pentru fiecare aplicaţie luată în considerare.

7.2.1. Date de caracterizare pentru cazul în care mecanismul IE funcţionează de sine stătător.

În cazul în care IE funcţionează de sine stătător, acesta extrage informaţie doar din sursele de informaţie parţială, prezentate în subcapitolul **6.2.4** şi efectuează implicarea înainte şi înapoi în graful instrucţiunilor dependente, obţinând în acest fel date despre valorile operanzilor acestora. Acesta ar constitui cazul cel mai simplu în care IE ar putea fi implementat în *hardware,* el putând fi adăugat ca o etapă componentă a *pipeline*-ului, aşa cum este prezentat în figura 7.1.

a. Distribuţia instrucţiunilor ce conţin informaţie IE, raportate la numărul total de instrucţiuni. (Figura 7.2.)

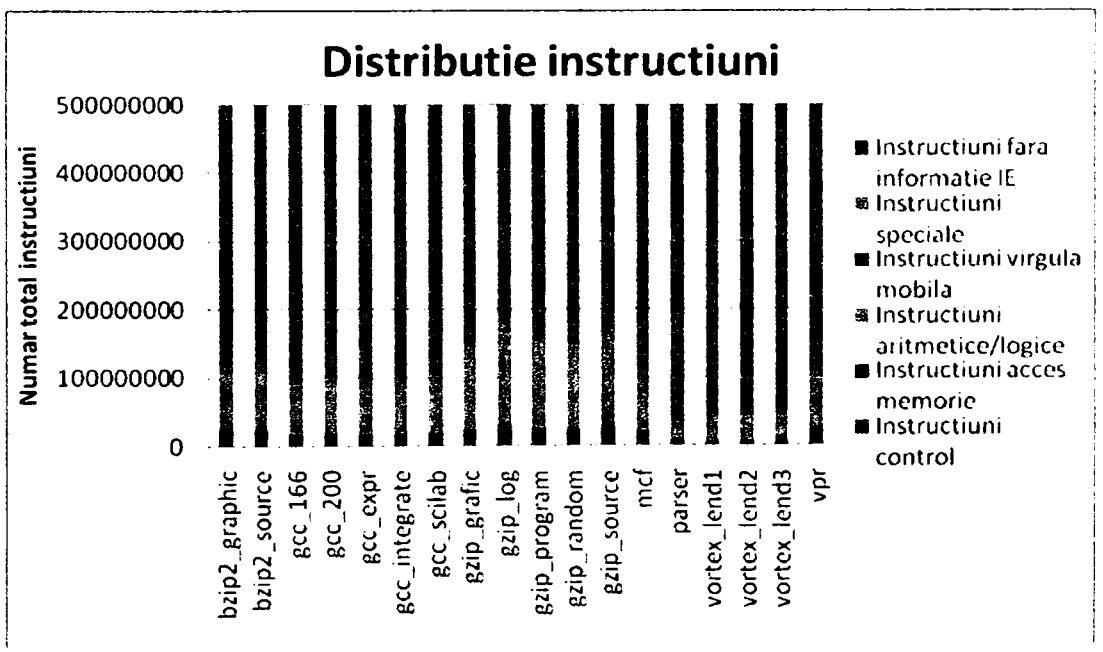


Figura 7.2. Distribuţia instrucţiunilor. Această figură prezintă instrucţiunile ce conţin informaţie parţială obţinută în urma aplicării IE, în fiecare dintre cele cinci categorii de instrucţiuni.

Datele prezentate în figura 7.2 relevă faptul că atunci când IE funcţionează de sine stătător, în medie, numărul total de instrucţiuni ce conţin informaţie parţială reprezintă 21% din totalul de instrucţiuni executate. Valoarea minimă fiind deţinută de aplicaţia *parser* cu 7%; iar valoarea maximă de aplicaţia *gzip* (datele de intrare log) pentru care numărul de instrucţiuni cu date parţiale este de 36% din numărul total de instrucţiuni. Pentru alte aplicaţii cum ar fi *bzip2, mcf* sau *gcc* numărul de instrucţiuni cu date IE este aproximativ egal cu 20%.

Procentajele obţinute relevă capacitatea IE de a genera şi propaga informaţie parţială.

b. Distribuţia procentuală a instrucţiunilor ce conţin informaţie IE, raportată la numărul total de instrucţiuni din categoria respectivă. (Figura 7.3.)

În figură sunt prezentate procentajele instrucţiunilor ce conţin date obţinute cu ajutorul IE pentru fiecare din cele cinci categorii de instrucţiuni, considerate.

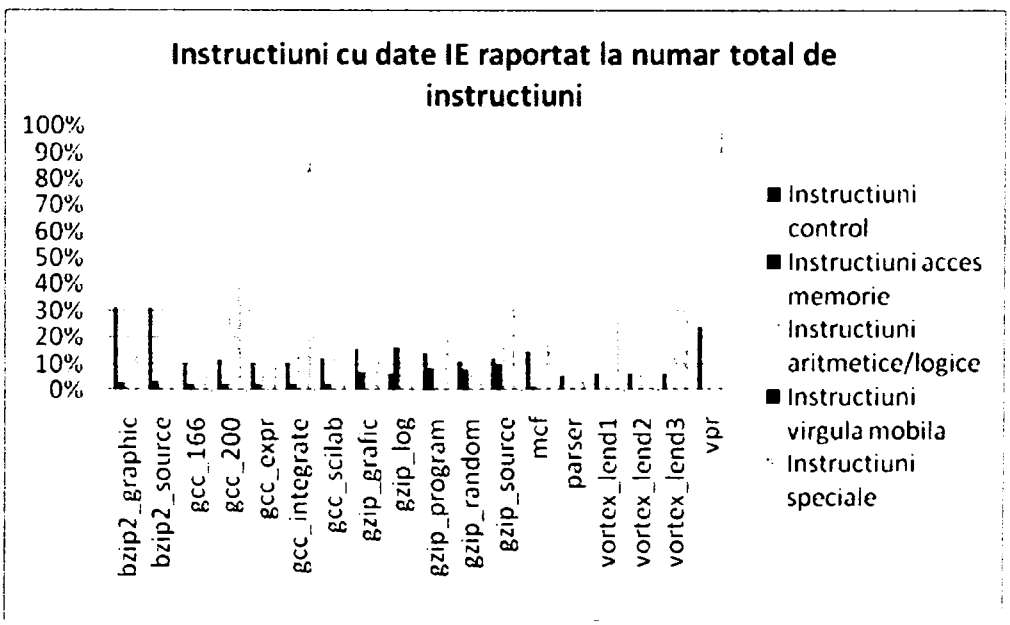


Figura 7.3. Instrucţiuni cu date IE: În această figură sunt reprezentate numărul de instrucţiuni cu date IE, raportate la numărul total de instrucţiuni din categoria respectivă.

Trebuie menţionat faptul că atât în cazul de referinţă în care simulatorul execută programul fără mecanismul IE, cât şi în cazul în care IE este operaţional, instrucţiuni în virgulă mobilă conţin doar aplicaţiile *gcc* şi *vpr.*

Datele prezentate în grafic permit următoarele concluzii:

- Mecanismul IE este capabil să genereze informaţie parţială pentru 100% din setul de instrucţiuni din categoria instrucţiunilor speciale, cu excepţia aplicaţiei *vpr* unde rata de succes este de 98%.
- Procentajul de instrucţiuni de accesare a memoriei ce conţin date IE este relativ scăzut, pentru majoritatea *benchmark*-urile studiate, singura aplicaţie *gzip (log)*, atingând valoarea de 16%. Pentru aplicaţii de acest gen IE reprezintă o soluţie viabilă pentru rezolvarea problemelor de performanţă create de dependenţele de date.
- Spre deosebire de dependenţele de date capacitatea IE de a soluţiona dependenţele de control este mai evidentă. Făcând media tuturor instrucţiunilor din această categorie din toate aplicaţiile studiate a rezultat că pentru 14% dintre instrucţiunile de control a fost posibilă determinarea datelor parţiale. Procentajul de instrucţiuni de control cu date IE cel mai scăzut îl deţine aplicaţia *parser* cu 6% iar cel mai ridicat aplicatia *bzip2* (ambele seturi de date de intrare) cu 31%.
- În medie rata de succes a IE de a genera informaţie parţială pentru instrucţiunile din categoria aritmetică/logică este relativ ridicată, 39%. Procentajul minim de instrucţiuni cu date obţinute în urma aplicării IE fiind deţinut de aplicaţia *parser*, 21%, iar procentajul cel mai ridicat de aplicaţia *gzip*, 48%.

c. Numărul total de biţi depistaţi de IE conţinuţi în cei doi regiştri operanzi şi în rezultatul fiecărei instrucţiuni. (Figura 7.4.)

Numărul de biţi reprezentaţi în graficul din figura 7.4 reprezintă media obţinută prin divizarea numărului total de biţi extraşi din cei doi regiştrii operanzi, precum şi din rezultat, determinaţi de IE, la numărul total de instrucţiuni ce conţin date IE dintr-o anumită categorie.

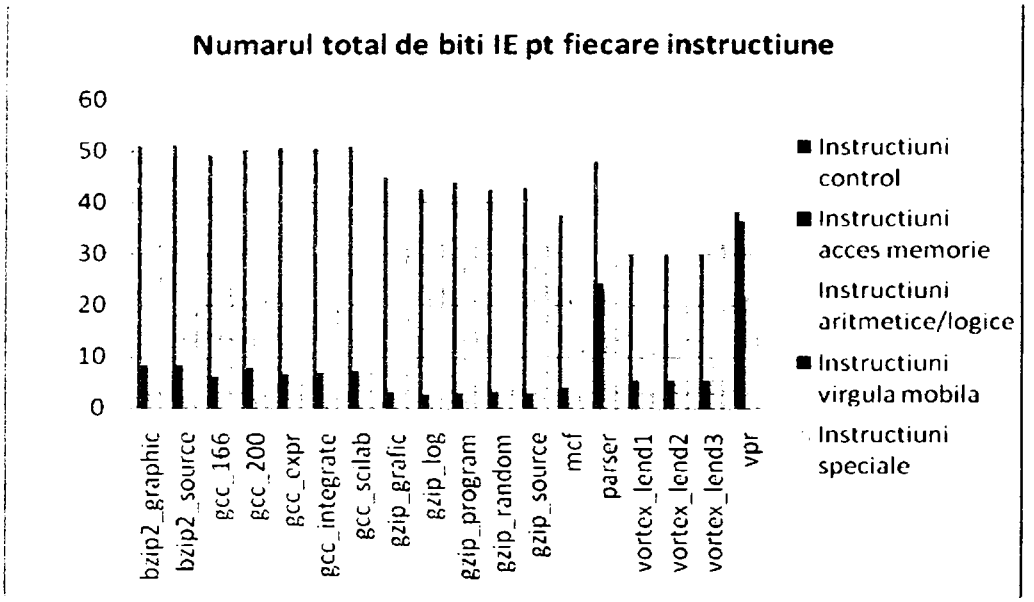


Figura 7.4. Număr de biţi IE. În această figură este reprezentată pentru fiecare instrucţiune media biţilor din regiştrii operanzi, ai căror valoare a fost cunoscută în urma aplicării IE.

Numărul maxim posibil de biţi ce pot fi determinaţi ca urmare a aplicării IE este 96 deoarece instrucţiunile simulate operează cu operanzi pe 32 de biţi.

Din datele prezentate în grafic se observă că:

- Numărul de biţi ai căror valori sunt cunoscute în urma IE este semnificativ în special pentru instrucţiunile de control, cele aritmetice logice şi cele speciale.

În ceea ce priveşte instrucţiunile de control media biţilor IE pentru toate aplicaţiile de testare este de aproximativ 43. Aceasta reflectă în mod clar probabilitatea ridicată de rezolvare a acestui tip de conflicte pe baza informaţiei IE. Numărul maxim de biţi ce conţin informaţie IE este deţinut de aplicaţia *bzip2* cu media de aproximativ 51 de biţi IE pentru fiecare instrucţiune, iar numărul minim îl deţine aplicaţia *vortex* cu media de aproximativ 30 de biţi IE.

- Pentru aplicaţii ca de exemplu *vpr* există o probabilitate mare de rezolvare a conflictelor de date. În medie numărul de biţi IE pentru fiecare din instrucţiunile de accesare a memoriei din această aplicaţie este de aproximativ 36.

- În plus pentru instrucţiunile aritmetice/logice aplicând procedeul IE se pot determina în medie valorile a 20 de biţi.

7.2.2. Date de caracterizare pentru cazul în care rezultatele tuturor instrucţiunilor ce şi-au încheiat execuţia sunt retransmise mecanismului IE.

În acest caz, pe lângă datele obţinute din sursele de informaţie parţială, rezultatele instrucţiunilor ce au fost retrase din execuţie sunt retransmise etapei *pipeline*-ului ce execută IE în urma unui *bypass* din ultima etapă *ruu_comit*. Utilizând datele obţinute această informaţie este apoi propagată de IE înainte şi înapoi în graful instrucţiunilor dependente obţinând în acest fel date despre valorile operanzilor acestora.

Spre deosebire de cazul în care IE funcţionează de sine stătător, acest caz reflectă limita maximă a cantităţii de informaţie parţială ce poate fi generată în urma aplicării IE, pentru situaţia în care IE funcţionează ca o etapă a pipeline relevând totodată şi potenţialul maxim al IE de rezolvare a diferitelor probleme de performanţă. Punerea în practică a mecanismului IE în felul acesta ar implica însă o creştere a nivelul de complexitate a structurilor hardware, fiind necesare pe lângă adăugarea unor etape în *pipeline* aferente IE şi adăugarea structurilor ce execută şi controlează bypass-ul rezultatelor.

a. Distribuţia instrucţiunilor ce conţin informaţie IE raportate la numărul total de instrucţiuni. (Figura 7.5.)

Din datele conţinute în acest grafic se observă că numărul total de instrucţiuni ce conţin date IE, pentru majoritatea aplicaţiilor studiate depăşeşte $3*10^8$ ceea ce reprezintă peste 60% din totalul de instrucţiuni executate.

