UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE "HERMANN OBERTH"
CATEDRA DE CALCULATOARE ŞI AUTOMATIZĂRI

PROIECT DE DIPLOMĂ

Conducător ştiinţific : Prof. dr. ing. Vinţan Lucian
Îndrumător: Conf. dr. ing. Florea Adrian

Absolvent:
Radu Ciprian Vasile
Specializarea CALCULATOARE

- Sibiu, 2008 -

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE "HERMANN OBERTH"
CATEDRA DE CALCULATOARE ŞI AUTOMATIZĂRI

IMPLEMENTAREA UNUI SISTEM MULTI-CORE CU MEMORIE PARTAJATĂ FOLOSIND MEDIUL UNISIM

Conducător ştiinţific : Prof. dr. ing. Vinţan Lucian
Îndrumător: Conf. dr. ing. Florea Adrian

Absolvent:
Radu Ciprian Vasile
Specializarea CALCULATOARE

Cuprins

1. Introducere în Multicore..3
    1.1. Legea lui Amdahl şi legea lui Gustafson 5
    1.2. Dependenţe 6
    1.3. „Race conditions”, excluziune mutuală, sincronizare şi încetinire paralelă 7
    1.4. Modele de consistenţă 8
    1.5. Taxonomia lui Flynn 9
    1.6. Tipuri de paralelism 10
        1.6.1. Paralelism la nivel de bit 10
        1.6.2. Paralelism la nivel de instrucţiuni 11
        1.6.3. Paralelism la nivel de date 12
        1.6.4. Paralelism la nivel de taskuri 12
    1.7. Hardware..13
        1.7.1. Memorie şi comunicare 13
        1.7.2. Clase de calculatoare paralele 14
            1.7.2.1. Calculatoare Multicore..14
            1.7.2.2. Multiprocesoare simetrice (SMP - Symmetric multiprocessing) 14
            1.7.2.3. Calculatoare distribuite..15
            1.7.2.4. Arhitecturi tip cluster...15
            1.7.2.5. Arhitecturi cu procesare paralelă masivă (Massive parallel processing) 15
            1.7.2.6. Grid computing...16
            1.7.2.7. Calculatoare paralele specializate...16
    1.8. Software...16
        1.8.1. Paralelizarea automată..17
        1.8.2. Aplicaţii 17
2. Arhitecturi multiprocesor cu memorie partajată..18
    2.1. Coerenţa memoriilor cache..19
        2.1.1. Protocolul MSI...23
        2.1.2. Protocolul MESI (Illinois)..26
        2.1.3. Protocolul MOSI...27
        2.1.4. Protocolul MOESI...28
    2.2. Modele de consistenţă a memoriei..30
    2.3. Sincronizarea proceselor..32
        2.3.1. Atomizări şi sincronizări..33
        2.3.2. Sincronizarea la barieră...34
3. Introducere în modelarea la nivel de tranzacţii...36
    3.1. Designul clasic al SoC...37
    3.2. Modelarea la nivel de tranzacţii..39
    3.3. TLM – metoda optimă de modelare a sistemelor on-chip (SoC) complexe.......................41
    3.4. Principii TLM...42
        3.4.1. Terminologie..42
        3.4.2. Procesul de modelare...43
        3.4.3. Acurateţea modelului...43
    3.5. Metodologia UNISIM pentru Modelarea la Nivel de Tranzacţii.......................................47
        3.5.1. Untimed TLM (UTLM) în mediul UNISIM...50
        3.5.2. Timed TLM (TTLM) în mediul UNISIM...51
    3.6. Utilizarea TLM în cadrul simulatorului..52
    3.7. Emularea Sistemului de Operare cu UNISIM...54
4. Simulare la nivel de tranzacţii cu UNISIM...59

4.1. Componente ale simulatorului.....61
4.2. Dezvoltarea simulatorului PowerPC-TLM.....70
4.2.1. Parametrii noului simulator PowerPC-TLM.....77
5. Simularea unei arhitecturi multicore cu memorie partajată.....79
5.1. Benchmarkuri.....79
5.2. Rezultate ale simulărilor.....81
5.3. Concluzii şi dezvoltări ulterioare.....87
Bibliografie.....90
ANEXA.....93
Instalarea simulatorului TLM PowerPC755 (snapshot august 2007).....93

1. Introducere în Multicore

„Old [conventional wisdom]:
Increasing clock frequency is the primary method of improving processor performance.
New [conventional wisdom]:
Increasing parallelism is the primary method of improving processor performance.
[...]
Even representatives from Intel, a company generally
associated with the 'higher clock-speed is better' position,
warned that traditional approaches to maximizing performance
through maximizing clock speed have been pushed to their limit."

Krste Asanovic et al.
The Landscape of Parallel Computing Research: A View from Berkeley
University of California, Berkeley. Technical Report No. UCB/EECS-2006-183
18 Decembrie 2006

Calculul paralel este o formă de calcul în care multe instrucțiuni sunt efectuate simultan, care funcționează pe principiul că de multe ori problemele mari pot fi împărțite în unele mai mici, care sunt apoi rezolvate concomitent ("în paralel"). Există mai multe forme diferite de calcul paralel: paralelism la nivel de bit, la nivel de instrucțiune, paralelism de date, paralelism la nivel de micro-threads şi threads şi paralelism la nivel de task. Calculul paralel este folosit de mai mulți ani, în principal în calcul de înaltă performanță, dar interesul pentru acesta a crescut în ultimii ani datorită constrângerilor fizice care previn scalarea frecvenței de procesare[1]. Calculul paralel a devenit paradigma dominantă în arhitectura calculatoarelor, în principal sub formă de procesoare multicore [Asa06]. Cu toate acestea, în ultimii ani, consumul de putere, cauzat de astfel de arhitecturi paralele, a devenit un motiv de îngrijorare [Asa06].

Calculatoarele paralele pot fi clasificate în funcție de nivelul la care hardware-ul sprijină paralelismul.

Programele paralele sunt mai dificil de scris decât cele secvențiale [Hen02], deoarece concurența introduce câteva noi clase de potențiale „buguri" software, din care „race conditions" (vor fi tratate pe parcursul acestei introduceri), sunt cele mai comune. Comunicarea şi sincronizarea între diferitele subtaskuri este de obicei una dintre cele mai mari bariere în calea obținerii unei bune performanțe a programului paralel. Accelerarea apărută prin paralelizarea unui program este dată de legea lui Amdahl.

1 Tehnică, folosită în arhitectura calculatoarelor, prin care frecvența de lucru a procesorului este crescută în vederea obținerii de creşteri de performanță

Din păcate însă, încă se merge pe procesarea secvențială. Pentru a rezolva o problemă, este construit un algoritm care produce un flux serial de instrucțiuni. Aceste instrucțiuni sunt executate pe o unitate centrală de prelucrare, pe un calculator. Doar o singură instrucțiune se execută la un moment dat. După ce o instrucțiune se execută, urmează a fi executată următoarea şi aşa mai departe.

Calculul paralel, pe de altă parte, utilizează simultan mai multe elemente de prelucrare pentru a rezolva o problemă. Acest lucru este realizat prin spargerea problemei în părți independente, astfel încât fiecare element de prelucrare a acestuia să poate executa o parte din algoritm simultan. Elementele de procesare pot fi diverse şi includ resurse, cum ar fi un singur computer cu mai multe procesoare, mai multe calculatoare aflate în rețea, hardware-ul specializat, sau orice combinație de mai sus.

Scalarea frecvenței a fost motivul dominant pentru îmbunătățiri de performanță în computerul de la mijlocul anilor 1980 până în 2004. Timpul de rulare a unui program este egal cu numărul de instrucțiuni, înmulțit cu durata medie de execuție per instrucțiune. Menținând toți ceilalți parametrii constanți, la creşterea frecvenței de tact scade timpul mediu necesar pentru a executa o instrucțiune. Astfel, o creştere în frecvență, scade timpul de rulare pentru toate programele computaționale [Hen02].

Consumul de putere dinamică al unui chip este dat de ecuația:

$$P = C \cdot V^2 \cdot F$$

unde P este puterea, C este reactanța capacitivă per ciclu de ceas (proporțională cu numărul de tranzistori pentru care semnalele de intrare se schimbă), V este tensiunea şi F este frecvența procesorului (cicli pe secundă) [Rab96]. Deci, creşteri ale frecvenței de procesare duc la creşteri ale puterii utilizate într-un procesor. Creşterea consumului de putere de către procesor a condus în cele din urmă la acțiunea companiei Intel (mai 2004) de anulare a procesoarelor Tejas şi Jayhawk. Acest moment este, în general, citat ca sfârşitul perioadei în care scalarea frecvenței era paradigma dominantă în arhitectura calculatoarelor [Fly04].

Legea lui Moore este o observație empirică a faptului că densitatea de tranzistori într-un microprocesor sau memorie DRAM se dublează la fiecare 18 până la 24 de luni. În ciuda problemelor consumului de energie[2], precum şi a previziunilor că această lege nu va mai fi valabilă, legea lui Gordon Moore este încă în vigoare. Odată cu sfârşitul metodei de scalare a frecvenței, tranzistorii suplimentari (care nu mai sunt utilizați pentru scalarea frecvenței) pot fi folosiți pentru a

2 Energia reprezintă produsul dintre putere si timpul în care ea se consumă: E = Pt. Însă, chiar dacă puterea consumată creşte, consumul de energie se poate reduce, cu condiția ca timpul în care acea putere s-a consumat să scadă.

adăuga hardware de calcul paralel.

1.1. Legea lui Amdahl şi legea lui Gustafson

Teoretic, accelerația introdusă de paralelizare ar trebui să fie liniară, dublarea numărului de elemente de procesare ar trebui să înjumătățească timpul de rulare. Cu toate acestea, foarte puțini algoritmi paraleli, ating această accelerație optimă. Cei mai mulți au o accelerație liniară pentru un număr mic de elemente de procesare, care devine constantă pentru un număr mare (de elemente de procesare).

Potențialul de speedup (accelerare) al unui algoritm paralel pe o platformă de calcul este dat de legea lui Amdahl, formulată inițial de către Gene Amdahl, în anii 1960 [Amd67]. Această lege afirmă că o mică parte a programului, care nu poate fi paralelizată va limita accelerația globală care poate fi obținută din restul programului, din partea paralelizabilă a acestuia. Orice probleme matematice sau de inginerie suficient de complexe, de obicei, vor consta din mai multe părți, unele paralelizabile, iar altele secvențiale. Accelerarea S pentru un sistem cu N procesoare este, prin definiție:

$S = \frac{T_S}{T_N}$, unde:

T_s = timpul de execuție pentru cel mai rapid algoritm secvențial care rezolvă problema pe un monoprocesor;

T_N = timpul de execuție al algoritmului paralel executat pe un sistem multiprocesor, cu N microprocesoare.

Dacă notăm cu f fracția (procentajul) din algoritm care are un caracter eminamente secvențial, $f \in [0,1]$, putem scrie:

$$T_N = f \cdot T_S + \frac{(1-f) \cdot T_S}{N}$$

adică,

$$S = \frac{T_S}{f \cdot T_S + \frac{(1-f) \cdot T_S}{N}}$$

sau: $$S=\frac{1}{f+\frac{1-f}{N}}$$ Legea lui G. Amdahl, $1 \leq S \leq N$

Această lege sugerează că un procentaj (f) oricât de scăzut de calcule secvențiale impune o limită superioară a accelerării ($\frac{1}{f}$) care poate fi obținută pentru un anumit algoritm paralel, pe un sistem multiprocesor, indiferent de numărul N al procesoarelor din sistem şi topologia de interconectare a acestora [Vin00].

Legea lui Gustafon este o altă lege în ingineria calculatoarelor, strâns legată de legea lui Amdahl. Ea poate fi formulată astfel:

$$S(P)=P-\alpha\cdot(P-1)$$

unde P este numărul de procesoare, S este accelerația şi α este partea neparalelizabilă a programului [ACM88]. Legea lui Amdahl presupune o dimensiune fixă a problemei şi faptul că mărimea părții secvențiale este independentă de numărul de procesoare. Legea lui Gustafson nu face aceste ipoteze.

1.2. Dependenţe

Înțelegerea dependențelor de date este fundamentală în punerea în aplicare a algoritmilor paraleli. Nici un program nu poate rula mult mai rapid decât cel mai lung lanț de dependențe de calcule (cunoscut sub numele de „cale critică”), deoarece calculele care depind de calcule prealabile din lanț trebuie să fie executate înainte. Cu toate acestea, cei mai mulți algoritmi nu constau doar dintr-un singur lanț de instrucțiuni dependente; sunt, de obicei, posibilități de a executa independent instrucțiuni, în paralel.

Fie P_i şi P_j două fragmente de program. Condițiile lui Bernstein's [Ber66] descriu când cele două porțiuni de program sunt independente şi pot fi executate în paralel. Fie I_i toate variabilele de intrare pentru P_i şi O_i toate variabilele de ieşire, de asemenea, şi pentru P_j vom avea I_j şi respectiv O_j. Considerând că P_j urmează lui P_i , spunem că P_i şi P_j sunt independente, dacă acestea îndeplinesc următoarele condiții:

- $I_j \cap O_i = \varnothing$
- $I_i \cap O_j = \varnothing$

- $O_i \cap O_j = \emptyset$

Încălcarea primei condiții introduce aşa numită dependență Read After Write (RAW sau flow dependence), care presupune că prima declarație produce un rezultat utilizat de către cea de-a doua declarație. Cea de-a doua condiție constituie o anti-dependență (WAR – Write After Read) şi înseamnă că prima declarație suprascrie o variabilă pe care o foloseşte cea de-a doua expresie. Cea de-a treia şi ultima condiție, marchează dependența de ieşire (WAW – Write After Write). Când două variabile scriu în aceeaşi locație, rezultatul final este dat de cea de-a doua declarație [Roo00].

1.3. „Race conditions”, excluziune mutuală, sincronizare şi încetinire paralelă

Subtaskurile unui program paralel sunt adesea numite fire de execuție. Unele arhitecturi de calculatoare utilizează versiuni mai mici de fire cunoscute sub denumirea de fibre, în timp ce alte versiuni utilizează versiuni mai mari cunoscute sub numele procese. Cu toate acestea, "firele" de execuție (threads) reprezintă un termen generic pentru subtaskuri. De multe ori firele vor trebui să-şi actualizeze unele variabile care sunt partajate între ele. Instrucțiunile între cele două programe pot fi întrețesute în orice ordine. De exemplu, luați în considerare următorul program:

Firul A	Firul B
1A: Citeşte în variabila V locația de memorie X	1B: Citeşte variabila V locația de memorie X
2A: V = V + 1	2B: V = V + 1
3A: Scrie variabila V înapoi în X	3B: Scrie variabila V înapoi în X

Dacă instrucțiunea 1B este executată între 1A şi 3A, sau în cazul 1A este executată între 1B şi 3B, programul va produce date incorecte. Această problemă este cunoscută ca „race condition”. Programatorul trebuie să utilizeze un mecanism de blocare (lock) pentru a crea o excluziune mutuală. O blocare (un lock) este o construcție a limbajului de programare care permite unui fir de a lua controlul asupra unei variabile şi de a nu permite altor fire de a citi sau scrie acea variabilă, până când nu este deblocată. Firul care deține blocarea este liber să execute secțiunea sa critică (secțiunea de program care necesită acces exclusiv la unele variabile) şi apoi va debloca datele, adică atunci când această secțiune critică se încheie. Prin urmare, pentru a garanta executarea corectă a programului, programul de mai sus poate fi rescris pentru a utiliza mecanismul de blocare:

Firul A	Firul B
1A: Blochează (Lock) variabila globală V	1B: Blochează (Lock) variabila globală V

2A: Citeşte în variabila V locația de memorie X	2B: Citeşte în variabila V locația de memorie X
3A: V = V + 1	3B: V = V + 1
4A: Scrie variabila V înapoi în X	4B: Scrie variabila V înapoi în X
5A: Deblochează (Unlock) variabila V	5B: Deblochează (Unlock) variabila V

Un fir va putea bloca variabila V, în timp ce alte fire nu vor putea bloca acea variabilă V până când ea nu va fi deblocată. Acest lucru garantează executarea corectă a programului. Blocările, sunt necesare pentru a asigura o execuție corectă a programului, dar pot încetini foarte mult un program.

Blocarea de mai multe variabile folosind blocări non-atomice creează posibilitatea de apariție a interblocărilor (deadlock). O blocare atomică (indivizibilă) blochează mai multe variabile, toate în acelaşi timp. Dacă o variabilă nu poate fi blocată, atunci nu se va mai bloca niciuna dintre ele. Dacă două fire au nevoie de a bloca aceleaşi două variabile şi utilizează operații non-atomice, este posibil ca un fir să blocheze una dintre ele şi cel de-al doilea fir să o blocheze pe cea de-a doua variabilă. Într-un astfel de caz, nici un fir nu poate finaliza şi rezultă un deadlock.

Multe programe paralele necesită ca subtaskurile lor să acționeze sincron. Acest lucru necesită utilizarea de bariere. Barierele sunt, de obicei, implementate folosind mecanismele software de blocare. O clasă de algoritmi, fără blocare (lock - unlock) şi fără aşteptare (bariere), există şi nu necesită bariere şi mecanisme de blocare. Cu toate acestea, această abordare este, în general, dificil de pus în aplicare în mod corect şi necesită structuri de date adecvate, care să asigure o proiectare adecvată.

Nu toate programele paralelizate duc la o creştere a vitezei de execuție, comparativ cu varianta lor serială. In general, când un program este împărțit în mai multe fire de execuție, aceste fire pot petrece o bună parte din timp comunicând unele cu altele. În cele din urmă, încărcarea produsă de comunicare domină timpul petrecut pentru rezolvarea problemei şi timpul de execuție al aplicației paralele creşte, în loc să scadă (comparativ cu timpul de execuție în varianta neparalelă).

În funcție de cât de des trebuie firele de execuție să se sincronizeze sau să comunice unele cu altele, există paralelism cu granularitate fină (fine-grained) - multe comunicații pe secundă, cu granularitate medie (coarse-grained) – nu comunică de multe ori pe secundă şi paralelism cu granularitate mare (embarrassing parralelism) – atunci când se comunică foarte rar sau chiar deloc. Evident, ultima categorie este cea mai convenabilă pentru că presupune foarte mult paralelism.

1.4. Modele de consistenţă

Leslie Lamport a definit pentru prima dată conceptul de consistență secvențială.

Limbajele de programare paralelă şi calculatoarele paralele trebuie să aibă un model de consistență (de asemenea cunoscut ca şi model de memorie). Modelul de consistență defineşte regulile după care au loc operațiile la nivelul memoriei şi felul în care rezultatele sunt produse.

Unul dintre primele modele de consistență a fost cel secvențial al lui Leslie Lamport. Consistența secvențială este proprietatea unui program paralel care asigură că execuția sa paralelă produce aceleaşi rezultate ca şi un program secvențial. Mai exact, un program este consistent secvențial, dacă „... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”[3] [Lam79].

Un model de consistență des întâlnit la nivel software este cel al memoriei tranzacționale. Acest model împrumută din teoria bazelor de date conceptul de tranzacții atomice, pe care îl aplică acceselor la memorie.

Matematic, aceste modele pot fi reprezentate în mai multe moduri. Rețelele lui Petri, care au fost introduse în teza de doctorat a lui Carl Adam Petri în 1962, au fost o încercare timpurie de a codifica regulile modelelor de consistență. Teoria fluxului de date (Dataflow theory) a fost construită mai târziu pe baza acestor rețele, iar arhitecturile orientate pe flux de date (Dataflow architectures[4]), au fost create pentru a pune în aplicare ideile acestei teorii. Modele logice, cum ar fi Logica Temporală[5] a Acțiunilor (TLA – Temporal Logic of Actions[6]) a lui Lamport şi diagrame de evenimente, au fost dezvoltate pentru a descrie comportamentul sistemelor concurente.

1.5. Taxonomia lui Flynn

Michael J. Flynn a creat unul dintre cele mai timpurii sisteme de clasificare pentru calculatoare şi programe paralele (şi secvențiale), acum cunoscut ca Taxonomia lui Flynn. Flynn a clasificat

3 ... rezultatele produse de orice execuție paralelă sunt aceleaşi ca în cazul în care toate operațiile ale tuturor procesoarelor au fost executate într-o ordine secvențială, precum şi dacă operațiile asociate fiecărui procesor apar în această secvență, în ordinea specificată prin program (tr. n.)

4 Arhitecturi de calcul care contrastează în mod direct tradiționala arhitectură a lui von Neumann. Aceste arhitecturi nu au un Program Counter (contor de program) sau, cel puțin teoretic, execuția unei instrucțiuni este exclusiv determinată pe baza disponibilității parametrilor de intrare ai instrucțiunilor. Deşi până acum nu s-a înregistrat niciun succes comercial la nivel hardware de astfel de arhitecturi, ele sunt foarte relevante în multe arhitecturi software actuale, precum Sisteme de Gestiune a Bazelor de Date şi frameworkuri (cadre) de calcul paralel

5 Termenul de logică temporală este folosit pentru a descrie orice sistem de reguli şi simboluri pentru a reprezenta şi raționa cu propoziții exprimate în termeni temporali, care au legătură cu timpul

6 Model logic care combină logica temporală cu logica acțiunilor, scopul fiind acela de a descrie comportamentul sistemelor concurente

calculatoarele şi programele, după câte seturi de instrucţiuni şi date folosesc.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Fig. 1: Taxonomia lui Flynn

Clasificarea single-instruction-single-data (SISD) este echivalentă cu un program complet secvenţial. Single-instruction-multiple-data (SIMD) este similară cu a face aceeaşi operaţiune, în mod repetat, pe un set mare de date (spre exemplu aplicaţiile de procesare a semnalelor). Multiple-instruction-single-data (MISD) este folosit foarte rar. În timp ce arhitecturi de calculatoare de acest fel au fost concepute (cum ar fi reţelele sistolice), s-au materializat foarte puţine aplicaţii care să se potrivescă cu această clasă. Multiple-instruction-multiple-data (MIMD) sunt de departe cele mai frecvente tipuri de programe paralele. În cadrul SIMD s-au remarcat calculatoarele vectoriale, dar şi în cazul arhitecturilor de calcul, cele mai des întâlnite sunt cele din categoria MIMD.

Potrivit David A. Patterson şi John L. Hennessy, „Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."[7][Hen02].

1.6. Tipuri de paralelism

1.6.1. Paralelism la nivel de bit

De la apariţia tehnologiei VLSI (Very Large Scale Integration) de fabricaţie a chipurilor, în anii 1970, până în aproximativ 1986, arhitecturile de calculatoare au evoluat prin dublarea dimensiunii cuvântului (câtă informaţie poate un procesor să execute pe ciclu) [Cul99]. Creşterea dimensiunii cuvântului reduce numărul de instrucţiuni maşină care trebuie executate pentru a efectua operaţii pe variabile ale căror dimensiuni sunt mai mari decât lungimea cuvântului. Istoric, microprocesoarele pe 4 biţi au fost înlocuite cu cele pe 8 biţi, apoi pe 16 biţi, apoi pe 32 de biţi. Această tendinţă a fost marcată de introducerea de procesoare pe 32 de biţi, care a fost un standard pentru arhitecturile de calcul, pentru două decenii. În anii 2003 - 2004, odată cu arhitecturile x86-64, procesoarele pe 64 de biţi au devenit un lucru obişnuit.

7 Unele maşini sunt, desigur, hibrizi ale acestor categorii, însă acest model clasic a supravieţuit, pentru că este simplu, uşor de înţeles, şi oferă o bună aproximare la prima vedere. De asemenea, este - probabil din cauza uşurinţei de a fi înţeles - cel mai des folosit de sistem. (tr. n.)

1.6.2. Paralelism la nivel de instrucţiuni

O maşină RISC[8] cu cinci faze pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Write back) poate fi considerată reprezentativă pentru paralelismul la nivel de instrucțiuni (ILP – Instruction Level Parallelism). Un program este, în esență, un flux de instrucțiuni ce trebuie executate de către un procesor. Aceste instrucțiuni pot fi reordonate şi combinate în grupuri care apoi sunt executate în paralel, fără a schimba rezultatul programului. Acest lucru este cunoscut sub denumirea de paralelism la nivel de instrucțiune. Progresul în domeniul ILP a dominat arhitecturile de calculator de la mijlocul anilor 1980 până la mijlocul anilor 1990 [Cul99].

Toate procesoarele moderne au mai multe faze de procesare pipeline. Fiecare fază pipeline corespunde unei acțiuni diferite pe care procesorul o efectuează asupra unei instrucțiuni. Cu alte cuvinte, un procesor pipeline cu N faze poate avea până la N instrucțiuni aflate în diferite faze de execuție. Aşa cum am precizat mai sus, procesorul pipeline canonic este cu 5 faze de execuție dar acest lucru nu este fix: Pentium 4 avea 35 de faze de execuție [Pat04].

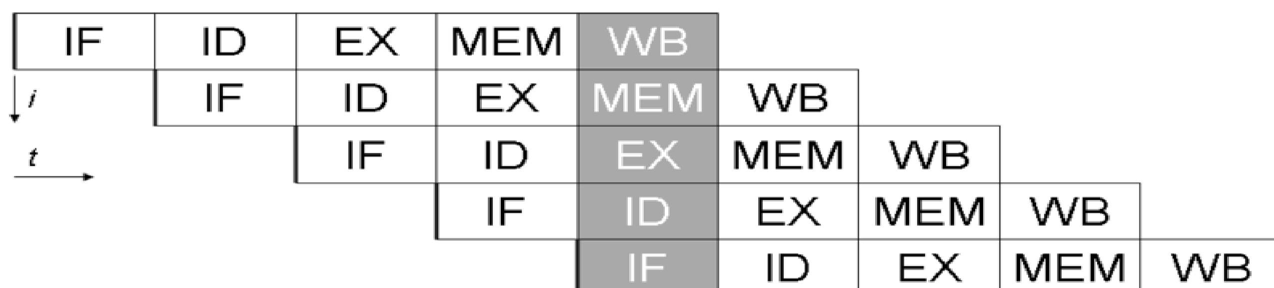


Fig. 2: Pipeline cu 5 faze

În plus față de paralelismul la nivel de instrucțiune adus de pipeline-uri, unele procesoare pot trimite spre execuție mai mult de o instrucțiune la un moment dat. Acestea sunt cunoscute ca procesoare superscalare. Instrucțiunile pot fi grupate împreună numai dacă nu există nici o dependență de date între ele. Algoritmul lui Tomasulo este unul dintre cei mai cunoscuți şi utilizați algoritmi care permit exploatarea ILP şi execuția out-of-order.

8 Reduced Instruction Set Computer – arhitectură de microprocesor care se bazează pe un set redus de instrucțiuni, mai simple, mai rapide deci, tocmai pentru a oferi o performanță mai mare. Există mai multe propuneri pentru o definiție precisă a RISC, dar acest termen este încet înlocuit de unul mai descriptiv: arhitectură load-store

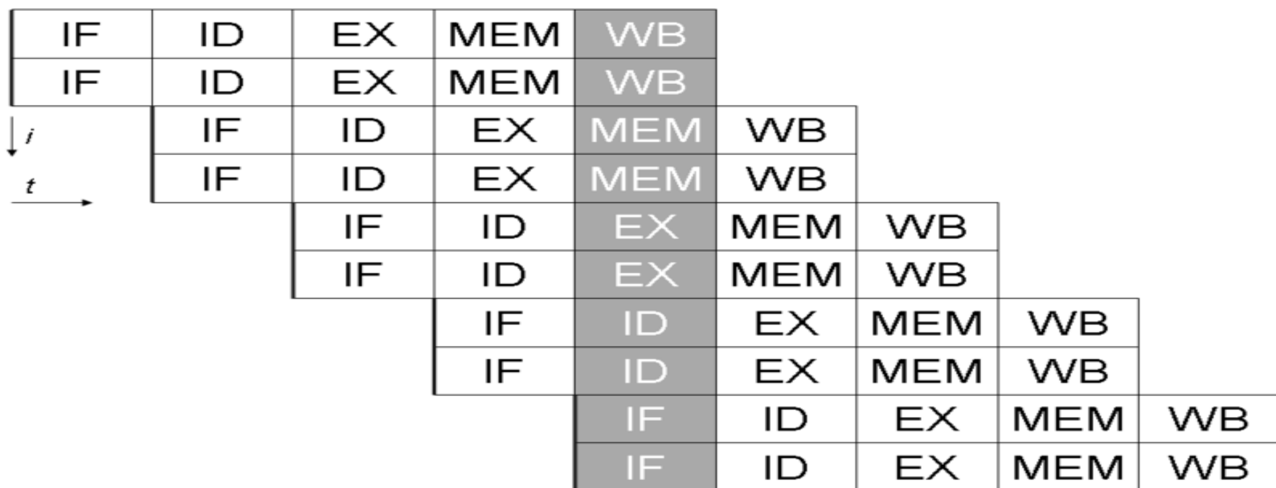


Fig. 3: Pipeline cu 5 faze al unui procesor superscalar, capabil de a emite două instrucțiuni pe ciclu. Poate avea două instrucțiuni în fiecare etapă a pipeline, pentru un total de până la 10 instrucțiuni executate simultan.

