Teme de Proiect An IV Calculatoare
la disciplina **Simularea şi Optimizarea Arhitecturilor de Calcul**

A. CARACTERISTICI COMUNE

Scrieţi in simulator pentru o arhitectură superscalară parametrizabilă. Scopul principal este acela de a determina diferiţi parametri de performanţă, configuraţii optime, pentru o arhitectură Harvard de memorie (cache-uri de instrucţiuni şi date separate).

Principalii parametri ai arhitecturii sunt:

- **FR (rata de fetch)** - specifică numărul de instrucţiuni citite simultan din cache sau memorie într-un ciclu de tact; poate lua valori de 4, 8 sau 16 instrucţiuni.
- **IBS (instruction buffer size)** - dimensiunea buffer-ului de prefetch, măsurată în număr de instrucţiuni; plaja de valori: 4 (minim FR), 8, 16, 32; buffer-ul de prefetch este o coadă ce lucrează după principiul FIFO (first in first out). Vor fi citite FR instrucţiuni simultan de la adresa specificată de PC (program counter) şi depuse în partea superioară a buffer-ului. În acelaşi ciclu de execuţie, instrucţiuni din partea inferioară sunt expediate spre unităţile de decodificare şi execuţie. O intrare în buffer va conţine câmpurile:
  OPCODE - codul operaţiei executată de instrucţiunea respectivă;
  PC_crt - adresa (Program Counter-ul) instrucţiunii curente;
  DATE / INSTR - adresa din / la care se citesc / se scriu date din sau în memorie, în cazul instucţiunilor cu referire la memorie, respectiv adresa instrucţiunii ţintă în cazul instrucţiunilor de salt.
- **IRmax (issue rate maxim)** - numărul maxim de instrucţiuni, lansate în execuţie simultan într-un ciclu de execuţie, din buffer-ul de prefetch. Poate lua valorile: 2, 4, 8, 16 (maxim FR) instrucţiuni. Dacă rata de fetch este mai mică decât numărul maxim de instrucţiuni executate concurent într-un ciclu, atunci performanţa este limitată de procesul de *fetch instrucţiune*. Considerăm execuţia instrucţiunilor "*in order*" - ordinea iniţială a instrucţiunilor. O instrucţiune va fi executată abia după ce toate celelalte instrucţiuni de care ea depinde au fost executate.
- **Latenţa** - numărul de cicli necesari execuţiei instrucţiunilor aritmetice, de salt şi cele cu referire la memorie (în cazul în care accesele pentru obţinerea datei sunt cu hit în cache). Iniţial are valoarea 1.
- **Cache-ul de instrucţiuni (IC)** şi **Cache-ul de date (DC)** - sunt cache-uri mapate direct, organizate în blocuri de capacităţi parametrizabile [4, 8, 16 (maxim IBS) locaţii]. Încărcarea şi evacuarea datelor în cache se face la nivel de bloc şi nu la nivel de locaţie.
  V – bit de validare (0 – nu e valida data; 1 – valida;). Iniţial are valoarea 0. Este necesar numai pentru programe automodificabile la cache-urile de instrucţiuni. La prima înscriere în cache este setat pe 1.
  D – bit Dirty. Este necesar la scrierea în cache-ul de date (vezi pct.4).

  SIZE_IC, SIZE_DC - dimensiunea cache-urilor de instrucţiuni respectiv de date au plaja de valori de la 64 locaţii (128, 256, ...) până la 8192 locaţii.

  TAG = data *div* SIZE_D(I)C
  BLOC_OFFSET = [data *mod* SIZE_D(I)C] *div* BLOC_SIZE (FR)

  BLOC_SIZE, FR - dimensiunea în locaţii a blocului din cache-ul de date respectiv instrucţiuni;
  data – data emisă din program;
  TAG - câmp de identificare al datei;

BLOC_OFFSET - offset-ul de bloc din cache;
TagE, TagC – Tag emis din program respectiv Tag citit din cache