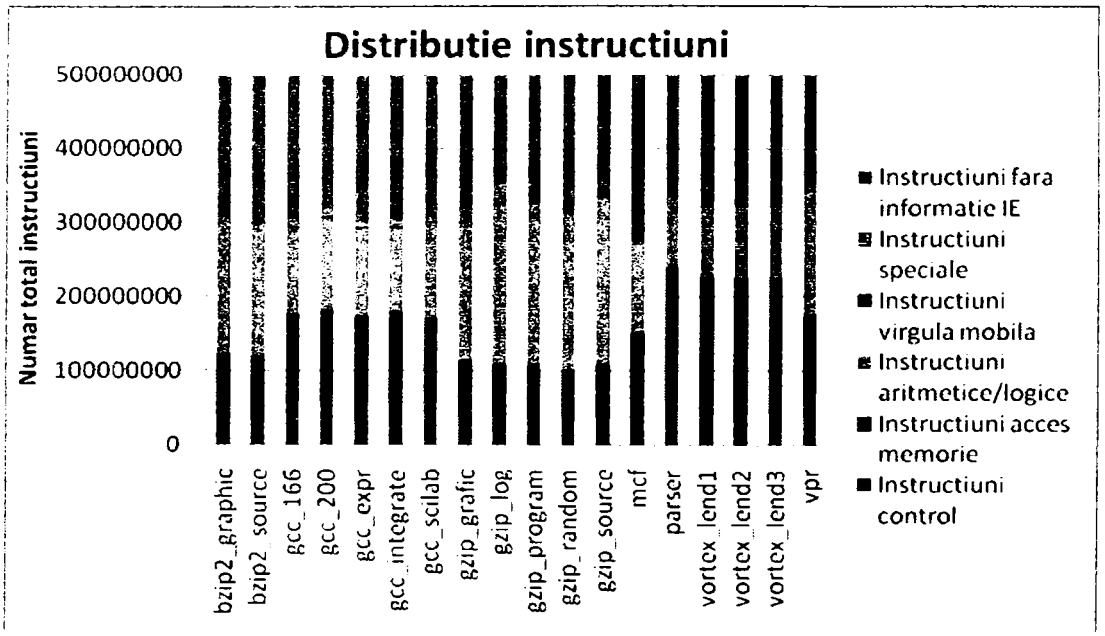


Figura 7.5. Distribuţia instrucţiunilor: Această figură prezintă instrucţiunile ce conţin informaţie parţială obţinută în urma aplicării IE, în fiecare dintre cele cinci categorii de instrucţiuni.

La un capăt al spectrului se află aplicaţia *mcf* cu 55% iar la celălalt capăt se află aplicaţia *gzip* (setul de date de intrare log) pentru care numărul de instrucţiuni ce conţin informaţie IE este de 77% din numărul total de instrucţiuni. Aceste rezultate relevă potenţialul foarte ridicat al IE de a genera informaţie parţială despre regiştri operanzi şi despre rezultatul instrucţiunilor, majoritatea instrucţiunilor conţinând date IE.

b. Distribuţia procentuală a instrucţiunilor ce conţin informaţie IE raportat la numărul total de instrucţiuni din categoria respectivă. (Figura 6.6.)

În figura următoare sunt prezentate procentele instrucţiunilor ce conţin date IE din fiecare dintre cele cinci categorii, instrucţiuni de control, instrucţiuni de accesare a memoriei, instrucţiuni aritmetice/logice, instrucţiuni în virgulă mobilă şi instrucţiuni speciale.

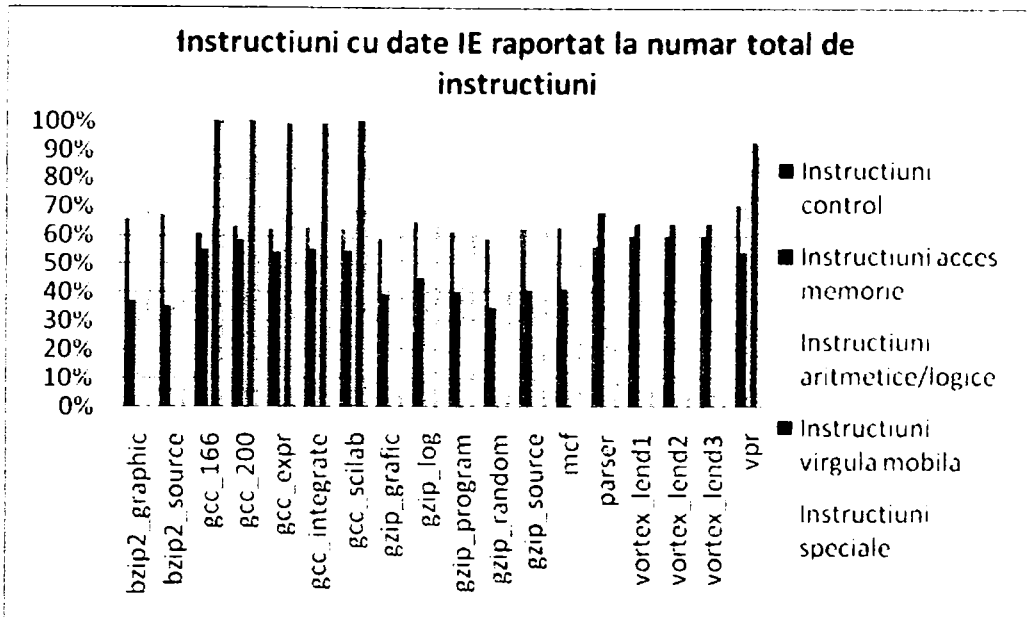


Figura 7.6. Instrucţiuni cu date IE: În această figură sunt reprezentate numărul de instrucţiuni cu date IE, raportate la numărul total de instrucţiuni din categoria respectivă.

Referitor la datele din figură trebuie menţionat faptul că în afară de aplicaţia *gcc* şi *vpr*, pentru celelalte aplicaţii componenta în virgulă mobilă în cele $5*10^8$ de instrucţiuni simulate este nulă chiar şi în cazul de referinţă în care simulatorul execută programul fără interferenţă IE.

Din datele prezentate în grafice se pot discerne următoarele concluzii:

- În toate aplicaţiile simulate pentru categoria instrucţiunilor speciale mecanismul IE este capabil să genereze informaţie parţială pentru întreg setul de instrucţiuni ce fac parte din această categorie.
- Pentru aplicaţiile ce au o componentă de instrucţiuni în virgulă mobilă rata de succes al IE de a genera informaţie parţială este mai mare de 95%.
- Datele prezentate în grafic semnalează de asemenea potenţialul utilizării IE în soluţionarea dependenţelor de date. Media instrucţiunilor din categoria citire/scriere din memorie cu date IE, a tuturor aplicaţiilor studiate este de 50%. Aplicaţia *gzip* (setul de date random) şi aplicaţia *bzip2* (setul de date source) deţin procentajul de instrucţiuni cu date IE cel mai scăzut 35% aplicaţia *vortex* are un procentaj de 64% pentru toate seturile de date studiate iar aplicaţia *parser* deţine

procentajul maxim de 68%.

- În afară de acestea capacitatea IE de a soluţiona dependenţele de control este evidentă. Făcând media tuturor instrucţiunilor din această categorie din toate aplicaţiile studiate se observă că 62% dintre instrucţiunile de control conţin informaţii IE. Procentajul de instrucţiuni de control cu date IE cel mai scăzut îl deţine aplicaţia *parser* cu 56% iar cel mai ridicat aplicaţia *vpr* cu 71%.
- Analizând potenţialul IE de a genera informaţie parţială pentru instrucţiunile din categoria aritmetica/logică se observă că în medie rata de succes a IE este de 75%. Procentajul minim de instrucţiuni cu date IE este deţinut de aplicaţia *mcf* cu 63%, iar procentajul cel mai ridicat este deţinut de aplicaţia *bzip2* cu 87%.

c. Numărul total de biţi depistaţi de IE conţinuţi în cei doi regiştrii operanzi şi în rezultatul fiecărei instrucţiuni. (figura 6.7.)

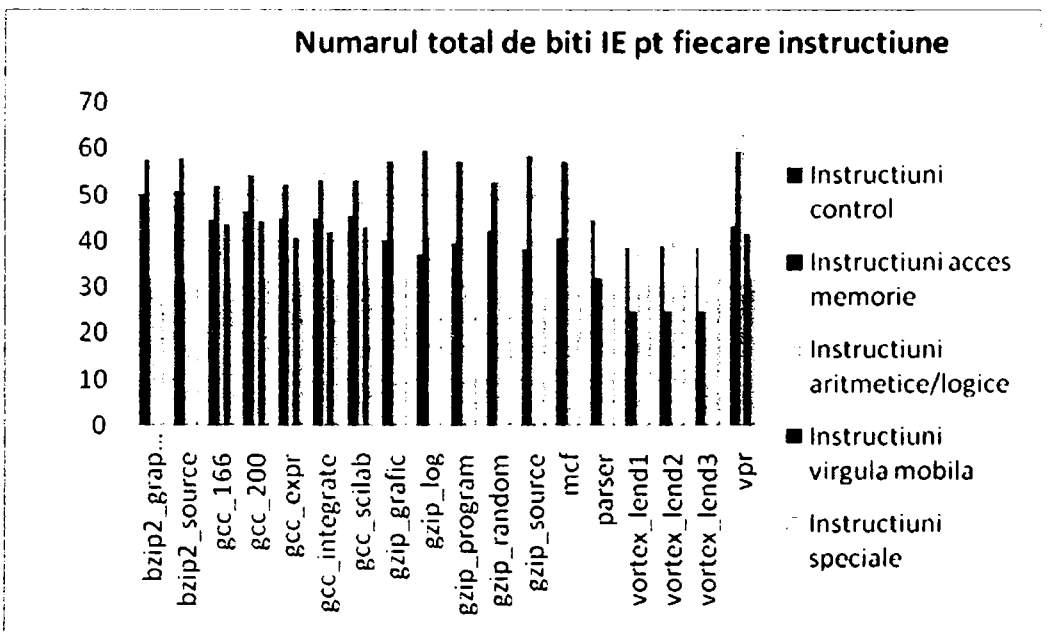


Figura 7.7. Număr de biţi IE. În figură este reprezentată pentru fiecare instrucţiune media biţilor din regiştrii operanzi ai căror valoare a fost cunoscută în urma aplicării IE.

Numărul de biţi reprezentaţi în grafic reprezintă media obţinută prin divizarea numărului total de biţi extraşi din cei doi regiştrii operanzi precum şi din rezultat, determinaţi de IE, la numărul total de instrucţiuni ce conţin date IE, dintr-o anumită categorie. Deoarece instrucţiunile simulate operează asupra unor operanzi pe 32 de biţi numărul maxim posibil de biţi ce pot fi determinaţi în urma aplicării IE este 96.

Din datele din grafic se observă că:

- Numărul de biţi ai căror valori sunt cunoscute în urma IE este semnificativ pentru toate cele cinci categorii de instrucţiuni studiate.
- În cazul instrucţiunilor de control media biţilor IE pentru toate aplicaţiile de testare este de aproximativ 43 ceea ce reflectă în mod clar probabilitatea ridicată de rezolvare a conflictelor de control pe baza informaţiei IE. Pentru această clasă de instrucţiuni numărul maxim de biţi ce conţin informaţie IE este deţinut de aplicaţia *bzip2* cu media de aproximativ 50 de biţi IE pentru fiecare instrucţiune, iar numărul minim este deţinut de aplicaţia *gzip* (setul de date de intrare log) cu media de aproximativ 37 de biţi IE.
- Probabilitatea mare de rezolvare a conflictelor de date este de asemenea reflectată în datele prezentate în grafic. În medie numărul de biţi IE pentru fiecare din instrucţiunile de accesare a memoriei este de aproximativ 50. La un capăt al spectrului aflându-se aplicaţia *gzip* (setul de date de intrare log) cu 59 de biţi IE pentru fiecare instrucţiune iar la celălalt capăt se află aplicaţia *vortex* cu 24 de biţi IE.
- Pentru instrucţiunile aritmetice/logice aplicând procedeul IE se pot determina în medie valorile a 52 de biţi.

d. Numărul total de instrucţiuni şi numărul total de biţi IE, aflaţi la fiecare nivel de dependenţă, pentru un set reprezentativ al aplicaţiilor studiate.

Scopul acestui studiu este acela de a discerne şi a evalua capacitatea şi limitele IE de a propaga rezultatele de intrare în lanţurile de instrucţiuni dependente. Este de asemenea interesant de cuantificat distanţa maximă faţă de instrucţiunile ce şi-au terminat execuţia până, la care se poate susţine propagarea

acestor date. În legătură cu aceasta a fost introdusă noţiunea de "nivel de dependenţă".

Modul prin care s-a efectuat etichetarea şi parcurgerea grafului de instrucţiuni dependente, în vederea stabilirii nivelului maxim de propagare este prezentat în figura 7.8. Figura prezintă un exemplu de graf al instrucţiunilor dependente pentru un interval de timp, cuprins între momentul **t** şi **t+k**.

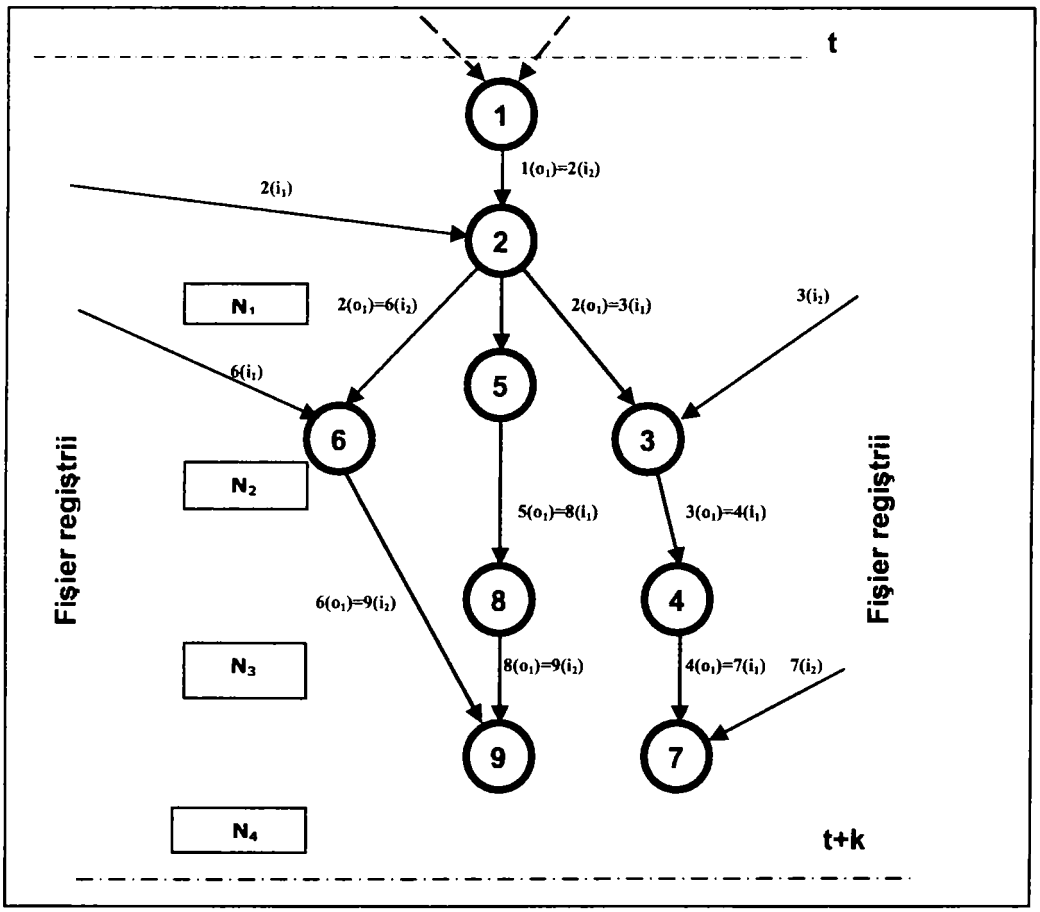


Figura 7.8. Nivel de propagare. În această figură este prezentat un exemplu de interpretare a nivelului de dependenţă într-un graf de instrucţiuni.