1.6.3. Paralelism la nivel de date

Paralelismul la nivel de date este paralelismul inerent din buclele de program, care se axează pe distribuirea datelor în diferite noduri de calcul pentru a fi prelucrate în paralel. „Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure."[9] [Cul99]. Multe aplicații ştiințifice şi de inginerie prezintă paralelism la nivel de date.

O dependență la nivelul unei bucle de program este dependența unei iterații a buclei de ieşirea produsă de una sau mai multe iterații precedente ale acelei bucle. Astfel de dependențe previn paralelizarea buclelor de program. Cu cât dimensiunea problemei devine mai mare, cu atât cantitatea de paralelism de date disponibil, de obicei, creşte [Cul99].

1.6.4. Paralelism la nivel de taskuri

Paralelismul la nivel de taskuri este caracteristica unui program paralel de a permite realizarea de operații complet diferite fie aceleaşi seturi de date sau pe seturi diferite [Cul99]. Acest lucru este opus cu paralelismul la nivel de date, unde acelaşi operații se fac pe aceleaşi sau diferite seturi de de date. Paralelismul la nivel de taskuri nu este scalabil, de obicei, cu dimensiunea problemei [Cul99].

9 Paralelizarea buclelor de program adesea duce la secvențe de operații (nu neapărat identice) sau funcții care sunt realizate pe elemente ale unor structuri mari de date. (tr. n.)

1.7. Hardware

1.7.1. Memorie şi comunicare

Memoria principală într-un calculator paralel este fie una partajată (caz în care memoria este partajată de toate elementele de procesare, într-un spaţiu de adrese unic), fie distribuită (fiecare element de procesare are propriul spaţiu local de adrese) [Hen02]. Memoria distribuită se referă la faptul că memoria este distribuită logic, dar de multe ori implică faptul că este, de asemenea, distribuită fizic. Memoria partajată distribuită este o combinaţie între cele două abordări şi presupune că fiecare element de procesare are propriul element de memorie şi acces la memoria altor elemente de procesare, nelocale. Accesele la memoria locală sunt, de obicei, mai rapide decât accesele la memoria nelocală.

Arhitecturile de calcul în care toată memoria principală poate fi accesată cu aceeaşi latenţă şi lăţime de bandă sunt cunoscute ca sisteme cu acces uniform la memorie (UMA - Uniform Memory Access). De regulă, doar un sistem cu memorie partajată, nedistribuită fizic, poate atinge aceste cerinţe (uniformitate a lăţimii de bandă şi a timpului de acces) şi se încadrează în categoria UMA. Un sistem care nu are această proprietate este cunoscut ca unul cu acces neuniform la memorie (NUMA - Non-Uniform Memory Access). Sistemele cu memorie distribuită fac parte din această categorie.

Sistemele de calcul folosesc memoriile cache, care sunt memorii de mici dimensiuni, rapide, situate aproape de procesor pentru a stoca temporar copii ale valorilor din memorie (aflate într-o anumită vecinătate, atât spaţială cât şi temporală). Sistemele paralele de calcul au probleme cu memoriile cache care pot stoca aceeaşi valoare în mai mult de o singură locaţie, cu posibilitatea de a crea execuţii incorecte ale programului. Aceste computere necesită un sistem de coerenţă, care monitorizează valorile din cache şi le invalidează sau le modifică strategic, asigurându-se astfel o execuţie corectă a programului. Supravegherea magistralei (Bus snooping) este una dintre cele mai comune metode pentru monitorizarea valorilor care sunt accesate. Proiectarea sistemelor de coerenţă a cache-urilor, de înaltă performanţă, scalabile, este o problemă foarte dificilă în arhitectura calculatoarelor. Ca rezultat, arhitecturile cu memorie partajată (SMA – Shared Memory Architecture) nu se scalează la fel de bine ca şi sistemele cu memorie distribuită [Hen02].

Comunicaţiile procesor-memorie şi procesor-procesor pot fi implementate în hardware în mai multe moduri, inclusiv prin intermediul memoriei comune (multiportată sau multiplexată), folosind

switchuri crossbar[10], o magistrală comună sau o rețea de interconectare cu un număr vast de topologii (plasă bidimensională, stea, inel, arbore, hipercub, arbore gras, plasă n-dimensională etc.).

Calculatoarele paralele care folosesc rețele de interconectare trebuie să aibă un fel de rutare, pentru a permite transmiterea de mesaje între nodurile care nu sunt conectate direct. Acest mediu folosit pentru comunicarea dintre procesoare este de obicei ierarhic în sistemele paralele de mari dimensiuni.

1.7.2. Clase de calculatoare paralele

Calculatoarele paralele pot fi aproximativ clasificate în funcție de nivelul la care hardware-ul sprijină paralelismul.

1.7.2.1. Calculatoare Multicore

Un procesor multicore este un procesor care include mai multe unități de execuție ("nuclee"). Aceste procesoare diferă de cele superscalare, care pot emite mai multe instrucțiuni per tact de la un singur flux de instrucțiuni (thread, fir); prin contrast, un procesor multicore poate emite mai multe instrucțiuni pe ciclu de la mai multe fluxuri de instrucțiuni (fire). Fiecare nucleu într-un procesor multicore poate fi superscalar şi de asemenea, la fiecare ciclu, fiecare nucleu poate emite mai multe instrucțiuni, de la un flux de instrucțiuni.

Multithreadingul simultan (tehnologia HyperThreading de la Intel este cea mai cunoscută) a fost cea mai devreme (prima) formă de arhitectură comercială pseudo-multicore. Un procesor capabil de multithreading simultan are doar o unitate de execuție ("nucleu"), doar că, atunci când unitatea de execuție este în aşteptare (cum ar fi un miss în cache), foloseşte unitatea de execuție pentru a procesa un al doilea fir. Intel Core şi Core 2 sunt familii de procesoare Intel multicore adevărate, la fel cum este şi Cell de la IBM, proiectat pentru utilizarea în cadrul Sony Playstation 3, sau IBMQ22 Blade.

1.7.2.2. Multiprocesoare simetrice (SMP - Symmetric multiprocessing)

Un multiprocesor simetric (SMP) este un sistem de calcul cu mai multe procesoare identice care partajează aceeaşi memorie şi se conectează prin intermediul unui bus (magistrală) [Hen02]. Astfel de arhitecturi au însă dezavantajul că busul reprezintă un punct central, toate

10 Switch (comutator) care conectează multiple intrări la multiple ieşiri într-o manieră matriceală

elementele de procesare se află în competiţie unele cu altele, pentru a obţine acces la bus, deoarece acesta este cel care asigură comunicarea cu memoria principală. Busul este aşadar un element arhitectural care limitează viteza de comunicaţie, un bottleneck care reduce scalabilitatea arhitecturilor SMP. Ca rezultat, un SMP în general, nu cuprinde mai mult de 32 procesoare [Hen02]. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."[11] [Hen02].

1.7.2.3. Calculatoare distribuite

Un calculator distribuit este un sistem de calcul cu memorie distribuită în care elementele de procesare sunt legate între ele printr-o reţea de interconectare. Astfel de sisteme sunt extrem de scalabile.

1.7.2.4. Arhitecturi tip cluster

Un cluster este un grup de computere slab cuplate care lucrează împreună (printr-o reţea de interconectare), astfel încât, în unele privinţe ele pot fi privite ca un singur calculator [Web08]. În timp ce maşinile într-un cluster nu trebuie să fie simetrice, echilibrarea încărcării este mai dificilă, dacă acestea nu sunt simetrice.

1.7.2.5. Arhitecturi cu procesare paralelă masivă (Massive parallel processing)

Un procesor masiv paralel (MPP) este un singur computer cu mai multe procesoare aflate în reţea. MPPurile au multe caracteristici ale clusterelor, dar ele sunt de obicei mai mari, au „cu mult mai mult” de 100 de procesoare [Hen02]. Într-un MPP, „each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect.”[12] [PCM07]. Din această categorie merită amintit supercomputerul Blue Gene /L cunoscut ca fiind cel mai rapid supercomputer din lume.

---

11 Din cauza dimensiunilor mici ale procesoarelor şi a reducerii semnificative în cerinţele de lăţime de bandă pentru bus, realizată de memoriile cache de mari dimensiuni, astfel de multiprocesoare simetrice sunt extrem de eficiente, cu condiţia ca o cantitate suficientă de lăţime de bandă pentru memorie să existe. (tr. n.)

12 Fiecare procesor conţine propria memorie şi o copie a sistemului de operare şi a aplicaţiei. Fiecare subsistem comunică cu restul prin intermediul unei reţele de interconectare de mare viteză (tr. n.)

1.7.2.6. Grid computing

Grid Computing este cea mai distribuită formă de calcul paralel. Se face uz de calculatoarele din Internet pentru a lucra la o anumită problemă. Din cauza lățimii de bandă scăzute şi a latenței extrem de mare disponibile pe Internet, grid computing de obicei se referă numai la probleme extrem de uşor de paralelizat (exemplu: SETI @ Home).

1.7.2.7. Calculatoare paralele specializate

În domeniul calculului paralel, există dispozitive specializate de calcul paralel, care rămân în domenii de interes de nişă. În timp ce nu sunt specifice domeniului, ele tind să fie aplicabile numai câtorva clase de probleme paralele.

Amintim din această categorie procesoarele grafice (GPU – Graphics Procesing Unit) şi procesoarele vectoriale.
Un GPU este un co-procesor care a fost puternic optimizat pentru calculele de procesare grafică. Procesarea grafică este dominată de date ce pot fi paralelizate, în special operațiile matriceale din algebra liniară.

Cray-1 este cel mai cunoscut procesor vectorial. Un procesor vectorial este un sistem de calcul care poate executa aceeaşi instrucțiune pe seturi mari de date. Procesoarele vectoriale conțin instrucțiuni de nivel înalt care sunt capabile să opereze cu şiruri lineare de numere, numite vectori. Un exemplu de operație vectorială este:

$$A = B \times C$$

unde A, B, şi C sunt vectori de 64 de elemente, pe 64 de biți, în virgulă mobilă. Acestea sunt strâns legate de SIMD, din clasificarea lui Flynn.

1.8. Software

Limbajele de programare concurentă, bibliotecile, API-urile şi modelele de programare paralelă au fost create pentru calculatoarele care permit calculul paralel. Acestea pot fi în general împărțite în clase, în funcție presupunerile pe care le fac referitor la arhitectura hardware: cu memorie partajată, distribuită, sau distribuită partajat. Limbajele de programare bazate pe memorie partajată comunică prin manipularea de variabile partajate. Cele orientate pe memorie distribuită

utilizează schimbul de mesaje. POSIX Threads (PThreads) şi OpenMP sunt două dintre cele mai larg utilizate API-uri de memorie partajată, întrucât Message Passing Interface (MPI) este cel mai des folosit API de message-passing. Un concept de programare paralelă a programelor este aşa numitul concept al viitorului, în care o parte a unui program promite că va livra datele necesare, într-o altă parte a unui program, undeva în viitor.

1.8.1. Paralelizarea automată

Paralelizarea automată a unui program secvențial de către un compilator este "Sfântul Graal" al calcului paralel. În ciuda unor decenii de muncă făcută de către cercetători în compilatoare, paralelizarea automată a avut doar succes limitat [She05]. Limbajele de programare paralelă rămân fie explicit paralele sau, în cel mai bun caz, parțial implicite, în care un programator oferă compilatorului directive pentru paralelizare. Câteva limbaje de programare paralelă complet implicite există (SISAL, Parallel Haskell, şi Mitrion-C, pentru FPGAuri), dar acestea nu se utilizează pe scară largă fiind mai degrabă nişte limbaje de nişă.

1.8.2. Aplicaţii

Din moment ce calculatoarele paralele devin tot mai răspândite şi mai rapide, devine fezabilă rezolvarea de probleme care anterior luau prea mult timp pentru a rula. Calculul paralel este utilizat într-o gamă largă de domenii, de la bioinformatică, la economie. Cele mai cunoscute tipuri de probleme specifice calcului paralel sunt următoarele [Asa06]:

- Algebra liniară densă (datele sunt sub formă de vectori şi matrice dense);
- Algebra liniară rară (datele sunt tot sub formă de tablouri, dar conțin multe valori nule, motiv pentru care se şi stochează comprimat);
- Metode spectrale (cum ar fi Transformata Fourier a lui Cooley-Tukey);
- Probleme cu N corpuri (N-body problems, probleme în care obiectele depind, interacționează foarte mult cu alte obiecte), cum ar fi simularea Barnes-Hut;
- Probleme tip grilă structurată (Structured grid problems), precum metodele Lattice Boltzmann;
- Probleme tip grilă nestructurată (analiza elementelor finite);
- Simulări Monte Carlo (calculele depind de rezultate statistice);

- Logică combinaţională (cum ar fi tehnicile criptografice de tip forţă brută);
- Parcurgerea grafurilor (cum ar fi algoritmii de sortare);
- Programare dinamică;
- Metode de tip branch and bound;
- Modele grafice (cum ar fi detectarea modelelor Markov ascunse şi construirea reţelelor Bayesiane);
- Simulări de tip automat finit de stări.

2. Arhitecturi multiprocesor cu memorie partajată

De vreme ce arhitecturile uniprocesor îşi ating limitele fizice, o bună înţelegere a arhitecturilor paralele devine esenţială, pentru oricine lucrează în domeniul arhitecturilor de calcul. Importanţa arhitecturilor multiprocesor este din ce în ce mai mare, având în vedere faptul că acum tot mai multe calculatoare sunt proiectate ca arhitecturi multicore.

Astfel de arhitecturi afectează însă modelul de programare al aplicaţiilor care trebuie să fie de acum scrise paralel, pentru a putea beneficia de puterea de calcul pe care o oferă arhitecturile multiprocesor, iar acest lucru înseamnă că nu doar arhitecţii ci şi programatorii trebuie să înţeleagă contextul mai dificil şi mai neobişnuit în care sunt nevoiţi să îşi realizeze munca: calculul paralel.

Arhitecturile multiprocesor cu memorie partajată fac parte din categoria MIMD (Multiple instruction streams, multiple data streams), din cadrul taxonomiei lui Flynn. MIMD reprezintă arhitectura cea mai des utilizată pentru sistemele multiprocesor, deoarece oferă flexibilitate, dar şi pentru că poate fi construită pe baza microprocesoarelor existente deja [Hen02].

În cadrul arhitecturilor MIMD se disting două categorii:

- arhitecturi cu memorie partajată;
- arhitecturi cu memorie distribuită.

Arhitecturile cu memorie partajată se numesc aşa deoarece folosesc o singură memorie principală, care este folosită, divizată, pentru toate procesoarele din cadrul sistemului. Interconectarea elementelor de procesare cu această memorie partajată se face în general cu ajutorul unei magistrale. Relaţia memorie – procesoare este una caracterizată de simetrie pentru că fiecare

procesor poate accesa memoria principală cu aceeaşi latenţă. Astfel, aceste arhitecturi se mai numesc şi arhitecturi multiprocesor cu memorie partajată simetric, sau arhitecturi UMA (Uniform Memory Access).

Celălalt tip de arhitecturi, cele cu memorie distribuită vin să adreseze problema scalabilității pe care o au arhitecturile cu memorie partajată: memoria poate fi partajată fizic, fiecare procesor putând avea propria memorie cu care poate comunica direct. Legătura dintre elementele de procesare şi memoriile ataşate acestora se face printr-o rețea de interconectare.

Comunicarea inter-procesoare se poate face în esență în două feluri: prin intermediul unui spațiu de adrese partajat[13] sau prin intermediul schimbului de mesaje, sincron sau asincron.

Ne vom axa în cele ce urmează pe arhitecturile multiprocesor cu memorie partajată şi vom prezenta trei dintre cele mai importante concepte care guvernează arhitecturile paralele: coerența memoriilor cache, consistența memoriei şi sincronizarea proceselor.

2.1. Coerenţa memoriilor cache

Coerența memoriilor cache este o problemă care afectează arhitecturile multiprocesor, cu memorie partajată (SMA – Shared Memory Architectures).

În cazul arhitecturilor uniprocesor această problemă nu apare deoarece există un singur procesor care să citească şi să scrie date din/în memorie. În plus, se poate face o singură operație asupra memoriei la un moment dat, astfel că, atunci când o locație din memorie se schimbă, toate operațiile următoare, care implică citirea acelei locații de memorie, vor accesa valoarea corect modificată.

În sistemele multiprocesor există două sau mai multe procesoare care lucrează în paralel, existând deci posibilitatea ca o locație de memorie să fie accesată în acelaşi moment de timp, de mai multe procesoare. Atât timp cât acea locație de memorie este accesată doar pentru citire (niciun procesor nu o modifică), partajarea ei se poate face fără probleme. Dar, atunci când valoarea este modificată de un procesor, există riscul ca celelalte procesoare să lucreze cu o copie veche, invalidă, a locației de memorie partajată.

Exemplul de mai jos ilustrează problema numită coerența cache-urilor.

---

13 este cazul primei categorii de arhitecturi MIMD, dar poate fi cazul şi arhitecturilor distribuite, cu memorie partajată, numite DSM – Distributed Shared-Memory, sau NUMA (Non-Uniform Memory Access), pentru că spațiul de adrese este cel partajat, memoria fiind de fapt distribuită fizic

Pas	Eveniment	Conținut cache P_0	Conținut cache P_1	Conținut memorie principală (X)
0		?	?	1
1	P_0 citeşte locația X	1	?	1
2	P_1 citeşte locația X	1	1	1
3	P_1 scrie 0 în X (WT)	0	1	0
	(WB)	0	1	1

Fig. 4: Problema coerenței cache-urilor pentru o singură locație de memorie (X) citită şi scrisă de două procesoare P_0 şi P_1. S-a presupus că inițial, nici una din cele 2 cache-uri nu conține variabila globală X şi că aceasta are valoarea 1 în memoria globală. În pasul 3 P_1 are o valoare incoerentă a variabilei X, considerând cache-uri de tip WT (Write Through). Pentru cache-uri WB (Write Back) situația este similară.

Aşa cum arată şi exemplul de mai sus, coerența cache-urilor se referă la integritatea datelor din memoriile cache locale, ale unei resurse partajate (memoria principală). Simplu spus, putem afirma că un sistem este coerent dacă orice citire a unei locații de memorie returnează valoarea cea mai recent scrisă, pentru acea locație de memorie. Ne referim aşadar la două aspecte, coerență şi consistență, ambele esențiale în scrierea corectă a programelor pentru arhitecturi cu memorie partajată.

Coerența defineşte ce valori pot fi returnate în urma unei citiri, iar consistența specifică când o valoare scrisă va fi returnată în urma unei operații de citire.

Atunci când mai multe procesoare păstrează în cache-urile lor locale (private) copii ale unor locații dintr-o memorie partajată, orice modificare a unei astfel de locații, la nivel de cache (locală deci), poate cauza o inconsistență la nivelul global al memoriei partajate.

O definiție completă a coerenței este însă următoarea: un sistem de memorie este coerent dacă:

1. O citire de către un procesor oarecare, P, a unei locații X, urmată de o scriere a acelei locații X, de către P, returnează întotdeauna valoarea scrisă de P, atunci când niciun alt procesor nu scrie în locația X, între cele două operații (citire şi scriere) făcute de P;
2. O citire a locației X, de către P, urmată de o scriere a aceleiaşi locații, dar de către un alt procesor, întoarce valoarea scrisă, dacă cele două operații sunt suficient de separate în timp şi dacă nu are loc nicio altă scriere, între cele două accese la locația X;
3. Operațiile de scriere ale aceleiaşi locații de memorie sunt serializate. Prin serializarea acestora se înțelege că: două scrieri ale aceleiaşi locații de memorie, de către oricare două procesoare, sunt văzute în aceeaşi ordine de către toate procesoarele.

Prima proprietate asigură păstrarea ordinii din cadrul programului.

A doua proprietate pune în evidență importanța coerenței memoriei: dacă un procesor ar citi o valoare veche din memorie am putea trage concluzia că acea memorie este incoerentă.

Serializarea operaților de scriere este o proprietate care asigură faptul că toate scrierile făcute la nivelul unei locații de memorie sunt văzute de către procesoare în ordinea în care acestea au fost făcute.

Incoerența cache-urilor poate fi cauzată de: partajarea datelor, migrarea proceselor, sau de operații de intrare/ieşire şi sunt necesare o serie de reguli de care trebuie ținut cont, pentru ca asigurarea coerenței memoriilor cache să fie posibilă. Iată, spre exemplu, doar 3 astfel de reguli [Ega07]:

- "Make updates to a location appear atomic";[14]
- "A write is performed globally when no processor can access the old data value";[15]
- "Cache coherence is maintained if a read always returns the latest write and the latest write is globally performed".[16]

În vederea soluționării acestei probleme, există mai multe mecanisme de asigurare a coerenței memoriilor cache:

- **„Snooping"** (procesul prin care fiecare memorie cache supraveghează liniile de adresare a memoriei, pentru locațiile de memorie păstrate. Atunci când o operație de scriere asupra unei locații de memorie este observată, locația de memorie corespunzătoare din cache este invalidată);
- **„Snarfing"** (procesul prin care se monitorizează atât adresa locației de memorie cât şi noua valoare, astfel ca actualizarea locației de memorie să poată fi făcută de către controllerul memoriei cache, atunci când modificarea a avut loc extern, la nivelul unui alt cache, al unui alt procesor);
- mecanisme bazate pe **directori** (mențin un director central al blocurilor cache).

În cele ce urmează, ne vom orienta atenția asupra mecanismului „snooping", principala strategie de păstrare a coerenței memoriilor cache, în cadrul sistemelor multiprocesor cu memorie partajată, bazate pe magistrală (bus). Busul este un mecanism convenabil de asigurare a consistenței memoriilor cache deoarece permite tuturor procesoarelor din cadrul arhitecturii CMP (Chip Multi-Processors) să „observe" tranzacțiile care se fac la nivelul memoriei.

Există două protocoale principale în cadrul categoriei de protocoale de coerență bazate pe supravegherea busului:

14 Actualizările unei locații trebuie să apară atomice (tr. n.)

15 O scriere a unei locații de memorie se face global atunci când niciun procesor nu mai poate accesa vechea valoare (tr. n.)

16 Coerența cache-urilor este menținută dacă o citire returnează de fiecare dată ultima valoare scrisă şi dacă ultima scriere este realizată global (tr. n.)

- **write invalidate**: procesorul care vrea să modifice o locație de memorie într-un cache al său va cauza mai întâi invalidarea tuturor celorlalte copii păstrate în restul memoriilor cache (de la celelalte procesoare) şi de abia apoi va actualiza blocul cache (locația de modificat);
- **write update** sau write broadcast: procesorul care face modificarea emite valoarea actualizată tuturor celorlalte cache-uri astfel că toate copiile locației de memorie partajată rămân identice.

Diferențele dintre cele două tipuri majore de protocoale „snooping" se bazează pe următoarele trei caracteristici:

- într-un protocol de tip write update, multiple scrieri ale aceleiaşi locații de memorie, fără operații intermediare de citire, necesită multiple acțiuni de tip write broadcast. O singură invalidare inițială este însă suficientă, într-un protocol de tip write invalidate;
- un protocol cu invalidare lucrează la nivelul blocurilor cache, în timp ce un protocol de tip update lucrează la nivelul cuvintelor din cadrul blocului cache. Fiecare scriere a unui cuvânt cauzează emiterea unui mesaj de tip broadcast, în vreme ce doar prima scriere a unui cuvânt, din cadrul unui bloc, trebuie să emită un mesaj de invalidare (pentru că acesta va cauza invalidarea întregului bloc);
- întârzierea dintre momentul scrierii unei locații de memorie de către un procesor şi citirea acelei locații de către un altul este de obicei mai mică în cazul unei scheme de tip write broadcast, comparativ cu o schemă cu invalidare, unde procesorului care face citirea îi este mai întâi invalidată locația de memorie şi de abia apoi este citită, fiind întârziat până când o copie a acelei locații poate fi obținută.

Având în vedere faptul că lățimea de bandă a memoriei şi a busului sunt aspecte esențiale în sistemele multiprocesor, bazate pe bus, cu memorie partajată (simetric), protocoalele de tip write invalidate sunt cele mai des folosite, în detrimentul celor de tip write update, deoarece generează mai puțin trafic la nivelul busului şi la nivelul memoriei principale. În plus, protocoalele de tip write broadcast cauzează mai multe probleme la nivelul modelelor de consistență a memoriei, reducând astfel potențialele câştiguri de performanță prin tehnica de update.

Având în vedere argumentele de mai sus, ne vom orienta în cele ce urmează asupra schemelor de tip „snoopy" bazate pe invalidare. Unele dintre cele mai cunoscute şi utilizate protocoale de coerență a cache-urilor, din această categorie, sunt următoarele:

- MSI;
- MESI (Illinois);
- MOSI;
- MOESI.

În cele ce urmează, vom prezenta cele patru protocoale enumerate mai sus, pornind de la cel mai

simplu dintre ele şi prezentându-le succesiv pe următoarele prin prisma asemănărilor şi deosebirilor care există între acestea.

2.1.1. Protocolul MSI

Diagrama următoare ilustrează funcționarea protocolului MSI. Fiecare tranziție, dintr-o stare în alta, este descrisă sub forma Eveniment/Acțiune, marcându-se astfel, pentru fiecare stare, fiecare situație posibilă, adică, care este acțiunea care trebuie executată în vederea păstrării coerenței, ținând cont de contextul în care ea trebuie luată, context marcat de starea blocului şi de operația, evenimentul, care se efectuează asupra acestuia. În situația în care tranziția se face fără a efectua nicio acțiune, lipsa acțiunii se marchează cu `-`.

S-au folosit următoarele notații pentru a marca evenimentele şi acțiunile care au loc în cadrul protocoalelor de coerență ce urmează a fi descrise:

PrRdHit	un procesor a efectuat o citire cu Hit la nivelul unui bloc dintr-un cache de-al său (local)
PrRdMiss	un procesor a efectuat o citire cu Miss la nivelul unui bloc dintr-un cache de-al său (local)
PrWrHit	un procesor a efectuat o scriere la nivelul unui bloc dintr-un cache de-al său (local)
PrWrMiss	un procesor a încercat să scrie o locație care nu se află într-un cache de-al său (local)
BusRd	un alt procesor (extern) vrea să citească o locație de memorie partajată (aflată şi în alte cache-uri asociate altor procesoare) şi face acest lucru prin intermediul busului
BusRdX	un alt procesor (extern) urmează să modifice o locație de memorie partajată şi cere deci acces eXclusiv la ea, prin intermediul busului
WB	Write Back, marchează acțiunea de scriere a unei locații de memorie modificate, pe nivelul de memorie superior

Fig. 5: Semnificația notațiilor folosite pentru descrierea evenimentelor şi acțiunilor care au loc în cadrul protocoalelor de coerență prezentate

Protocolul MSI este un protocol de coerență a cache-urilor cu trei stări, literele din numele protocolului identificând stările posibile în care se poate afla blocul din cache. Pentru MSI avem următoarele stări:

- **Modified (M)** – acest bloc a fost modificat (dirty) în cache-ul curent. Data a devenit inconsistentă cu nivelul superior de memorie (de exemplu memoria principală). Un cache care are un bloc în starea M este obligat să scrie valoarea blocului în nivelul superior de memorie, când această locație este evacuată.
- **Shared (S)** – acest bloc este partajat, nu a fost modificat şi există în cel puțin un cache. Cache-ul poate deci evacua blocul fără să îl scrie în nivelul superior de memorie, pentru că acesta nu este dirty, ci doar partajat la nivelul mai multor memorii cache.