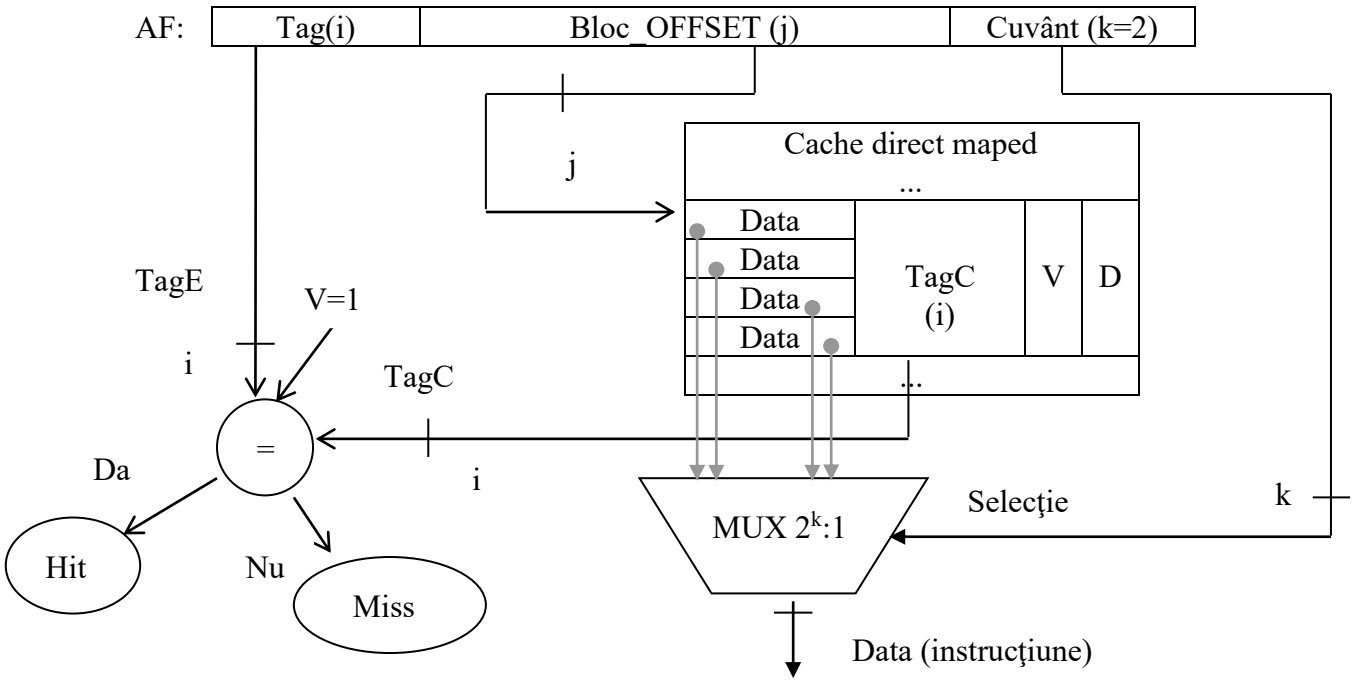

Figura 1. Structura cache-urilor de Instrucţiuni/Date

În cazul cache-urilor mapate direct, datele vor fi memorate în acelaşi loc de fiecare dată când sunt accesate. Din acest motiv vom şti la fiecare acces ce dată va fi evacuată din cache.

La o căutare în cache (IC sau DC) se ia tag-ul valorii căutate şi se verifică dacă ea există la indexul sau la offset-ul de bloc respectiv. În caz afirmativ spunem că avem acces cu HIT, altfel MISS în cache şi trebuie actualizat cache-ul.

- **Memoria principală -** (care se accesează numai la *miss* în cache) va avea o latenţă parametrizabilă de N_PEN (10, 15, 20) tacţi procesor. În cazul acceselor de date, sunt introduse penalizări numai pentru instrucţiunile LOAD (la STORE nu e nevoie din cauza procesorului de ieşire care pipelineizează scrierea în memorie, făcând-o transparentă pentru procesor). Este posibilă execuţia a două instrucţiuni cu referire la memorie de genul: Load + Load sau Load + Store.
- Presupunem existenţa unui număr suficient de mare (maxim IRmax) de **seturi de regiştrii generali**: un set de regiştrii generali este necesar pentru execuţia unei instrucţiuni de tip aritmetico-logic sau cu referire la memorie.
- Presupunem un *branch prediction* perfect, adică cunoaşterea în permanenţă a adresei corecte a următoarei instrucţiuni ce se va executa.

Programul va simula fişiere *trace* (*.trc), rulate pe arhitectura HSA (Hartfield Superscalar Architecture). Este vorba de 8 benchmark-uri Stanford, care cuprind probleme clasice de sortare, problema turnurilor din Hanoi, problema damelor, generare de permutări şi înmulţiri de matrici.

Rezultatele generate sunt sunt rata de procesare (număr de instrucţiuni raportat la număr de cicli de execuţie), rate de miss în cache-uri (IC, DC), procentul din timpul total cât buffer-ul de prefetch este gol. Se vor determina parametri optimi şi factorii de limitare în fiecare din cazuri.

CARACTERISTICI DISTINCTE

1.1. Generaţi rezultate urmate de grafice privind influenţa ratei de fetch (FR) asupra ratei de procesare IR(FR), asupra ratei de miss în cache-urile de date şi instrucţiuni $R_{missDC}(FR)$ şi $R_{missIC}(FR)$.
1.2. Studiaţi influenţa dimensiunii buffer-ului de prefetch asupra ratei de procesare IR(IBS) şi asupra procentului din timpul total cât buffer-ul de prefetch este gol IBE (instruction buffer empty) - procesorul stă, nu are instrucţiuni de executat.
1.3. Studiaţi influenţa capacităţii cache-ului de instrucţiuni asupra ratei de procesare IR(SIZE_IC) şi asupra ratei de miss la cache-ul de instrucţiuni $R_{missIC}(SIZE\_IC)$.

2.1. Determinaţi influenţa numărului maxim de instrucţiuni ce pot fi trimise simultan în execuţie asupra ratei de procesare IR(IRmax).
2.2. La acest punct nu se va mai considera număr nelimitat de seturi de regiştrii generali. Se va determina numărul optim de seturi de regiştrii (2, 3, 4, ...IRmax) în variantele cu cache de date uniport (o singură instrucţiune cu referire la memorie se poate executa) sau biport (două instrucţiuni cu referire la memorie se pot executa: L+L sau L+S).
2.3. Pentru valoarea optimă determinată la punctul 2.2. a numărului de seturi de regiştrii, studiaţi comparativ performanţa (rata de procesare) pe două tipuri de cache de date (uniport sau biport).

3.1. Se va modela şi simula un cache de instrucţiuni şi date perfect din punct de vedere al ratei de hit (100% hit). Se va dovedi o metrică ideală cu care vom compara rezultatele obţinute folosind cache-uri de instrucţiuni şi date reale (rată de hit diferită de 100%). Grafic IR(Rata de hit).
3.2. Simulaţi execuţia realistă a branch-urilor penalizând 0 respectiv 1 sau 2 cicli, la fiecare branch trimis spre execuţie. La 0 cicli penalizare avem varianta optimă de predicţie perfectă. Reprezentaţi grafic comparativ rata de procesare în cele trei situaţii.