Rezultatul fiecărei instrucţiuni, indiferent de codul acesteia este retransmis instrucţiunilor dependente de ea. Mecanismului IE execută propagarea înainte şi înapoi până când nu se mai generează date IE noi. Instrucţiunile din graful de instrucţiuni dependente sunt etichetate în urma unei parcurgeri recursive a acestui

graf şi fiecăreia i se atribuie un anumit nivel de dependenţă, în funcţie de numărul de instrucţiuni ce se află între instrucţiunea curentă şi o instrucţiune ce şi-a terminat execuţia de care ea este dependentă. Dependenţele pot fi satisfăcute sau de rezultatele instrucţiunilor precedente sau din fişierul de regiştri.

În exemplul de faţă se presupune că instrucţiunile citesc unul sau doi operanzi şi produc un singur rezultat.

Am notat cu:

- **i1** şi **i2** cei doi regiştrii operanzi de intrare, iar cu
- **o1** rezultatul fiecărei instrucţiuni.

De exemplu spunem că:

- la momentul **t** instrucţiunea **1** şi-a încheiat execuţia. Rezultatul ei **o1** este retransmis mecanismului IE.
- pentru satisfacerea dependenţei operandului **i1** instrucţiunea **9** are un nivel de propagare **patru,** valoarea acestui operand fiind produsă de instrucţiunea **8,** care pentru a-şi produce rezultatul depinde de finalizarea instrucţiunilor **5**, **2** si **1**. Pentru satisfacerea dependenţei operandului **i2** instrucţiunea **9** are un nivel de propagare **trei** valoarea acestui operand fiind produsă de instrucţiunea **6** care pentru a-şi produce rezultatul depinde de finalizarea instrucţiunilor **2** si **1**. Spunem astfel că nivelul maxim de propagare pentru instrucţiunea 9 este 4.

În figura 7.8. am notat cu $N_{1\text{-}4}$ nivelul maxim de propagare pentru instrucţiunele prezentate.

În figura 7.9. este prezentată pentru fiecare dintre aplicaţiile luate în studiu numărul de instrucţiuni aflate la acelaşi nivel de propagare.

Dintre aplicaţiile studiate nivelul minim de propagare a datelor este 12 pentru aplicaţia *mcf*, iar nivelul maxim este 34 pentru aplicaţia *gzip*.

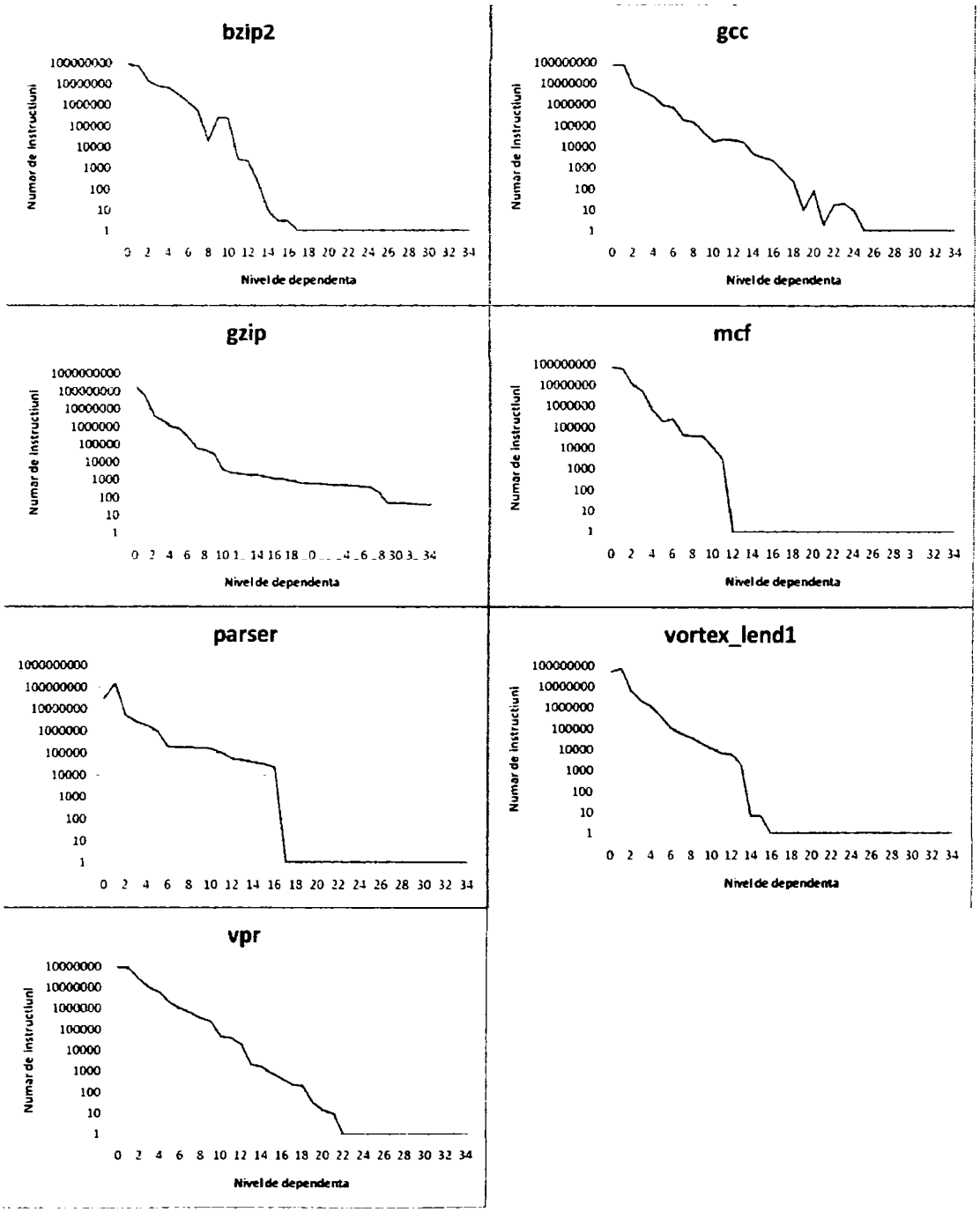


Figura 7.9. Distanţa de propagare - instrucţiuni. În această figură este prezentat, la fiecare nivel de dependenţă, numărul de instrucţiuni ce deţin informaţie parţială obţinută în urma aplicării IE.

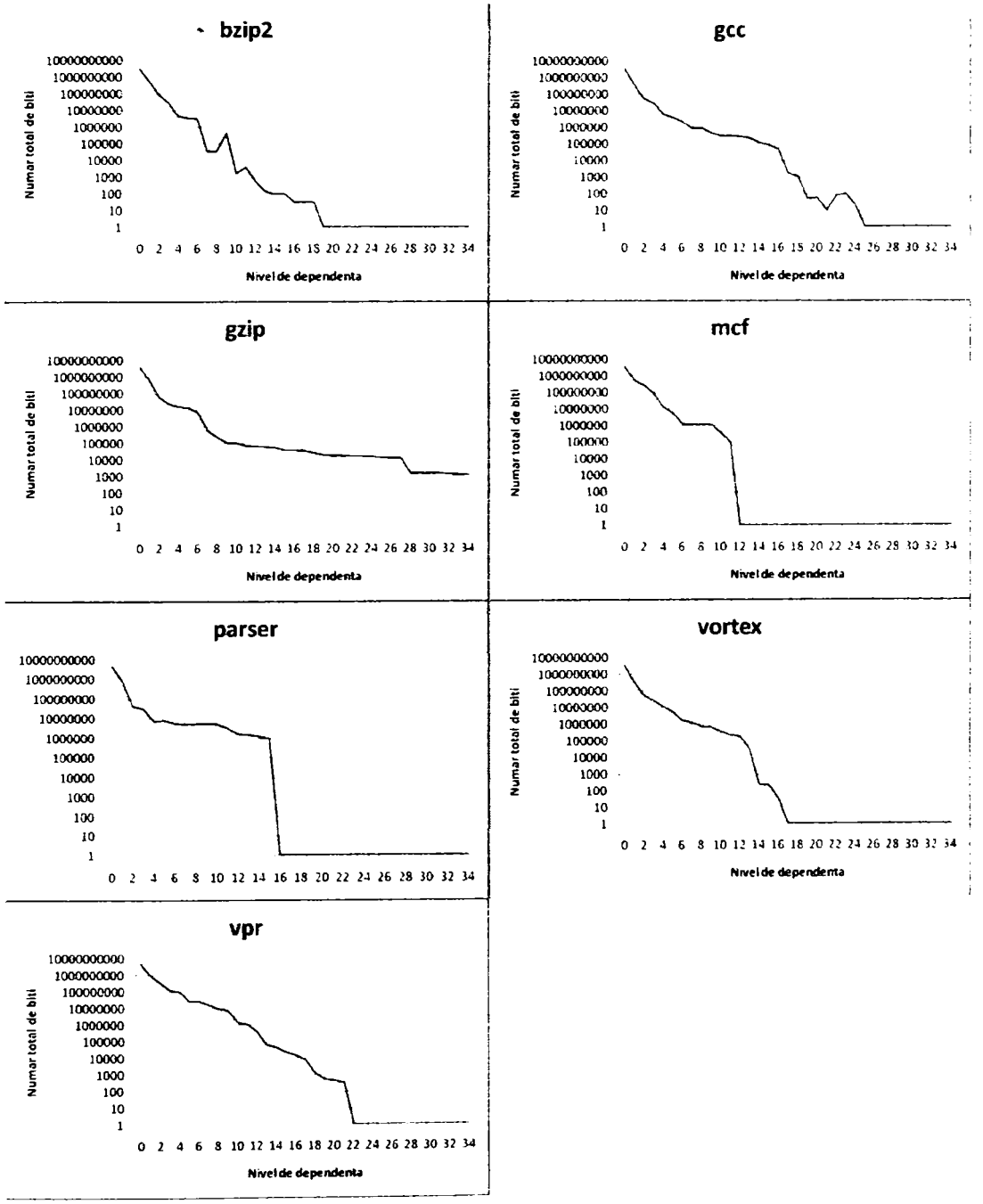


Figura 7.10. Distanţa de propagare - biţi. Figura prezintă numărul total de biţi ai căror valoare a fost determinată în urma aplicării IE, la fiecare nivel de dependenţă.

În figura 7.10 este reprezentat numărul total de biţi IE de la fiecare nivel de propagare. Scopul acestui studiu este de a estima numărul total de biţi la fiecare nivel de propagare ce reflectă totodată capacitatea IE de a susţine propagarea informaţiei de intrare la distanţe mari în lanţurile de instrucţiuni dependente.

7.2.3 Date comparative privitoare la eficienţa IE în cele două cazuri studiate.

Comparativ cu cazul în care rezultatele tuturor instrucţiunilor ce şi-au terminat execuţia sunt retransmise mecanismului IE, distribuţia instrucţiunilor relevă faptul că numărul total de instrucţiuni ce conţin date IE, în condiţiile în care mecanismul IE funcţionează de sine stătător este mai scăzut. În acest din urmă caz media obţinută este de doar 21% din totalul instrucţiunilor executate ceea ce este echivalent cu o scădere de 60% faţă de cazul precedent. Trebuie însă remarcat faptul ca şi în cazul în care IE funcţionează de sine stătator, pentru o serie de aplicaţii ca de exemplu *gzip* numărul de instrucţiuni ce conţin date IE reprezintă un procent de peste 30% din total.

În ceea ce priveşte instrucţiunile speciale, în ambele cazuri studiate, IE este capabil să determine informaţie parţială despre aproape întreg setul de instrucţiuni din această categorie (98% respectiv 100%). Procentul de 100% fiind obţinut în cazul în care rezultatele instrucţiunilor sunt retransmise IE.

În legătură cu probabilitatea de rezolvare a conflictelor de date, se constată că aceasta este mai ridicată când rezultatele instrucţiunilor sunt retransmise IE. În acest caz media instrucţiunilor de citire/scriere din memorie ce conţin date IE pentru toate aplicaţiile studiate este de 50%, IE putând determina valorile a aproximativ 50 dintre cei 96 de biţi posibili ai regiştrilor operanzi şi ai rezultatului fiecărei instrucţiuni.

Cu toate că eficienţa de a genera informaţie parţială pentru clasa de instrucţiuni de accesare a memoriei este mai scăzută în cazul în care IE funcţionează de sine stătător pentru unele dintre aplicaţii ca de exemplu *gzip* s-au determinat date IE pentru 10% din aceste instrucţiuni, determinându-se în medie valorile a 32 de biţi pentru fiecare instrucţiune.

Aşa cum era de aşteptat şi în rezolvarea conflictelor de control IE are un potenţial mai ridicat când îi sunt retransmise rezultatele instrucţiunilor, în medie

pentru 62% dintre instrucţiunile din această clasă a fost posibilă determinarea de informaţie parţială şi per global 50 dintre cei 96 de biţi ai fiecărei instrucţiuni au fost determinaţi de IE. Pentru această clasă de instrucţiuni succesul IE este mare şi atunci când funcţionează de sine stătător, putând genera date parţiale, în medie pentru 14% din totalul de instrucţiuni, putând ajunge chiar până la 31% pentru aplicaţii ca de exemplu *bzip2*. Totodată numărul de biţi IE pentru fiecare instrucţiune este în medie de 43.

În ceea ce priveşte instrucţiunile aritmetice/logice IE de sine stătător poate determina date parţiale în medie pentru 39% din totalul de instrucţiuni faţă de 75% când rezulatele instrucţiunilor ce au fost retrase din execuţie îi sunt retransmise. Numărul de biţi IE pentru fiecare instrucţiune este de 20 comparativ cu 52.