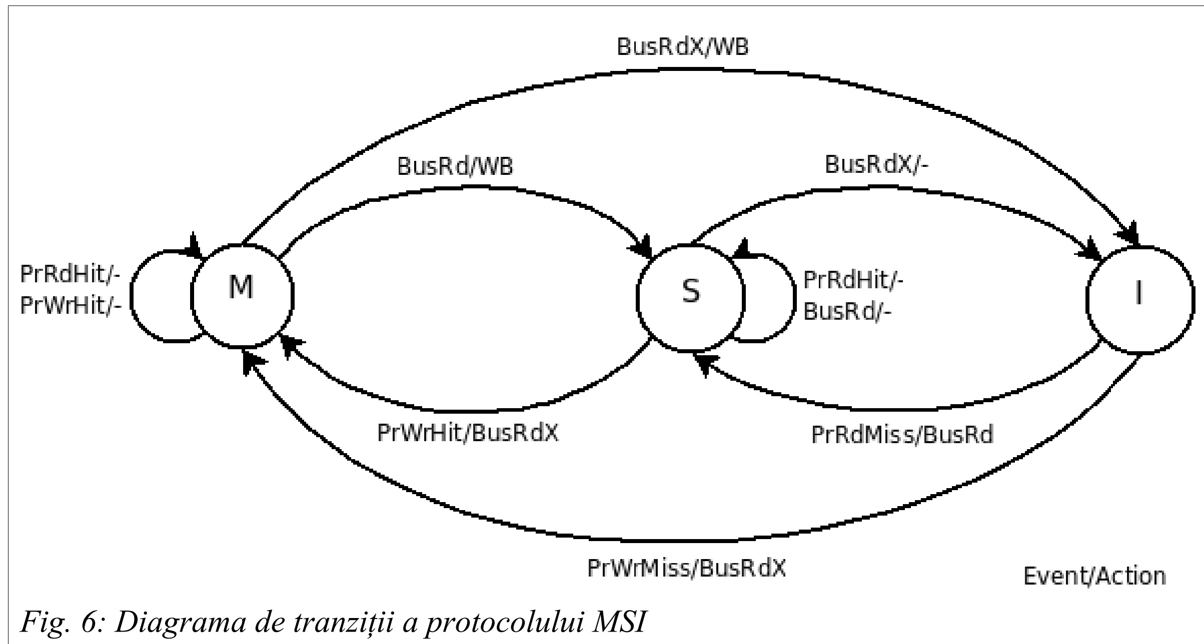


Fig. 6: Diagrama de tranziții a protocolului MSI

- **Invalid (I)** – blocul este invalid, trebuie adus din memorie, sau din alt cache pentru a putea fi utilizat.

Stările prezentate anterior sunt menținute prin comunicații între cache-uri şi memoria de nivel superior, realizate prin intermediul busului. Cache-urile au diferite responsabilități când scriu/ citesc blocuri, sau când "observă" că alte cache-uri efectuează o operație de scriere sau de citire.

Când o cerere de citire ajunge la cache pentru un bloc aflat în stările M sau S, cache-ul oferă acea locație de memorie. Dacă blocul nu se află în cache (este în starea I), trebuie să se verifice ca acest bloc să nu se afle în starea M în oricare alt cache. Dacă un alt cache are acest bloc în starea M, cache-ul care deține blocul trebuie să-l scrie în memoria de nivel superior şi să treacă blocul în starea S (shared). Imediat după ce blocul a fost scris în nivelul superior de memorie, cache-ul îşi poate obține datele pe care le-a cerut din memoria de nivel superior. Abia acum cache-ul poate răspunde cererii. După ce a răspuns cererii, intrarea din cache este în starea S (a avut loc o operație de citire).

Când o cerere de scriere ajunge la cache pentru un bloc aflat în starea M, cache-ul modifică

data local. Dacă blocul este în starea S (shared), cache-ul este responsabil să anunțe toate celelalte cache-uri care dețin acest bloc, că trebuie sa îşi invalideze blocul. După ce această operație s-a efectuat, data poate fi modificată local.

Tabelul de mai jos reprezintă o sistematizare a modului de funcționare al acestui protocol.

Starea blocului cache	**Eveniment**	**Acțiune**	**Comentarii**
I	PrRdMiss	BusRd	Un procesor a încercat să citească o locație de memorie, nu a găsit-o în cache-ul său, motiv pentru care emite o cerere de citire pe bus. Numai dacă nici un alt cache al altui procesor nu are acea locație de memorie, informația în cauză va fi preluată din memoria principală.
I	PrWrMiss	BusRdX	Un procesor vrea să scrie o locație de memorie pe care nu o are în cache-ul asociat lui. Din acest motiv, el va cere acces exclusiv, la acea locație, prin intermediul busului.
S	PrRdHit	-	Un procesor citeşte o locație partajată. Nicio acțiune nu trebuie luată.
S	BusRd	-	Un procesor citeşte o locație partajată, dintr-un cache al unui alt procesor. Nicio acțiune nu trebuie luată.
S	PrWrHit	BusRdX	Un procesor vrea să scrie o locație partajată, motiv pentru care cere acces exclusiv la ea.
S	BusRdX	-	Un alt procesor a cerut acces exclusiv la o locație de memorie, motiv pentru care blocul cache va fi invalidat.
M	PrRdHit	-	Un procesor citeşte o locație modificată de el. Nicio acțiune nu trebuie luată.
M	PrWrHit	-	Un procesor scrie o locație modificată de el. Nicio acțiune nu trebuie luată.
M	BusRd	WB	Un procesor doreşte să citească o locație modificată de un alt procesor. Acea locație de memorie, modificată de un cache extern, va fi mai întâi scrisă în memoria principală.
M	BusRdX	WB	Un procesor doreşte să scrie o locație modificată de un alt procesor. Acea locație de memorie, deja modificată de un cache extern, va fi mai întâi scrisă în memoria principală.

Fig. 7: Descrierea funcționării protocolului MSI

2.1.2. Protocolul MESI (Illinois)

Acest protocol este unul dintre primele protocoale de tip write-invalidate, fiind introdus odată cu apariția procesorului Intel Pentium, pentru a susține mai bine memoriile cache de tip write-back, care înlocuiau memoriile de tip write-through (existente în Intel 486) [Zah03].

Numele acestui protocol vine de la cele patru stări pe care le conține:

- **Modified (M)** – Linia cache este prezentă numai în cache-ul curent şi a fost modificată, dirty, față de valoarea din memoria principală. Memoria cache va fi nevoită să evacueze în memoria principală linia aflată în această stare, înainte de a permite citiri ale acelei locații, de acum invalidă, din memoria principală;
- **Exclusive (E)** – Linia cache este prezentă în cache-ul curent şi este validă, în sensul că este aceeaşi cu cea din memoria principală;
- **Shared (S)** – Indică faptul ca această linie este partajată, se află şi în alte cache-uri de pe această maşină;
- **Invalid (I)** – Această linie din cache este invalidă, procesorul trebuie să o caute în alta parte (fie în memoria principală, fie chiar în alt cache).

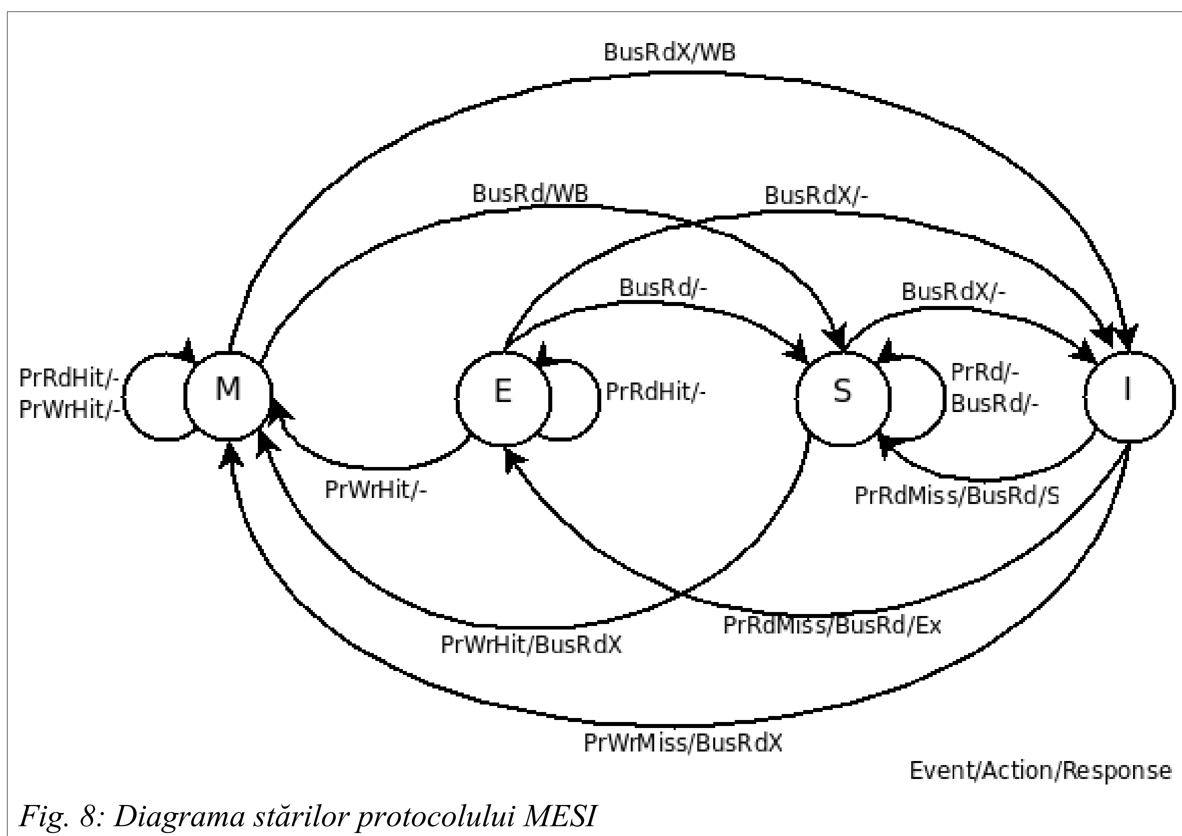


Fig. 8: Diagrama stărilor protocolului MESI

Cele patru stări sunt asociate fiecărei linii cache, fiind codificate pe doi biți. Atât starea E cât şi starea S sunt valide, dar exclusiv înseamnă că locația de memorie este prezentă numai într-un singur cache.

Acest protocol se comportă bine în situațiile în care nu există foarte multe date partajate, dar este mai complex decât protocolul MSI, fiindcă se bazează pe răspunsul venit prin snooping: după cum se poate observa din diagrama de tranziții, un read cu miss pentru un bloc aflat în starea Invalid, va cauza aducerea acelui bloc în cache. Blocul adus va intra în starea E, în cazul în care el nu se mai află şi în alt cache al altui procesor, sau în starea S, în caz contrar.

Noutatea introdusă de acest protocol, comparativ cu cel prezentat anterior, constă în starea Exclusiv. Aceasta nouă stare duce, față de protocolul MSI, la o optimizare a traficului. Datele aflate in starea Exclusive nu mai trebuie scrise înapoi in memoria principală la momentul evacuării, acesta fiind avantajul principal față de protocolul MSI prezentat anterior.

Un cache care deține o linie în starea Exclusive trebuie sa urmărească toate citirile făcute de celelalte procesoare din sistem şi în caz de potrivire să îşi treacă linia în starea de Shared.

Avantajul adus de protocolul MESI este deci un trafic mai scăzut pe bus, dar, ca şi dezavantaj, el prezintă o complexitate mai crescută a hardware-ului.

2.1.3. Protocolul MOSI

La fel ca şi MESI, acest protocol este o e extensie a protocolului MSI, în sensul că introduce o nouă stare O – Owned. Un bloc cache aflat în această stare marchează faptul că el este cel care deține locația de memorie în cauză, servind cererilor venite din partea altor procesoare. Altfel (cazul MSI), un acces – cu miss în propriul cache – al unui procesor la un bloc ce se află partajat în cache-urile altor procesoare, ar necesita accesul memoriei principale (latență mare), în lipsa unei reguli după care unul (şi numai unul!) din procesoarele care dețin o copie validă a acelui bloc să o pună la dispoziție pe busul comun.

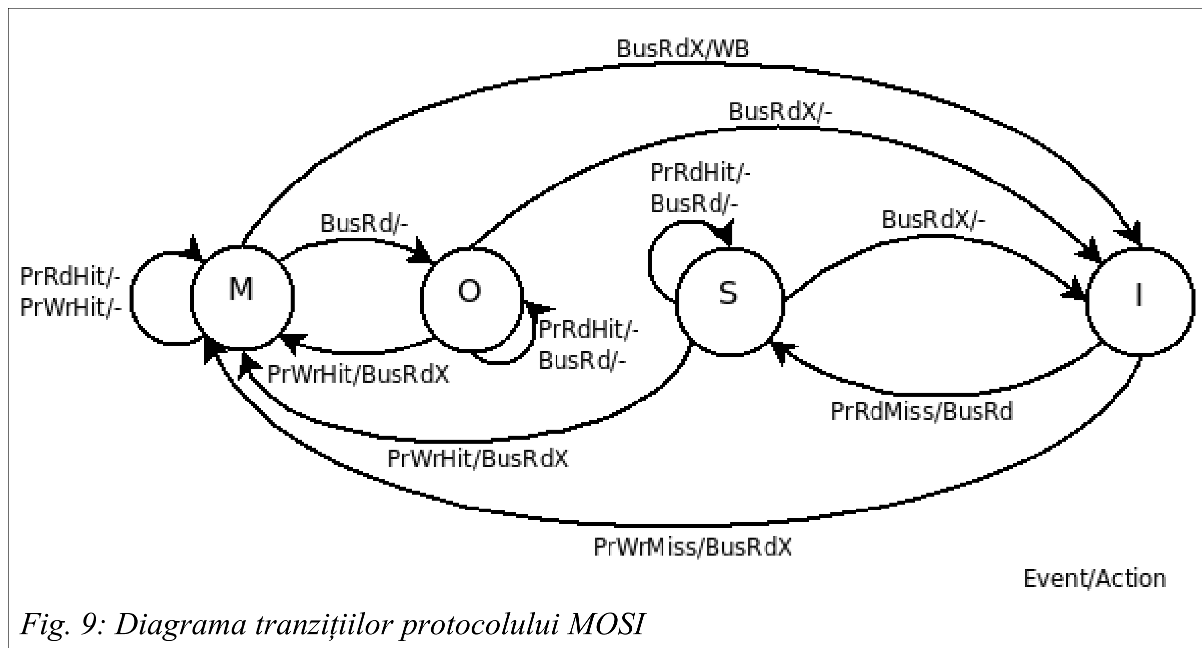


Fig. 9: Diagrama tranzițiilor protocolului MOSI

După cum se poate vedea şi din diagramă, atunci când un procesor doreşte să citească o locație de memorie care se află modificată extern, în cache-ul asociat altui procesor, acea locație nu va mai fi scrisă mai întâi (WB) în memoria principală, ci va trece în starea nouă (Owned), valoarea modificată fiind pusă pe bus, la dispoziția procesorul care a efectuat citirea. Evident, pentru fiecare bloc cache poate exista la un moment dat un singur deținător (Owner).

De asemenea, mai merită observat faptul că, atunci când un bloc aflat în starea Owned este modificat, scris, el va tranzita în starea Modified numai după ce, în prealabil, s-a cerut acces exclusiv la acel bloc, pentru că, un bloc aflat în starea Owned poate fi modificat şi partajat în acelaşi timp. Trecerea în starea Modified duce la pierderea ownershipului, iar – dacă în prealabil nu s-ar fi cerut acces exclusiv pe bus la acel bloc, implicând invalidarea copiilor sale, aflate în starea Shared în alte cache-uri - alte procesoare ar fi putut risca să citească o valoare veche, pentru că blocul aferent se află în cache-urile asociate lor în starea Shared.

Cu prețul unei complexități hardware mai ridicate, comparativ cu MSI, acest protocol are avantajul că poate reduce semnificativ numărul de scrieri (WB – Write Back) şi citiri la / de la nivelul memoriei principale.

Comparativ cu MESI, putem afirma că acest protocol are cam aceeaşi complexitate (tot 4 stări), introduce starea Owned, dar nu conține starea E, care contribuie şi ea la reducerea numărului de scrieri la nivelul de memorie ierarhic superior.

2.1.4. Protocolul MOESI

Reprezintă un protocol complet de coerență a memoriilor cache, deoarece conține cinci stări, adică toate stările întâlnite în general în protocoalele de coerență ale memoriilor cache. Acest protocol conțin următoarele stări [AMD07]:

- **Modified (M)** – Linia cache este prezentă numai în cache-ul curent şi a fost modificată, dirty, față de valoarea din memoria principală. Memoria cache va fi nevoită să evacueze în memoria principală linia aflată în această stare, înainte de a permite citiri ale acelei locații, de acum invalidă, din memoria principală;
- **Owned (O) –** O linie cache aflată în această stare marchează faptul că ea este cea care deține locația de memorie în cauză, servind cererilor venite din partea altor procesoare. Linia este partajată, dar diferă de valoarea aflată în memoria principală;
- **Exclusive (E)** – Linia cache este prezentă în cache-ul curent şi este validă, în sensul că este aceeaşi cu cea din memoria principală;
- **Shared (S) –** Indică faptul ca această linie este partajată, se află şi în alte cache-uri de pe această maşină;
- **Invalid (I)** – Această linie din cache este invalidă, procesorul trebuie să o caute în alta parte (fie în memoria principală, fie chiar în alt cache).

Acest protocol este o variantă îmbunătățită a protocoalelor MESI respectiv MOSI. Datorită stării în plus pe care o permite față de MESI (O – Owned), este evitată situația în care valorile modificate trebuie scrise în memoria principală atunci când un alt procesor încearcă să citească acea locație. Starea Owned îi permite procesorului să dețină dreptul de a modifica o locație partajată (shared).

Dar starea Owned nu este o noutate, cel puțin nu față de MOSI. Deosebirea este totuşi evidentă: acest protocol nu renunță nici la starea Exclusive, având deci cinci stări.

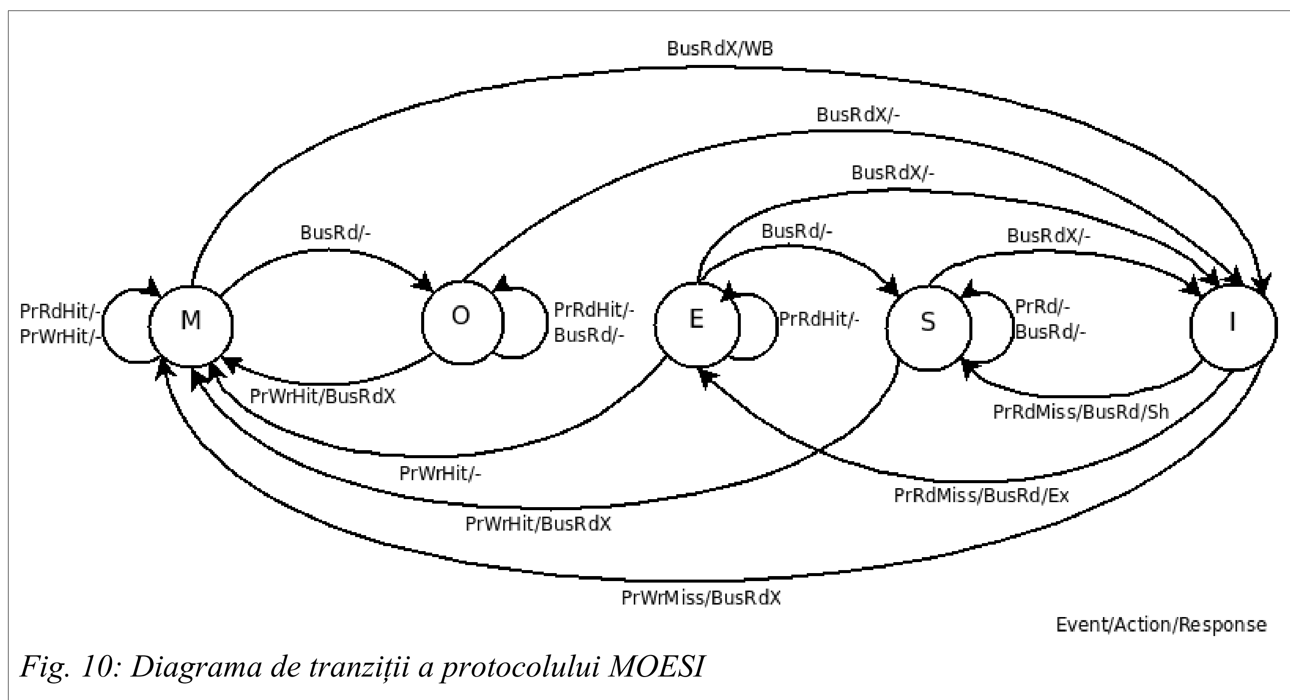


Fig. 10: Diagrama de tranziții a protocolului MOESI

MOESI este util mai ales atunci când latența comunicației dintre procesoare este semnificativ mai bună decât cea dintre procesor şi memorie, sistemele multicore cu cache de nivel doi (L2 Cache) fiind un exemplu în acest sens. Prin faptul că acest protocol adaugă protocolului MESI şi starea Owned, avantajul său este evident legat de reducerea numărului de scrieri în memoria principală.

Putem observa în concluzie că, cele patru protocoale de coerență a memoriilor cache prezentate mai mai sus se află în strânsă legătură: MSI e o primă variație a celui mai simplu protocol de tip write invalidate posibil, cel cu două stări (Valid şi Invalid), MOSI şi MESI aduc îmbunătățiri lui MSI, iar MOESI le înglobează pe toate, fiind cel mai general dintre ele, după cum o arată şi diagrama de mai jos.

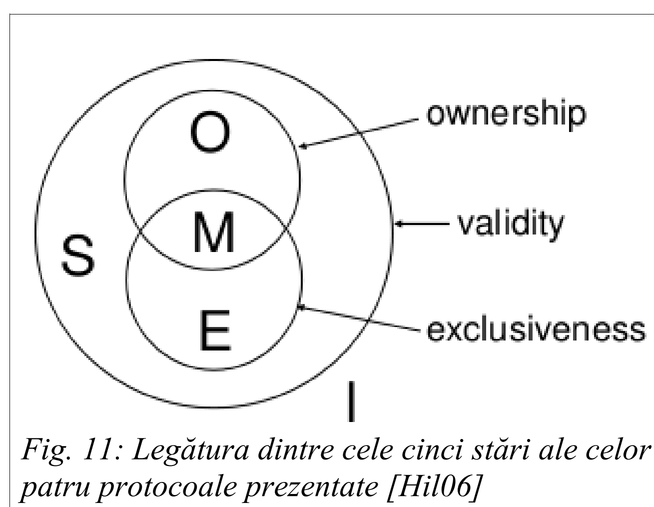


Fig. 11: Legătura dintre cele cinci stări ale celor patru protocoale prezentate [Hil06]

2.2. Modele de consistenţă a memoriei

Coerența cache-urilor asigură o viziune consistentă a memoriei pentru diversele procesoare din sistem. Nu se răspunde însă la întrebarea "CÂT de consistentă?", adică în ce moment trebuie să vadă un procesor că o anumită variabilă a fost modificată de un altul?

(P1)	(P2)
A=0;	B=0;
- - - -	- - - -
A=1;	B=1;
L1: if(B==0)...	L2: if(A==0)...

Fig. 12: Exemplu care ilustrează problema consistenței memoriei („data race”) [Hen02]

În mod normal, este imposibil pentru ambele programe rezidente pe procesoare diferite (P1, P2) să evalueze pe adevărat condițiile L1 şi L2, în condițiile în care scrierile se fac imediat şi sunt văzute de ambele procesoare imediat. Şi totuşi, acest fapt s-ar putea întâmpla. Să presupunem că variabilele globale, partajate, A şi B sunt pe '0' în cache-urile ambelor procesoare. Dacă de exemplu, între scrierea A=1 în P1 şi invalidarea lui A în P2 se scurge un timp suficient de îndelungat, atunci este posibil ca P2 să ajungă la eticheta L2 şi să evalueze (în mod evident eronat) A==0 ca adevărat!

Cea mai simplă soluție constă în forțarea fiecărui procesor care scrie o variabilă partajată, de a-şi întârzia această scriere până în momentul în care toate invalidările (WI) cauzate de către procesul de scriere, se vor fi terminat. Această strategie se numeşte consistență secvențială şi reprezintă de altfel cea mai simplă strategie de păstrare a consistenței memoriei.

Deşi consistența secvențială prezintă o paradigmă simplă din punctul de vedere al programatorului, totuşi ea reduce performanța arhitecturilor multiprocesor cu memorie partajată, în special pe sistemele având un număr mare de procesoare sau, cu căi de interconectare de latențe ridicate.

Un model de asemenea simplu din punctul de vedere al programatorului dar care permite o implementare mai eficientă, se bazează pe asumarea faptului că programele aflate în execuție pe diversele procesoare din sistem, sunt sincronizate. Un program este sincronizat dacă toate accesele la o variabilă partajată sunt ordonate prin operații de sincronizare. Astfel, accesarea unei date este ordonată printr-o operație de sincronizare dacă, în orice instanță posibilă de execuție, scrierea unei variabile partajate de către un procesor şi respectiv accesarea (scriere/citire) ei de către un alt procesor, sunt separate între ele printr-o pereche de operații de sincronizare. O astfel de operație este executată după "WRITE" de către procesorul care scrie iar cealaltă operație este executată de către celălalt procesor înainte de accesul său la data partajată. Numim aceste operații de sincronizare UNLOCK - pentru că deblochează un procesor (proces de scriere) blocat, respectiv LOCK - pentru că îi dă dreptul unui procesor de a citi o variabilă partajată.

Aşadar, un program este sincronizat permiţând consistenţa variabilelor partajate, dacă fiecare operaţie de scriere executată de către un procesor urmat de un acces la aceeaşi dată a unui alt procesor, se separă ca mai jos:

WRITE (X)
|
|
UNLOCK(S)
|
|
LOCK(S)
|
|
ACCESS(X)

În acest caz, gestiunea consistenţei este lăsată în seama programatorului, a Sistemului de Operare. Situaţiile în care variabilele partajate pot fi actualizate fără ordonarea operaţiilor impusă prin sincronizare sunt numite „data races", deoarece rezultatul execuţiei depinde de viteza procesoarelor fiind astfel impredictibil. Programele sincronizate vor fi considerate astfel ca fiind „data-race-free" [Hen02].

2.3. Sincronizarea proceselor

Cheia sincronizării proceselor în SMA (Shared Memory Architectures) este dată de implementarea unor aşa-zise procese atomice. Un proces atomic reprezintă un proces care odată iniţiat, nu mai poate fi întrerupt de către un alt proces (se spune că procesul se află într-o secţiune critică). Spre exemplu, să considerăm că pe un sistem multiprocesor cu memorie partajată se execută o aplicaţie care calculează o sumă globală prin nişte sume locale calculate pe fiecare procesor, ca mai jos:

```
LocSum = 0;
For i =1 to Max
    LocSum = LocSum + LocTable[i]; // secvenţa executată în paralel de
                                   // către fiecare procesor!
```

Proces | LOCK
Atomic | GlobSum = GlobSum + LocSum; // secțiunea critică
| UNLOCK

Procesul LOCK/UNLOCK este atomic, în sensul că numai un anumit procesor execută acest proces la un moment dat. În caz contrar s-ar putea obține rezultate hazardate pentru variabila globală "GlobSum". Astfel de exemplu, P1 citeşte GlobSum = X, P2 la fel, apoi P1 scrie GlobSum = GlobSum + LocSum1 iar apoi P2 va scrie GlobSum = GlobSum + LocSum2 = X + LocSum2 (incorect !). Este deci necesar ca structura hardware să poată asigura atomizarea unui proces software.