4. Modelaţi şi simulaţi procesul de scriere în cache prin prisma celor două strategii posibile: *write back* şi *write through*. Write back e preferată în majoritatea implementărilor actuale deoarece scrierea are loc la viteza memoriei cache iar multiplele scrieri în bloc necesită doar o scriere în nivelul cel mai de jos al memoriei. Cu write through, informaţia e scrisă în ambele locuri: în blocul din cache şi în blocul din memoria principală. Prin write back informaţia e scrisă doar în blocul din cache. Write back implică evacuare efectivă a blocului - cu penalităţile de rigoare - în memoria principală. Rezultă este necesar un bit **Dirty** asociat fiecărui bloc. Starea acestui bit indică dacă blocul e *Dirty* (modificat cât timp a stat în cache), sau C*lean* (nemodificat). Dacă bitul este curat, blocul nu e scris la miss, deoarece nivelul inferior – memoria principală - are copia fidelă a informaţiei din cache. Dacă avem *citire din cache cu miss* şi *Dirty* setat pe 1 atunci vom avea o penalizare egală în timp cu timpul necesar evacuării blocului - existent în cache dar nu cel care îl solicit - la care se adaugă timpul necesar încărcării din memorie în cache a blocului necesar în continuare. La **write through** nu există evacuare de bloc la cache-urile mapate direct, dar există penalităţi la fiecare scriere în memorie. Se vor genera graficele IR(BLOC_SIZE) şi $R_{missDC}(BLOC\_SIZE)$ în cele două ipostaze: scriere în cache prin write back şi scriere în cache prin write through. Se va studia comparativ realismul, prin rata de procesare, introdus prin cele două tehnici de scriere faţă de situaţia când nu se foloseşte nici una din aceste tehnici.

5. Considerăm următoarea tehnică de îmbunătăţire a performanţei unui procesor. Aceasta constă în: presupunând că ştim adresele de acces ale tuturor instrucţiunilor Load şi Store din buffer-ul de prefetch, atunci în momentul execuţiei unui anumit Load, am putea considera executate toate

instrucţiunile Load din buffer-ul de prefetch care accesează acelaşi bloc, cu condiţia ca locaţia respectivă să nu fie alterată de un Store intercalat. Acest lucru poate fi realizat într-un mecanism Out of Order unde există buffere pentru Load-uri şi Store-uri sau într-un Trace Procesor. Folosind această tehnică, rata de procesare poate fi influenţată de doi factori: dimensiunea cache-ului de date (la un cache de capacitate redusă - mai puţine blocuri - probabilitatea de acces în acelaşi bloc fiind mai mare), şi de capacitatea buffer-ului de prefetch - într-un buffer de prefetch mai mare fiind mai probabil să găsesc mai multe instrucţiuni Load care accesează acelaşi bloc de date.

5.1. Studiaţi influenţa tehnicii prezentate, numită *multiple load*, coroborată cu dimensiunea cache-ului de date şi tehnică de scriere write back, asupra ratei de procesare.

5.2. Studiaţi influenţa tehnicii *multiple load*, coroborată cu dimensiunea buffer-ului de prefetch şi tehnică de scriere write back, asupra ratei de procesare.

5.3. Realizaţi un grafic comparativ privind rata de procesare cu şi fără *multiple load*, cu tehnică de scriere write back.

6. Implementarea unei memorii Trace Cache în cadrul arhitecturii HSA. Determinarea câştigului de performanţă faţă de un sistem care integrează doar un cache de instrucţiuni. Se va varia lungimea liniei din Trace Cache, gradul de asociativitate, dimensiunea buffer-ului de prefetch. În primă fază, se consideră predicţia perfectă a salturilor, ulterior se va simula inclusiv folosirea unui predictor realist de branch-uri (adaptiv pe două niveluri, neuronal, etc).

7. Implementarea unor algoritmi de evacuare a blocurilor din cache în vederea îmbunătățirii performanței arhitecturilor *single* și *multicore* (http://www.jilp.org/jwac-1/JWAC-1%20Program.htm). Dezvoltarea de simulatoare trace-driven pe framework-ul JWAC-1 (ISCA-2010 - http://www.jilp.org/jwac-1/) folosind trace-urile propuse. Metrici folosite (Speed-up, Rata de hit / miss).

8. Implementarea unor algoritmi de evacuare a blocurilor din cache în vederea îmbunătățirii performanței arhitecturilor *single* și *multicore* (http://crc2.ece.tamu.edu/). Dezvoltarea de simulatoare trace-driven pe framework-ul **ChampSim** (ISCA-2017 - https://github.com/ChampSim/ChampSim) folosind trace-urile propuse. Metrici folosite (Speed-up, Rata de hit / miss).

B. CARACTERISTICI COMUNE

➢ **Metodologia de simulare:**

Simularea este efectuată pe benchmark-urile Stanford, 8 fişiere trace cu extensia (*.tra). Aceste fişiere sunt o prelucrare a programelor scrise în mnemonică de asamblare (*.ins) şi a trace-urilor originale (*.trc), cu scopul de a evidenţia toate salturile (inclusiv cele care nu se fac).

Întregul fişier trace (*.trc) este o înlănţuire de triplete <***TipInstr AdrCrt AdrDest***>, unde ***TipInstr*** poate lua una din valorile ‘B’ – branch, ‘L’ – load, ‘S’ – store; ***AdrCrt*** reprezintă valoarea registrului PC – adresa instrucţiunii curente, iar ***AdrDest*** reprezintă adresa de memorie a datei accesate – în cazul instrucţiunilor cu referire la memorie (‘L’ / ‘S’) sau adresa destinaţie a saltului – în cazul instrucţiunilor de salt şi ramificaţie (‘B’).