În concluzie chiar dacă eficienţa mecanismului IE este mai redusă când funcţionează de sine stătător el poate totuşi genera informaţie parţială despre un număr semnificativ de regiştri operanzi şi rezultate pentru majoritatea claselor de instrucţiuni. Cu toate că potenţialul IE de rezolvare a conflictelor de date şi de control este mai ridicat în situaţia în care îi sunt retransmise rezulatele instrucţiunilor ce şi-au încheiat execuţia, datele experimentale demonstrează totuşi că IE poate fi folosit cu succes în îmbunătăţirea performanţei şi atunci când funcţionează de sine stătător.

Alegerea uneia sau a celeilalte soluţii este o decizie ce trebuie luată în faza de implementare, evaluând pe de-o parte potenţialul mai ridicat al IE de a genera informaţie parţială atunci când rezultatele instrucţiunilor ce şi-au finalizat execuţia îi sunt retransmise, iar pe de altă parte complexitatea determinată de adăugarea structurilor hardware şi de control necesare pentru a face posibil *bypass*-ul rezultatelor din etapa *ruu_commit* în etapa ce execută IE.

7.3 Concluzii.

În acest capitol s-a efectuat un studiu al limitelor şi posibilităţilor de utilizare a mecanismului IE în situaţia în care acesta este implementat în *hardware* ca o etapă a *pipeline*-ului. A fost prezentat mediul de simulare care a fost folosit pentru obţinerea datelor de caracterizare bazat pe simulatorul de performanţă al unui

procesor *out-of-order* ce face parte din suita SimpleScalar. Au fost descrise pe scurt operaţiunile efectuate în fiecare etapă a *pipeline*-ului inclusiv etapa ce execută IE.

Pentru obţinerea datelor experimentale de evaluare a potenţialului mecanismului IE a fost necesară parcurgerea următoarelor etape ce constituie totodată o parte din contribuţiile acestui capitol:

- transpunerea în cod a algoritmului ce guvernează funcţionarea IE;
- adaugarea la *sim-outorder* a codului sursă al IE;
- modificarea componentei de analiză a SimpleScalar într-o structură de analiză asemănătoare FAF şi integrarea acesteia în *sim-outorder*;
- integrarea IE ca o etapă în pipeline-ul simulat de *sim-outorder*;
- conceperea unei metode de evaluare a potenţialului IE de a propaga informaţia parţială bazată pe parcurgerea grafului de instrucţiuni dependente şi integrarea în *sim-outorder* a codului sursă ce permite obţinerea estimării distanţei de propagare.

Capacitatea IE de a genera informaţie parţială a fost evaluată în două cazuri diferite de implementare, când rezultatele instrucţiunilor sunt retransmise IE şi când acesta funcţionează de sine stătător.

Instrucţiunile ce alcătuiesc aplicaţiile software la nivelul codului maşină au fost împărţite în cinci categorii şi s-au prezentat rezultate legate de procentajul de instrucţiuni ce conţin date IE, raportate la numărul total de instrucţiuni din fiecare categorie.

În plus de aceasta pentru a înţelege potenţialul folosirii informaţiei IE în rezolvarea conflictelor de date şi de control au fost generate date statistice referitoare la numărul total de biţi din regiştrii operanzi şi din rezultatul fiecărei instrucţiuni a căror valoare a fost cunoscută în urma aplicării IE. Nu în cele din urmă au fost estimate limitele IE de a susţine propagarea informaţiei parţiale prin măsurarea distanţei de propagare în graful de instrucţiuni dependente. Metodele de evaluare precum şi datele experimentale obţinute reprezintă o altă contribuţie importantă a acestui capitol.

Toate rezultatele experimentale relevă în mod clar capacitatea IE de a genera şi propaga informaţie parţială despre valorile din regiştrii operanzi precum şi potenţialul utilizării acestei informaţii pentru soluţionarea unor evenimente cu conotaţie negativă asupra perfomanţei în execuţie a aplicaţiilor *software* pe platformele *out-of-order*.

8. Concluzii, dezvoltări ulterioare, contribuţii originale.

8.1 Concluzii finale

Prezenta teză este centrată pe principalele activităţi desfăşurate pentru asigurarea unui nivel înalt de performanţă a sistemelor computaţionale, activităţi ce se desfăşoară pe tot parcursul proiectării unei noi platforme *hardware.*

Sunt prezentate contribuţiile autorului atât la nivel teoretic cât şi aplicativ, legate de această importantă activitate care începe în faza de explorare iniţială, continuă pe tot parcursul proiectării şi se termină după ce produsul a fost realizat fizic.

În partea de debut a lucrării este introdusă noţiunea de performanţă a sistemelor cu microprocesor, **acesta fiind de fapt obiectivul principal al tezei** şi se face o trecere în revistă a principalelor mărimi asociate ei, fiind prezentate de asemenea diferitele mijloace de evaluare, a modului de operare şi a principalelor tipuri de simulare (*trace driven* si *execution driven).*

Sinteza efectuată poate fi apreciată ca o primă contribuţie originală a acestei lucrări.

În continuare în cadrul tezei sunt prezentate rezultatele obţinute în trei direcţii de dezvoltare a conceptului de performanţă, care constitue şi obiectivele lucrării. Toate acestea reprezintă etape parcurse în procesul de fabricare al unui microprocesor şi între ele există o strânsă interconectivitate:

- Îmbunătăţirea structurii şi a calităţii mijloacelor prin care se face evaluarea performanţei.
- Cercetarea proprietăţilor, a modului de comportare în rulare a aplicaţiilor *software* şi a cerinţelor impuse de acestea structurilor *hardware* ale unui sistem computaţional.

- Conceperea unor mecanisme care să conducă la îmbunătăţirea vitezei de lucru a microprocesorului pe baza rezultatelor obţinute în urma caracterizării aplicaţiilor.

Obiectivele tezei au fost concretizate practic prin:

1. Un nou concept pentru proiectarea unui utilitar de analiză prin simulare a microprocesoarelor şi a aplicaţiilor soft denumit "Flexible Analysis Framework" (FAF). Noul concept, prezentat pe larg în capitolul 3 al tezei, derivă dintr-o necesitate practică, consecinţă a faptului că o analiză făcută asupra utilitarelor utilizate curent, relevă faptul că acestea, în mare parte, nu corespund întocmai standardelor enunţate în tabelul 2.1 din capitolul 2 al tezei.

Prin modul în care a fost gândit noul concept elimină dezavantajele aferente celor mai multe dintre utilitarele uzuale şi creează cadrul pentru eliminarea şi a aspectelor cu conotaţie negativă, datorate segmentării expertizei în analiza performanţei, între diversele departamente participante la proiectarea procesorului, fiindu-i conferite următoarele avantaje: un grad ridicat de universalitate, posibilitatea de a fi folosit pentru analiza platformelor hardware eterogene, indiferent de arhitectura acestora; capacitatea de a opera atât cu fişiere *trace* cât şi în cadrul simulatoarelor de performanţă; uşurinţa de a fi înţeles şi modificat pentru a servi unor situaţii speciale ce reclamă noi metode de analiză; influenţa scăzută asupra vitezei de simulare când operează integrat în simulatorul de performanţă.

Noul concept a servit drept fundament la edificarea unei structuri modulare de analiză de tip arbore, care este folosită în practică, în primul rând pentru studiul proprietăţilor aplicaţiilor *software* şi în al doilea rând pentru estimările de performanţă ce servesc la luarea deciziilor arhitecturale legate de structurile *hardware* aflate în faza de proiect.

În contextul utilitarelor uzuale, structura de analiză realizată conform noului concept se individualizează printr-o serie de atribute, calificate drept noutăţi, motiv pentru care s-a întocmit documentaţia necesară obţinerii unui brevet de invenţie.

2. Un studiu comparativ asupra modului de comportare în rulare a unor aplicaţii care presupun un regim computaţional dificil, specific aplicaţiilor din domeniul ştiinţific şi comercial, în care sunt evaluate şi comparate caracteristicile cu influenţă asupra performanţei.

Studiul realizat este o consecinţă a faptului că deşi cunoaşterea proprietăţilor aplicaţiilor *software* şi a modului în care acestea interacţionează cu *hardware*-ul pe care urmează să ruleze reprezintă o etapă crucială în analiza performanţei, sursele de informare documentară relevă faptul că multe dintre studiile efectuate relativ la acest aspect se limitează la analiza unui număr redus de aplicaţii sau de parametrii. În plus de aceasta sunt puţine studii ce-şi propun estimări comparative în legătură cu proprietăţile aplicatiilor ce reclamă un regim computaţional dificil cum sunt, de exemplu, cele din domeniul ştiintific şi cel comercial.

Studiul prezentat în cap.4 al tezei face o evaluare complexă a acestui tip de aplicaţii, prezentându-se rezultate şi concluzii legate de un număr important de parametrii de caracterizare cum sunt, distribuţia instrucţiunilor, localizarea spaţială şi temporală a accesărilor memoriei, influenţa organizării memoriei cache asupra numărului total de miss-uri, efectele execuţiei unor aplicaţii paralelizate într-un mediu multiprocesor asupra miss-urilor în *cache* şi nu în ultimul rând efectul precitirii datelor asupra performanţei în execuţie.

Un mijloc general acceptat în caracterizarea aplicaţiilor software este analiza fişierelor *trace*. În lucrare sunt prezentate metode curente de colectare şi validare a acestora. Validarea, etapa în urma căreia se confirmă faptul că modul de comportare reflectat de către fişierul *trace,* este identic cu comportamentul *hardware*-ului de referinţă este evidenţiată în teză printr-un exemplu în care indicatorii de performanţă (numărul de *miss*-uri în memoria cache L2) obţinuţi din mediul de simulare sunt comparaţi cu datele obţinute pe platforme *hardware* existente folosind regiştrii de performanţă care echipează microprocesorul.

În mare parte rezultatele experimentale obţinute în analiza de performanţă a aplicaţiilor ce reclamă un regim computaţional greu, au fost generate în urma folosiririi structurii de analiză ce modelează conceptul FAF. Pe lângă datele privind efectele organizării memoriei *cache* asupra numărului de *miss*-uri şi implicit asupra

performanţei, cele mai importante concluzii desprinse în urma analizei efectuate sunt următoarele:

- Evaluarea performanţei aplicaţiilor ştiintifice doar pe baza componentei în virgulă mobilă poate conduce la rezultate eronate datorită existenţei în aceste aplicaţii a unei importante componente întregi.
- Mecanismele de precitire au cel mai benefic efect asupra aplicaţiilor a căror mod de accesare a datelor este predictibil.
- S-au observat interesante asemănări între proprietăţile aplicaţiilor ştiinţifice reale şi a celor comerciale.
- Este improprie emiterea unor concluzii cu caracter general pentru întregul spectru al aplicaţiilor ştiinţifice bazat doar pe analiza unor aplicaţii din acestă categorie.

3. Cercetarea posibilităţilor şi limitelor unui posibil mecanism hardware nespeculativ, denumit "Implication Engine" (IE), de rezolvare a dependenţelor de date, şi a dependenţelor de control, ce apar în cazul procesării în paralel a instrucţiunilor.

Partea a doua a lucrării se concentrează, pe parcursul a trei capitole în special pe cea de-a treia etapă componentă a analizei de performanţă şi anume conceperea unei arhitecturi performante cu structuri hadware optimizate în vederea obţinerii unui nivel maxim de performanţă.

În cap.5 se prezintă într-un format condensat problemele de performanţă asociate cu execuţia unei aplicaţii software pe un microprocesor superscalar, *out-of-order* şi se face o sinteză a principalelor mecanisme *hardware* moderne folosite pentru soluţionarea acestora.

Folosind simulatorul de procesor superscalar *sim-outorder* component al familiei SimpleScalar şi un subset al suitei SPEC-CPU 2000, sunt evaluate efectele asupra creşterii vitezei de execuţie a unor mecanisme de îmbunătăţire a performanţei folosite în mod uzual cum sunt, nivelul doi al memoriei *cache*, predictorul hibrid al instrucţiunilor de *branch* precum şi mecanismul de dezambiguare al accesărilor memoriei. Rezultatele sunt edificatoare, majoritatea acestor mecanisme îmbunătăţind viteza de execuţie a aplicaţiilor studiate de câteva ori. Trebuie făcută remarca că o seamă de mecanisme de îmbunătăţire a

performanţei printre care se numără predictorul instrucţiunilor de *branch*, predictorul valorilor regiştrilor operanzi, predictorul alias-urilor între instrucţiunile de citire/scriere din memorie precum şi altele de acest gen, mecanisme a căror eficacitate este incontestabilă, au dezavantajul că sunt bazate pe prezumţii speculative iar de multe ori speculaţiile se dovedesc false, fapt ce crează probleme legate de viteza de procesare.

Pentru eliminarea dezavantajului enunţat anterior propun un concept nou pentru un mecanism pe care l-am denumit **„Implication Engine" (IE)**. Acest mecanism este destinat îmbunătăţirii performanţei microprocesorului iar prin faptul că este nespeculativ abordează în mod diferit soluţionarea problemelor de performanţă. În cap.6 sunt prezentate ideile care au stat la baza edificării lui:

- examinarea ponderii diferitelor clase de instrucţiuni în urma caracterizării unui set larg de aplicaţii software, solicitante pentru structurile componente ale unui microprocesor.
- observarea similitudinilor între modul de funcţionare a unor blocuri alcătuite din porţi logice şi anumite instrucţiuni în limbaj de asamblare ce sunt executate de microprocesor.
- evaluarea beneficiilor ce pot rezulta ca urmare a cunoaşterii valorilor datelor din regiştrii înainte ca acestea să devină cunoscute în urma execuţiei regulate.
- examinarea beneficiilor asupra performanţei ce pot rezulta în urma cunoaşterii şi folosirii valorilor parţiale din regiştrii operanzi ai instrucţiunilor.
- observaţia că valorile parţiale din regiştri operanzi pot fi determinate în urma aplicării implicării înainte şi înapoi, un principiu similar celui executat de algoritmului de testare a circuitelor combinaţionale PODEM.

Sunt identificate sursele de informaţie parţială şi se prezintă prin exemple modul prin care IE determină valorile biţilor conţinuţi în regiştrii operanzi aplicând implicarea înainte şi înapoi. Lucrarea scoate în evidenţă zonele unde informaţia determinată de IE poate fi folosită cu succes pentru rezolvarea unor evenimente cu conotaţie negativă asupra performanţei şi implicit pentru creşterea vitezei de execuţie.

Sunt propuse de asemenea două metode prin care IE poate fi implementat în hardware atât într-un microprocesor superscalar cât şi într-unul procesor CMP.