O implementare a acestor procese atomice este realizată în felul următor. Se porneşte de la constatarea faptului că instrucțiunile atomice tip "Read - Modify - Write" sunt dificil de implementat, necesitând ajutorul hardware - ului. O alternativă o constituie perechea de instrucțiuni "load locked" ("ll") şi "store-ul condiționat" ("sc"). Aceste instrucțiuni sunt utilizate în secvența: dacă conținutul locației de memorie specificată de "ll" se modifică înainte de apariția unui "sc" la aceeaşi locație de memorie, atunci acesta ("sc") nu se execută propriu-zis. La fel, dacă procesorul procesează o comutare de context (ex. CALL/RET, întreruperi etc.) între cele 2 instrucțiuni, de asemenea "sc"-ul practic nu se face. Practic instrucțiunea condiționată "sc Reg, Adresă", returnează în registrul "reg" '1' sau '0' după cum s-a făcut sau nu s-a făcut. Cu aceste precizări, se prezintă implementarea unei operații atomice de tipul "fetch and increment". Ea returnează valoarea unei locații de memorie şi o incrementează atomic. Iată mai jos un "fetch and increment" atomic, implementat prin mecanismul "ll/sc":

```
rep:    ll R2, 0(R1)
        add R2,R2,#1
        sc R2, 0(R1)
        begz R2,rep
```

2.3.1. Atomizări şi sincronizări

În cazul proceselor de sincronizare, dacă un anumit procesor "vede" semaforul asociat unei variabile globale pe '1' (LOCK - ocupat), are 2 posibilități de principiu:

- Să rămână în bucla de testare a semaforului până când acesta devine '0' (UNLOCK) - strategia

"spin lock"

- Să abandoneze intrarea în respectivul proces, care va fi pus într-o stare de aşteptare şi să inițieze un alt proces disponibil - comutare de task-uri.

Prima strategie, deşi des utilizată, poate prelungi mult alocarea unui proces de către un anumit procesor. Pentru a dealoca un proces (variabila aferentă), procesorul respectiv trebuie să scrie pe '0' semaforul asociat. Este posibil ca această dealocare să fie întârziată datorită faptului că simultan, alte N procesoare doresc să testeze semaforul în vederea alocării resursei (prin bucle de tip Read - Modify - Write). Pentru evitarea acestei deficiențe este necesar ca o cerere de bus de tip "Write" să se cableze ca fiind mai prioritară decât o cerere de bus în vederea unei operații tip "Read - Modify - Write". Altfel spus, dealocarea unei resurse este implementată ca fiind mai prioritară decât testarea semaforului în vederea alocării resursei.

A doua strategie prezintă deficiențe legate în special de timpii mari determinați de dealocarea/ alocarea proceselor (salvări/restaurări de contexte).

În ipoteza că într-un SMA nu există mecanisme de menținere a coerenței cache-urilor, cea mai simplă implementare a verificării disponibilității unei variabile globale este următoarea (spin lock):

```
            li R2, #1; R2 <- ‘1’
test:   lock exchg R2,0(R1)          ; atomică
            bnez R2,test
```

Dacă însă ar exista mecanisme de coerență a cache-urilor, semaforul ar putea fi ataşat local. Primul avantaj ar consta în faptul că testarea semaforului ('0' sau '1') s-ar face din cache-ul local fără să mai fie necesară accesarea busului comun. Al doilea avantaj - de natură statistică - se bazează pe faptul dovedit, că e probabil ca într-un viitor apropiat procesorul respectiv să dorească să testeze din nou semaforul (localitate spațială şi temporală).

2.3.2. Sincronizarea la barieră

Este o tehnică de sincronizare deosebit de utilizată în programele cu bucle paralele. O barieră forțează toate procesele să aştepte până când toate au atins bariera, abia apoi permițându-se continuarea acestor procese. O implementare tipică a unei bariere poate fi realizată prin 2 bucle succesive: una atomică în vederea incrementării unui contor sincron cu fiecare proces care ajunge la barieră iar cealaltă în vederea menținerii în aşteptare a proceselor până când aceasta îndeplineşte o

anumită condiție (test); se va folosi funcția "spin (cond)" pentru a indica acest fapt.

Se prezintă implementarea tipică a unei bariere:

```
        LOCK(counterlock)
/*Proces*/ if(count == 0) release = 0; // şterge release la început
/*atomic*/ count++;              // contorizează procesul ajuns la barieră
        UNLOCK(counterlock)
     if(count == total) { // toate procesele au ajuns la barieră
              count = 0;
              release = 1;
     } else { // mai sunt procese de ajuns
          spin(release == 1); // aşteaptă până ce ajunge şi ultimul
     }
```

"total" – nr. maxim al proceselor ce trebuie să atingă bariera
"release" – utilizat pentru menținerea în aşteptare a proceselor la barieră

Există totuşi posibilitatea de exemplu, ca un procesor (proces) să părăsească bariera înaintea celorlalte care ar sta în bucla "spin (release==1)" şi ar rezulta o comutare de task-uri chiar în acest moment. La revenire vor vedea "release==0" pentru că procesul care a ieşit a intrat din nou în barieră. Rezultă deci o blocare nedorită a proceselor în testarea "spin".

Soluția în vederea eliminării acestui hazard constă în utilizarea unei variabile private asociate procesului (local_sense). Bariera devine astfel:

```
local_sense = !local_sense;
LOCK(counterlock);
count++;
UNLOCK(counterlock);
if(count == total) {
         count = 0;
         release = local_sense;
} else {
         spin(release == local_sense);
}
```

Dacă un proces iese din barieră urmând ca mai apoi să intre într-o nouă instanță a barierei, în timp ce celelalte procese sunt încă în barieră (prima instanță), acesta nu va bloca celelalte procese întrucât el nu resetează variabila "release" ca în implementarea anterioară a barierei. Din punctul de vedere al programatorului secvențial, prima implementare a barierei era corectă.

3. Introducere în modelarea la nivel de tranzacţii

De-a lungul evoluției industriei microelectronicii, dezvoltatorii de sisteme dedicate (embedded systems) au încercat continuu să-şi îmbunătățească productivitatea pentru a putea exploata numărul de tranzistori de pe un chip, număr aflat într-o continuă creştere.

Răspunsul pentru această provocare a fost de a creşte gradul de abstractizare folosit pentru implementarea sistemelor dedicate. De la tranzistori la porți logice, de la porți logice la RTL[17], productivitatea designului a fost suficient de ridicată pentru a putea menține pasul cu evoluția tehnologică.

Designul sistemelor dedicate folosind RTL nu poate însă să gestioneze complexitatea sistemelor care integrează 500 de milioane de tranzistori, printr-un proces tehnologic de 90 nm. Pentru a rezolva această problemă, creată de diferențele tot mai mari care se manifestă între productivitatea desginului şi capacitatea proceselor tehnologice, sau adoptat două direcții majore:

- creşterea nivelului de abstractizare pentru a specifica şi modela sisteme dedicate;
- adoptarea unei paradigme noi de design.

Modelarea la nivel de tranzacții (TLM – Transaction Level Modeling) cu SystemC[18] reprezintă o abordare care adresează prima din cele două direcții. Această soluție, a cărei succes a fost dovedit în industrie, rezolvă problemele critice apărute în designul unui sistem dedicat.

În anul 1965, Gordon Moore a prezis faptul că numărul de tranzistori pe care un circuit integrat îi va conține se va dubla o dată la fiecare 18 luni. Această predicție, cunoscută sub numele de „Legea lui Moore”, a fost uimitoare, dovedindu-se mai precisă decât până şi Moore ar fi crezut.

Creşterea exponențială a complexității circuitelor integrate, aşa cum legea lui Moore o specifică, a dus la apariția, în cadrul industriei semiconductorilor, a sistemelor dedicate, numite şi System-on-Chip (SoC).

System-on-Chip reprezintă conceptul prin care diferite componente electronice sunt integrate pe un singur chip, pentru a forma un sistem electronic complet. Acest concept este posibil datorită progreselor uluitoare care s-au făcut în domeniul circuitelor integrate, ducând procesul de

17 Register Transfer Level – o modalitate de descriere a operațiilor unui circuit digital sincron; comportamentul circuitului este definit prin modul în care semnalele circulă între regiştri hardware şi prin operațiile logice care se efectuează asupra semnalelor

18 SystemC este un set de rutine şi macrouri implementate în C++, care permite simularea proceselor concurente, fiecare proces fiind descris prin sintaxă C++

fabricare al acestora la scară nanometrică, spre nanotehnologie.

Un SoC conține atât componente hardware cât şi părți software, fiind alcătuit din blocuri reutilizabile, create pentru a realiza nişte sarcini specifice. Astfel de sisteme se folosesc tot mai des în zilele noastre putând fi întâlnite spre exemplu în camere digitale, telefoane mobile etc..

3.1. Designul clasic al SoC

Dezvoltarea unui circuit integrat se realizează prin utilizarea explicită a unei combinații de unelte de design electronic automat (EDA – Electronic Design Automation) şi constituie o metodologie inginerească riguroasă, un proces de concepere, verificare, validare şi livrare, a designului circuitului integrat, în vederea producerii acestuia, la un nivel foarte ridicat din punct de vedere calitativ.

În mod tradițional, pentru realizarea sistemelor electronice dedicate, se apelează la procesul de design ilustrat în figura de mai jos. Un astfel de model pleacă de la o privire de ansamblu preluată din specificațiile sistemului, după care se împarte în două căi de activități, complet separate:

- dezvoltarea componentei hardware a sistemului;
- dezvoltarea componentei software a sistemului.

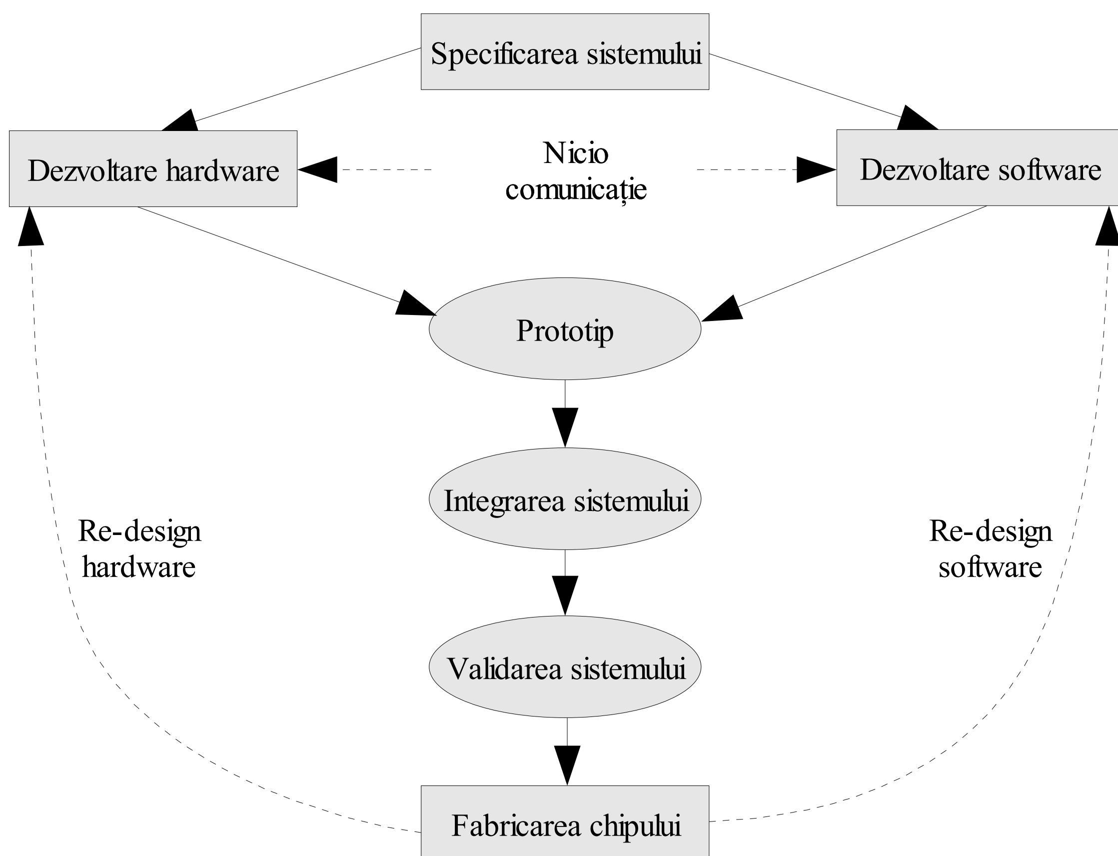


Fig. 13: Modelul de design clasic

Se poate observa că nu există nicio comunicație între cele două căi principale ale procesului de design. Ambele componente, hardware şi software, se dezvoltă separat, până când un prototip al sistemului este disponibil.

Modelarea hardware-ului începe cu dezvoltarea codului RTL. Această etapă presupune descrierea modelelor hardware folosind limbaje de descriere hardware, precum VHDL sau Verilog. Modelele create astfel sunt verificate funcțional prin simulări care să ateste corectitudinea comportamentului acestora. Apoi, se face o sinteză care să ducă la obținerea listei de componente logice si conexiunile dintre acestea (logic netlist). După ce aceasta este realizată, are loc dispunerea componentelor pe placă (pe baza "gradului de conexiune" între modulele componente, in vederea optimizării dispunerii acestora), utilizarea algoritmilor de rutare pentru realizarea conexiunilor dintre componente, analize de timing şi alte operații care duc până la o verificare fizică a circuitului integrat proiectat. În acest moment, designul hardware este gata, fabricarea unui prototip al sistemului poate începe.

Modelarea software-ului pentru sistemul dedicat se face separat de partea hardware. Inginerii software dezvoltă codul necesar sistemului fără a discuta cu inginerii hardware. Cu toate că partea software ar putea începe mai devreme în acest fel, testarea codului necesită interacțiunea cu componentele hardware, fapt care presupune maparea codului RTL pe un emulator sau pe un sistem FPGA prototip. Acest proces este însă unul foarte costisitor pentru că presupune utilizarea unor echipamente scumpe. O altă abordare, mai rea, este de a aştepta după chip de test obținut după fabricarea prototipului propus de modelarea hardware.

Concluzia este că software-ul este întotdeauna validat mai târziu decât hardware-ul.

Odată ce prototipul sistemului este gata, componenta software este integrată în acesta, urmând etapele de integrare şi validare ale sistemului. Dacă apar erori, fie în hardware, fie în software, procesul de design va fi reiterat, până când comportamentul sistemului va fi cel aşteptat.

Începând cu anii '90, s-a folosit aşa numitul proces de „co-verificare" (hardware-software codesign), care permitea simularea hardware-ului RTL împreună cu software-ul dedicat. Simularea rula însă la viteza mică a RTL (sute de cicli bus pe secundă). Astfel, prin această metodă se puteau rula numai părți mici ale codului software. Având în vedere că aplicațiile software devin din ce în ce mai complexe, co-verificarea nu mai reprezintă o variantă fezabilă. Industria SoC are nevoie de o co-simulare a hardware-ului şi a software-ului, care să permită în esență simularea hardware-ului la o viteză mai mare.

Modelul clasic de design al unui SoC nu mai este o variantă fezabilă în momentul de față datorită complexității existente. Abordarea clasică, prin care echipe separate lucrează la diverse

modele incoerente, a trebuit să fie înlocuită de o abordare nouă, care să permită legarea diverselor etape din cadrul procesului de design, printr-o metodologie centralizată, în vederea unei modelări la nivel de sistem.

3.2. Modelarea la nivel de tranzacţii

Soluția poartă numele de Modelare la Nivel de Tranzacții (Transaction Level Modeling – TLM). Această abordare este bazată pe evenimente provenite din specificațiile hardware şi software ale sistemului, reprezentând o soluție viabilă pentru designul la nivel de sistem.

TLM se bazează pe limbaje de programare de nivel înalt, aşa cum este SystemC şi pune în evidență conceptul de separare a comunicației, față de procesul computațional din cadrul unui sistem.

În abordarea TLM, componentele sunt modelate ca şi module care pot executa o mulțime de procese concurente, care calculează şi le reprezintă comportamentul. Aceste module comunică prin mesaje sub formă de tranzacții folosind canale de comunicație abstracte. Interfețele TLM sunt implementate în cadrul canalelor de comunicație pentru a încapsula protocoale de comunicație. Pentru a putea comunica, un proces trebuie doar să acceseze aceste interfețe, prin porturile pe care le are modulul cu care se doreşte comunicarea. Interfețele joacă aşadar un rol esențial în filozofia TLM, deoarece permit separarea comunicației de partea computațională din cadrul sistemului.

TLM defineşte o tranzacție ca fiind acel transfer de date (comunicație) sau sincronizare dintre două module la un moment dat (adică un eveniment SoC determinat de specificațiile hardware şi software ale sistemului). Poate fi orice structură, de cuvinte sau biți. Definiția tranzacției poate fi redefinită ca un proces care este „conştient" de protocolul de la nivelul unei magistrale (bus). Spre exemplu, tranzacția poate include informație cu privire la lățimea de bandă a busului sau capacitatea acestuia de a trimite date în rafală (burst). Astfel de detalii sunt extrem de utile arhitecților care doresc să realizeze o analiză fină a arbitrărilor care care au loc între modulele interconectate, din cadrul unui SoC.

Pe parcursul întregului proces de design al SoC, TLM este referința unică dintre diferitele echipe de lucru, implicate în următoarele activități esențiale:

- dezvoltarea rapidă, mai devreme a părții software;
- analiza arhitecturii sistemului;
- verificarea funcțională.

Apariția TLM a contribuit esențial la crearea unui nou ciclu de dezvoltare, pentru designul sistemelor dedicate. Figura următoare prezintă noul ciclu de dezvoltare apărut.

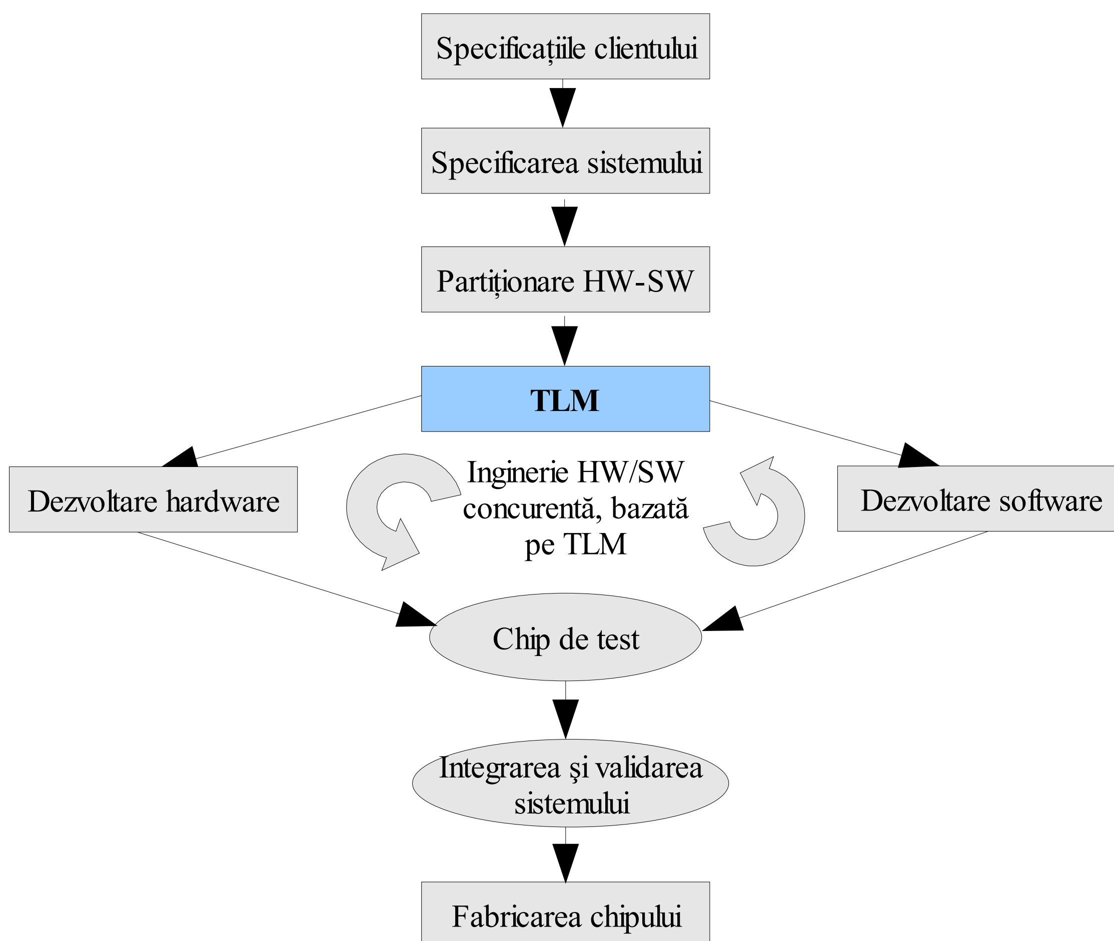


Fig. 14: Modelul de design nou, bazat pe tranzacții

Designul unui SoC porneşte de obicei de la specificațiile clientului, etapă în care cerințele sistemului trebuie bine identificate. Pe baza acestor cerințe, rezultă o specificare preliminară a sistemului, cu ajutorul căreia arhitecții hardware şi software realizează o partiționare a sistemului în cele două mari componente.

Urmează etapa TLM. După ce s-a realizat o platformă TLM, se intră cu ambele procese (hardware şi software) într-o etapă de dezvoltare concurentă. În această fază, TLM serveşte pe post de referință unică pentru echipele software şi hardware, permițând o dezvoltare mai devreme a software-ului şi o mai bună viziune asupra arhitecturii sistemului. De asemenea, TLM foloseşte procesului de testare al sistemului dedicat.

Rezultatul este obținerea unui veritabil co-design, hardware/software, aceasta fiind una dintre diferențele remarcabile între modelul tradițional şi cel nou.

Atunci când primul chip de test este gata şi componentele software, precum drivere,

firmware, vor fi terminate. Atât partea hardware, cât şi cea software, vor fi mai bine verificate şi integrate, astfel că probabilitatea, ca sistemul să fie funcțional din prima încercare, creşte.

3.3. TLM – metoda optimă de modelare a sistemelor on-chip (SoC) complexe

În cadrul procesului de modelare a unui sistem dedicat (SoC), se pune problema alegerii acelei metodei care să permită o modelare optimă, din punctul de vedere al balansării celor două criterii importante care stabilesc performanța modelului simulat: viteza şi acuratețea.

O metodă de modelare este aceea care presupune folosirea unui model algoritmic, analitic, care are ca avantaj major faptul că permite o viteză de simulare foarte ridicată. Dezavantajul acestei metode este acela că un model algoritmic nu conține detalii de implementare, nu are nicio informație despre componentele hardware şi software ale sistemului.

În cealaltă extremă se află metoda modelării pur logice, care are loc la nivelul transferului între regiştri (RTL). Modelele RTL se descriu prin limbaje de descriere hardware (precum VHDL) având avantajul unui grad ridicat de fidelitate, comparativ cu implementarea reală a hardware-ului. Această metodă permite deci realizarea unei analize precise a sistemului modelat, din punct de vedere al performanței, al funcționalității. Se poate observa că această metodă este complementară primei metode, ținând cont şi de faptul că viteza de simulare este foarte scăzută în cazul modelării pur logice.

Modelarea la nivel de tranzacții este o soluție care încearcă să balanseze cele două abordări extreme amintite mai sus, ținând cont de următoarele criterii fundamentale:

- viteză: modelul trebuie să simuleze miliarde de cicli pentru o testare completă sau măcar reprezentativă a sistemului, iar acest lucru trebuie să poată fi realizat într-un timp acceptabil;
- acuratețe: modelul trebuie să păstreze suficiente detalii ale sistemului pentru ca rezultatele simulărilor să poată fi considerate ca fiind de încredere;
- simplitate: este de preferat ca modelul să fie cât mai uşor şi rapid realizat, cu un efort minim.

TLM se plasează undeva între modelarea precisă, la nivel de ciclu şi modelarea algoritmică, care permite viteză ridicată de simulare, dar căreia îi lipseşte acuratețea simulării la nivel de tact, de tip execution-driven.

Modelarea la nivel de tranzacții realizează un balans între viteză şi acuratețe, fiind o excelentă platformă complementară platformei RTL. În plus TLM are avantajul flexibilității, o modelare la acest nivel putând fi apropiată sau îndepărtată de oricare dintre extreme. Spre exemplu,

se poate trece cu uşurinţă de la o modelare care să nu conţină informaţii de timing, fiind mai simplă şi mai rapidă, la o modelare care să adauge şi astfel de informaţii, apropiindu-se deci de modelarea la nivel de ciclu, dar fără a fi la fel de lentă ca aceasta.

Figura de mai jos [Ghe05], prezintă o comparaţie între trei metode de modelare: cea pur logică (RTL – Register Transfer Level), cea cu precizie la nivel de ciclu (CA – Cycle Accurate) şi cea la nivel de tranzacţii (TLM – Transaction Level Modeling). Se poate observa o eficienţă a TLM net superioară celorlalte două metode, atât din punctul de vedere al simulării, cât şi din punctul de vedere al timpului necesar modelării propriu zise.

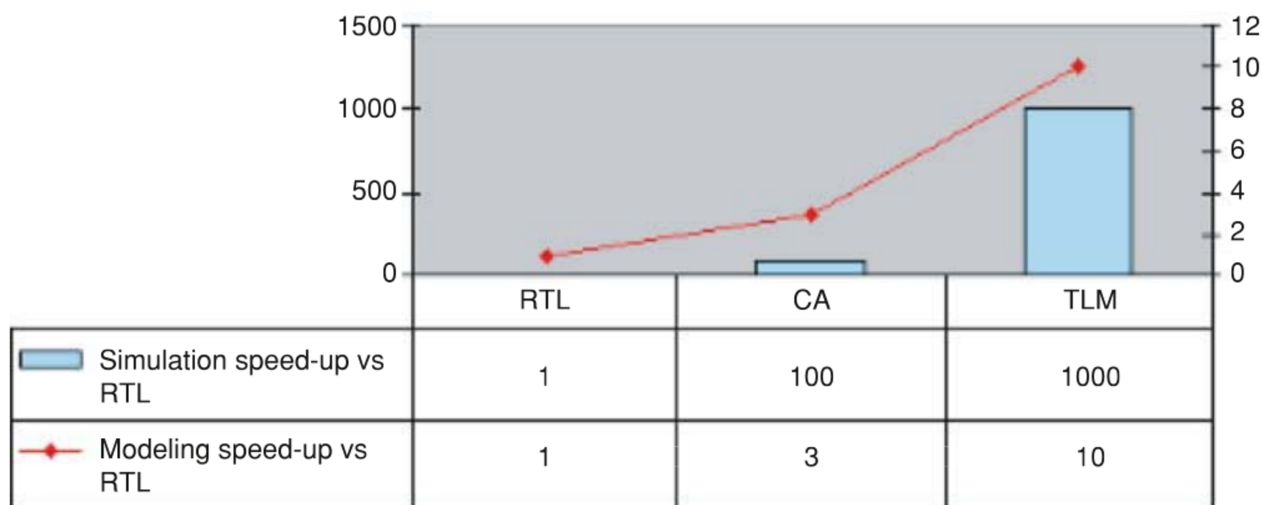


Fig. 15: Eficienţa mai multor metode de modelare: RTL, CA (cycle accurate) şi TLM

3.4. Principii TLM

3.4.1. Terminologie

TLM modelează fiecare componentă din cadrul unui sistem electronic sub forma unui **modul**. Starea internă a componentei este încapsulată la nivelul modulului printr-un set de variabile, iar comportamentul componentei este reprezentat prin **procese concurente** sau **fire de execuţie**, care pot fi executate în paralel.

Comunicaţiile dintre module se realizează prin **interconexiuni** sau **canale**. În funcţie de gradul de acurateţe necesar modelului, un canal poate fi reprezentat de un simplu ruter, de un bus, de o reţea de interconectare, sau de alte structuri. Prin intermediul canalelor de comunicaţie are loc separarea părţii de comunicare, de partea computaţională.

Conectarea modulelor şi a canalelor de comunicaţie se face prin intermediul **porturilor**.

O dată ce conexiunile s-au realizat, informația poate circula în cadrul sistemului modelat. **Tranzacția** este mulțimea datelor care se transmit de la un modul la altul. Un modul **inițiator** este acel modul care inițiază o tranzacție în cadrul sistemului, iar un modul **țintă** (destinație) este cel care recepționează tranzacția şi emite un răspuns corespunzător emițătorului. Oricare două tranzacții consecutive pot avea dimensiuni diferite, în funcție de cantitatea de date vehiculată între două evenimente de sincronizare a sistemului.

Sincronizarea sistemului este o acțiune explicită care are loc între două module comunicante care trebuie să se coordoneze (lucru absolut necesar pentru a putea asigura un caracter predictibil, determinist al sistemului).

3.4.2. Procesul de modelare

Un sistem complet este construit cu ajutorul TLM prin conectarea modulelor şi canalelor de comunicație între ele. După ce platforma este integrată, simularea sistemului are loc prin rularea componentei software, fie nativ, fie prin cross-compilare.