Există totuşi o deficienţă a trace-urilor cu extensia *.trc: nu evidenţiază salturile care nu se fac. Din acest motiv s-au generat noile trace-uri (fişierele *.tra). Acestea conţin doar branch-urile (atât cele care se fac cât şi cele care nu se fac) şi exclude instrucţiunile Load / Store. Forma acestor fişiere este următoarea:

```
BT      12      30
BS      32      98
BM      100     33
NT      36      37
BRA     2       100
MOV     PC, RA  (RETURN)
```

Reprezentarea salturilor se face tot sub forma unor triplete <***TipBr AdrCrt AdrDest***>, unde ***TipBr*** se prezintă sub forma unei codificări pe două caractere, primul dintre ele indică dacă saltul se face (‘B’ – saltul se face, ‘N’ – saltul nu se face), iar al doilea caracter indică tipul saltului: ‘T’ sau ‘F’ – salturi condiţionate, ‘S’ – apeluri de tip Call, ‘M’ – apeluri de tip Return, ‘R’ – salturi necondiţionate. Alegerea acestei codificări a fost inspirată de mnemonicile întâlnite în sursa în limbaj de asamblare (BT, BF – salt condiţionat, BSR – instrucţiune de tip Call, BRA – salt necondiţionat şi MOV PC, RA – instrucţiune de tip Return).

➢ **Automatul de predicţie:**

Este descris printr-un şir de caractere cu un format mai special, ce prezintă atât numărul de stări, tranziţiile între stări cât şi predicţia aferentă fiecărei stări. Pentru o mai bună înţelegere a funcţionării automatului exemplificăm:

Automatul **BCBAADCD:12** - pe 2 biţi

Tabelul care descrie funcţionarea primului automat este următorul:

Stare Curentă	Stare următoare		Predicţie
	Pentru intrare = 0	Pentru intrare = 1	
A	B	C	0
B	B	A	0
C	A	D	1
D	C	D	1

Prima parte a şirului până la caracterul ‘:’ reprezintă tranziţiile pentru starea ‘A’ cu intrare 0, apoi cu intrare 1, apoi tranziţiile din ‘B’ pentru aceleaşi intrări, etc. Numărul din a doua parte este “văzut” în binar sub forma “0010” şi reprezintă ieşirile asociate fiecărei stări în parte (stării ‘A’ îi este asociat cel mai puţin semnificativ bit, în acest caz bitul ‘0’, următorul bit lui ‘B’, bitul ‘0’, următorul bit lui ‘C’, bitul ‘1’, etc).

CARACTERISTICI DISTINCTE

1. Să se proiecteze o structură de predicţie a salturilor de tip BTB care poate fi **complet-asociativă** (vezi figura 2) sau **mapată direct**.

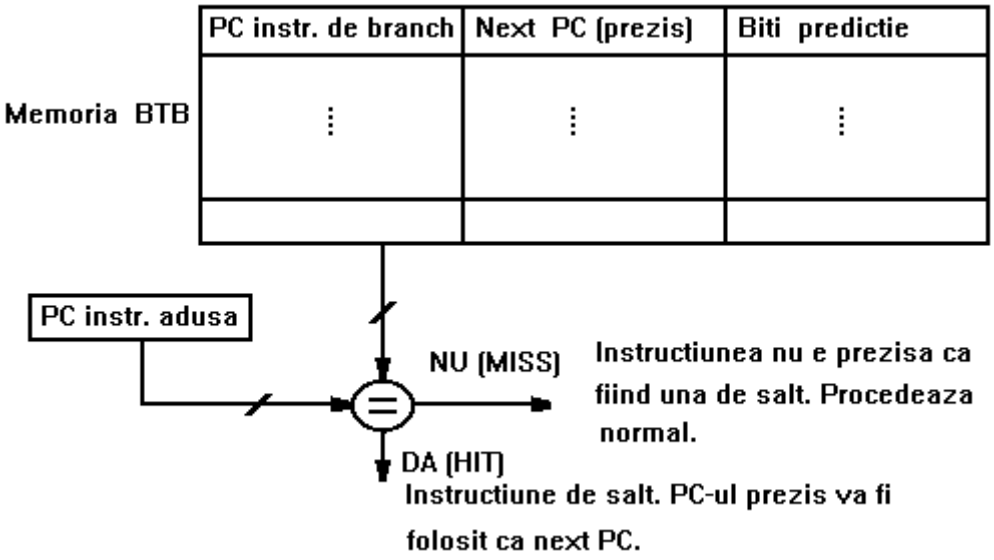

Figura 2. Structură de predicţie BTB asociativă

În cazul proiectării unui BTB complet asociativ evacuarea se face prin metoda LRU (cel mai puţin recent folosit).

În principiu, simulatorul dezvoltat va funcţiona astfel:

a) Iniţializare simulator:
- Dimensiunea tabelei (număr de intrări)
- Iniţializarea cu 0 a adreselor destinaţie
- Iniţializare stare automat de predicţie
- Tipul de arhitectură ales (mapat direct / complet asociativ)

b) Simularea propriu-zisă

Se stabileşte de către utilizator benchmark-ul de tip trace care va fi utilizat. Din acest benchmark, se citesc secvenţial instrucţiunile de ramificaţie şi se compară predicţia reală din trace cu cea propusă din tabelă. Aici pot să apară 3 cazuri distincte: predicţie corectă, predicţie incorectă, predicţie incorectă datorată exclusiv adresei de salt incorecte din tabelă. Acest ultim caz se poate datora faptului că adresa de salt din tabelă a fost modificată de exemplu de către un alt salt, având astfel un **fenomen de interferenţă al salturilor** dar pot fi şi alte cauze posibile. În final se actualizează locaţia folosită din tabela de predicţii (starea automatului de predicţie, câmpurile LRU şi nextPC).

c) Generarea de rezultate

La finele simulării propriu-zise se generează rezultate semnificative precum numărul total de salturi executate, procentajul de predicţii corecte, incorecte şi respectiv afectate de interferenţe ale salturilor. Se cere să se prezinte sub formă grafică funcţiile:
- **Ap = f (tip_arhitectură)**
- **Ap = f (dim_tabelei)**
- Să se repete rezultatele utilizând automatul de predicţie pe 1 bit: **ABAB:2**. Realizaţi grafic: **Ap = f (tip_automat)**

2. Să se implementeze o schemă de predicţie corelată pe două niveluri de tip GAg (Global History Register, Global Prediction History Table) – vezi figura 3.