În capitolul 7 se efectuează un studiu asupra potenţialului şi a limitărilor IE în a genera informaţie parţială, despre regiştrii operanzi ai instrucţiunilor. Acestea sunt dovedite în urma unui studiu de caracterizare minuţios a soluţiei în care IE este implementat ca o etapă a pipeline-ului unui microprocesor superscalar.

Pentru evaluare, IE a fost modelat în *software* şi integrat în *sim-outorder,* iar pentru obţinerea datelor de caracterizare a fost folosită o structură de analiză asemănătoare cu FAF. Pentru a studia distanţa până la care IE poate propaga informaţia parţială a fost utilizată noţiunea de nivel de dependenţă şi a fost conceput un mecanism de analiză, bazat pe parcurgerea recursivă a grafului instrucţiunilor dependente.

Analiza IE a fost efectuată în două situaţii :

- când funcţionează de sine stătător utilizând doar datele provenite din sursele de informaţie parţială.

- când rezultatele instrucţiunilor retrase din execuţie sunt retransmise IE, aceasta pentru a stabili nivelul maxim de informaţie parţială ce poate fi determinată de IE.

Rezultatele experimentale relevă clar potenţialul ridicat al IE de a genera informaţie parţială şi de a o propaga în graful de instrucţiuni dependente. Totodată procentajul instrucţiunilor de accesare a memoriei şi instrucţiunilor condiţionale ce conţin informaţie IE este foarte ridicat fiind evidentă posibilitatea de soluţionare a conflictelor de date şi control în urma folosirii valorilor parţiale din regiştrii operanzi obţinute de IE.

În final se poate concluziona că obiectivele tezei au fost realizate autorul aducandu-şi contribuţii originale în toate cele trei domenii abordate.

8.2 Dezvoltări ulterioare

Domeniul IT se află într-o permanentă schimbare aplicaţiile devin din ce în ce mai complexe alimentate fiind de cerinţele tot mai ample ale utilizatorilor precum şi de posibilităţile tehnologice actuale care permit realizarea unor platforme *hardware* tot mai performate.

În acest context pe baza rezultatelor obţinute fiecare din cele trei probleme abordate în teză au o paletă largă de dezvoltări viitoare.

1. În cazul activităţii de caracterizare din punct de vedere computaţional a aplicaţiilor rulate, îmi propun să extind studiile de caracterizare de genul celor prezentate în teză spre un spectru larg de aplicaţii ştiintifice şi comerciale precum şi spre aplicaţii noi, insuficient caracterizate, acesta din două motive:

- Aplicaţiile software în prezent se îndreaptă spre domenii total necunoscute cu câţiva ani în urmă, ca de exemplu "*social networking*", „*web analytics*" sau "*cloud computing*". Astfel de aplicaţii deoarece operează asupra unor seturi de date de ordinul tera sau peta *bytes*-ilor şi rulează în medii virtuale, pe clustere de sisteme distribuite conectate prin reţele, modifică de multe ori modul tradiţional de procesare în care datele sunt transmise spre nodurile de procesare, distribuind în schimb procesarea spre nodurile care stochează datele. Caracteristicile acestui gen de aplicaţii nu sunt îndeajuns cunoscute fabricanţilor de platforme *hardware*. Aceştia necunoscându-le proprietăţile şi caracteristicile de execuţie nu pot optimiza sistemele computaţionale pe care le proiectează pentru ca aceste aplicaţii să ruleze la un nivel ridicat de performanţă.
- În ceea ce priveşte aplicaţiile tradiţionale cu cât spectrul aplicaţiilor caracterizate este mai larg cu atât munca de optimizare a componentelor sistemelor computaţionale poate fi canalizată mai precis.

2. Pentru dezvoltarea şi îmbunătăţirea calităţii mijlocelor folosite în analiza aplicaţiilor software, îmi propun extinderea structurii de analiză FAF cu noi *analyzer*-e şi *profiler*-e. Acesta în condiţiile în care, aşa cum am arătat anterior, pe piaţă apar în permanenţă noi aplicaţii *software*.

3. În ceea ce priveşte IE dezvoltările ulterioare vizează următoarele aspecte:

- Cuantificarea efectelor folosirii informaţiei parţiale generate de IE pentru soluţionarea unor evenimente cu conotaţie negativă asupra performanţei unui microprocesor superscalar cum sunt: dezambiguarea accesărilor memoriei, detectarea rapidă a direcţiei instrucţiunilor condiţionale sau detectarea unui *hit* sau *miss* în memoria cache.
- Extinderea numărului de aplicaţii software utilizate pentru evaluarea potenţialului IE.

- Examinarea unor alte posibile aplicaţii ale IE ca de exemplu:
  - eliminarea etapei de distribuire şi execuţie în cazul instrucţiunilor a căror rezultat este determinat în totalitate în urma IE.
  - *bypass*-ul valorilor parţiale IE spre instrucţiunile dependente. Lucrări ca de exemplu [8.1] demonstrează faptul că instrucţiunile ce operează cu întregi nu au nevoie de întregul set de valori din regiştrii operanzi pentru a-şi putea începe execuţia putând fi astfel lansate în execuţie pe baza informaţiei generată de IE.
- Nu în ultimul rând îmi propun modelarea în software a IE într-un microprocesor CMT aplicând metoda descrisă în teză. Pentru ca acest lucru să fie posibil este necesară parcurgerea următoarelor etape:
  - integrarea IE într-un simulator de performanţă a unui procesor CMT.
  - conceperea metodologiei de analiză pentru evaluarea potenţialului utilizării informaţiei parţiale IE în această implementare.
  - cuantificarea beneficiilor asupra performanţei rezultate în urma utilizării IE.

8.3 Contribuţii

Cap 2

S-a efectuat o sinteză asupra:

1. Indicilor de performanţă utilizaţi în evaluarea performanţei microprocesoarelor.
2. Mijloacelor de evaluare a performanţei;
   a. principalele tipuri de simulatoare.
   b. principalele tehnici de simulare.
   c. aplicaţiile de testare.
3. Etapelor parcurse în simulare, în vederea îmbunătăţirii performanţei unei platforme *hardware* noi.

Cap 3

Este prezentat conceptul "***Flexible Analysis Framework***" – FAF în următoarele etape:

1. S-a procedat la analiza critică a utilitarelor şi a modului în care se efectuează activitatea de analiză a performanţei. S-au definit:
   a. conceptul FAF
   b. arhitectura conceptului FAF
   c. modalitatea de structurare funcţie de context de utilizare
2. Domenii de utilizare a conceptului:
   a. pentru a caracteriza diverse aplicaţii ce rulează pe platforme *hardware* existente şi compararea datelor obţinute cu date realizate pe platformele aflate în fază de proiect.
   b. studiul accesărilor locaţiilor de memorie pentru întregul spectru al aplicaţiilor software incluzând cele de tip comercial şi ştiintific.
   c. pentru a obţine informaţii despre calitatea fişierelor *trace*, utilizate de către proiectanţii viitoarelor sisteme procesor.
3. S-a exemplificat: folosirea FAF în două situaţii :
   a. în cadrul simulatorului de performanţă:
      i. s-a ilustrat modul de integrare a FAF într-un simulator de performanţă
      ii. s-au prezentat rezultate exeperimentale legate de utilizarea FAF pentru obţinerea unor informaţii legate de *map*-area diferitelor evenimente ce apar în execuţia instrucţiunilor spre codul sursă care le-a generat.
      iii. s-a efectuat o ierarhizare a celor mai frecvent executate funcţii dintr-un subset al aplicaţiilor SPEC CPU2000 şi s-a cuantificat contribuţia fiecărei funcţii la numărul total de *miss*-uri în memoriile *cache* şi predicţii greşite a instrucţiunilor de *branch*.
   b. în analiza fişierelor *trace.*
      i. s-au prezentat rezultate exerimentale legate de modul în care instrucţiunile specifice diferitelor aplicaţii accesează diversele segmente de memorie

4. Conceptul s-a materializat printr-un utilitar. Pentru concept şi utilitar sunt scoase în evidenţă contribuţiile proprii în cadrul echipei care a construit utilitarul în baza conceptului.
5. Prin implementarea utilitarului sunt aduse următoarele contribuţii:
   a. Îmbunătăţirea condiţiilor de lucru în simulare prin faptul că noul utilitar (FAF), având o structură modulară este uşor de înţeles şi utilizat.
   b. Îmbunătăţirea tehnicilor de simulare cu un utilitar ieftin şi non invaziv.
   c. Îmbunătăţirea tehnicilor de simulare cu un utilitar care permite o gamă variată de tipuri de analiză şi obţinerea unor date diverse, într-o singură simulare.
   d. Îmbunătăţirea tehnicilor de simulare cu un utilitar care permite obţinerea informaţiilor despre modul de rulare al programelor de testare şi evaluare a performanţei, atât pe platformele existente cât şi pe cele în faza de proiect
6. S-a prezentat conceptul şi aplicarea unei metode de validare a rezultatelor produse de simulatoarele de performanţă când rulează în modul de simulare *execution driven,* utilizând rezultatele de analiză generate de FAF.
7. **FAF este subiectul unui brevet de invenţie trimis spre aprobare la „United States Patent and Trademark Office".**

Cap 4

S-a efectuat un studiu în care s-au aplicat un spectru larg de tehnici de caracterizare, atât aplicaţiilor din spectrul aplicaţiilor *ştiinţifice*, cât şi celor din spectrul aplicaţiilor *comerciale* scop pentru care:

1. s-a efectuat un studiu al lucrărilor de specialitate ce au ca subiect caracterizarea aplicaţiilor *software* în scopul identificări proprietăţilor acestora, definitorii pentru nivelul de performanţă cu care ele vor rula pe platformele *hardware*.
2. în urma unui proces se selecţie au fost identificate un set de aplicaţii ştiinţifice şi comerciale reprezentative pentru categoria de aplicaţii solicitante pentru sistemele computaţionale.
3. s-a definit o metodologie şi o infrastructură.

4. s-a analizat distribuţia instrucţiunilor.
5. s-a procedat la analiza accesărilor locaţiilor de memorie din punct de vedere al localizării spaţiale şi temporale.
6. s-a examinat comportarea diferitelor aplicaţii în corelaţie cu configurarea memoriilor *cache* de instrucţiuni şi de date pentru sistemele uniprocesor.
7. s-au estimat efectele execuţiei unor aplicaţii paralelizate într-un mediu multiprocesor asupra ratei de *miss* în memoria cache.
8. s-a analizat influenţa instrucţiunilor de precitire a datelor asupra performanţei definite de **CPI**.
9. s-au tras o serie de concluzii referitor la proprietăţile aplicaţiilor studiate.
10. s-au efectuat studii comparative între aplicaţiile ştiinţifice şi cele comerciale, s-au tras o serie de concluzii legate de asemănările şi deosebirile dintre acestea.
11. **au fost publicate 3 articole în conferiţe şi jurnale internaţionale dintre care 2 sunt indexate ACM şi cotate fiecare, în alte 7 lucrări de specialitate, două dintre cele trei articole fiind publicate în calitate de prim autor.**

Cap 5

Este dedicat discutării tehnicilor de îmbunătăţire a performanţei microprocesoarelor funcţie de etapele de execuţie a instrucţiunilor.

Se face o sinteză a următoarelor aspecte:

1. procesarea *in-order* şi procesarea *out-of-order;*
2. tehnologia *pipeline;*
3. procesarea în paralel, conceptul ILP (*Instruction Level Parallelism*);
4. dependenţele de date şi de control;
5. tipuri de hazard RAW, WAR si WAW;
6. tehnici de îmbunătăţire a performanţei microprocesoarelor funcţie de etapele de execuţie a instrucţiunilor;
7. studiu al direcţiilor actuale de îmbunătăţire a microprocesoarelor

Cap 6

Este dedicat prezentării conceptului *Implication Engine* (IE). Pentru definirea acestui concept s-au parcurs următoarele etape:

1. prezentarea premiselor care au condus la definirea conceptului.
2. prezentarea algoritmului PODEM şi extinderea sa.
3. evaluarea eficacităţii algoritmului în generarea vectorilor de testare pentru circuitele logice.
4. exemple de funcţionare a PODEM şi similitudini între funcţionarea acestuia şi IE.
5. definirea conceptului IE:
    i. Identificarea surselor de informaţie parţială.
    ii. Exemple de funcţionare a implicării înainte şi înapoi asupra instrucţiunilor executate de un microprocesor.
    iii. Identificarea zonelor în care informaţia parţială generată de IE poate fi utilizată în scopul creşterii performanţei.
6. două modalităţi de implementare:
    i. într-un procesor superscalar
    ii. într-un procesor CMT

Cap 7

Este dedicat studiului limitelor şi posibilităţilor de utilizare a mecanismului IE pentru creşterea vitezei de execuţie a unei aplicaţii în situaţia în care acesta este implementat în *hardware* ca o etapă a *pipeline*-ului. Pentru aceasta s-au realizat următoarele:

1. modelarea în software a algoritmului ce guvernează funcţionarea IE;
2. adăugarea la *sim-outorder* a codului sursa al IE;
3. modificarea componentei de analiză a SimpleScalar într-o structură de analiză asemănătoare FAF şi integrarea acesteia în *sim-outorder*;
4. integrarea IE ca o etapă în *pipeline*-ul simulat de *sim-outorder*;
5. conceperea unei metode de evaluare a potenţialului IE de a propaga informaţia parţială bazată pe parcurgerea grafului de instrucţiuni dependente şi integrarea în *sim-outorder* a codului sursă ce permite estimarea distanţei de propagare.

6. modelarea în software a metodelor de evaluare.
7. estimarea capacităţii de rezolvare a conflictelor de date şi control rezultate în urma utilizării informaţiei parţiale generată de IE pe baza datelor experimentale obţinute.
8. evaluarea IE în două cazuri diferite de implementare, când rezultatele instrucţiunilor sunt retransmise IE şi când acesta funcţionează de sine stătător.

Preocupările autorului în domeniul arhitecturii microprocesoarelor s-au materializat de asemenea printr-o altă lucrare de cercetare care nu face obiectul tezei:

Arijit Biswas , Paul Racunas , **Razvan Cheveresan** , Joel Emer, Shubhendu S. Mukherjee, Ram Rangan, „Computing Architectural Vulnerability Factors for Address-Based Structures", publicată în Proceedings of the 32nd annual international symposium on Computer Architecture, June 2005 indexată ACM şi IEEE care a fost citată in 64 de lucrări de specialitate.

Bibliografie:

[1.1] R.Hiremane – *„From Moore's Law to Intel's Innovation – Prediction to Reality"* Technology@Intel Magazine, April 2005.

[1.2] W. Warner. "*Great moments in microprocessor history*". IBM developerWorks, 22 December 2004.