Pentru a putea asigura o funcționare corectă a simulării sistemului, se urmăresc două aspecte majore:

- toate tranzacțiile trebuie să fie blocante (inițiatorul îşi va continua execuția numai dacă tranzacția s-a terminat);
- toate evenimentele de sincronizare ale sistemului sunt potențiale puncte de planificare a rulării mediului de simulare, pentru a putea asigura o simulare precisă a concurenței.

Sincronizarea sistemului se poate face prin evenimente, semnale, întreruperi sau chiar prin polling[19]. Oricum s-ar realiza, trebuie precizat faptul că esența modelării la nivel de tranzacții constă în stabilirea când şi unde trebuie efectuate sincronizări ale sistemului simulat. Dacă se introduc prea multe puncte de sincronizare, modelul tinde să se apropie prea mult de variantele lui de la nivel de ciclu, sau chiar RTL, iar viteza simulării scade prea mult. Pe de altă parte, dacă se implementează prea puține puncte de sincronizare, se poate ajunge la o execuție incorectă a sistemului.

3.4.3. Acurateţea modelului

Acuratețea modelului marchează precizia şi corectitudinea cu care este construită varianta virtuală, de simulat, a sistemului real. Există doi factori majori care specifică acuratețea

19 O metodă de control al comunicației, folosită în sistemele de calcul, care presupune existența unui controller care verifică toate perifericele ataşate unui mediu comun de transmisie, pe rând, dacă au sau nu informație de trimis sau de recepționat

modelării unui SoC:
- granularitatea datelor comunicate;
- acuratețea temporală a sistemului.

Prin acuratețea temporală a sistemului se înțelege fidelitatea cu care este modelat comportamentul în timp al modelului. Astfel, modelul poate să nu conțină absolut nicio informație despre comportamentul său în timp (cât timp durează o operație de exemplu), sau poate să ajungă până la a include toate detaliile din punct de vedere temporal. În acest caz sistemul va evolua cu o precizie mare, la nivel de ciclu.

Evident, cele două situații prezintă abordările extreme ale acurateții temporale. Orice nivel situat între cele două extreme este considerat a avea un nivel aproximativ de precizie temporală.

Din acest punct de vedere, modelarea TLM propune două clase fundamentale (pentru care vom şi păstra terminologia din limba engleză, din motive de concizie):
- Untimed TLM;
- Timed TLM.

Varianta **Untimed TLM** este un model destinat fazelor inițiale ale dezvoltării sistemului, când viteza ridicată de simulare este principalul scop, iar informațiile de timp nu sunt deci obligatorii. Un astfel de model serveşte în primul rând programatorilor deoarece aceştia vor putea să realiza o modelare arhitecturală fără a fi nevoiți să intre în detalii legate partea hardware.

Într-un sistem de tip Untimed TLM nu există informații legate de timp pentru arhitectura simulată, nu există semnal de ceas. Procesele se execută concurent pentru a putea accesa resursele sistemului în acelaşi moment de timp. Cu toate acestea, un sistem de tip Untimed TLM trebuie să garanteze un comportament funcțional adecvat, o anumită ordine în cadrul execuției paralele a proceselor concurente. Acest lucru este îndeplinit în principal prin următoarele caracteristici:
- respectarea dependențelor cauzale dintre procese, folosind sincronizări la nivel de sistem;
- execuție concurentă a proceselor independente.

Pentru a putea asigura un comportament deterministic al sistemului, relațiile de cauzalitate dintre procese trebuie respectate. Acest lucru se realizează prin intermediul unei sincronizări explicite, la nivel de sistem. Aceasta asigură o desfăşurare a proceselor care implică o ordine parțială în cadrul execuției acestora. Toate dependențele dintre procese sunt asigurate, iar orice ordine de execuție particulară este permisă cât timp relațiile de dependență dintre procesele concurente sunt îndeplinite.

Sincronizarea la nivel de sistem poate permite, de exemplu, asigurarea consistenței datelor, a memoriei.

De asemenea, prin sincronizarea la nivel de sistem, se creează o metodă de validare eficientă a sistemului, prin modelul construit pentru simulare, deoarece se pot testa mai multe variante de execuție concurentă a proceselor care caracterizează sistemul.

În ceea ce priveşte modalitatea de execuție concurentă a proceselor, fără a include informații de timp, merită menționat că nucleul de simulare nu este preemptiv. În acest fel se asigură un comportament predictibil al stării proceselor şi al controlului acestora. Execuția sistemului va fi independentă de o implementare anume a nucleului de simulare.

La nivelul modelului Untimed TLM pot fi introduse **întârzieri funcționale**, pentru că de multe ori, astfel de întârzieri fac parte din specificațiile sistemului şi deci nu pot fi ignorate (spre exemplu: un decodor video procesează 30 de cadre pe secundă). Astfel de „timpi impliciți" introduc constrângeri în plus pentru execuția proceselor din cadrul sistemului modelat şi reduc mulțimea de posibile execuții întrețesute a proceselor.

Întârzierile funcționale trebuie introduse în aşa fel încât sincronizarea sistemului să poată cuprinde toate relațiile de dependență din sistem. În acest fel se asigură stabilitatea sistemului modelat (nu contează de exemplu că frecvența de tact poate varia sau că latența unei tranzacții pe bus depinde de gradul de încărcare al busului).

Modelul Untimed TLM cu întârzieri funcționale este aşadar un intermediar între modelul Untimed TLM pur şi modelul Timed TLM.

Modelarea de tip **Timed TLM** include informații legate de timp, cu privire la comportamentul modulelor şi a comunicațiilor dintre acestea. Astfel, modelarea în această variantă este una micro-arhitecturală, fiind mai puțin abstractă, orientându-se spre un nivel de acuratețe suficient de ridicat, care să poată corespunde sistemului real.

Principalele obiective în cadrul modelării de tip Timed TLM includ:

- măsurarea performanței micro-arhitecturii;
- optimizarea micro-arhitecturii prin varierea parametrilor acesteia;
- optimizarea componentei software pentru a putea satisface cerințele de timp real ale micro-arhitecturii corespunzătoare.

Pentru a realiza un model care să conțină informații legate de timp, se au în vedere două aspecte: întârzierile computaționale şi întârzierile de comunicație. Prin întârzieri computaționale se înțelege timpul necesar componentelor sistemului pentru a putea realiza funcțiile specifice (calculele). Prin întârzieri de comunicație se înțelege timpul consumat pentru a accesa şi

transfera (tranzacționa) date.

Trecerea timpului în cadrul unei componente a sistemului poate fi modelată în filozofia TLM în două feluri: prin inserarea informației de timp în cadrul modelului de tip Untimed TLM (adnotare), sau prin folosirea unui model, cu informații legate de timp, de sine stătător.

Aşadar, modelarea în variantă Timed TLM poate fi făcută în două feluri. Prima variantă prezentată mai sus se bazează pe reutilizarea modelului Untimed TLM construit anterior, la care ar trebui, cel puțin teoretic, să fie inserate numai informații de timing.

Cealaltă abordare presupune crearea unui model separat, care să modeleze evoluția în timp a sistemului. Comportamentul în timp presupune modelarea de întârzieri care sunt calculate în timpul execuției modelului. Un astfel de model este separat de modelul funcțional al sistemului, având încorporate informațiile de timing. Se pretează cel mai bine atunci când structura funcțională, algoritmic modelată, diferă mult de structura reală, de la nivel micro-arhitectural.

În figura de mai jos [1] este ilustrată modelarea Untimed TLM alături de modelarea Timed TLM. Modelul UTLM (Untimed TLM) reprezintă comportamentul funcțional al componentei simulate. Execuția UTLM are loc până când se ajunge la un punct de sincronizare. Apoi, se simulează modelul TTLM (Timed TLM) care va rula toate întârzierile în timp care sunt asociate componentei funcționale care tocmai a fost simulată. Tot acum, se simulează şi întârzierile cauzate de comunicații. După ce toate întârzierile au fost simulate, modelul UTLM îşi reia execuția.

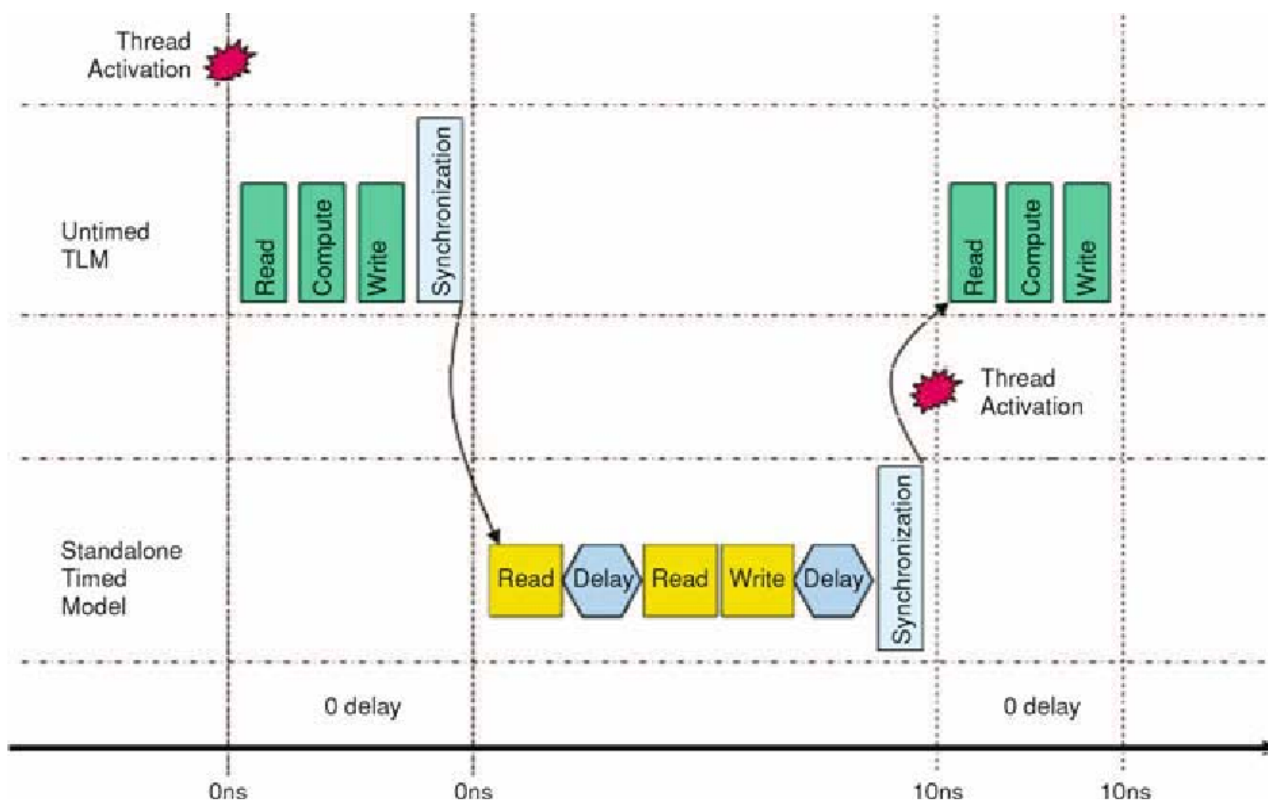


Fig. 16: Exemplu de execuție combinată: Untimed TLM cu Timed TLM

3.5. Metodologia UNISIM pentru Modelarea la Nivel de Tranzacţii

În metodologia TLM, interfețele au un rol foarte important, deoarece ele sunt cele care realizează separarea comunicației dintre module de implementarea acestora şi de partea lor computațională.

Prezentăm în cele ce urmează care este interfața folosită de UNISIM [UNI08] în procesul de modelare la nivel de tranzacții [Per07].

```
template <typename REQ, typename RSP>
class TlmSendIf : public virtual sc_interface
{
public:
   virtual bool Send(const Pointer<TlmMessage<REQ, RSP> > &message) = 0;
};
template <typename REQ, typename RSP>
class TlmMessage
{
public:
  Pointer<REQ> req;
  Pointer<RSP> rsp;
private:
  stack<sc_event *> event_stack;
};
```

Fig. 17: Interfața TLM folosită în UNISIM

Inițiatorul comunicației apelează metoda **Send** atunci când doreşte să realizeze o tranzacție cu un alt modul. Metoda Send returnează un rezultat de tip logic (boolean), care specifică dacă cererea poate sau nu să fie îndeplinită. În cazul în care cererea nu poate fi satisfăcută, modulul apelant are sarcina de a încerca trimiterea cererii mai târziu. Singurul parametru al metodei Send este mesajul, trimis de la modulul inițiator, la modulul țintă. Trebuie precizat faptul că mesajul este transmis prin referință şi nu prin copiere, pentru ca, atât modulul apelant cât şi modulul apelat să partajeze mesajul.

Mesajul este definit prin intermediul clasei generice **TlmMessage** şi conține trei elemente:

- cererea;
- răspunsul;
- o stivă de evenimente.

Inițiatorul are sarcina de a pune cererea în cadrul mesajului înainte de a apela Send. Dacă inițiatorul aşteaptă vreun răspuns din partea modulului apelat, va pune şi un eveniment pe stiva de evenimente şi va aştepta până când acel eveniment va fi notificat de către modulul apelat. Utilizarea conținutului răspunsului este aşadar validă numai după ce evenimentul asociat este notificat.

Atât cererea cât şi răspunsul pot avea orice tip de date, fiind definite generic.

Partajarea mesajului permite evitarea schimbului de informații între modulul inițiator şi modulul țintă. În acest fel, viteza simulării creşte, dar pe de altă parte, complică sarcina modulului

țintă: mesajul, dar mai ales partea lui de cerere, trebuie să rămână disponibil, alocat în memorie, până când modulul țintă se va folosi de cerere. Spre exemplu, chiar dacă inițiatorul nu aşteaptă după un răspuns (o scriere în memorie din partea procesorul de exemplu), el nu poate dealoca mesajul din memorie odată ce metoda Send (care este neblocantă) a returnat rezultatul, pentru că nu ştie dacă modulul țintă a preluat şi a terminat de utilizat mesajul. Această problemă nu ar putea fi rezolvată decât în situația în care programatorul are o bună şi completă înțelegere a întregului simulator, al fiecărui modul, iar acest lucru este evident complet nebenefic într-un mediu de simulare care mizează pe modularitate şi reutilizabilitate (aşa cum este UNISIM).

Din acest motiv, autorii UNISIM au introdus un **Garbage Collector**, care are rolul de a prelua sarcina dealocării mesajelor. Aceasta se va face astfel automat, fără a fi necesară intervenția programatorului. Garbage Collectorul are capacitatea de a aloca rapid memorie şi gestionează toate referințele la obiecte, folosind câte un contor pentru fiecare referință. Acest contor este incrementat de fiecare dată când apare o nouă referință către obiectul în cauză şi este decrementat atunci când o referință dispare. Utilizarea Garbage Collectorului este foarte facilă: o clasă **Pointer** este disponibilă, utilizarea ei implicând beneficierea de avantajele Garbage Colllectorului. Crearea de astfel de pointeri se face uşor, datorită supraîncărcării operatorului new. Exemplul următor ilustrează acest lucru:

```
Pointer<int> p_int = new(p_int) int(1);
```

Fig. 18: Exemplu de utilizare a clasei Pointer (şi implicit a Garbage Collectorului)

În continuare vom justifica utilizarea unei stive de evenimente. De cele mai multe ori un singur eveniment este suficient: acesta ar asigura o schemă de comunicare simplă, în care sunt implicate numai două evenimente. Însă, există şi multe situații în care sunt necesari mai mulți paşi pentru a realiza o comunicare. Spre exemplu, aşa cum figura următoare o arată, comunicarea între modulul A şi modulul C trebuie să se facă prin intermediul modului B (putem face analogia cu un model de procesare de tip NUMA, în care modulul A este un procesor, modulul C este o memorie, iar modulul B este un switch al rețelei de interconectare).

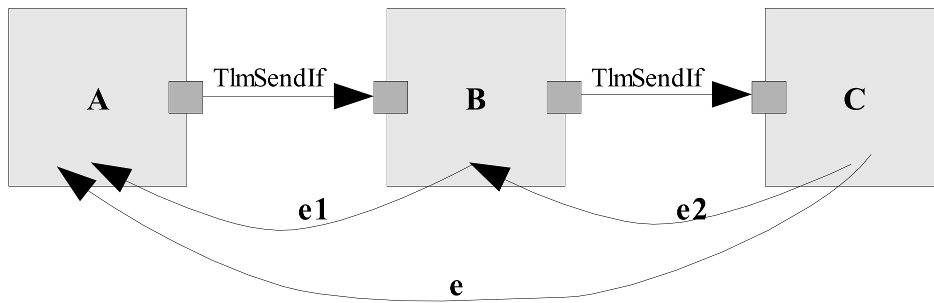


Fig. 19: Înlănțuirea comunicației

Pe exemplul ilustrat mai sus, se poate observa că timpul petrecut în modulul B afectează timpul total necesar tranzacției dintre modulele A şi C. Dacă se foloseşte un singur eveniment (e), timpul computațional care se consumă în modulul B nu va putea fi ştiut, deoarece răspunsul va fi direct notificat din modulul C, pentru modulul A. Este deci necesar un eveniment adițional. Vom avea pe acest exemplu două evenimente (e1 şi e2): modulul B va aştepta după modulul C (prin e2), iar modulul A va aştepta după modulul B (prin e1), nu direct după modulul C.

Pe caz general, o stivă de evenimente este folosită pentru a uşura sarcina programatorului atunci când se realizează o comunicație înlănțuită.

3.5.1. Untimed TLM (UTLM) în mediul UNISIM

În modelarea UTLM, întârzierile de timp, care pot apare între momentul trimiterii unei cereri emise de către un modul inițiator, până în momentul în care un răspuns este primit din partea modulului țintă, nu sunt considerate.

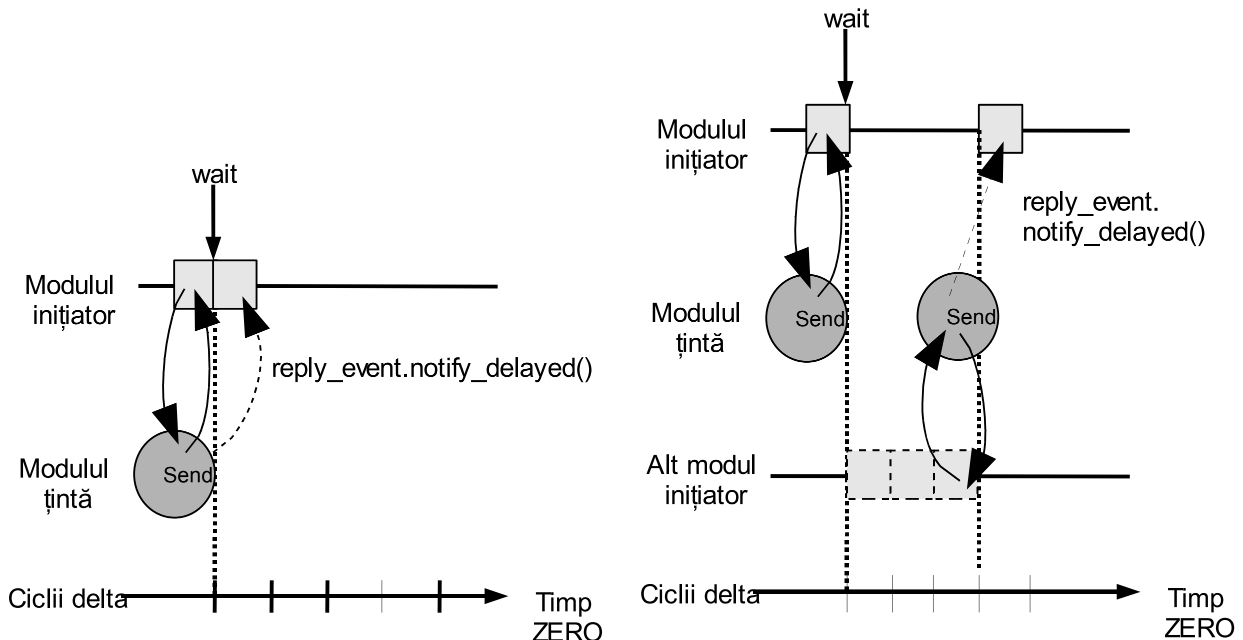


Fig. 20: Principiul UTLM în UNISIM

Se poate observa în figura de mai sus faptul că timpul simulat este fixat la zero. Doar ciclii delta[20] au loc în timpul unei simulări de tip UTLM. Modulul inițiator trimite o cerere modulului țintă (destinație) prin metoda Send şi apoi aşteaptă un răspuns, care va fi primit în următorul ciclu delta. Modulul inițiator este nevoit să aştepte după răspuns chiar dacă modelul este de tip UTLM (adică timpul de aşteptare nu este contorizat). Aşa cum arată diagrama din partea dreaptă a figurii de mai sus, răspunsul poate fi primit după mai mulți cicli delta şi poate fi chiar condiționat de răspunsul unui alt modul inițiator (modulul țintă nu poate emite un răspuns până ce nu a primit la rândul lui un răspuns de la un alt modul).

Forțând metodologia de modelare UTLM în acest fel are ca efect faptul că se poate realiza o tranziție uşoară de la UTLM la TTLM, permițând co-simularea Untimed TLM şi Timed TLM, după cum se va putea şi observa în continuare.

3.5.2. Timed TLM (TTLM) în mediul UNISIM

Prin folosirea modelării TTLM, întârzierile de timp, cauzate de comportamentul modulelor simulate, de comunicațiile inter-module, pot fi contorizate.

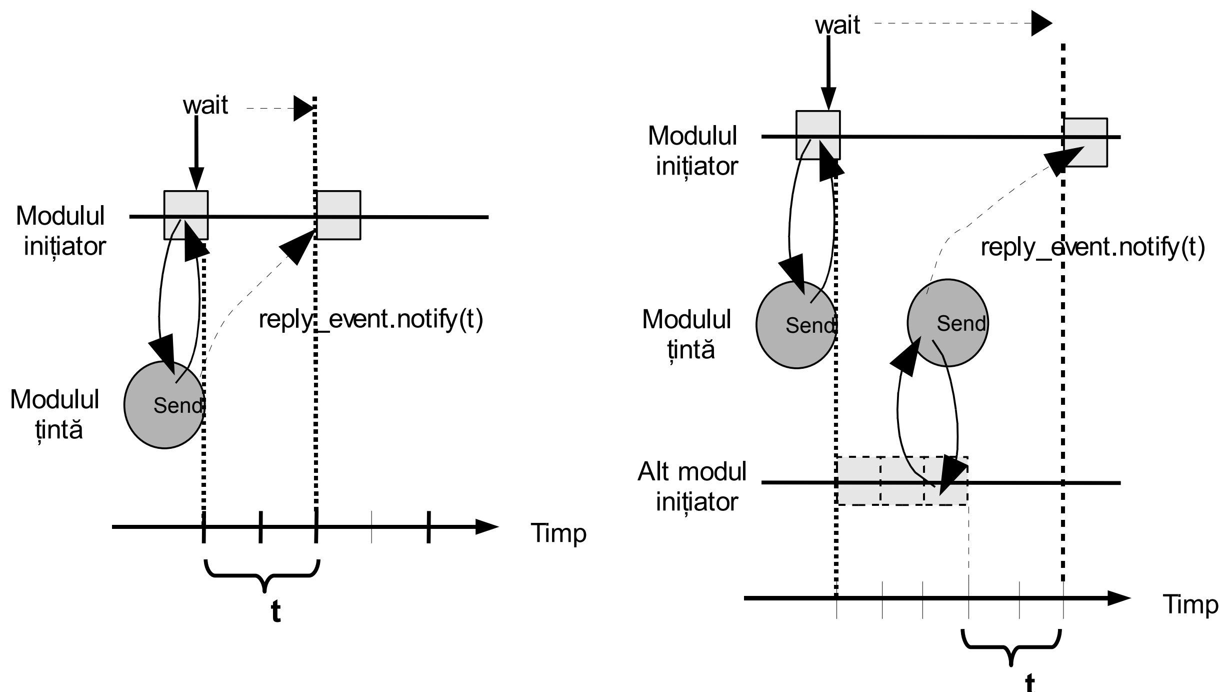


Fig. 21: Principiul TTLM în UNISIM

20 Un ciclu delta (delta cycle) reprezintă în filozofia SystemC perioada în care un modul îşi execută comportamentul. Acest timp nu este cuantificat, execuția unui proces fiind împărțită de nucleul SystemC în mai multe perioade de acest fel pentru a asigura o execuție concurentă a proceselor. Planificatorul SystemC este cel care decide ordinea de întrețesere a proceselor, ținând cont de relațiile de dependență specificate explicit.

Funcționarea modelului TTLM, după cum se poate vedea şi în figura de mai sus, este similară modelului UTLM, cu deosebirea că acum poate fi inserată informație de timp. Prin intermediul metodei notify(...), asociată fiecărui eveniment, se pot specifica duratele comunicațiilor dintre module. Dacă notificările s-ar face instantaneu (folosind notify(0)), modelul TTLM se reduce la modelul UTLM. Observația este importantă, pentru că arată faptul că tranziția, de la un model Untimed TLM, la un model Timed TLM, se poate face uşor.

3.6. Utilizarea TLM în cadrul simulatorului

În cadrul simulatorului creat de UNISIM folosind metodologia TLM, modelarea la nivel de tranzacții se foloseşte pentru a simula comunicația care are loc între memoria principală, busul, care implementează un protocol de tip snooping pentru asigurarea coerenței memoriilor cache şi unul sau mai multe procesoare, care sunt conectate împreună cu memoriile lor cache la magistrală.

Având în vedere metodologia TLM introdusă în UNISIM şi descrisă mai sus, ne interesează care este structura mesajelor vehiculate prin procesele de comunicație care eu loc între modulele precizate mai sus. Figura de mai jos prezintă cele două componente ale unui mesaj TLM (cererea şi răspunsul), esențiale din punct de vedere structural.

```
template <class ADDRESS, unsigned int DATA_SIZE>
class Request
{
public:
		enum Type		// Tipul cererii
			{
				READ,		// Citire
				READX,		// Modificare
				WRITE,		// Scriere
				INV_BLOCK,	// Invalidare bloc cache
				FLUSH_BLOCK	// Golire bloc cache
			};

		Type type;	// Tipul cererii
		bool global;	// Cerere globală sau locală
		ADDRESS addr;	// Adresa fizică a locației de memorie referită
		unsigned int size; // Dimensiunea transferului pe bus (<= DATA_SIZE)

		uint8_t write_data[DATA_SIZE];  // Datele de scris în memorie
};

template <unsigned int DATA_SIZE>
class Response
{
public:
		typedef enum
			{
				RS_MISS     = 0,  // Cache-ul nu are blocul
				RS_SHARED   = 1,  // Cache-ul are blocul nemodificat
				RS_MODIFIED = 2,  // Cache-ul are blocul modificat
				RS_BUSY     = 4   // Un răspuns nu poate fi oferit acum
			} ReadStatus; // Starea cererii de citire



		ReadStatus read_status;	// Starea unei cereri de citire (READ) sau
			// de modificare (READX)
		uint8_t read_data[DATA_SIZE]; // Datele citite prin snooping sau din
			// memorie
};
```

Fig. 22: Structura cererii şi a răspunsului din cadrul unui mesaj TLM al simulatorului

Elementele unei cereri sunt:

- tipul acesteia;
- adresa fizică a locației de memorie la care cererea se adresează;
- dimensiunea transferului pe bus, sau mai exact numărul de octeți care se cer (a fi citiți, sau scrişi);
- un vector de octeți care conține datele de scris, în cazul unei cereri de tip WRITE.

Cererea poate fi una globală sau una locală. În cazul în care aceasta este globală înseamnă că ea va trebui să fie interceptată şi de restul memoriilor cache (snooping).

Pe de altă parte, un răspuns conține informații cu privire la locația de memorie citită fie direct din memoria principală (caz în care starea cererii de citire nu interesează), fie prin snooping dintr-un alt cache (caz în care starea specifică dacă locația de memorie a fost găsită sau nu si dacă da, dacă este modificată sau nemodificată). Similar cererii, răspunsul conține un vector de octeți care va stoca octeții citiți.

3.7. Emularea Sistemului de Operare cu UNISIM

Pentru a putea simula la nivel user chiar şi un simplu benchmark, este necesară implementarea unor sarcini care de fapt sunt dedicate Sistemului de Operare. Astfel de sarcini sunt apelurile către Sistemul de Operare (apeluri sistem – system calls, în engleză). Apelul sistem este mecanismul utilizat de aplicații atunci când trebuie să ceară servicii oferite de către Sistemul de Operare.