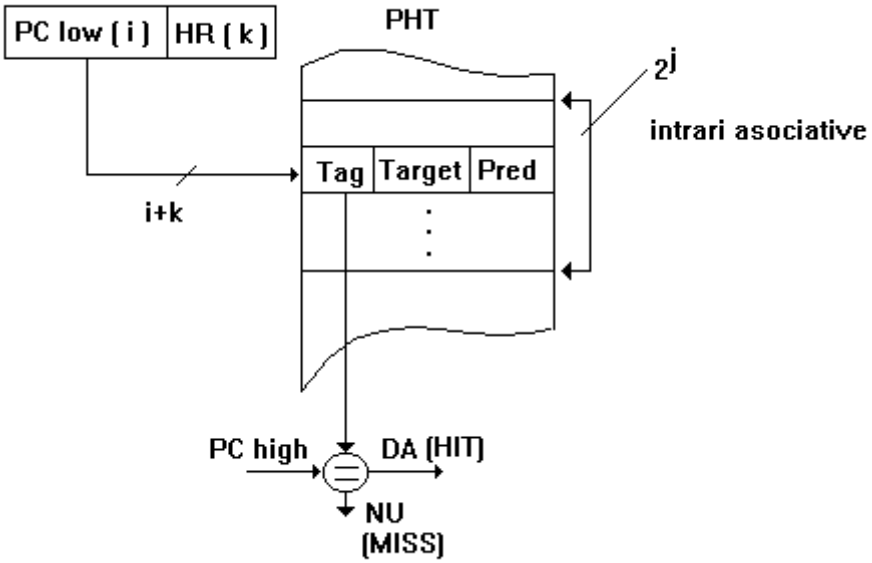

Figura 3. Structură de predicţie de tip GAg

Tabela de predicţii PHT (Prediction History Table) este adresată cu un index rezultat din concatenarea a **două informaţii ortogonale:** PClow (i biţi), semnificând gradul de localizare al saltului, respectiv registrul de predicţie (HR- History Register pe k biţi), semnificând "contextul" în care se situează saltul în program. Ambele contribuţii s-au făcut cu scopul eliminării interferenţelor branch-urilor în tabela de predicţie. Adresarea PHT exclusiv cu HR duce la serioase interferenţe (mai multe salturi pot accesa aceeaşi automat de predicţie din PHT), cu influenţe evident defavorabile asupra performanţelor. PHT poate avea diferite grade de asociativitate (iniţial considerăm tabela PHT complet asociativă). Un cuvânt din această tabelă are un format similar cu cel al cuvântului dintr-un BTB.

În cazul proiectării tabelei PHT complet asociativă evacuarea se face prin metoda LRU (cel mai puţin recent folosit).

Simulatorul dezvoltat va funcţiona astfel:

a) Iniţializare simulator:
- Iniţializarea număr biţi PClow (i = gradul de localizare al saltului)
- Iniţializarea dimensiunii registrului de deplasare HRglobal (k = gradul de corelare al saltului)
- Iniţializarea cu 0 a adreselor destinaţie
- Iniţializare stare automat de predicţie
- Tipul de arhitectură ales (mapat direct / complet asociativ)

b) Simularea propriu-zisă

Se stabileşte de către utilizator benchmark-ul de tip trace care va fi utilizat. Din acest benchmark, se citesc secvenţial instrucţiunile de ramificaţie şi se compară predicţia reală din trace cu cea propusă din tabelă. Aici pot să apară 3 cazuri distincte: predicţie corectă, predicţie incorectă, predicţie incorectă datorată exclusiv adresei de salt incorecte din tabelă. Acest ultim caz se poate datora faptului că adresa de salt din tabelă a fost modificată de exemplu de către un alt salt, având astfel un **fenomen de interferenţă al salturilor** dar pot fi şi alte cauze posibile. În final se actualizează corespunzător locaţia

folosită din tabela de predicţii (starea automatului de predicţie, câmpurile LRU şi nextPC) precum şi registrul de istorie al salturilor HRglobal.

c) Generarea de rezultate

La finele simulării propriu-zise se generează rezultate semnificative precum numărul total de salturi executate, procentajul de predicţii corecte, incorecte şi respectiv afectate de interferenţe ale salturilor. Se cere să se prezinte sub formă grafică funcţiile:

- **Ap = f (HRglobal)**
- **Ap = f (i)** ; i = size of PC_low
- **Ap = f (tip_arhitectură)**

3. Să se implementeze un modul care detectează *salturile dificil de prezis într-un anumit context*. Realizaţi un predictor adaptiv pe 2 niveluri (GAg, PAg, gshare) în care structura de predicţie va fi indexată cu informaţie de corelaţie extinsă (*path*). Determinaţi cantitativ şi calitativ în ce măsură este redus numărul de salturi dificile. Determinaţi câştigul în acurateţe faţă de schema originală (GAg nemodificată) în condiţii echivalente hardware (tabele de predicţie de aceeaşi capacitate). Funcţionarea simulatorului respectă subpunctele anterioare 2a÷2c.