[1.3] **R.Cheveresan**, S.Holban, "*Workload Characterization an Essential Step in Computer Systems Performance Analysis - Methodology and Tools*", Advances in Electrical and Computer Engineering, Suceava 2009.

[2.1] Lepak, K.M. Cain, H.W. Lipasti, M.H. *„Redeeming IPC as a performance metric for multithreaded programs"*, Parallel Architectures and Compilation Techniques, 2003. PACT 2003.

[2.2] John L. Hennesy, David A. Patterson *„Computer Architecture a Quantitative Approach",* Morgan Kauffman Publishers, 2003.

[2.3] *„The Computer Engineering Handbook, Second Edition",* CRC; 2 edition (January 7, 2008)

[2.4] K. Arnold, J.Gosling, D.Holmes, *„The Java Programming Lanuguage Third Edition"*, Prentice Hall PTR; 3 edition (June 5, 2000).

[2.5] M. Rosenblum , E. Bugnion , S. Devine , S. A. Herrod *„Using the SimOS Machine Simulator to Study Complex Computer Systems"*, Proceedings of the ACM Transactions on Modeling and Computer Simulation, 1997.

[2.6] *„Introduction to the Simics Full System Simulator without Equal",* Virtutech White Paper, 2002.

[2.7] S.S.Mukherjee, S.V.Adve, T.Austin, J.Emer, P.S.Magnusson *„Performance Simulation Tools",* Computer, v.35 n.2, p.38-39, February 2002.

[2.8] D.J,Lilja, J.J.Yi, *„Simulations of Computer Architectures: Simulators, Benchmarks, Methodologies and Recommendations",* IEEE Transactions on Computers, 2006.

[3.1] J. Anderson, L. Berc, J. Dean, S. Ghemawat,M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspruger, and W. Weihl. *„Continuous pro_ling: Where have all the cycles gone?"* In Proceedings of the 16th ACM Symposium of Operating Systems Principles, October 1997.

[3.2] J. Dean, J. Hicks, C. Waldspruger, W. Weihl, and G. Chrysos, *„Proleme:Hardware support for instruction-level profiling on out-of-order*

processors". In Proceedings of Annual International Symposium on Microarchitecture, 1997.

[3.3] *„Sun studio performance analyzer"*: developers.sun.com/prodtech/cc/ analyzerindex.html.

[3.4] *„Intel vtune performance analyzer":*
www.intel.com/cd/software/products/asmona/ eng/vtune/index.htm.

[3.5] M. Martonosi, A. Gupta, and T. Anderson. Memspy: *„Analyzing memory system bottlenecks in programs".* In Measurement and Modeling of Computer Systems, pages 1-12, 1992.

[3.6] A. Lebeck, D. Wood, *„Cache profiling and the spec benchmarks: A case study",* IEEE Computer, 27(10):15-26, October 1994.

[3.7] S. Graham, P. Kessler, and M. McKusick, *„gprof: a call graph execution profiler",* In SIGPLAN: Symposium on Compiler Construction, 1982.

[3.8] B. Cantrill, M. Shapiro, and A. Leventhal, *„Dynamic instrumentation of production systems"* , USENIX Annual Technical Conference, pages 15-28, 2004.

[3,9] A. Tamches and B. Miller. *„Fine-grained dynamic instrumentation of commodity operating system kernels",* 3rd Symposium on Operating System Design and Implementation, 1999.

[3.10] J. Roberts and C. Zilles, *„Tracevis: An execution trace visualization tool"* Workshop on Modeling, Benchmarking, and Simulation, 2005.

[3.11] E. Vlaovic and S. Davidson, *„Taxi: Trace analysis for x86 interpretation",* IEEE International Conference on Computer Design, 2002.

[3.12] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 (http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF).

[3.13] S. Microsystems, *„UltraSPARC User's Manual",* Sun Microsystems, Inc., July 1997.

[3.14] J. Mauro and R. McDougall, *„Solaris Internals – Core Kernel Architecture",* Sun Microsystems Press, 2001.

[3.15] T.M.Ismail, **R.Cheveresan**, M.Ramsay *„Apparatus for Application Level Analysis of Hardware Simulations"* Trimis spre patentare la United States Patents and Trademark Office.

[4.1] P. J. Denning and S. C. Schwartz, *„Properties of the working-set model"*, Commun. ACM, 15(3):191-198, 1972.

[4.2] **R. Cheveresan** , M. Ramsay , C. Feucht , I. Sharapov, *„Characteristics of workloads used in high performance and technical computing"*, Proceedings ofthe 21st annual international conference on Supercomputing, June 17-21, 2007,

Seattle, Washington.

[4.3] K. Rupnow , A. Rodrigues , K. Underwood , K. Compton, *„Scientific applications vs. SPEC-FP: a comparison of program behavior"*, Proceedings of the 20th annual international conference on Supercomputing, June 28-July 01, 2006, Cairns, Queensland, Australia.

[4.4]. F. Darema-Rogers , G. F. Pfister , K. So, *„Memory access patterns of parallel scientific programs"*, Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems, p.46-58, May 11-14, 1987, Banff, Alberta, Canada.

[4.5] L. Carrington, A. Snavely, X. Gao, and N. Wolter, *„Performance prediction framework for scientic applications"*, Lecture Notes in Computer Science, 2659, pages 926--935. Springer, January 2003.

[4.6] P.Trancoso , J-L. Larriba-Pey , Zheng Zhang , Josep Torrellas, *„The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors"*, Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, p.250, February 01-05, 1997.

[4.7] J. Weinberg , M. O. McCracken , E. Strohmaier , A. Snavely, *„Quantifying Locality In The Memory Access Patterns of HPC Applications"*, Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p.50, November 12-18, 2005.

[4.8] J. Peachey, R. Bunt, and C. Colbourn, *„Towards an intrinsic measure of program locality",* 16th Annual Hawaii International Conference on System Sciences, pages 128--137, 1983.

[4.9]. I. Sharapov , R. Kroeger , G. Delamarter , **R. Cheveresan** , M. Ramsay, *„A case study in top-down performance estimation for a large-scale parallel application"*, Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, March 29-31, 2006, New York, New York, USA.

[4.10] Transaction Processing Perfomance Council (www.tpc.org/tpcc).

[4.11] E. Strohmaier , H. Shan, *„Architecture Independent Performance Characterization and Benchmarking for Scientific Applications"*, Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04), p.467-474, October 04-08, 2004.

[4.12] A. J. Smith, *„Cache Memories"*, ACM Computing Surveys (CSUR), v.14 n.3, p.473-530, Sept. 1982.

[4.13] S. Ghosh , M. Martonosi , S. Malik, *„Cache miss equations": a compiler framework for analyzing and tuning memory behavior"*, ACM Transactions on Programming Languages and Systems (TOPLAS), v.21 n.4, p.703-746, July 1999.

[4.14] P. J. Denning, *„The working set model for program behavior"*, Communications of the ACM, v.11 n.5, p.323-333, May 1968.

[4.15] P. J. Denning , S. C. Schwartz, *„Properties of the working-set model"*, Communications of the ACM, v.15 n.3, p.191-198, March 1972.

[4.16] R. A. Uhlig , T. N. Mudge, *„Trace-driven memory simulation: a survey"*, ACM Computing Surveys (CSUR), v.29 n.2, p.128-170, June 1997.

[4.17] E. H. Gornish , E. D. Granston , A. V. Veidenbaum, *„Compiler-directed data prefetching in multiprocessors with memory hierarchies"*, Proceedings of the 4th international conference on Supercomputing, p.354-368, June 11-15, 1990, Amsterdam, The Netherlands.

[4.18] L. Spracklen , Y. Chou , S. G. Abraham, *„Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications"*, Proceedings of the 11th International Symposium on High-Performance Computer Architecture, p.225-236, February 12-16, 2005.

[4.19] S. J. Eggers, *„Simulation Analysis Data Sharing in Shared Memory Multiprocessors"*, University of California at Berkeley, Berkeley, CA, 1989.

[4.20] K. Keeton , D. A. Patterson , Y. Q. He , R. C. Raphael , W. E. Baker, *„Performance characterization of a Quad Pentium Pro SMP using OLTP workloads"*, Proceedings of the 25th annual international symposium on Computer architecture, p.15-26, June 27-July 02, 1998, Barcelona, Spain.

[4.21] P. Lusczek, J. Dongarra, J. Kepner, "*Design and Implementation of the HPC Challenge Benchmark Suite*", *CTWatch Quarterly,* Volume 2, Number 4A, Nov.2006.

[4.22] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and Horst D. Simon. *„TOP500 Supercomputer Sites"* , 28th edition, November 2006.

[4.23] J. Dongarra and P. Luszczek, *„Introduction to the HPC Challenge benchmark suite"*, Technical Report UT-CS-05-544, University of Tennessee, 2005.

[4.24] J. Dongarra, P. Luszczek, and A. Petitet, *„The linpack benchmark: Past, present and future"*, Concurrency: Practice and Experience, 15:803-820, 2003.

[4.25] Nas Parallel Benchmarks (http://www.nas.nasa.gov/Resources/Software/npb.html).

[4.26] D. H. Bailey, T. Harris, R. Van der Wigngaart, W. Saphir, A. Woo, and M.Yarrow, *„The NAS Parallel Benchmarks 2.0"*, Technical Report NAS-95-010, NASA Ames Research Center, 1995.

[4.27] T. Macke, *„Nab, a language for molecular manipulation",* PhD Thesis, The Scripps Research Institute, 1996.

[4.28] LS-DYNA, Livermore Software Technology Corporation (www.lstc.com).

[4.29] SAP Standard Aplication Benchmarks (www.sap.com/solutions/benchmark).

[4.31] CORPORATE SPARC International, Inc. *„The SPARC architecture manual (version9)",* Prentice-Hall, Inc. Upper Saddle River, NJ 1994.

[4.32] C. Ding and Y. Zhong, *„Predicting wholeprogram locality through reuse distance analysis",* ACM SIGPLAN Conference on Programming Language Design andImplementation. ACM Press, 2003.

[4.33] R.Kroeger, I.Sharapov, **R.Cheveresan**, M.Ramsay, G.Delamarter *„A Case Study in Top-Down Performance Estimation for A Computational Workload"* memoriu tehnic publicat intern Sun Microsystems Octombrie 2005.

[5.1] P.Kongetira, K. Aingaran, K.Olukotun, *„Niagara: a 32-way Multithreaded Sparc Processor"*, IEEE Micro, March 2005.

[5.2] Werner Bucholz, ed., *„Plannig a Computer System"*, New York: McGraw-Hill 1962.

[5.3] J.E.Smith, G.Sohi, *„The Microarchitecture of Superscalar Processors"*, Proceedings of the IEEE, vol. 83, pp. 1609--1624, Dec.1995

[5.4] G.F.Grohoski, *"Machine Organization of the IBM RISC System/6000 processor",* IBM Journal of Research and Development vol.34, pp.37-58, Jan. 1990.

[5.5] R.R. Oehler and R.D.Groves, *„IBM RISC System/6000 Processor Architecture",* IBM Journal of Research and Development, 1990.

[5.6] N.P. Jouppi and D.W. Wall, *„Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines"*, Proc.Architectural Support for Programming Languages and OS, Boston, Apr. 1989.

[5.7] J.E. Smith, A.R. Pleskun, *„Implementing Precise Interrupts in Pipelined Processors",* IEEE Transactions on Computers, vol. 37, No. 5 May 1998, New York, N.Y., pp. 562-573.

[5.8] K.D. Cooper, L. Torczon, *„Engineering a compiler"*, Morgan Kaufmann 2005.

[5.9] S. McFarling. *„Combining branch predictors".* Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[5.10] T.Y. Yeh, Y.N Patt, (1991). *"Two-Level Adaptive Training Branch Prediction*". Proceedings of the 24th annual international symposium on Microarchitecture. Albuquerque, New Mexico, Puerto Rico: ACM. pp. 51 – 61.

[5.11] M.H. Lipasti, J.P. Shen *„Exceeding the Dataflow Limit via Value Prediction"* Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, Paris, France, Dec. 1996.*

[5.12] C. R. Moore , D. M. Balser , J. S. Muhich , R. E. East, *„IBM Single Chip RISC Processor (RSC)"*, Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors, p.200-204, Oct. 11-14, 1992.

[5.13] E. Rotenberg, S. Bennett & J. E. Smith, Trace Cache: *„A Low Latency Approach to High Bandwith Instruction Fetching"*, IEEE Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996, pp. 24-34.

[5.14] L. Conway, B. Randell, D.P. Rozenberg, D.N, Senzig, *„Dynamic Instruction Scheduling"* RJ 565, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, Mar. 1969.

[5.15] M. Goshima, K. Nishino, Y. Nakashima, S.I. Mori, T. Kitamura, S. Tomita, *„A high-speed dynamic instruction scheduling scheme for superscalar processors"* In International Symposium on Microarchitecture, 2001.

[5.16] A. J. Smith, „Cache memories", *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sep. 1982.

[5.17] G.Z. Chrysos, J.S. Emer, *„Memory Dependence Prediction Using Store Sets"* ISCA 1998: 142-153.

[5.18] N. P. Jouppi, *„Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers"*, Proceedings of the 17th annual international symposium on Computer Architecture, p.364-373, May 28-31, 1990.

[5.19] D. C. Burger and T. M. Austin, *„The SimpleScalar tool-set, Version 2.0"*, Technical Report 1342, Department of Computer Science, UW, Jun. 1997.

[5.20] D.T. Marr, D.Koufaty, *„Hyperthreading Technology in the Netburst Microarchitecture"*, IEEE Micro, 2003.

[5.21] R.Kalla, B.Sinharoy and J.Tendler, "Power5: Ibm's next generation power microprocessor", In Proc. 14th Hot Chips Symp, Aug. 2003.

[5.22] L.A.Barroso, K.Gharachorloo, R. McNamara, A.Nowatzyk, S.Qadeer, B.Sano, S.Smith, R.Stets and B.Varghese, *„Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, Jun. 12--14 2000.

[5.23] J.M.Tendler, J.S.Dodson, J.S.Fields Jr, H.Le, B.Sinharoy, *„POWER 4 System Microarchitecture"*, IBM Journal of Research and Development 2002.

[5.24] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, *„Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor"*, ISCA'09, Jun. 2009.

[5.25] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer , M. Tremblay, *„Rock: A High-Performance Sparc CMT Processor"*, IEEE Micro, v.29 n.2, p.6-16, Mar. 2009.

[6.1] M.H. Lipasti, J.P. Shen *„Exceeding the Dataflow Limit via Value Prediction" Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, Paris, France, Dec. 1996.

[6.2]. P. Goel, *„An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits,"* IEEE Trans. on Comput., Vol.30, No.3, pp. 215-222, 1981

[6.3] P.Goel *„PODEM-X: An automatic test generation system for VLSI logic structures"* Proc. Of the International Fault-Tolerant Computing Symp, Aug.1980.