În afară de a procesa date în zona de memorie alocată, un program poate să mai necesite şi alte date sau servicii, furnizate de către sistem (spre exemplu utilizarea plăcii de rețea, a plăcii de sunet, a plăcii grafice, sau simplul schimb de informație cu alte programe).

Din cauza faptului că o utilizare greşită sau rău voită a sistemului poate cauza cu uşurință o cădere a sistemului, s-au impus mai multe nivele de control. Toate arhitecturile microprocesor moderne oferă mai multe nivele de privilegii, pe care programele pot rula.

Cel mai jos nivel de privilegii este şi cel în care aplicațiile se execută în mod normal. Acest nivel presupune limitarea spațiului de adrese al programului, astfel încât acesta să nu poată accesa sau modifica alte aplicații care rulează pe sistem sau chiar Sistemul de Operare. Totodată, la acest nivel de privilegii, aplicațiile nu pot accesa perifericele sistemului (cum sunt placa grafică sau placa de rețea).

În acest fel se asigură un control al sistemului. Dar, pe de altă parte, programele trebuie

să poată beneficia de facilităţile restricţionate în mod implicit, atunci când le este permis acest lucru.

Aici intervine Sistemul de Operare, care se execută la un nivel de privilegii înalt şi care permite aplicaţiilor să ceară servicii. Aceste cereri se fac prin intermediul apelurilor sistem.

Un apel sistem este de regulă implementat prin întreruperi software: dacă are permisiunea, sistemul intră pe un nivel de privilegii mai înalt, execută setul de instrucţiuni necesare (asupra cărora programul întrerupt nu are niciun control direct), după care are loc revenirea la nivelul de privilegii scăzut şi predarea controlului programului care a efectuat apelul sistem. Această modalitate de operare este o variantă de implementare a securităţii.

Deoarece s-au dezvoltat mai multe moduri de operare, cu diverse niveluri de privilegii, a fost necesar un mecanism pentru transferul în siguranţă al controlului, dintr-un mod de operare cu privilegii mai puţine, într-unul cu privilegii mai multe. Un cod cu mai puţine privilegii nu poate pur şi simplu să transfere controlul unui cod cu mai multe privilegii, la un moment oarecare de timp, sau atunci când procesorul se află într-o stare oarecare, pentru că ar fi încălcată securitatea.

De aceea, Sistemele de Operare pun la dispoziţie o librărie, scrisă de obicei în limbajul C (libc), aşa cum este spre exemplu glibc (GNU C library). Această librărie este un intermediar între Sistemul de Operare şi programe. Scopul ei este acela de a se ocupa de detaliile de nivel jos, care privesc pasarea informaţiei necesare către nucleul Sistemului de Operare (către kernel) şi trecerea pe un nivel de privilegii ridicat (modul de operare supervizat - supervisor). Avantajul folosirii unei astfel de librării este acela că reduce cuplajul dintre Sistemul de Operare şi aplicaţii, crescând portabilitatea.

Implementarea apelurilor sistem presupune transferul controlului de la program la nucleul Sistemului de Operare, iar acest transfer al controlului este specific arhitecturii hardware. O modalitate tipică de a implementa acest transfer de control, după cum am mai menţionat anterior, este utilizarea întreruperilor software. Folosind această abordare, programul care execută apelul sistem trebuie doar să seteze un registru cu numărul care identifică apelul sistem, după care să execute întreruperea software.

Figura de mai sus [Jon07] prezintă o versiune simplificată al procesului din spatele unui apel sistem, folosind metoda întreruperii. Fiecare apel sistem este multiplexat în kernel prin intermediul unui singur punct de intrare: registrul EAX specifică care apel sistem trebuie executat, lucru specificat în librăria C. După ce librăria C a încărcat identificatorul apelului sistem împreună cu alţi potenţiali parametrii ai acestuia, se invocă întreruperea software 0x80, care determină (prin handlerul ei) execuţia funcţiei system_call. Această funcţie este responsabilă cu toate apelurile sistem, şi foloseşte numărul din EAX pentru a şti despre ce apel sistem este vorba. După câteva teste, are loc

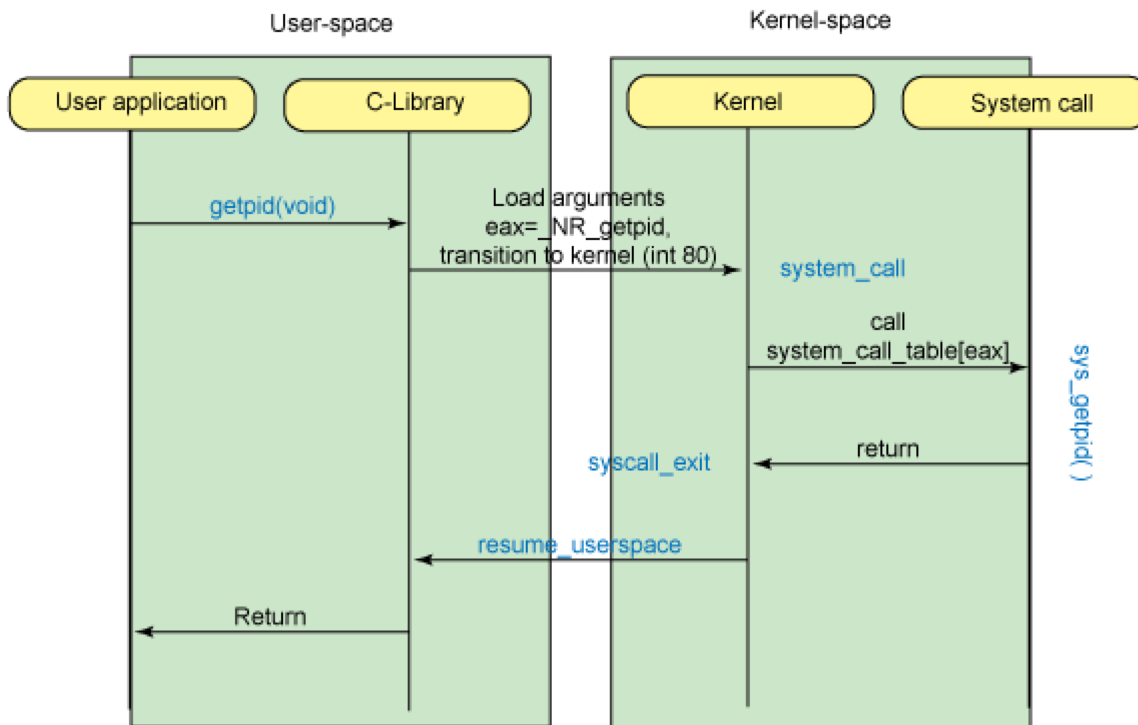


Fig. 23: O variantă simplificată a funcționării unui apel sistem, folosind metoda întreruperii

invocarea propriu-zisă a apelului sistem , folosind tabela apelurilor sistem (system_call_table). După ce execuția apelului sistem a luat sfârşit, se apelează syscall_exit, care determină revenirea din modul kernel în modul user, în librăria C. Din librăria C, execuția revine la programul utilizator, cel care a invocat apelul sistem.

Pentru multe procesoare RISC acesta este singura modalitate fezabilă, însă multe arhitecturi CISC suportă şi alte tehnici. Intel au propus SYSCALL/SYSRET [Gar06] (independent de Intel, cei de la AMD au venit cu SYSENTER/SYSEXIT, scopul şi utilitatea fiind aceleaşi). Acestea nu sunt altceva decât nişte instrucțiuni care permit un transfer rapid al controlului (este eliminat costul introdus de mecanismul de întreruperi).

Majoritatea Sistemelor de Operare au astăzi câteva sute de apeluri sistem. Spre exemplu Linux si FreeBSD au peste 300 de astfel de apeluri. Prezentăm în tabelul de mai jos doar câteva dintre cele mai cunoscute apeluri sistem [Bur04].

%eax	Nume	Sursă	%ebx	%edx	%ecx	%esx	%edi
3	read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	open	fs/open.c	const char *	int	int	-	-
6	close	fs/open.c	unsigned int	-	-	-	-
25	time	kernel/time.c	int *	-	-	-	-
39	mkdir	fs/namei.c	const char *	int	-	-	-

Fig. 24: Câteva apeluri sistem, valabile pentru nucleul Linux, versiunea 2.2

În prima coloană este specificat identificatorul apelului sistem. Acest număr se setează în registrul EAX. În ceilalţi regiştri specificaţi în tabel (EBX, EDX, ECX, ESX, EDI) se trec, dacă este cazul, valorile pentru restul parametrilor apelului sistem în cauză. Coloana sursă specifică unde se află fiecare apel sistem (considerând calea /usr/src/linux ca fiind implicită).

Având în vedere funcţionarea apelurilor sistem, descrisă mai sus, ne vom îndrepta în cele ce urmează atenţia către modalitatea de emulare a Sistemului de Operare (al execuţiei apelurilor sistem mai precis) aşa cum este ea realizată în UNISIM. Ca să putem explica mai bine acest lucru, să considerăm următorul program foarte simplu:

```
#include <stdio.h>
int main() {
        printf("hello world\n");
        return 0;
}
```

În urma simulării acestui program simplu, ne aşteptăm ca pe ecran să se afişeze mesajul „hello world”. Să detaliem puţin execuţia lui.

Afişarea pe ecran se face prin intermediul funcţiei **printf.** Această funcţie, va apela la rândul ei apelul sistem numit **write** (se poate observa acest lucru folosind unealta **strace** – este oferită de Linux şi permite afişarea tuturor apelurilor sistem pe care le execută un program dat ca parametru):

```
write(1, "hello world\n", 12);
```

Primul parametru specifică descriptorul de fişier al ieşirii standard (stdout), al doilea este mesajul de afişat, iar al treilea marchează lungimea mesajului (mai exact dimensiunea bufferului de afişat).

Compilând programul de mai sus pentru un procesor PowerPC, observăm că apelul sistem **write** corespunde următorului cod, în limbaj de asamblare:

```
li      r3, 1                    // r3 = descriptorul de fişier (1 -> stdout)
mr      r4, @hello               // r4 = adresa şirului „hello world\n”
li      r5, 12                   // r5 = lungimea şirului de afişat
li      r0,4                     // r0 = 4 (id-ul apelului sistem write)
sc                               // execută apelul sistem
```

Fig. 25: Exemplu de executare a unui apel sistem pe un procesor PowerPC

Acest cod foloseşte instrucţiunea de apel sistem, **sc**, pusă la dispoziţie de către setul de instrucţiuni ale procesorului PowerPC. Această instrucţiune ar trebui la rândul ei să ceară Sistemului de Operare să execute apelul sistem.

Pentru a putea identifica apelurile sistem, fiecare Sistem de Operare are propria tabelă a apelurilor sistem. Spre exemplu, Sistemul de Operare Linux asociază identificatorul 4 pentru funcţia **write**.

Alte apeluri sistem permit:

- accesul la fişiere (open, close, read, write);
- operaţii cu sistemul de fişiere (rmdir – stergere director, mkdir – creare director);
- obţinerea de informaţii de sistem (time – ceasul sistemului).

Toate aceste apeluri trebuie deci efectuate, pentru ca programul care le foloseşte să poată fi executat corect, să poată avea rezultatul aşteptat.

În condiţiile în care Sistemul de Operare nu este simulat, soluţia este emularea acestuia, prin translatarea apelurilor sistem, venite de la procesorul simulat (de la arhitectura simulată), către arhitectura nativă, cea care rulează simulatorul. Apelurile sistem vor fi deci rulate de către sistemul de operare gazdă.

Această facilitate este oferită în UNISIM prin **modulul Syscall** [Gir08]. În momentul de faţă, dezvoltatorii UNISIM pun la dispoziţie două astfel de module: unul care leagă procesoarele PowerPC de Linux şi unul care leagă procesoarele ARM de acelaşi Sistem de Operare Linux.

Atâta vreme cât Sistemul de Operare ţintă este apropiat de Sistemul de operare gazdă, translatarea apelurilor sistem se poate face relativ simplu.

De obicei, un apel sistem efectuează fie citirea unui buffer din memorie şi operarea asupra datelor citite, fie scrierea conţinutului unui buffer în memorie. Astfel, principalii paşi care se fac în timpul translatării unui apel sistem sunt următorii:

- preluarea parametrilor apelului sistem (dacă acesta are parametrii) din setul de regiştri ai

procesorului simulat;

- conversia structurilor de intrare ale apelului sistem, în structurile de intrare corespunzătoare apelului sistem nativ:
  - copierea datelor din memoria simulată în memoria nativă;
  - corectarea endianessului;
  - alocarea şi scrierea structurilor de intrare corespunzătoare apelului sistem nativ;
- efectuarea apelului sistem nativ;
- conversia structurilor de ieşire ale apelului sistem nativ, în structurile de ieşire corespunzătoare apelului sistem ţintă:
  - alocarea unei zone de memorie nativă, care corespunde structurii de apel sistem ţintă;
  - scrierea structurii alocate cu rezultatul apelului sistem efectuat nativ şi conversia endianessului (dacă este necesară);
  - copierea structurii de ieşire în memoria simulată;
  - setarea corectă a regiştrilor procesorului simulat.

4. Simulare la nivel de tranzacţii cu UNISIM

UNISIM [UNI08] propune două nivele de abstractizare pentru construirea de simulatoare:

- la nivel de ciclu (CLM – Cycle Level Modeling);
- la nivel de tranzacţii (TLM – Transaction Level Modeling).

Simularea la nivel de tact permite o acurateţe înaltă în evaluarea performanţelor componentelor hardware simulate, dar presupune şi foarte multă comunicaţie între modulele ce compun simulatorul, ducând la o scădere semnificativă a vitezei.

În schimb, simularea la nivel de tranzacţii, este orientată mai mult pe comunicaţiile dintre module (comunicaţia fiind un bottleneck real în cazul arhitecturilor CMP). Astfel, simulatoarele TLM pot fi mai puţin precise decât cele de tip CLM, dar rulează mult mai repede.

UNISIM nu se doreşte a fi un nou mediu de simulare, ci mai degrabă este un add-on al SystemC [Sys08]. De altfel, modelarea la nivel de tranzacţii este un standard de comunicaţii introdus odată cu apariţia SystemC 2.0 şi reprezintă o abordare de nivel înalt a modelării sistemelor digitale. TLM se caracterizează în esenţă prin faptul că permite o separare a detaliilor de implementare a modulelor, faţă de detaliile de implementare a comunicaţiilor care au loc între acestea.

Deoarece metodologia TLM permite construirea de sisteme care pot fi simulate cu o viteză ridicată, UNISIM a introdus două nivele de simulare:

– **user-level** (presupune simularea anumitor componente din cadrul unei arhitecturi hardware, în vreme ce alte componente necesare, cum este sistemul de operare în cadrul simulatoarelor execution-driven, sunt emulate);
– **system-level** (permite simularea la nivel de sistem, adică, pe lângă arhitectura hardware, sunt incluse şi sistemul de operare, perifericele).

La nivel TLM, UNISIM propune un simulator care corespunde arhitecturii Mac G3 şi include:

- microprocesor MPC755;
- memorii;
- chipset MPC107;
- PIC (Programmable Interrupt Controller);
- PIIX4 IDE controller;
- Discuri IDE;
- Framebuffer display.

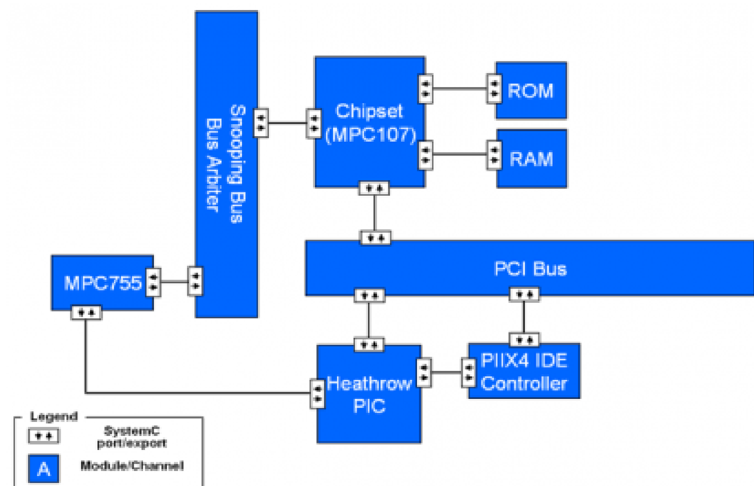


Fig. 26: Reprezentare schematică a arhitecturii emulate de către simulatorul TLM, propus de autorii UNISIM

Simulatorul a fost dezvoltat de către CEA („Commissariat a l'Energie Atomique”), Paris, Franţa, în colaborare cu cercetători de la „Universitat Politecnica de Catalunya” (UPC), Spania şi de la Universitatea din Michigan. Conform autorilor, simulatorul este capabil să booteze un sistem de operare Linux, pentru PowerPC şi să ruleze majoritatea benchmarkurilor SPEC 2006.

Două versiuni ale simulatorului sunt disponibile:

– **ppcemu**: simulator al procesorului PowerPC 755, care nu include niciun periferic şi nu simulează sistemul de operare. Apelurile sistem sunt transmise sistemului de operare gazdă.

– **ppcemu-system**: simulator al procesorului PowerPC 755, care conţine întreaga arhitectură prezentată mai sus şi simulează sistemul de operare. Apelurile sistem sunt rulate în cadrul simulatorului şi se accesează periferice virtuale.

Acest simulator este în dezvoltare. În momentul de faţă se lucrează spre exemplu introducerea mai multor versiuni ale procesorului ARM (datorită consumului redus de putere,

procesoarele ARM domină piața electronicelor mobile).

4.1. Componente ale simulatorului

- **Procesorul PowerPC 755**

Diagrama bloc a procesorului PowerPC 755 [Fre06] comercial este cea din figura următoare:

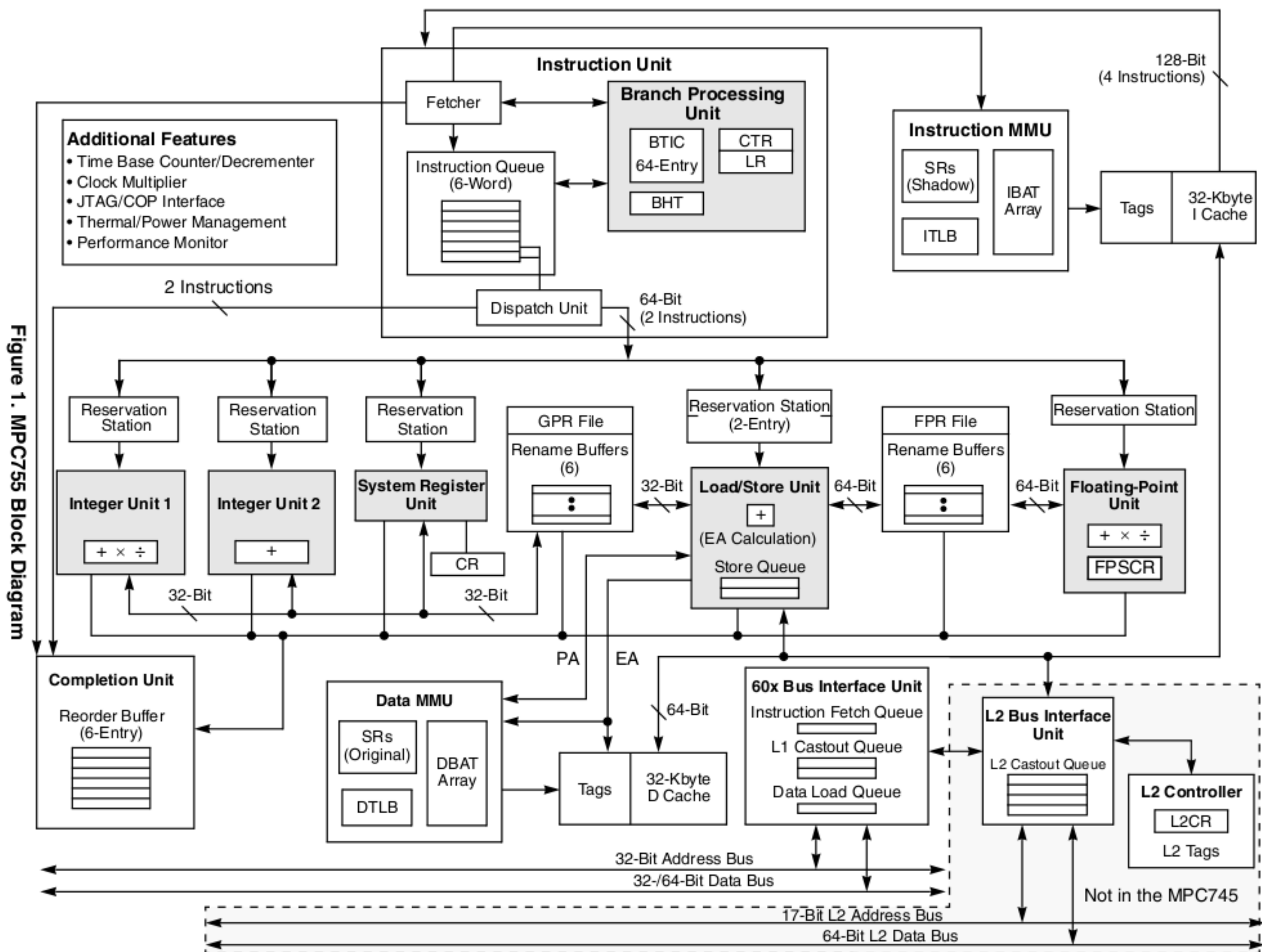


Fig. 27: Schema bloc a procesorului PowerPC 755

Procesorul PowerPC 755 implementat în cadrul simulatorului are însă deocamdată o arhitectură mai simplă. În esență, procesorul nu are o unitate de procesare a instrucțiunilor de salt, iar execuția este una de tip inorder (nu există implementată o arhitectură de tip Tomasulo: stații de rezervare, buffer de reordonare, renaming...).

Procesorul implementat în cadrul simulatorului TLM cuprinde în mare următoarele:

- set de 32 de regiștrii generali (GPR);

- set de 32 de regiştrii pe flotanţi (FPR);
- memorie cache de nivel 1, cu următoarele caracteristici:
  - cache de instrucţiuni:
    - 32 KB dimensiune;
    - dimensiunea blocului de 32 de octeţi;
    - asociativitatea de 8;
    - politică de înlocuire a blocurilor de tip LRU;
    - write-back;
    - protocol de coerenţă de tip write-invalidate, cu 2 stări: valid şi invalid.
  - cache de date:
    - 32 KB dimensiune;
    - dimensiunea blocului de 32 de octeţi;
    - asociativitatea de 8;
    - politică de înlocuire a blocurilor de tip LRU;
    - write-back;
    - protocol de coerenţă de tip write-invalidate, MESI.
- memorie cache de nivel 2, cu următoarele caracteristici:
    - 512 KB dimensiune;
    - dimensiunea blocului de 32 de octeţi;
    - asociativitatea de 8;
    - politică de înlocuire a blocurilor de tip LRU;
    - write-back;
    - protocol de coerenţă de tip write-invalidate, MESI.
- memory management unit (MMU):
  - TLB pentru instrucţiuni:
    - 128 de intrări;
    - asociativitatea de 2;
    - politică de înlocuire a blocurilor de tip LRU;
  - TLB pentru date:
    - 128 de intrări;
    - asociativitatea de 2;
    - politică de înlocuire a blocurilor de tip LRU;
- buffer de prefetch cu 6 intrări;
- interfaţă pentru BUS (procesorul poate comunica prin standardul TLM cu un bus care

implementează un protocol de tip snooping).

În plus, se permite specificarea frecvenței de lucru a procesorului (în MHz), a frecvenței de lucru cu magistrala (în MHz), a numărului maxim de instrucțiuni pe care procesorul le poate executa.

De asemenea, se poate monitoriza consumul de putere al memoriilor cache şi al TLB-urilor, cu ajutorul unei facilități oferite de UNISIM (bazată pe CACTI).

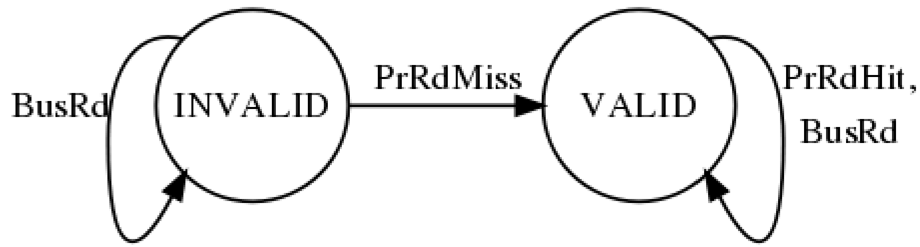
- **Memoria cache**

Este implementată într-o manieră generică. Funcționarea generală a unei memorii cache (decodificarea adresei, evacuarea blocurilor, căutarea unui anumit bloc, etc.), este implementată separat de blocul memoriei cache. Dacă spre exemplu se doreşte crearea unei memorii cache care să folosească un bit de dirty, atunci modificările necesare se fac doar la nivelul blocului, ele nu afectează funcționarea generală a memoriei cache. Mai mult, diagramele de tranziții pentru protocoalele de coerență sunt şi ele separate de implementarea generală a memoriei cache.

Implementarea memoriei cache permite specificarea tuturor parametrilor importanți: dimensiunea cache-ului, asociativitatea, mărimea unui bloc, politica de scriere a datelor în memoria principală (write-back sau write-through), algoritmul de evacuare a blocurilor (LRU sau Pseudo LRU).

Există implementate două tipuri de protocoale de coerență a cache-urilor, ambele de tip write-invalidate:

- protocol simplu, cu 2 stări, valid şi invalid, folosit pentru cache-ul de instrucțiuni



PrRdHit - processor read hit
PrRdMiss - processor read miss
BusRd - bus read

Instruction Cache Coherency Protocol

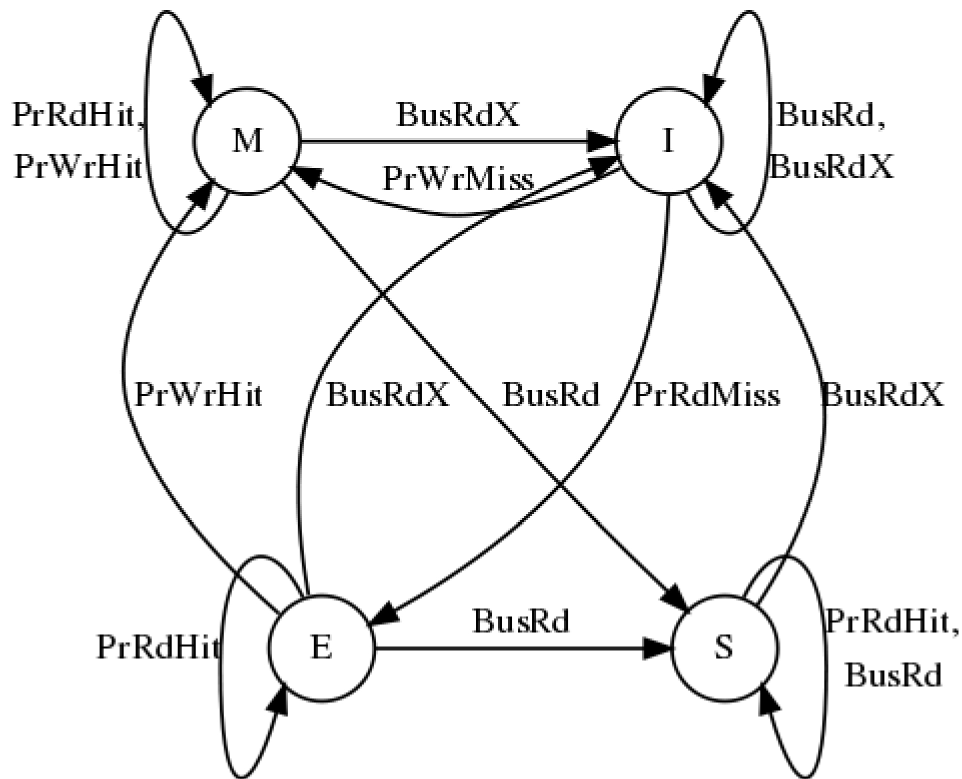
Fig. 28: Diagrama de stări a protocolului de coerență folosit în cadrul simulatorului pentru cache-ul de instrucțiuni (IL1)

După cum o arată şi diagrama de mai sus, protocolul folosit pentru menținerea coerenței unei memorii cache de instrucțiuni este unul simplu. Un bloc oarecare se poate afla în două stări: valid sau invalid. Inițial, fiecare bloc este în stare invalidă.