4. Implementaţi un predictor contextual de tip PPM complet pentru predicţia salturilor din benchmark-urile Stanford. Comparaţi acurateţea de predicţie obţinută cu cea a predictorului GAg de la punctul 2. Studiaţi fezabilitatea unui predictor simplificat bazat pe context compus dintr-un predictor Markov de ordin maxim şi unul de ordin 0. Funcţionarea simulatorului respectă subpunctele anterioare 2a÷2c.

5. Implementarea unui predictor de salturi condiţionate de tip **perceptron** *simplu, fast path-based şi idealized piecewised*. Analiză comparativă asupra costurilor. Modelare şi implementare sub platformă academică SPEC 2000 şi industrială standardizată CBP – campionatul mondial de predicţia salturilor. Simularea prin programare distribuita a benchmark-urilor.

6. Implementarea unor predictoare de salturi condiţionate de tip *state-of-the-art*: **O-GEHL, L-TAGE, GTL**. Analiză comparativă asupra costurilor. Modelare şi implementare sub platformă academică SPEC 2000 şi industrială standardizată CBP – campionatul mondial de predicţia salturilor. Simularea prin programare distribuita a benchmark-urilor.

7. Dezvoltarea de simulatoare trace-driven pe framework-ul CBP (ISCA-2011) folosind trace-urile propuse. Metrici folosite (Acuratetea predictiei) Simularea unor predictoare de salturi de tip „*state-of-the-art*" pentru platforma Campionatului Mondial al Predictoarelor de Salturi (http://www.jilp.org/jwac-2/cbp3_framework_instructions.html, http://www.jilp.org/jwac-2/program/JWAC-2-program.htm, http://www.jilp.org/jwac-2/program/07_seznec.tgz, http://www.jilp.org/jwac-2/program/cbp3_07_seznec.pdf, http://www.jilp.org/jwac-2/program/cbp3_07_seznec.pptx, etc).

C.

1. Realizaţi un scheduller minimal care să cuprindă următoarele tehnici de rearanjare a instrucţiunilor în scopul eliminării statice a dependenţelor reale de date (RAW) şi a dezambiguizării referinţelor la memorie. Simularea se face pe trace-urile benchmark-urilor SPEC folosite de simulatorul PSATSIM sau pe fisierele (.ins, .trc) aferente benchmark-urilor Stanford.

a) Fuzionarea instrucţiunilor (*merging*)

Tehnica "*merging*" implică combinarea a două instrucţiuni într-una singură. Există trei categorii de astfel de instrucţiuni. Prima categorie, numită **MOV Merging** implică o pereche de instrucţiuni în care prima din ele este MOV. A doua categorie numită **Immediate Merging** se caracterizează prin faptul că ambele instruţiuni au ca operanzi sursă valori immediate. A treia categorie se numeşte **MOV Reabsorption** şi are ca a doua instrucţiune o instrucţiune MOV sau instrucţiunea ce se va infiltra va fi convertită la tipul primei instrucţiuni.

- **MOV Merging**

Când apare o dependenţă reală de date între o instrucţiune MOV şi o instrucţiune care încearcă să promoveze în sus în structura de cod a programului, se verifică faptul dacă cele două instrucţiuni pot fi procesate în paralel. În caz afirmativ instrucţiunea ce se infiltrează îşi continuă drumul ascendent prin basic block. În continuare vom prezenta tipurile de situaţii ce pot să apară în cadrul tehnicii MOV merging.

– Combinarea cu o **instrucţiune de adunare**.

a) Secvenţa iniţială:

```
MOV R6, R7
ADD R3, R6, R5      /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
MOV R6, R7; ADD R3, R7, R5
```

b) Secvenţa iniţială:

```
MOV R6, #4
ADD R3, R6, #5      /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
MOV R6, #4; MOV R3, #9
```

Prin înlocuirea registrului R6 cu valoarea imediată respectivă instrucţiunea de adunare devine MOV.

– Combinarea cu o **instrucţiune *store***

Secvenţa iniţială:

```
MOV R3, #0
ST (R1, R2), R3      /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
MOV R3, #0; ST (R1, R2), R0
```

Registrul R3 este înlocuit cu R0 deoarece toate procesoarele RISC au registrul R0 cablat la 0.

– Combinarea cu o **instrucţiune relaţională**

Secvenţa iniţială:

```
MOV R4, #4
```

```
GT B1, R4, R3          /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
MOV R4, #4; LTE B1 R3, #4
```

Registrul R4 este înlocuit cu valoarea imediată memorată de el şi instrucţiunea GT devine LTE pentru a permite operanzilor instrucţiunilor interschimbarea.

– Combinarea **instrucţiunilor gardate**

a) Secvenţa iniţială:

```
EQ B3, R0, R0              /* B3 := true */
TB3 ADD R10, R11, R12      /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
EQ B3, R0, R0; ADD R10, R11, R12
```

Instrucţiuni de tipul EQ Bi, R0, R0 şi NE Bi, R0, R0 sunt folosite pentru a înlocui instrucţiunile MOV Bi, #true sau MOV Bi, #false pe care arhitectura HSA nu le pune la dispoziţie. Deoarece B3 este întotdeauna *true* garda TB3 aferentă instrucţiunii de adunare poate fi înlăturată. Dacă B3 ar fi evaluată întotdeauna la *false* atunci instrucţiunea care se infiltrează va fi înlocuită cu un NOP.

b) Secvenţa iniţială:

```
MOV B1, B2
TB1 LD R4, (R0, R6)        /* instrucţiunea care se infiltrează */
```

Secvenţa combinată:

```
MOV B1, B2; TB2 LD R4, (R0, R6)
```

Pentru eliminarea dependenţelor de date, garda instrucţiunii LD devine acum B2.

c) Secvenţa iniţială:

```
MOV B1, B2
BT B1, label               /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
MOV B1, B2; BT B2, label
```

d) Secvenţa iniţială:

```
EQ B1, R0, R0
BT B1, label               /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
BRA label
```

Dacă registrul boolean care constituie gardă pentru instrucţiunea care se infiltrează are valoare constantă saltul fie va fi eliminat fie va fi transformat într-unul necondiţionat (BRA).