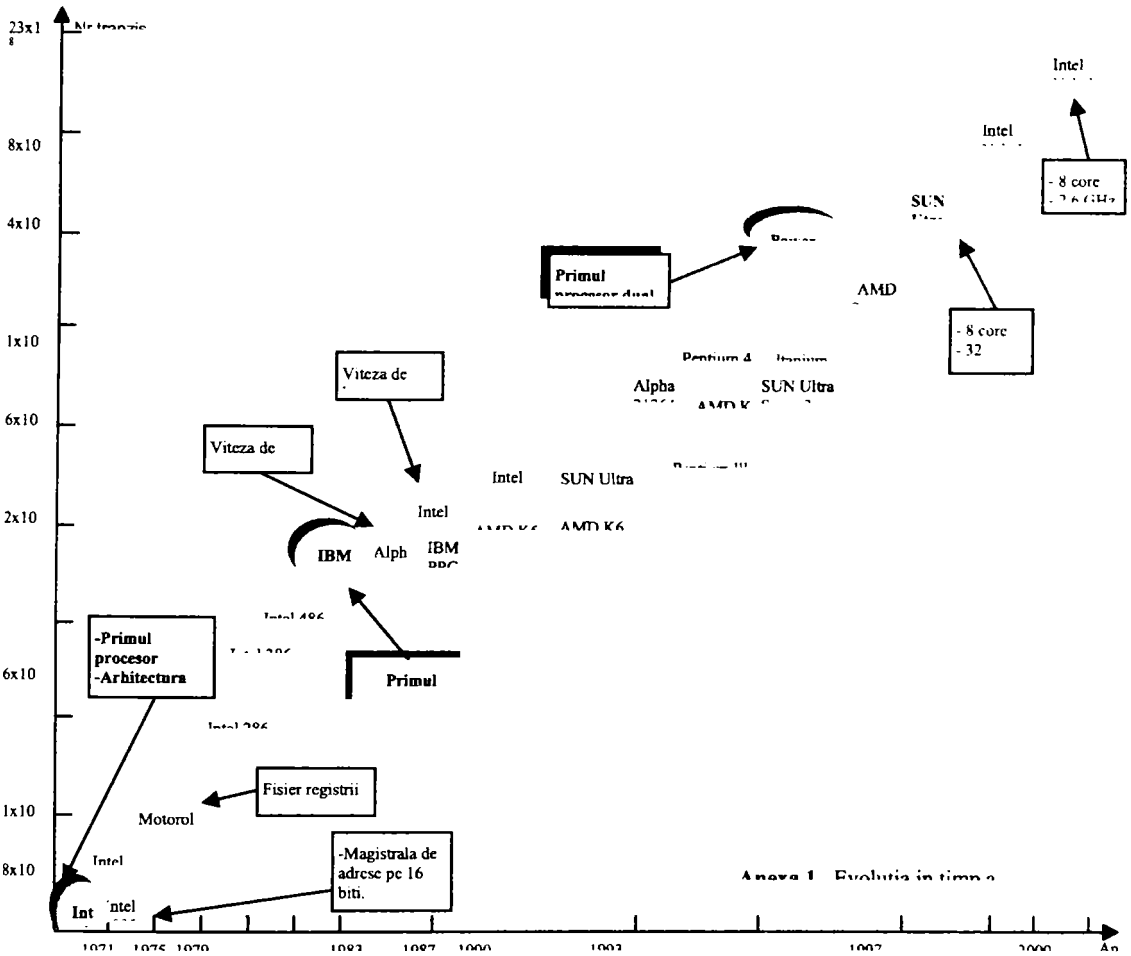
[6.4] B.Ayari, B.Kaminska „A New Dynamic Test Vector Compaction for Automatic Test Pattern Generation" *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 13, No. 3. (1994), pp. 353-358.

[6.5] F. Brglez and H. Fujiwara, *„A Neutral Netlist of 10 Combinational Benchmark Circuits",* Proc. IEEE Int'l Symp. Circuits and Systems, IEEE Press, Piscataway, N.J., 1985, pp. 695-698

[6.6] G.Z. Chrysos, J.S. Emer, *„Memory Dependence Prediction Using Store Sets"* ISCA 1998: 142-153.

Anexe:

Anexa 1



Anexa 1 Evolutia in timp a

Anexa 2

Programe utilizate pentru evaluarea performanţei microprocesoarelor

1) Programele SPEC - CPU2000. Programele SPEC - CPU2000 a fost creat pentru estimarea performanţelor microprocesorului, prin executarea unor programe ce solicită la maxim unitatea de calcul. Performanţa microprocesorului este evaluată prin măsurarea timpului total de execuţie al acestor programe de testare. Aplicaţiile componente sunt împărţite în două categorii:

1.1.) CINT 2000 – setul de programe ce operează cu întregi (Tabelul 1).

1.2.) CFP 2000 – setul de programe ce operează cu numere în virgulă mobilă (Tabelul 2).

1.1 Setul de programe CINT 2000. Este compus din 12 programe ce operează cu întregi şi operatori logici. Dintre acestea 11 sunt scrise în „C" iar unul este scris în „C++".

Tabelul 1

164.gzip	Una din versiunile popularului program de comprimare/decomprimare de date *gzip*. Acest program este scris în C. Programul executa o serie de operatiuni de comprimare/decomprimare pe o serie de fişiere de marimi de 28MBytes. Aplicâtia include fişierele executabile, un set de date generate la intimplare şi un fişier comprimat ce contine fişierele sursa. Toate operatiunile sunt executate în memorie pentru a evalua exclusiv munca executata de procesor şi de sistemul de memorie.

175.vpr	Program de proiectare circuite integrate scris în C. VPR este un program de pozitionare şi de interconectare a diferitelor parti componente ce alcâtuiesc circuitele integrate. Programul de fapt implementeaza un circuit integrat destinat unei anumite aplicaţii folosind o matrice de porti programabile. În timpul testului rezolva probleme legate de pozitionarea şi interconectarea acestor porti logice pentru a genera circuitul integrat ce ofera funcţionalitatea solicitata.
176.gcc	Compilator pentru programele scrise în C. Programul se bazeaza pe versiunea 2.7.2.2 a compilatorului gcc. Aplicaţia urmareste pasii executati de către un compilator în compilarea unui fişier sursa scris în C, cind se utilizeaza un număr mare de flag-uri de optimizare. Programul de testare executa compilarea unui set de 5 programe scrise în C de marime 3.7MBytes.
181.mcf	Program de optimizare a traficului în sistemul de transport în comun, scris în C. Aplicâtia a fost proiectata pentru planificarea frecventei vehiculelor implicâte în transportarea pasagerilor în vederea minimizarii costurilor şi pentru generarea unui orar de operare optim.
186.crafty	Program de sah scris în C. Datorita modului de execuţie neliniar acest program poate fi folosit cu succes pentru evaluarea erformanţelor predictorului instrucţiunilor conditionale în microprocesoarele moderne. Programul rezolva 5 situatii diferite pe tabla de sah variind complexitatea algoritmului de alegere a celor trei mutari ce pot constitui mutarea urmatoare.
197.parser	Program de analiză sintactica a textelor scrise în limba engleza scris în C. Programul contine un dictionar de 60000 de cuvinte şi analizeaza fraze de pina la 770 Kbytes.
252.eon	Program de vizualizare asistata de calculator scris în C++. Eon este un program de urmărire probabilistica a traiectoriei unei raze de lumina utilizat pentru crearea de imagini tridimensionale. Programul transmite imaginea unui scaun ce sta în coltul unei camere. Pentru rezolvarea problemei sunt utilizati trei algoritmi diferiti.
253.perlbmk	Varianta scurtata a versiunii v5.005_03 a popularului program de scriptare Perl. Aplicâtia este utilizata pentru rezolvarea a patru probleme diferite: 1 Conversie email în HTML. 2 Lucrul cu specdiff (parte componenta a SPEC). 3 Identificarea numerelor perfecte. Generarea unei secvente de date la intimplare.
254.gap	Program de calcul pentru algebra liniara scris în C. Gap rezolva o serie de probleme de calcul combinatoric, operatii cu grupuri de permutari şi alte probleme de algebra liniara.

255.vortex	Program de baze de date orientata pe obiecte scris în C. Programul simuleaza modul de operare a trei baze de date interconectate (lista de adrese, lista de piese de schimb şi datele geometrice). Aplicâtia a fost modificâtă pentru a reduce influenta în execuţie a sistemului de discuri în acest fel majoritatea

	operatiunilor fiind efectuate în memoria RAM. Aplicâtia este rulata de trei ori, de fiecare data se executa o serie diferita de operatii de scriere, stergere şi cautare pentru a simula diferitele tipuri de operatii ce sunt executate în bazele de date.
256.bzip2	Utilitar de comprimare date scris în C. Programul reprezinta o alta varianta de program de comprimare în care sunt utilizate o imagine, un program şi un fişier text ca intrari. Volumul total al datelor este de aproximativ 20 Mbytes.
300.twolf	Program de proiectare a circuitelor integrate. Versiunea originala a programului este utilizata pentru creerea imaginilor litografice folosite în productia de microcipuri. Aplicâtia detemina locul şi interconexiunile intre diferitele grupuri de tranzistori ce alcâtuiesc un microcip.

Tabelul 2

1.2.) Setul de programe CFP 2000. Este compus din 14 programe ce operează cu operanzi în virgulă mobila; 6 sunt scrise în „Fortran-77", 4 în „Fortran-90" iar 4 în „C".

168.wupwise	Program utilizat în fizica cuantica scris în Fortran 77. Aplicâtia rezolva una dintre ecuatiile cele mai importante în teoria interactiunilor intre particule, ecuatia Iattice-Dirac, prin metoda BICGStab.
171.swim	Program de meteorologie scris în Fortran 77. Rezolva o ecuatie diferentiala referitoare la apele de suprafata. În trecut aplicâtia a fost folosita pentru evaluarea erformanţelor super calculatoarelor.
172.mgrid	Program de rezolvare a problemelor ce necesita lucrul în planuri diferite, scris în Fortran 77. Calculeaza un cimp de potential tridimensional.
173.applu	Program de calcul folosit în dinamica fluidelor şi în chimia computationala scris în Fortran 77.
177.mesa	Librarie grafica tridimensionala scris în C
178.galgel	Program de calcul folosit în dinamica fluidelor scris în Fortran 90. Aplicâtia executa calculul numeric al parametrilor scurgerii lichidelor în spatii restrinse.
179.art	Program de simulare a retelelor neuronale,scris în C. Programul utilizeaza un sistem de retele neuronale pentru identificarea de obiecte.
183.equake	Program de simulare al propagarii undelor seismice scris în C. Programul simuleaza propagarea undelor seismice în vai neuniforme folosind metoda elementelor finite.
187.facerec	Program de procesare de imagini scris în Fortran 90. Aplicâtia este utilizata în recunoasterea fizionomiilor.
188.ammp	Program de chimie computationala scris în C. Programul

	permite modelarea moleculelor biologice mari.
189.lucas	Program utilizat în teoria numerelor scris în Fortran 90. Aplicâtia executa testul Lucas-Lehmer pentru a determina daca numerele Marsenne (2^p -1) sunt prime.
191.fma3d	Program de simulare a raspunsului la solicitari mecanice scris în Fortran 90. Aplicâtia utilizeaza metoda elementului finit pentru a simula raspunsul inelastic, tranzitoriu dinamic al solidelor şi structurilor tridimensionale supuse unor impulsuri intermitente sau unor sarcini momentane.
200.sixtrack	Program de simulare utilizat în studiul energiei nucleare de mare putere scris în Fortran 77. Aplicâtia simuleaza un accelerator de particule şi verifica variatia dinamica de exemplu a stabilitatii pe termen lung a fluxului.
301.apsi	Program de prezicere a starii vremi scris în Fortran 77. Aplicâtia calculeaza raspindirea elementelor poluante în functie de conditiile meteorologice.

2. Aplicaţii stiintifice:

2.1) HPC Challenge [http://icl.cs.utk.edu/hpcc] (Tabelul 3)

Este o suită de programe de testare care datorită modului de accesare neordonat al memoriei solicită la maxim ierarhia de memorie din microprocesoare. Cele mai importante dintre aceste aplicaţii sunt:

Tabelul 3

Linpack	program de algebra liniara ce rezolva sisteme de ecuatii liniare.
RandomAcess	program ce masoara rata aleatoare de actualizare a memoriei. (cunoscut şi sub numele de GUPS sau Giga Updates Per Second)
STREAM	program ce testeaza latimea totala a benzii de accesare a memoriei intr-un anumit sistem
FFT	program ce calculeaza Transformata Fourier rapida

2.2) NAS [www.nas.nasa.gov/Sortware/NPB] (Tabelul 4)

Aplicaţiile paralele; au fost elaborate în anii 1990 pentru a analiza scalabilitatea platformelor hardware. Aceste aplicaţii reprezintă variante la scară mai mică ale aplicaţiilor ştiintifice.

Tabelul 4.

NAS BT	aplicaţie ce efectueaza un calcul pe o grila 3D a unui sistem de ecuatii bloc-tridiagonal.
NAS CG	aplicaţie ce rezolva conjugatul gradient şi opereaza pe o matrice simetrica cu elemente pozitiv definite.

2.3) HPC, High Performance Computing (Tabelul 5) Aplicaţii ştiintifice reale a căror mod de execuţie este relativ neregulat. Cele trei aplicaţii ştiinţifice alese sunt reprezentative pentru domenii ca de exemplu fizica particulelor, chimie computaţională şi analiză structurală.

Tabelul 5

GTC	(Gyro-Kinetic Toroidal Code) aplicaţie stiintifica care simuleaza interactiunea dintre particule intr-un reactor de fuziune nucleara de tip Tokamak.
NAB	(Nucleic Acid Builder) este o aplicaţie ce permite construirea de biomolecule cu preponderenta acizi nucleici şi proteine. Aceasta aplicaţie este utilizata pentru a masura fortele ce incearca sa aduca atomii în stare de echilibru după ce particulele au fost supuse unor acceleratii initiale aleatorii.
LS Dyna	Aplicaţie ce calculeza elemente finite ce este utilizata în analizarea fenomenelor fizice neliniare ce este utilizata de exemplu în estimarea modului de deformare al diferitelor parti componente ale unei masini în urma unui accident.

3.Aplicaţii comerciale. (Tabelul 6) Au fost alese şi două aplicaţii comerciale răspândite

Tabelul 6

TPC-C	Aplicaţie proiectată pentru a studia procesul de tranzacţionare online. Este centrată în jurul unor utilizatori ce executaţii în legatură cu o bază de date.
SAP SD	Aplicaţie de *business* ce simuleaza procesarea comenzilor şi plăţilor.