Prima dată când procesorul va dori să citească dintr-un bloc, va avea loc o citire cu miss, în urma căreia, blocul va fi adus din memoria principală, starea lui devenind una validă.

Dacă un alt procesor va încerca să obțină informația dintr-un bloc prin intermediul busului, făcând snooping (evenimentul BusRd), blocul îşi va păstra starea.

- protocolul MESI (4 stări; Modified, Shared, Exclusive, şi Invalid)



PrRdHit - processor read hit; PrRdMiss - processor read miss
PrWrHit - processor write hit; PrWrMiss - processor write miss
BusRd - bus read; BusRdX - bus exclusive read

MESI Cache Coherency Protocol

Fig. 29: Diagrama de stări a protocolului de coerență folosit în cadrul simulatorului

Protocolul MESI este utilizat pentru asigurarea coerenței celorlalte două memorii cache din cadrul arhitecturii: memoria cache de date, aflată pe nivelul unu şi memoria cache unificată, aflată pe nivelul doi, în cadrul ierarhiei de memorie utilizată.

- **Busul cu protocol de tip snooping**

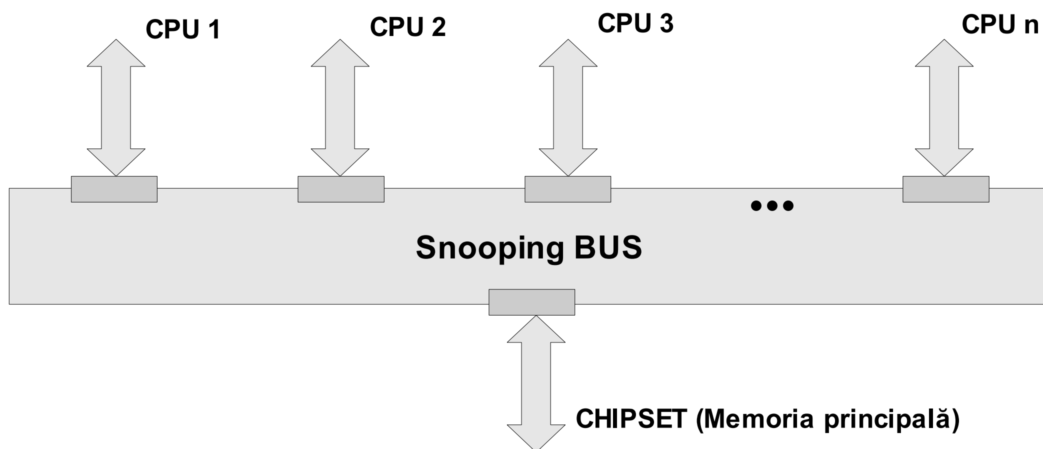


Fig. 30: Reprezentare schematică u busului folosit în cadrul simulatorului

Implementat cu ajutorul standardului TLM, busul existent în cadrul simulatorului permite conectarea a unuia sau mai multor procesoare, cu chipsetul. În cadrul simulării la nivel user, chipsetul este înlocuit direct de modulul care simulează memoria principală.

Busul respectă protocolul de tip snooping, pentru asigurarea coerenței cu memoriile cache ale procesoarelor conectate. Mecanismul „snooping”, este principala strategie de păstrare a coerenței memoriilor cache, în cadrul sistemelor multiprocesor cu memorie partajată, bazate pe magistrală (bus). Busul este un mecanism convenabil de asigurare a coerenței memoriilor cache deoarece permite tuturor procesoarelor din cadrul arhitecturii CMP (Chip MultiProcessors) să „observe” tranzacțiile care se fac la nivelul memoriei.

Busul are un controller, care are rolul de a recepționa toate cererile care vin din exterior, din partea procesoarelor, sau a chiptsetului. Toate cererile intră în cozi de aşteptare (câte una pentru fiecare procesor, plus una pentru chipset) şi sunt servite pe rând, câte una la fiecare tact al magistralei.

Pentru aceasta, s-a implementat un mecanism simplu prin care este simulat tactul busului (frecvența de lucru a busului este parametrizabilă). La fiecare ciclu de bus, se caută primul procesor pentru care coada de mesaje nu este goală. Se ia primul mesaj din coadă şi se tratează. Dacă nu există mesaje din partea nici unui procesor, este verificată coada care ține mesajele primite din partea chipsetului. În cazul în care există mesaje de la chipset, primul dintre ele este extras din coadă şi tratat. După ce un mesaj a fost tratat, are loc un proces de sincronizare, în care, este calculat timpul rămas din cadrul tactului. Acel timp va fi unul de aşteptare: de abia după trecerea acelui timp este preluat următorul mesaj.

În situația în care mesajul primit este de la chipset, au loc următoarele acțiuni (mesajele de răspuns, către chipset sunt prioritare mesajelor de cerere venite din partea chipsetului):

- dacă mesajul este de tip răspuns, chipsetul este notificat;
- în cazul unei cereri de scriere, mesajul este direcționat către primul procesor care îl acceptă;
- în cazul unei cereri de citire, mesajul este trimis tuturor procesoarelor. Procesorul care are blocul cerut va răspunde cu informația cerută. Dacă niciun procesor nu are informația cerută, va fi emis un mesaj gol, către chipset (un miss).

În situația în care mesajul este primit de la un procesor (fiecare procesor are un număr de identificare) există trei cazuri:

- dacă mesajul nu este unul global, el este trimis direct chipsetului (un mesaj global este acel mesaj care trebuie ascultat – snooped - şi de celelalte procesoare);
- dacă mesajul este adresat memoriilor cache ale celorlalte procesoare (de exemplu invalidarea unui bloc), el este trimis direct tuturor celorlalte procesoare;
- în cazul în care mesajul este o cerere de scriere, el va fi trimis tuturor celorlalte procesoare, dar şi chipsetului (cu precizarea că mai întâi este trimis celorlalte procesoare şi de abia apoi chipsetului);
- în final, o cerere de citire este trimisă tuturor celorlalte procesoare. Se aşteaptă un răspuns din partea acestora şi numai în cazul în care toate procesoarele au răspuns cu miss, mesajul este trimis către chipset.

Aşadar UNISIM propune, în general, dar şi prin intermediul acestui simulator, a unei metodologii de simulare bazată pe crearea şi interconectarea de module. Spre exemplu, procesorul PowerPC este un modul, busul este un alt modul, etc.

Pentru a simula un comportament hardware al unui modul, SystemC permite ca fiecare modul să aibă un semnal de tact, deci să funcționeze la o anumită frecvență. Practic, un modul rulează independent, având propriul fir de execuție. Simularea latențelor de comunicație se poate face uşor, prin primitiva *wait(...)*.

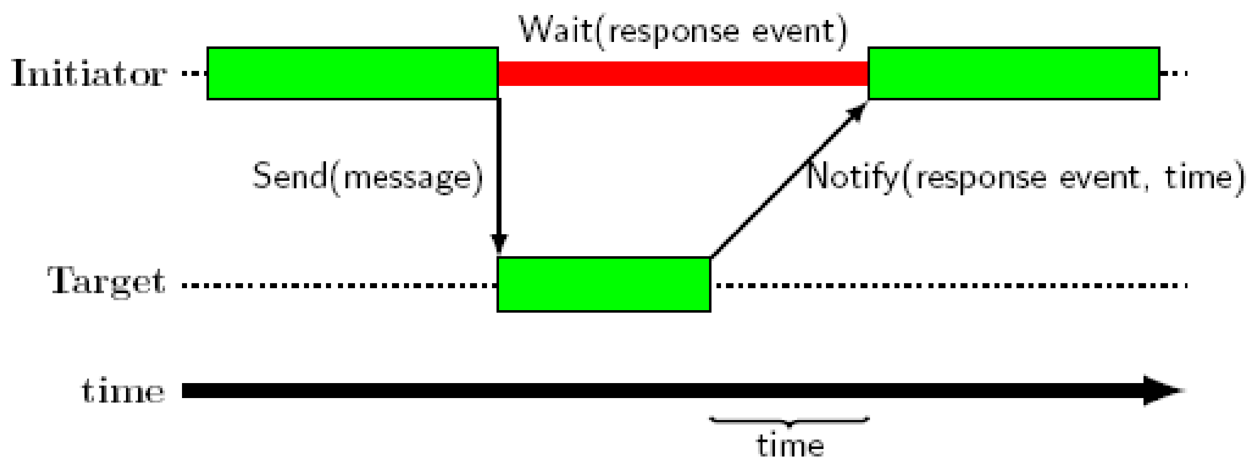


Fig. 31: Reprezentare schematică a modelului de tip Timed TLM, folosit în cadul simulatorului

În cadrul UNISIM, modulele au fost clasificate în două categorii.

Există astfel module care simulează componente ale arhitecturii hardware (procesor, cache, bus) şi pentru care se urmăresc factori de performanţă.

Pe de altă parte, există şi module care nu simulează componente hardware, ci îndeplinesc diverse funcționalități, mai degrabă software, dar necesare pentru o simulare la nivel de sistem. Aceste tipuri de module au fost numite Capabilități („Capabilities”), de către dezvoltatorii UNISIM.

În figura următoare sunt prezentate ambele tipuri de module.

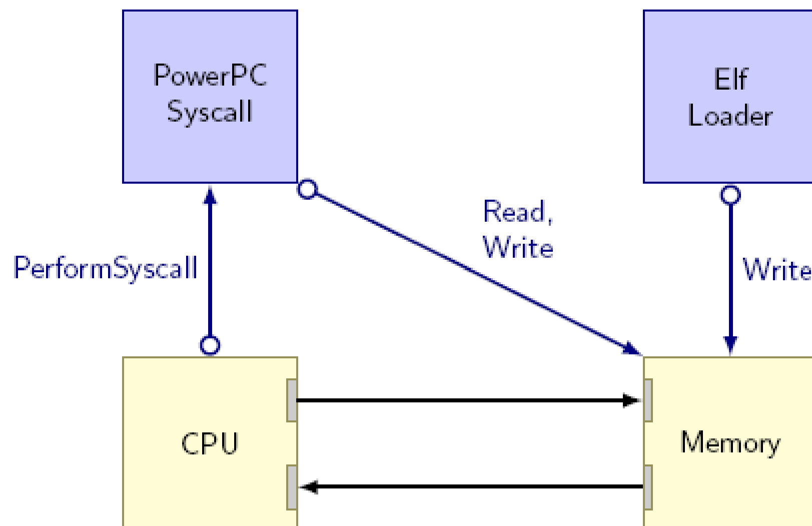


Fig. 32: Modularizarea componentelor arhitecturale dar şi software simulate sau emulate în cadrul UNISIM

ElfLoader serveşte la încărcarea programelor compilate în memorie, iar PowerPC Syscall foloseşte la emularea sistemului de operare (apelurile către sistemul de operare sunt translatate către sistemul de operare gazdă).

Simulatorul TLM oferă însă şi alte module, de exemplu pentru estimarea consumului de

putere şi energie consumate de memoriile cache, sau un modul GDB (GNU DeBugger) care poate fi utilizat la verificarea programelor care rulează pe arhitectura emulată.

Conectarea modulelor se face prin intermediul porturilor pe care modulele le au. Tipic, acestea sunt de două feluri: de import şi de export. Porturile de import permit preluarea de informații de la alt modul (un server) şi folosirea lor de către modulul care le-a preluat (şi care este un client). Aşadar, modulele pot oferi servicii pentru alte module, sau pot cere servicii din partea unor alte module.

Un exemplu simplu de conectare a două module este următorul:

```
bus->memory_import << memory->memory_export
```

Fig. 33: Exemplu de interconectare a busului cu memoria principală

Busul este conectat la o memorie principală, astfel că va putea să- folosească serviciile oferite, adică să scrie şi să citească din memorie.

Simularea în UNISIM, folosind metodologia TLM, beneficiază de facilitățile SystemC şi permite măsurarea timpului scurs în urma rulării sistemului, a latențelor introduse prin primitiva wait(...).

Practic simulatorul măsoară prin intermediul metodologiei TLM care este timpul total de comunicație, la nivelul busului, din cadrul arhitecturii dezvoltate. De asemenea, pot fi măsurați şi alți factori de performanță. De exemplu memoria cache raportează accesele unui modul de estimare a consumului de putere. Acesta foloseşte CACTI şi oferă informațiile specifice pentru estimarea consumului de putere.

Figura următoare prezintă care sunt rezultatele oferite de către simulatorul la nivel de tranzacții implementat de autorii UNISIM:

```
Simulation finished
Simulation statistics:
simulation time: 93.79 seconds
simulated time : 2.39704 seconds (exactly 2397040139165 ps)
simulated instructions : 709134708 instructions
host simulation speed: 7.56088 MIPS
time dilatation: 39.1274 times slower than target machine
```

Fig. 34: Rezultatele oferite de simulator

- simulation time – indică cât a durat simularea, pe maşina pe care a rulat simulatorul;
- simulated time – arată care a fost timpul total de comunicație, inclusiv procesele de asigurare a coerenței cache-urilor (la nivelul busului practic);
- simulated instructions – marchează numărul total de instrucțiuni maşină rulate de către

procesorul PowerPC 755 simulat;

- host simulation speed – reprezintă raportul dintre „simulation time” şi „simulated time” şi vrea să reprezinte cu ce viteză a fost executat un benchmark, prin intermediul procesorului simulat. Având în vedere faptul că „simulated time” reprezintă în fapt timpul de comunicaţie pe bus, această metrică este în opinia noastră eronată şi deci inutilă;
- time dilation – este raportul dintre „simulation time” şi „simulated time”. Prin aceeaşi raţiune ca mai sus şi această metrică este inutilă.

Exceptând rezultatele oferite de către simulator referitor la estimarea consumului de putere, care nu sunt prezentate aici, cam acestea sunt rezultatele oferite, cel puţin în mod direct de către acest simulator.

Este deci evident faptul că scopul urmărit de către dezvoltatorii săi a fost acela de a realiza o arhitectură cât mai completă şi mai apropiată de cea reală, urmărindu-se în primul rând obţinerea unui simulator la nivel de sistem.

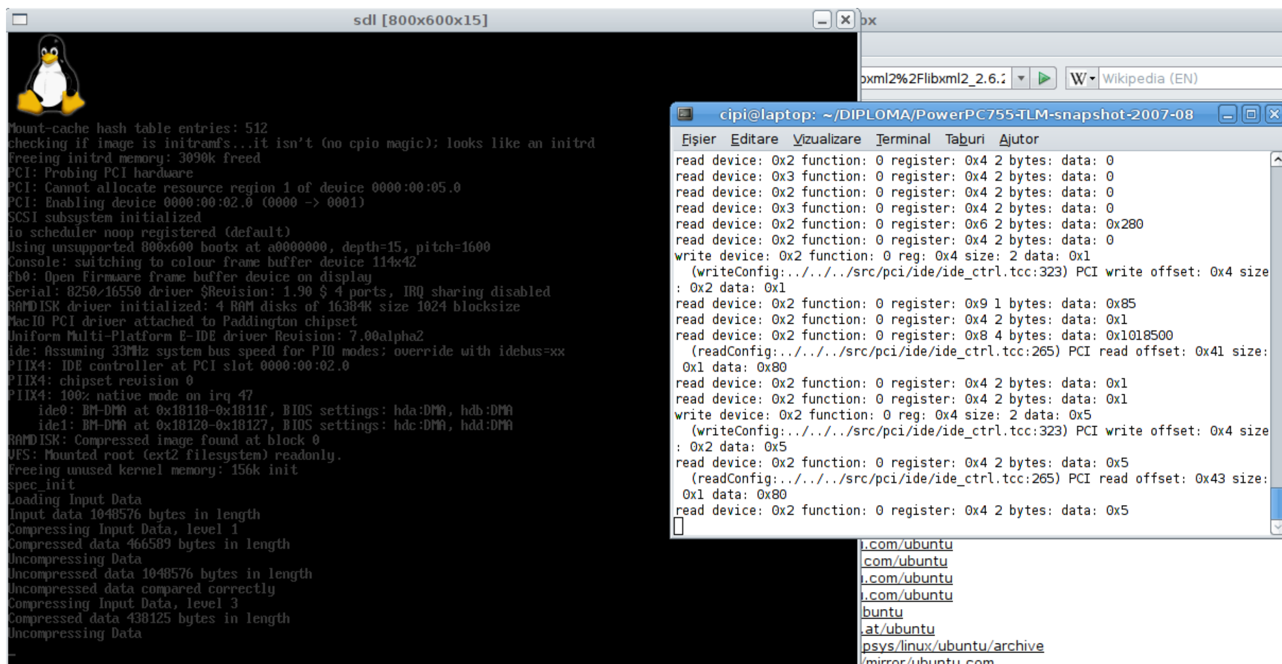


Fig. 35: Rularea, la nivel de sistem, cu simulatorul PowerPc-TLM a benchmarkului gzip pe un Sistem de Operare Linux emulat şi el

Figura de mai sus arată faptul că simulatorul TLM propus de către autorii UNISIM este capabil să booteze un Sistem de Operare Linux, încărcat în memoria emulată, şi să ruleze aplicaţii pe el.

4.2. Dezvoltarea simulatorului PowerPC-TLM

Deşi simulatorul PowerPC-TLM a fost gândit şi proiectat pentru a suporta mai multe procesoare conectate la magistrală, pentru a putea realiza acest lucru, practic, o arhitectură

multiprocesor, cu memorie partajată, au mai fost necesare o serie de dezvoltări ale acestuia.

Unul din primele obiective urmărite a fost acela de a obține o un simulator biprocesor (dual-core) funcțional. Pe scurt, s-a urmărit rezolvarea principalelor probleme care caracterizează arhitecturile SMA (Shared-Memory Architecture): coerența memoriilor cache, consistența memoriei principale şi sincronizarea proceselor, programarea paralelă. În acest sens, s-a urmărit în primul rând implementarea conceptului de „snooping", deoarece, datorită modularității de care dă dovadă simulatorul, adăugarea unui nou procesor în cadrul arhitecturii a fost o problemă rezolvată uşor.

Pentru a evita problemele în plus pe care le cauzează ierarhiile pe mai multe nivele de memorii cache, memoria cache unificată, de nivel 2 a procesorului PowerPC 755 a fost dezactivată. Cu această simplificare arhitecturală, s-a urmărit mai întâi implementarea conceptului de „snooping". Datorită metodologiei de modelare la nivel de tranzacții, s-a lucrat la nivelul comunicațiilor care au loc în cadrul arhitecturii, procesor-bus, bus-memorie, dar şi a „snoopingului", modelat ca cereri şi răspunsuri, vehiculate prin intermediul busului, sub formă de mesaje TLM.

În versiunea inițială a simulatorului, mesajele emise de către procesor busului erau acumulate într-o coadă fiind tratate de către bus pe unul după altul, în ordinea sosirii lor. După cum testele efectuate pe arhitectura biprocesor creată, au arătat-o, utilizarea unui astfel de mecanism de tratare a cererilor venite de la procesoare este unul nesigur. Iată un exemplu: procesorul P1 trimite busului un mesaj de tip WRITE, care marchează faptul că doreşte scrierea unei locații din memoria principală (WB – Write Back). Să spunem că protocolul de coerență utilizat a dictat acest lucru. Mesajul va fi primit de bus şi va intra în coada cu mesaje, aferentă P1, aşteptând să fie tratat de către bus. Între timp, celălalt procesor, P2, emite către bus o cerere de citire (READ) a aceleiaşi locații de memorie. Deoarece scrierea cerută de P1 nu a avut încă loc şi pentru că busul tocmai a terminat de tratat un mesaj precedent de la P1, busul va trece la a trata un mesaj de la P2, adică cererea de citire se face înainte de scriere. Ca rezultat, consistența memoriei este violată! Ne putem întreba: de ce busul a trecut la tratarea unui mesaj de la P2 şi nu a rămas la P1? Răspunsul este simplu: deoarece trebuie să existe o logică de arbitrare la nivelul magistralei în ceea ce priveşte mesajele venite de la procesoare. Concluzia este că acea abordare, modalitate de funcționare a busului era una problematică.

Acesta este motivul pentru care s-a optat pentru o altă abordare, mai simplă, dar care să asigure consistența memoriei principale. S-a optat pentru o consistență secvențială a memoriei, folosindu-se în acest sens conceptul de deținător al busului, „bus owner". Practic, atunci când un procesor se află pe bus, niciun alt procesor nu mai poate comunica cu busul, până când magistrala

nu este eliberată de deținător. O astfel de abordare elimină problema din exemplul de mai sus: P2 nu va mai putea să să emită cererea de citire până ce mesajul WRITE, scrierea deci, venită de la P1 nu a fost tratată. Desigur, această abordare nu este una perfectă, deoarece poate duce la fenomenul de înfometare, „starvation”. În practică însă, nu am avut această problemă pentru că se comunică totuşi foarte des cu busul, iar astfel „ownershipul” se schimbă des, astfel că „lupta” pentru obținerea busului este prezentă, dar nu are de fiecare dată acelaşi „câştigător”.

Unul din scopurile urmărite este acela de a studia influența protocoalelor de coerență a memoriilor cache. În acest sens, pe lângă cele două protocoale existente în cadrul simulatorului (INSN – util numai la un cache de instrucțiuni – şi MESI), s-au mai adăugat alte trei protocoale similare: MSI, MOSI şi MOESI (descrise în capitolul 2). Profitând de decuplajul care este realizat între implementarea propriu-zisă a unei memorii cache şi un protocol de coerență oarecare, introducerea celor trei noi protocoale de coerență s-a realizat relativ uşor.

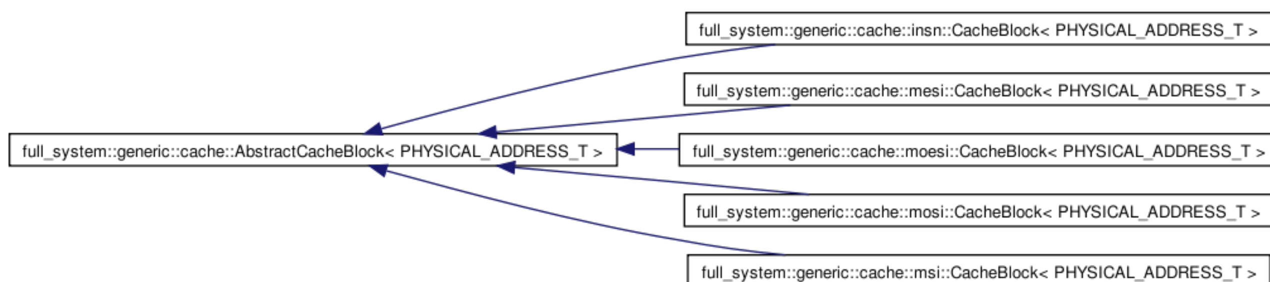


Fig. 36: AbstractCacheBlock reprezintă un schelet al unui bloc cache, care este acelaşi indiferent de protocolul de coerență folosit. Numai informațiile specifice, precum stările posibile ale blocului, sunt specificate, suprascrise, la nivelul fiecărui protocol în parte

Implementarea tranzițiilor dintr-o stare în alta este separată deci de detaliile de implementare ale funcționării unui bloc cache. Practic, pentru fiecare nou protocol a fost necesară, folosind această abordare, adăugarea stărilor pe care le foloseşte şi descrierea tranzițiilor dintre acestea. Pentru fiecare protocol, s-au folosit câte două tablouri bidimensionale[21]: unul care arată care este acțiunea care trebuie luată, în funcție de starea şi evenimentul blocului cache şi unul care evidențiază care este starea următoare în care blocul trebuie să treacă, în funcție de starea lui curentă şi evenimentul care a cauzat actualizarea stării.

Desigur, pentru simularea arhitecturii biprocesor cu memorie partajată, sunt necesare şi aplicații paralele. Având în vedere că pentru simulatoarele la nivel de ciclu, dezvoltatorii UNISIM au introdus emularea standardului POSIX Threads, varianta cea mai fezabilă de a putea rula

21 Protocoalele de coerență care aşteaptă un răspuns pentru a decide dacă blocul trece în starea Shared sau în starea Exclusive, adică MESI şi MOESI, folosesc tablouri tridimensionale, deoarece au nevoie şi de informația Shared sau Not Shared (Exclusive)

aplicații multi-fir pe simulatorul PowerPC-TLM a fost aceea de a migra funcționalitatea PThread de la nivel CLM, la nivel TLM. Această migrare s-a făcut fără mari modificări, singurul lucru mai dificil fiind implementarea conceptului de procesor în aşteptare („sleeping"). De ce acest concept? Pentru că fiecare fir de execuție creat de aplicația paralelă, mai exact fiecare proces, este asociat spre execuție unui alt procesor. În afară deci de procesorul care rulează firul principal al aplicației, restul procesoarelor trebuie să înceapă să execute instrucțiuni numai după ce a fost creat un nou fir, creare care se face cu funcția `pthread_create(...)` şi care presupune pregătirea contextului asociat noului procesor şi asignarea lui unui procesor aflat în aşteptare.

Folosind thread-uri POSIX la nivelul simulatorului înseamnă că simulatorul trebuie să se ocupe de scheduling-ul lor complex (incluzând migrația thread-urilor) aşa cum ar face un sistem de operare. Implementarea unui scheduler ar fi o sarcină foarte mare şi dezvoltatorii simulatorului au decis să se limiteze la o subclasă a benchmark-urilor cu fire. S-au limitat la următoarele funcții:

- pthread_create şi pthread_exit pentru pornirea şi oprirea unui fir.
- pthread_self pentru a putea afla id-ul firului.
- pthread_join – aşteaptă după celalalt thread să termine.
- pthread_mutex_init şi pthread_mutex_destroy folosit pentru a defini şi distruge un mutex.
- pthread_mutex_lock şi pthread_mutex_unlock pentru a putea folosi un mutex declarat anterior.

Numărul de fire care se pornesc într-o aplicație de test nu trebuie aşadar să depăşească numărul de procesoare din sistemul simulat.

O altă problemă esențială a calculului paralel ce trebuie tratată deci pentru dezvoltarea şi utilizarea unei arhitecturi paralele o constituie cea a mecanismelor de sincronizare. În vederea asigurării unor astfel de mecanisme în cadrul simulatorului, ne-am îndreptat atenția asupra a două dintre cele mai des folosite mecanisme: secțiune critică de program (marcată prin perechea de operații Lock – Unlock) şi sincronizarea la barieră.

În privința secțiunii critice, a asigurării accesului exclusiv pentru un sigur proces, pentru o anumită porțiune de cod, a atomicității execuției (până şi o banală incrementare a unei variabile se realizează prin trei instrucțiuni maşină), există două abordări majore: una hardware şi una software.

Abordarea hardware presupune practic utilizarea instrucțiunilor atomice. Procesorul PowerPC oferă în acest sens perechea de instrucțiuni **`lwarx`** (load and reserve) şi **`stwarx`** (store and reserve) care în esență permit accesul exclusiv la o locație de memorie. Figura de mai jos prezintă spre exemplificare implementarea instrucțiunii atomice Test&Set.

```
loop: lwarx  r5,0,r3 #load and reserve
      cmpwi  r5, 0     #done if word
      bne    $+12      #not equal to 0
      stwcx. r4,0,r3   #try to store non-zero
      bne-   loop      #loop if lost reservation
```

Fig. 37: Implementarea instrucțiunii atomice Test and Set în procesorul PowerPC [Pow00]

Se poate observa că instrucțiunea Test and Set va sta într-o buclă până când va obține acces exclusiv la locația de memorie vizată (semnul „-” de la instrucțiunea bne – branch not equal îl informează pe predictorul de salturi că acest salt este probabil NT – Not Taken).