- **Immediate Merging**

Această tehnică implică orice pereche de instrucţiuni care au valori imediate pe post de al doilea operand sursă.

Secvenţa iniţială:

```
SUB R3, R6, #3
ADD R4, R3, #1             /* instrucţiunea care se infiltrează */
```

Secvenţa modificată:

```
SUB R3, R6, #3; ADD R4, R6, #-2
```

- **MOV Reabsorption**

Acest tip de combinare implică transformarea instrucţiunii MOV într-o instrucţiune de acelaşi tip cu prima.

Secvenţa iniţială:

```
ADD R3, R4, R5
MOV R6, R3                  /* instrucţiunea care se infiltrează */
```

Secvenţa combinată:

```
ADD R3, R4, R5; ADD R6, R4, R5
```

Ideea aflată la baza acestei metode este de a absorbi instrucţiunea MOV prin *renaming* aplicat registrului destinaţie al primei instrucţiuni reducând astfel expansiunea codului. În cazul în care prima instrucţiune este una cu referire la memorie (Load sau Store) duplicarea instrucţiunilor poate duce la reducerea performanţei datorită numărului limitat de porturi de date ale cache-ului.

b) Analiza anti-alias a referinţelor la memorie

Dependenţele de date nu apar doar între regiştri, ci şi între locaţii de memorie referite în instrucţiunile LD şi ST. La fel ca celelalte dependenţe ele provoacă deseori degradarea performanţei procesoarelor. Pentru a face distincţie între locaţiile de memorie referite de cele două tipuri de instrucţiuni, folosim tehnica numită analiză anti-alias statică a memoriei (static memory disambiguation). Pentru a decide dacă o instrucţiune LD poate fi inserată în faţa unei instrucţiuni ST, lucru ce se poate face în siguranţă doar dacă cele două adrese diferă, adresele locaţiilor de memorie sunt comparate şi este returnată una din valorile:

- **Diferit:** Adresele sunt întotdeauna diferite.
- **Identic:** Adresele sunt întotdeauna identice.
- **Eşuează:** Adresele nu se pot distinge.

Dacă valoarea returnată este “Diferit” instrucţiunea LD poate fi inserată în faţa instrucţiunii ST. De asemenea, dacă valoarea returnată este “Identic” instrucţiunea LD poate fi înlocuită cu o instrucţiune MOV ca în exemplul următor:

Secvenţa iniţială:

```
ST (R0, R5), R6
LD R10, (R0, R5)
```

Secvenţa devine:

```
MOV R10, R6
ST (R0, R5), R6
```

Pe de altă parte, în cazul în care instrucţiunea LD este urmată de o instrucţiune ST, pentru ca aceasta din urmă să poată fi infiltrată în faţa primeia **trebuie** ca cele două adrese să fie distincte întotdeauna.

Secvenţa de cod ce va fi analizată este următoarea:

```
MOV R6, R5
DIV R5, R6, #120
ASL R13, R5, #4
SUB R13, R13, R5
ADD R13, R13, #-60
ST (R0, R17), R13
ADD R17, R17, #4
ADD R18, R18, #1
LES B1, R18, #25
BT B1, L10 (#0)
ADD R19, R19, #104
LES B1, R19, #2600
BT B1, L11 (#0)
LD R18, 12(SP)
LD RA, 0(SP)
ADD SP, SP, #128
```

```
                MOV PC, RA (#0)
_Innerproduct:
                SUB SP, SP, #128
                MOV R10, R5
                MOV R13, R8
                ST (R0, R10), R0
                MOV R8, #1
                ASL R5, R13, #1
                ASL R5, R5, #2
                ADD R5, R5, R13
                ADD R13, R5, R6
                ASL R9, R9, #2
                ADD R6, R13, #4
                ADD R7, R7, #104
L16:
                ADD R13, R7, R9
                LD R5, (R0, R6)
                LD R13, (R0, R13)
                MULT R5, R5, R13
                LD R13, (R0, R10)
                ADD R13, R5, R13
                ST (R0, R10), R13
                ADD R6, R6, #4
                ADD R8, R8, #1
                LES B1, R8, #25
                BT B1, L16 (#0)
                ADD SP, SP, #128
                MOV PC, RA (#0)
```

D.

1. Generarea fișierelor *trace* pentru aplicațiile paralele din suita **PARSEC** și **SPLASH-2** folosindu-vă de simulatorul multicore **multi2sim**. Practic, în urma simulării unui benchmark Parsec cu simulatorul multi2sim se va genera pentru fiecare CORE în parte pe care a rulat benchmark-ul un fișier trace. Trace-urile rezultate vor trebui simulate pe arhitectura **SMPCache** (configurația multicore). Realizați o documentație aferentă evoluției trace-urilor considerând protocoalele de coerență **MSI / MESI** și exemplificați diferențele.

2. Implementarea unui **Automatic Design Space Explorer** aferent unei arhitecturi superscalare (Hatfield – vezi punctul **A**), pentru determinarea optimului din punct de vedere a ratei de procesare folosind algoritmi de căutare locali (în vecinătatea unei configurații microarhitecturale) de tip *Hill Climbing*, *Simulated Annealing*.