Anexa 3

Tabelul 1. Tipuri de *analayzer-e.*

Nr.crt	Denumire	Comentarii
1	**Analyzer-ul generic**	Este utilizat în aplicaţii unde nu se intentionează împărţirea şirului de informaţie în categorii diferite; cum ar fi caracterizarea numărului total de *traps* întîlnite într-un fişier *trace*, sau caracterizarea numărului total de instrucţiuni citite de toate procesoarele dintr-un simulator de performanţă. Este util ca *radacină a arborelui de caracterizare*, când se intenţionează folosirea mai multor module în paralel.
2	**Analyzer-ul identificatorului de procesor**	Decodifică şirul de informaţii şi le clasifică în funcţie de identificatorul procesorului. Este utilizat în sistemele multiprocesor. În cazul acestor sisteme informaţiile vor fi separate în funcţie de identificatorul procesorului care le-a generat.
3	**Analyzer-ul *thread*-ului soft**	Clasifică şirul informaţiilor de analizat în funcţie de identificatorul *thread*-ului respectiv. Programele actuale pot avea mai multe *thread*-uri soft, ce pot executa în paralel, fiecare având un identificator unic.
4	**Analyzer-ul *user/kernel***	Decodifică şirul de instrucţiuni şi îl separă în funcţie de apartenenţa instrucţiunii: la programul utilizator, sau la instrucţiunile pe care sistemul de operare le execută, ca urmare a rulării programului utilizator. Este utilizat în cuantificarea activităţii sistemului de operare, ca urmare a execuţiei unui anumit program utilizator, sau pentru caracterizarea doar a programului utilizator, după ce activitatea sistemului de operare a fost filtrată.
5	**Analyzer-ul de context.**	Decodifică şi extrage identificatorii de context, din şirul de informaţie de analizat şi clasifică, informaţia în functie de aceşti identificatori. Este utilizat când se doreşte caracterizarea în parte a fiecarui proces ce rulează în paralel. Sistemul de operare permite mai multor programe să ruleze in paralel, atribuindu-le identificatori unici de context.
6	**Analyzer-ul segmentelor de memorie**	Imparte şirul de informaţii de analizat în funcţie de segmentul de memorie virtuală pe care acestea le apelează (*stack, heap, text, data, library,* etc.). Utilitarul permite analiza atât a segmentelor de memorie alocate programului utilizator cât şi cele ale sistemului de operare. Acest *analyzer* oferă o modalitate de a corela diferitele evenimente cum sunt "*miss*" în TLB sau în memoriile cache, cu segmentul de memorie responsabil pentru aceste evenimente. Pentru caracterizare analyzer-ul de segment citeşte un fişier de configurare ce contine mapările segmentelor de memorie ale tuturor proceselor active în sistem la un moment dat.

7	**Analyzer-ul funcţiilor**	Extrage adresele din şirul de informaţii şi face conexiunea cu funcţia din codul sursă, din care acestea provin. Utilitarul permite efectuarea acestui tip de analiză atât pentru programele utilizator cât şi pentru functiile sistemului de operare. *Analyzer*-ul permite *map*-area anumitor evenimente cum ar fi *miss* în TLB, în memoriile *cache*, sau predicţia greşită a instrucţiunilor de control condiţionale spre codul sursă responsabil pentru generarea lor. Asemeni *analyzer*-ului segmentelor de memorie acest *analyzer* citeşte un fişier de configure care conţine informaţie extrasă din tabelul de simboluri a aplicaţiei respective şi care permite *map*-area codului sursă al fiecarei funcţii, către un interval de adrese din memoria virtuală din sistemul *hardware* simulat.

Tabelul 2. Tipurile de *profiler*-e.

Nr.crt.	Destinaţia *profiler*-ului	Funcţia *profiler*-ului
1	**Frecvenţă instrucţiuni.**	Decodifică fiecare instrucţiune din sirul de informatii de analizat şi raporteaza numărul de instrucţiuni ce fac parte dintr-una din categoriile mari de instrucţiuni, cum ar fi instrucţiunile ce accesează memoria, instrucţiunile de control, instrucţiunile "*trapped*", instrucţiunile aritmetice etc.
2	**Tip de instrucţiuni.**	Decodifică fiecare instrucţiune şi raportează de câte ori a fost intâlnit un anumit tip de instrucţiune.
3	**Analiza "*trap*":**	Numară şi caracterizează fiecare tip de "*trap*" pe care îl întâlneşte în şirul de informaţii de analizat.
4	***Memory footprint.* (Analiza cantităţii de memorie accesată pentru a citi un set de instrucţiuni).**	Decodifică adresa fizică de accesare a memoriei, pentru citirea fiecarei instrucţiuni şi raportează informaţii despre cresterea în timp a cantităţii de memorie accesate, pentru a citi respectivul set de instrucţiuni. Cantitatea de memorie accesată, reprezintă numarul total de cuvinte unice accesat de o anumita secventă de instrucţiuni.
5	***Working set.* (Analiza cantităţii de memorie accesată pentru a citi datele necesare procesării unui anumit set de instrucţiuni).**	Decodifică adresa fizică de memorie accesată de instrucţiunile de citire/scriere din memorie şi raportează informaţii despre cantitatea de memorie accesată în timp, de o anumită secvenţă de instrucţiuni. Cantitatea de memorie accesată reprezintă numărul total de cuvinte unice accesat de o anumita secvenţă de instrucţiuni în "n" accesări anterioare.
6	**Mod acces memorie (Caracterizarea modului de acces a datelor din memorie).**	Decodifică fiecare instrucţiune şi raportează într-o "histogramă" distanţa dintre două instrucţiuni şi două accese de memorie consecutive.

7	**Alternarea între instrucţiunile de citire/scriere din memorie.**	Identifică fiecare instrucţiune de accesare a memoriei şi raportează într-un tabel procentajul instrucţiunilor de citire urmate de instrucţiuni de citire, procentajul instrucţiunilor de citire urmate de instrucţiuni de scriere, etc.
8	**Localizare temporală.**	Identifică fiecare operaţie de accesare a memoriei şi raportează o histogramă a numărul total de accesări ale memoriei, efectuate între două accesări succesive, ale aceleiaşi locaţii.
9	**Localizare spatială.**	Decodifică adresa fizică de accesare a memoriei pentru fiecare instrucţiune de citire/scriere şi raportează scorul localizării spaţiale prin efectuarea diferenţei între adresele de memorie ce sunt accesate de două instrucţiuni apropiate în timp.
10	**Identificarea schimbărilor de context.**	Identifică schimbările de context ce apar în urma execuţiei aplicaţiei şi raporteaza suma instrucţiunilor întâlnite în fiecare context.
11	**Contor evenimente**	Identifică şi cuantifică numărul de evenimente de un anumit tip în *pipeline*.
11	**Verificarea corectitudinii secvenţei PC-urilor.**	Asigură ca setul de instrucţiuni de analizat reprezintă o secventă legală din punct de vedere al standardului microarhitecturii procesoarelor Sparc. Acest *profiler* verifică corectitudinea PC-urilor instrucţiunilor în jurul instrucţiunilor de control conditionale, "*trap*", întreruperi etc. şi raportează erorile întâlnite. Poate fi utilizat doar interconectat la *analyzer*-ul identificatoruli de procesor.

Anexa 4

Instrucţiunile ce intră în componenţa celor cinci clase de instrucţiuni prezentate în capitolul 7.

Instrucţiuni de control		
1	**j**	Jump
2	**jal**	jump and link
3	**jr**	jump register
4	**jalr**	jump and link register
5	**beq**	branch == 0
6	**bne**	branch != 0
7	**blez**	branch <= 0
8	**bgtz**	branch > 0
9	**bltz**	branch < 0
10	**bgez**	branch >= 0
11	**bct**	brach FCC TRUE
12	**bcf**	branch FCC FALSE
Instrucţiuni acces memorie		
1	**lb**	load byte
2	**lbu**	load byte unsigned
3	**lh**	load half (short)
4	**lhu**	load half (short) unsigned
5	**lw**	load word
6	**dlw**	load double word
7	**l.s**	load single-precision FP
8	**l.d**	load double-precision FP
9	**sb**	store byte
10	**sbu**	store byte unsigned
11	**sh**	store half (short)
12	**shu**	store half (short) unsigned
13	**sw**	store word
14	**dsw**	store double word
15	**s.s**	store single-precision FP
16	**s.d**	store double-precision FP
Instrucţiuni aritmetice/logice		
1	**add**	integer add
2	**addu**	integer add unsigned
3	**sub**	integer subtract
4	**subu**	integer subtract unsigned
5	**mult**	integer multiply
6	**multu**	integer multiply unsigned
7	**div**	integer divide

8	**divu**	integer divide unsigned
9	**and**	logical AND
10	**or**	logical OR
11	**xor**	logical XOR
12	**nor**	logical NOR
13	**sll**	shift left logical
14	**srl**	shift right logical
15	**sra**	shift right arithmetic
16	**slt**	set less than
17	**sltu**	set less than unsigned
Instrucţiuni virgulă mobilă		
1	**add.s**	single-precision (SP) add
2	**add.d**	double-precision (DP) add
3	**sub.s**	SP subtract
4	**sub.d**	DP subtract
5	**mult.s**	SP multiply
6	**mult.d**	DP multiply
7	**div.s**	SP divide
8	**div.d**	DP divide
9	**abs.s**	SP absolute value
10	**abs.d**	DP absolute value
11	**neg.s**	SP negation
12	**neg.d**	DP negation
13	**sqrt.s**	SP square root
14	**sqrt.d**	DP square root
15	**cvt**	int., single, double conversion
16	**c.s**	SP compare
17	**c.d**	DP compare
Instrucţiuni speciale		
1	**nop**	no operation
2	**syscall**	system call
3	**break**	declare program error

Lista figurilor

Figura 7.2. Distribuţia instrucţiunilor. Această figură prezintă instrucţiunile ce conţin informaţie parţială obţinută în urma aplicării IE din fiecare dintre cele cinci categorii de instrucţiuni. (IE funcţionează de sine stătător)

Figura 7.3. Instrucţiuni cu date IE: În această figură sunt reprezentate numărul de instrucţiuni cu date IE raportate la numărul total de instrucţiuni din categoria respectivă. (IE funcţionează de sine stătător)

Figura 7.4. Număr de biţi IE. În această figură este reprezentată pentru fiecare instrucţiune media biţilor din regiştrii operanzi ai căror valoare a fost cunoscută în urma aplicării IE. (IE funcţionează de sine stătător)

Figura 7.5. Distribuţia instrucţiunilor**:** Această figură prezintă instrucţiunile ce conţin informaţie parţială obţinută în urma aplicării IE din fiecare dintre cele cinci categorii de instrucţiuni. (IE îi sunt retransmise rezultatele instrucţiunilor)

Figura 7.6. Instrucţiuni cu date IE**:** În această figură sunt reprezentate numărul de instrucţiuni cu date IE raportate la numărul total de instrucţiuni din categoria respectivă. (IE îi sunt retransmise rezultatele instrucţiunilor)

Figura 7.7. Număr de biţi IE**.** În figură este reprezentată pentru fiecare instrucţiune media biţilor din regiştrii operanzi ai căror valoare a fost cunoscută în urma aplicării IE. (IE îi sunt retransmise rezultatele instrucţiunilor)

Figura 7.8. Nivel de propagare. În această figură este prezentat un exemplu de interpretare a nivelului de propagare într-un graf de instrucţiuni dependente.

Figura 7.9 Distanţa de propagare **-** instrucţiuni. În această figură este prezentat, la fiecare nivel de propagare, numărul de instrucţiuni ce deţin informaţie parţială obţinută în urma aplicării IE.

Figura 7.10 Distanţa de propagare - biţi. (IE funcţionează de sine stătător)

Lista tabelèlor

Lista abrevierilor

CPU_{time} - Central Processing Unit time timpul de execuţie al unităţii centrale.

CPI - Cycles per instruction, valoarea medie a numărului de perioade de tact petrecute în procesarea unei instrucţiuni,

IPC - Instructions per clock, numărul total de instrucţiuni executate într-o perioadă de tact.

NI - Numărul total de instrucţiuni executate într-o aplicaţie.

CT - Numărul total de perioade de tact petrecute în execuţia unui program.

PC - Durata unui tact.

MFLOPS - Million floating point operations per second, milioane de instrucţiuni în virgulă mobilă pe secundă.

MIPS – Million instructions per second

FAF - Flexible Analysis Framework

TLB - Translation lookaside buffer

DTLB – Data translation lookaside buffer

UML - Unified Modeling Language

OLTP - aplicaţii comerciale de tip

HPC - High Peformance Computing

CMT - Chip Multi Threaded

PC - Program counter

SMT - Simultaneous Multithreading

CMP - Chip Multiprocessors

SST - Simultaneous Speculative Threading

IE - Implication Engine

Titluri recent publicate în colecția „TEZE DE DOCTORAT" seria 10: Știința Calculatoarelor

1. **Rodica Țirtea** – *Contribuții la îmbunătățirea dependabilității și securității informației, ISBN: 978-973-625-422-2*, (2007);

2. **Ionel Muscalagiu** – *Contribuții la implementarea, evaluarea și îmbunătățirea performanțelor tehnicilor de căutare asincrone în cadrul programării bazate pe constrângeri distribuite, ISBN: 978-973-625-592-2*, (2007);

.....

20. **Versavia Maria Ancuța** – *Problema Consensului în calculul tolerant la erori, ISBN: 978-973-625-895-4*, (2009);

21. **Răzvan Virgil Bogdan** - *A data security perspective on information transmission over distributed systems, ISBN: 978-973-625-909-8*, (2009);

22. **Cosmin Cernăzanu-Glăvan** – *Contribuții la antrenarea rețelelor neuronale. Învățarea pe baza corecției erorii cu exemple negative, ISBN: 978-973-625-979-1*, (2009);

23. **Dacian Tudor** – *A Grid Service Layer for Shared Data Programming, ISBN: 978-606-554-004-0*, (2009);

24. **Petru-Florin Mihancea** – *A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment, ISBN: 978-606-554-006-4*, (2009);

25. **Sebastian Fuicu** – *Contribuții la optimizarea resurselor în rețele locale fără fir, ISBN: 978-606-554-008-8*, (2009);

26. **Răzvan Traian Chevereșan** – Contribuții la îmbunătățirea performanței sistemelor cu microprocesor, ISBN: 978-606-554-043-9, (2010).

EDITURA POLITEHNICA

Lista completă a tezelor publicate sub sigla Editurii Politehnica poate fi consultată la adresa: **http://www.editurapolitehnica.upt.ro**