3. Implementarea unui **Automatic Design Space Explorer** aferent unei arhitecturi superscalare (vezi simulatorul PSATSim – arhitectura PowerPC), pentru analiza multiobiectiv (performanță, consum de energie) folosind tehnici de optimizare non-Pareto (suma ponderată a obiectivelor, algoritmul VEGA (*Vector Evaluated Genetic Algorithm*), ordonare lexicografică).

4. Implementarea unui **Automatic Design Space Explorer** aferent unei arhitecturi superscalare (vezi simulatorul PSATSim – arhitectura PowerPC), pentru analiza multiobiectiv (performanță, consum de energie) folosind tehnici de optimizare Pareto: NSGA-II (*Non-dominated Sorting Genetic Algorithm*) [27], SPEA2 (*Improving the strength Pareto evolutionary algorithm*) [28].

5. Implementarea unui **Automatic Design Space Explorer** aferent unei arhitecturi superscalare (vezi simulatorul PSATSim – arhitectura PowerPC) pentru identificarea arhitecturii optime din perspectivă multi-obiectiv (CPI, Energie) şi care să implementeze mecanismul de Meta-Optimizare (nivelul de meta-optimizare poate rula mai mulți algoritmi de optimizare în paralel: NSGA-II şi SPEA – abordare random şi ponderată).

Bibliografie

[1] **Chen I.K., Coffey J.T., Mudge T.** – *Analysis of Branch Prediction via Data Compression,* Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Cambridge, MA, USA, October 1996, pp. 128-137.

[2] **Florea A., Vinţan L.** – *Simularea şi optimizarea arhitecturilor de calcul în aplicaţii practice*, Editura MatrixRom, Bucureşti, 2003, ISBN 973-685-605-4.

[3] **Florea A.** – *Predicţia dinamică a valorilor în microprocesoarele generaţiei următoare*, Editura MatrixRom, Bucureşti, 2005, ISBN 973-685-980-0.
[4] **Jimenez D., Lin C.** – *Neural methods for dynamic branch prediction*. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.
[5] **Jimenez D. –** *Fast Path-Based Neural Branch Prediction*, in the Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), December 2003.
[6] **Mudge T. Chen, I., Coffey, J.** – *Limits to Branch Prediction*, Technical Report, University of Michigan, January 1996.
[7] **Nair, R.** – *Optimal 2-Bit Branch Predictors*, IEEE Transaction on Computers, No. 5, 1995.
[8] **Vinţan N. L., Florea A.** – *Microarhitecturi de procesare a informaţiei*, Editura Tehnică, Bucureşti, ISBN 973-31-1551-7, 2000 (312 pg.).
[9] **Vintan L., Gellert A., Florea A., Oancea M., Egan C.** – *Understanding Prediction Limits through Unbiased Branches*, "Lecture Notes in Computer Science", vol. 4186-0480, pp. 483-489, Springer-Verlag, ISSN 0302-9743, Berlin Heidelberg, 2006.
[10] **Florea A., Gellert A., Radu C., Calborean H., Crapciu A.,** An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture, The 6th EUROSIM Congress on Modelling and Simulation, (EUROSIM 2007), ISBN 978-3-901608-32-2, September 9-13, Ljubljana, Slovenia 2007, (special session: *Education in Simulation / Simulation in Education I*).
[11] **Gellert A., Florea A., Vintan M., Egan C., Vintan L.,** Unbiased Branches: An Open Problem, Proceedings of the Twelfth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2007), Korea, Seoul, August 2007.
[12] **Vinţan L., Egan, C.** – *Extending Correlation in Branch Prediction Schemes*, International Euromicro'99 Conference, Milano, Italy, September 1999**.**
[13] **Vinţan N. L., Florea A. –** *Branch Prediction: a Criticism and a Novel Scheme* - SINTES 10, Craiova, 2000.
[14] **Vinţan L., Florea A. -** *Investigating New Branch Prediction through Quantitative Approach* – Beyond 2000: Engineering Research Strategies, Sibiu, 1999.
[15] **http**://webspace.ulbsibiu.ro/adrian.florea/html/
[16] **http**://webspace.ulbsibiu.ro/lucian.vintan/html/
[17] Seznec A. – *The Idealistic GTL Predictor*, Journal of Instruction Level Parallelism, no. 9, May, 2007
[18] Seznec A. – *The L-TAGE Branch Predictor*, Journal of Instruction Level Parallelism, no. 9, May, 2007
[19] Seznec A. – *Analysis of the OGEHL predictor*, Proceedings of the 32th International Symposium on Computer Architecture (IEEE-ACM), Madison, June 2005.
[20] http://arco.unex.es/smpcache/#MemoryTraces
[21] http://parsec.cs.princeton.edu/download.htm
[22] http://www.multi2sim.org/wiki/index.php5/Main_Page
[23] http://www.jilp.org/jwac-1/JWAC-1%20Program.htm
[24] http://www.jilp.org/jwac-2/program/JWAC-2-program.htm
[25] http://www.crhc.illinois.edu/TechReports/2007reports/magellan.pdf
[26] http://www.autonlab.org/tutorials/hillclimb02.pdf
[27] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182-197, 2002.
[28] E. Zitzler, M. Laumanns, L. Thiele, and others, "SPEA2: Improving the strength Pareto evolutionary algorithm," in *Eurogen*, pp. 95–100, 2001.
[29] Chiş Radu, *Developing Effective Multi-Objective Optimization Methods for Complex Computing Systems*, PhD Thesis, Lucian Blaga University of Sibiu, September 2017.