Pe baza instrucțiunii Test and Set, operațiile Lock şi respectiv Unlock se vor realiza astfel:

```
lock: li    r4,1            #obtain lock
loop: bl    test_and_set #test and set
      bne-  loop            #retry until old = 0
                            #delay subsequent instructions until previous ones complete
      isync
      blr                   #return
```

```
unlock: sync       #delay until prior stores finish


       li   r1,0
       stw  r1,0(r3)   #store zero to lock location
       blr             #return
```

Fig. 38: Ilustrarea achiziționării unei blocări asupra unei locații de memorie printr-o operație de sincronizare atomică, de tip Read-Modify-Write [Pow00]

Abordarea software înseamnă folosirea unor algoritmi pentru programarea concurentă, în vederea realizării excluziunii mutuale. Astfel de algoritmi pot exista la nivelul nucleului Sistemului de Operare şi pot asigura excluziunea mutuală de fiecare dată când programatorul cere acest lucru. Spre exemplu, în standardul POSIX Threads, se folosesc următoarele funcții:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex);  // initializare mutex (obiect asociat sectiunii
                             // critice)
pthread_mutex_lock(&mutex);
          // sectiunea critica
pthread_mutex_unlock(mutex);
```

În implementarea operațiilor de Lock şi Unlock am optat pentru abordarea software, deoarece cea hardware era mai dificilă (trebuia extins setul de instrucțiuni al procesorului PowerPC 755 din simulator). Ca şi algoritm software, am folosit algoritmul lui Lamport (numit şi algoritmul brutarului), pentru că oferă o soluție generalizată, ce se poate scala pe N procese. Algoritmul, pseudo-cod este următorul:

```
begin integer j;
 L1 : choosing [i] : = 1 ;
     number[i] : = 1 + maximum (number[l],..., number[N]);
     choosing[i] : = 0;
     for j = 1 step l until N do
       begin
         L2: if choosing[j] ≠ 0 then goto L2;
         L3: if number[j] ≠ 0 and (number [j],j) < (number[i],
                i) then goto L3;
       end;
     critical section;
     number[i] : = 0;
     noncritical section;
     goto L 1 ;
end
```

Fig. 39: Algoritmul lui Lamport, în pseudo-cod, unde (a, b) < (c, d) este echivalent cu (a < c) or ((a == c) and (b < d)) [Lam74]

Este remarcabil faptul că acest algoritm reuşeşte să asigure secțiunea critică fără a se folosi de instrucțiuni atomice de nivel jos! Desigur, atunci când operația de Lock nu a reuşit (din cauză că alt proces se află sau urmează să intre în secțiunea critică), procesul trebuie să reîncerce (buclele L2 şi L3). Evident în practică procesul trebuie să intre într-o stare de aşteptare (idle), pentru a nu cauza blocaje. În implementarea noastră, am profitat de facilitățile oferite de metodologia TLM: procesorul care nu a reuşit să facă un Lock va evita să tot reîncerce să-l obțină (spin lock) şi va intra într-o stare de aşteptare, până când va primi un mesaj de notificare de la alt procesor, cum că a executat un Unlock. Astfel, cererea din cadrul mesajul tranzacționat a fost completată cu un nou tip de mesaj: MUTEX_UNLOCK.

În ceea ce priveşte implementarea acestui algoritm în cadrul simulatorului nostru, mai trebuie precizat faptul că buclele L2 şi L3 din cadrul algoritmului au fost înlocuite fiecare cu două

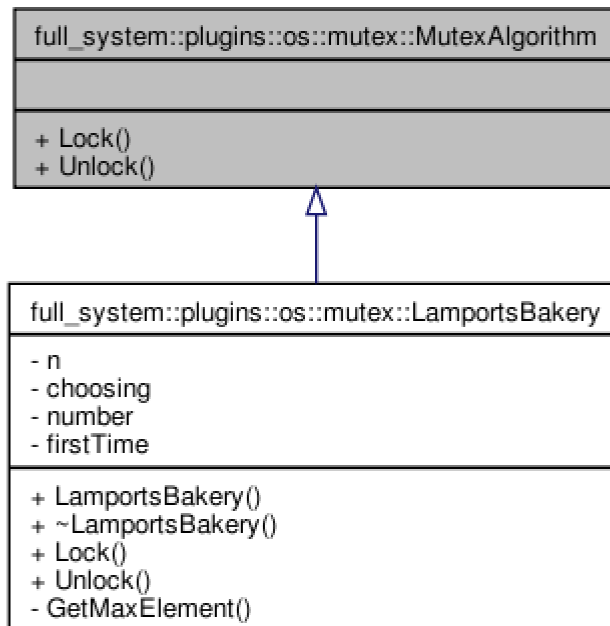


Fig. 40: Diagramă UML de clasă care ilustrează introducerea algoritmul lui Lamport în cadrul simulatorului. Clasa LamportsBakery implementează interfața MutexAlgorithm, care nu este altceva decât o specificație pură despre cum trebuie "arate" un algoritm pentru excluziune mutuală.

teste condiționate. Practic, dacă procesul nu poate face Lock, fie din cauza condiției L2, fie din cauza L3, metoda Lock(...) va returna **false** (fireşte, dacă operația reuşeşte, rezultatul returnat va fi **true**). Dacă operația Lock a eşuat, procesul va intra în starea de aşteptare şi va încerca mai târziu, atunci când a primit o notificare cum că o operație de Unlock a avut loc. Atunci când va reîncerca, algoritmul se va relua, numai că procesul nu mai este acum la prima încercare şi deci nu mai trebuie să ia un „bon de ordine”. Aici intervine de fapt necesitatea variabilei firstTime (vezi figura de mai sus): fiecare proces primeşte un „bon de ordine” numai o singură dată.

De altfel, o abordare similară am folosit pentru sincronizarea la barieră, pentru care am creat o notificare numită BARRIER_UNLOCK. Scopul unei bariere este acela de a bloca toate procesele până când toate care au ajuns în acel punct din program, marcat de barieră. În simulator, când ultimul proces ajunge la barieră, el trimite un mesaj de notificare. Celălalt procesor, care a ajuns deja la barieră trebuie să primească o astfel de notificare pentru a putea continua. Totodată, se foloseşte si ideea de barieră cu sens: într-o variabilă asociată barierei se memorează care a fost ultimul procesor care a accesat-o. Astfel, API-ul Pthread disponibil simulatorului a fost îmbogățit cu alte trei funcții:

```
pthread_barrier_init(&barrier, n); // initializare bariera
```

```
                                              //(n - cate procese vor astepta)

      pthread_barrier_wait(&barrier); // asteptarea la bariera

      pthread_barrier_destroy(&barrier); // distrugerea unei bariere
```

unde `barrier` este de tipul:

```
typedef struct {
      unsigned int needed; // cate procese(procesoare) vor astepta
      unsigned int called; // cate procese(procesoare) au ajuns la bariera
      pthread_mutex_t mutex; // mutex folosit pentru incrementarea lui called
} pthread_barrier_t;
```

4.2.1. Parametrii noului simulator PowerPC-TLM

În urma dezvoltărilor descrise mai sus, simulatorul oferă următorii parametri de intrare (aceştia pot fi văzuți prin simpla rulare, fără parametrii, a executabilului ppcemu_pthread_tlm):

- n (numărul de procesoare, unul sau două);
- i <nr> (numărul maxim de instrucțiuni maşină care să fie executate; pentru două procesoare, primul care ajunge să execute <nr> instrucțiuni încheie simularea);
- m <fm[MHz]> (frecvența memoriei principale – specifică cât de mult va aştepta busul ca să primească o valoare citită din memoria principală – latența memoriei);
- b <fb[MHz]> (frecvența busului – specifică cât de des poate procesorul să comunice cu busul – latența busului);
- c <file> reprezintă un fişier în care se pot configura mai mulți parametrii ai memoriilor cache. Exemplu:

```
# Level 1 Data Cache

# 32 * 1024 = 32768 = 32 KB
dl1.cacheSize = 32768
dl1.blockSize = 32
dl1.associativity = 8
# Replacement policy can be LRU or PSEUDO_LRU (LRU is used by default)
dl1.replacementPolicy = LRU
# Available coherence protocols are: MESI(used by default), MSI, MOSI, MOESI
# INSN cannot be used with a data cache
dl1.coherenceProtocol = MESI
```

Fig. 41: Parametrii de configurare pentru memoriile cache pot fi specificați într-un fişier properties[22]

- p (dacă acest parametru este specificat, simulatorul va oferi şi rezultate privind consumul de putere);
- l <file> (dacă este specificat permite obținerea unu fişier de logging care conține detaliat toate operațiile făcute de memoriile cache, de procesor, bus, memoria principală – dezavantajul este că încetineşte foarte mult rularea simulatorului, dar avantajul este că reprezintă o modalitate bună de a realiza teste).

La ieşire, simulatorul oferă informații privind:

- timpul de comunicație pe bus;
- numărul de instrucțiuni executat de procesoare;
- informații cu privire la cache-uri: numărul de accese cu hit/miss pt. citiri/scrieri/accese prin snooping, dar şi numărul de operații Write Back cauzate de protocoalele de coerență;
- estimarea consumului de putere.

  Toate aceste rezultate sunt date pe fiecare procesor în parte şi pot fi afişate în consolă, sau pot fi salvate în format CSV[23] într-un fişier, prin parametrul de intrare „-r <file>”. În acest fel ele pot fi mai uşor interpretate, folosite.

22 Fişierele .properties sunt un tip de fişier text folosit în aplicațiile Java ca fişier de configurare. În esență un astfel de fişier mapează nume cu valori.
23 Comma Separated Values – valori separate prin virgulă

5. Simularea unei arhitecturi multicore cu memorie partajată

5.1. Benchmarkuri

Proiectanții de sisteme paralele se confruntă cu o problemă în ceea ce priveşte aplicațiile software. În acest moment există puține aplicații paralele reale pentru a le ghida modelele hardware, iar utilizatorii nu doresc să scrie noi aplicații pentru sisteme care nu există. Rezultatul este că studiile efectuate pentru a evalua caracteristicile sistemului îşi bazează adesea concluziile pe programe de "jucărie" care se aseamănă puțin, sau sunt doar o parte, din codurile pe care oamenii le vor rula de fapt pe aceste sisteme.

Benchmark-urile **SPLASH-2** [Woo95] modelează diferite probleme din domeniul ştiințific şi ingineresc, aplicațiile fiind concepute pentru dezvoltatorii de hardware şi software care lucrează în domeniul calculului paralel. Ele au fost dezvoltate la Universitatea Stanford în ideea de a oferi un grad de consistență si comparabilitate studiilor. Suita SPLASH-2 constă în 8 aplicații complete si 4 kernel-uri: Barnes, Cholesky, FFT, FFM, LU, Ocean, Radiosity, Radix, Raytrace, Volrend, Water-Nsq, Water-Sp.

Din cadrul acestei suite, am folosit următoarele benchmarkuri:

- **RADIX** este o aplicație care sortează numere întregi, procesând fiecare cifră a numărului în parte. Algoritmul realizează o singură iterație pentru fiecare cifră a cheilor. După fiecare iterație procesorul dă mai departe cheile asignate si generează o histograma locală. Histogramele locale sunt apoi acumulate într-o histogramă globală. În final fiecare procesor foloseşte histograma globală pentru a-şi permuta cheile într-o nouă listă pentru iterația următoare. În acest pas de permutare are loc o comunicare a fiecărui procesor cu fiecare.
- **LU** – acest kernel factorizează o matrice în produsul dintre matricea triunghiulară superioară şi cea inferioară. Matricea densă A (n* n) este descompusă într-un şir N*N de B*B blocuri (n=N*B) pentru a exploata localitatea temporală a elementelor din submatrici. Pentru a reduce comunicațiile, blocurile sunt asignate procesoarelor folosind un algoritm “2D scatter decomposition”, blocul fiind modificat doar de procesorul care îl deține. Dimensiunea blocului trebuie să fie destul de mare pentru ca rata de miss în cache să fie mică şi destul de mică pentru ca informația să fie distribuită uniform.
- **FFT** – constă într-un algoritm complex optimizat pentru a minimiza comunicarea dintre procesoare. Datele de intrare sunt n puncte complexe care vor fi transformate şi alte n puncte

complexe denumite radicalul unității (roots of unity). Ambele seturi sunt organizate ca matrice de $\sqrt{n} x \sqrt{n}$ partiționate astfel încât fiecărui procesor sa-i fie repartizat un set contiguu de coloane care sunt alocate în memoria lui locală. Fiecare procesor transpune o submatrice contiguă de $\sqrt{n}/p x \sqrt{n}/p$ de la fiecare procesor şi transpune o submatrice local.

Princeton **A**pplication **R**epository for **S**hared-Memory **C**omputers (**PARSEC**) [Bie08] este o suită foarte recentă de benchmark-uri compusă din programe multi-fir. Suita a fost creată la Universitatea Princeton în colaborare cu Intel. Benchmark-urile includ algoritmi importanți de data mining şi câteva aplicații de ultima oră de la Universitatea Princeton si Stanford: blackscholes (analiză financiară), bodytrack (computer vision), canneal (inginerie), dedup, facesim (animație), ferret (căutarea de similitudini), fluidanimate (animație), freqmine (data mining), streamcluster (data mining), swaptions (analiză financiară), vips (procesare media), x264(procesare media). Algoritmii implementati de aceste benchmark-uri sunt considerați folositori, dar cererile computaționale sunt considerate prea mari pentru platformele din prezent.

Din cadrul PARSEC am folosit în simulări aplicația BlackScholes. Aceasta este un benchmark Intel. Cu ajutorul ei se calculează prețul unui portofolio analitic folosindu-se de ecuația cu derivate parțiale Black-Scholes. Acest benchmark a fost ales pentru a reprezenta gama largă de programe folosite pentru a rezolva ecuațiile parțiale diferențiale. Programul este limitat de numărul de operații in virgulă mobilă pe care le poate efectua procesorul.

În afară de benchmarkurile descrise mai sus, am mai folosit şi trei aplicații dezvoltate de noi:

MERGESORT – este un algoritm de sortare de ordinul n log n. Avantajul acestui algoritm ar fi acela că el este foarte uşor de paralelizat, setul de intrare distribuindu-se in mod uniform pe fiecare procesor.

Paşii acestui algoritm sunt:

4. împarte lista nesortată în 2 de aceeaşi dimensiune
5. împarte fiecare dintre cele 2 liste recursiv pana când dimensiunea listei este 1, caz in care lista este returnată
6. interclasează cele 2 subliste într-una singură sortată

QUICKSORT – la fel ca şi MERGESORT acest algoritm de sortare este foarte uşor de paralelizat datorita naturii sale de divide et impera. Unul din avantajele quicksort-ului paralel față

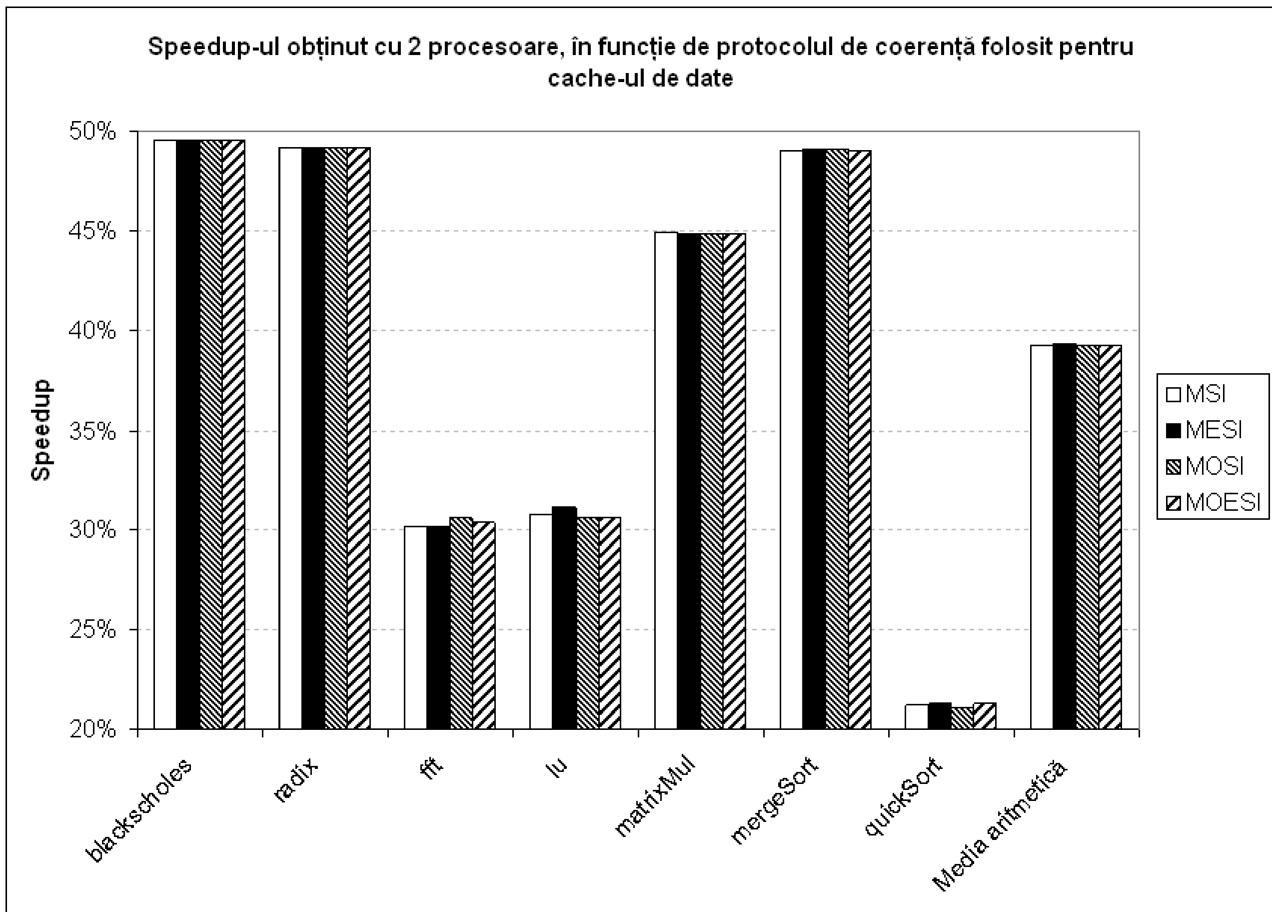
de alți algoritmi paraleli de sortare este acela ca nu este nevoie de nici o sincronizare. Un nou thread este pornit imediat ce o sublistă este disponibila pentru el. Când toate thread-urile se încheie, sortarea este terminata. Dezavantajul acestui algoritm este acela ca datele de intrare nu se partajează în mod uniform pe fiecare procesor.

MATRIXMUL – acest benchmark implementează algoritmul de înmulțire a doua matrice. Avantajul acestui algoritm este acela ca el se scalează bine pe n procesoare.

Benchmarkurile folosite din suita SPLASH-2, dar şi cel din cadrul PARSEC folosesc macrouri (aşa numitele PARMACS – PARallel MACroS) la nivelul codului C/C++, pentru a marca zonele sincronizare: bariere, lock, unlock. Astfel, utilizatorul poate să vină cu propria implementare a operațiilor de sincronizare dintre procesoare. În cazul nostru, aceste macrouri au fost înlocuite cu apeluri către funcții din API-ul Pthread.

5.2. Rezultate ale simulărilor

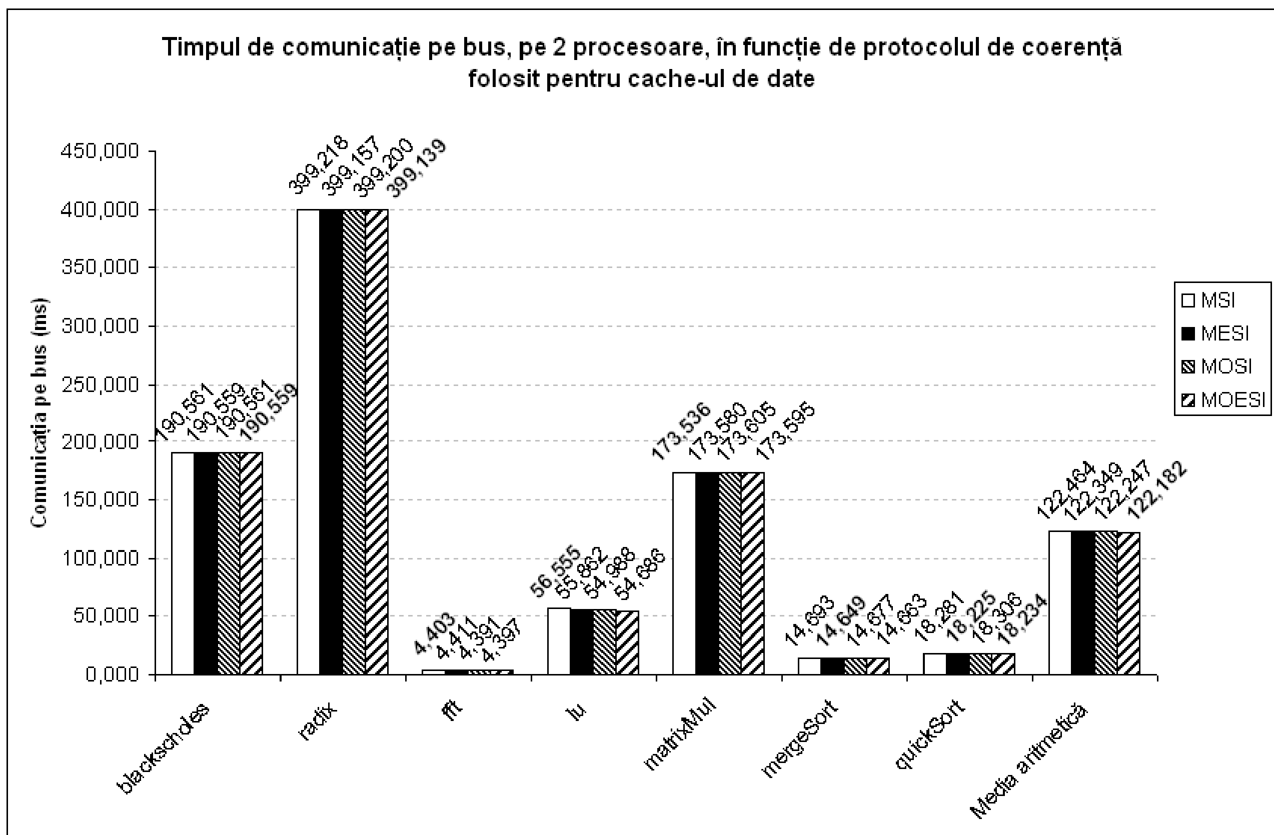
Prezentăm în cele ce urmează câteva din rezultatele obținute în urma simulărilor realizate cu ajutorul simulatorului PowerPC-TLM. Toate benchmarkurile au rulat cu valorile implicite, recomandate, ale paremetrilor de intare.



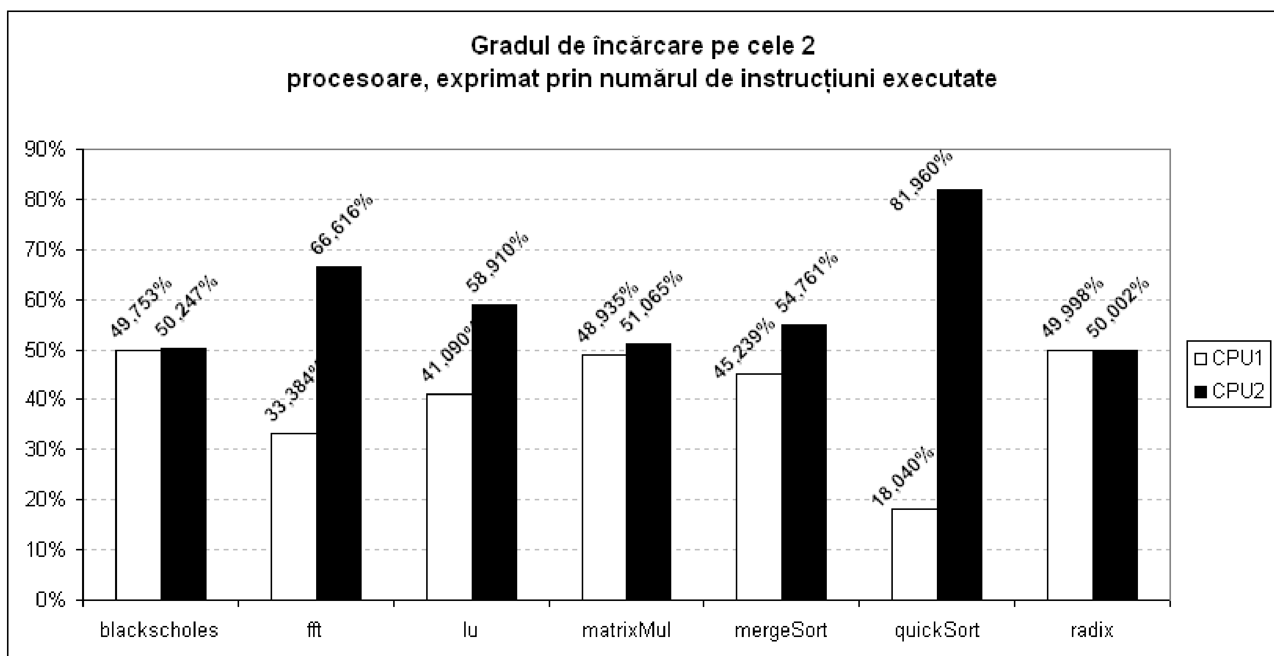
Acest grafic arată care este câştigul de performanță, în termeni de timp de comunicație la nivelul busului, comparând varianta uniprocesor cu varianta biprocesor. Se poate astfel observa că benchmarkurile blackscholes, radix (peste 49%) şi chiar mergesort au un speedup foarte apropiat de 50% (care înseamnă valoarea maximă, practic o dublare a timpului de execuție).

Benchmarkul quicksort, ale cărui valori de sortat nu se repartizează uniform pe cele două procesoare (datorită poziției nesimetrice a pivotului) duce la o accelerare de numai aproximativ 21%, comparativ cu mergesort care oferă un speedup de circa 49%.

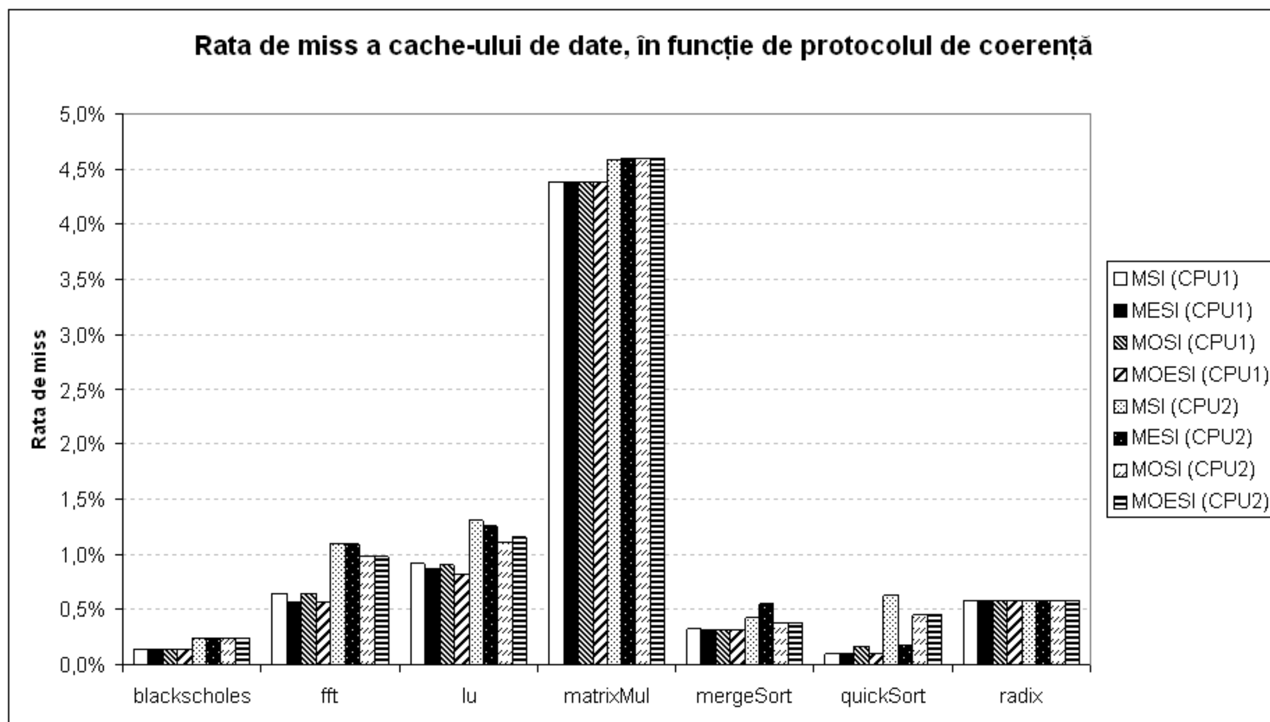
Ca o ultimă concluzie, merită remarcat că MESI duce per global la cea mai ridicată performanță, pe aceste simulări, de 39,3%, cu circa 0,1% mai mult decât MSI, MOSI, şi MOESI.



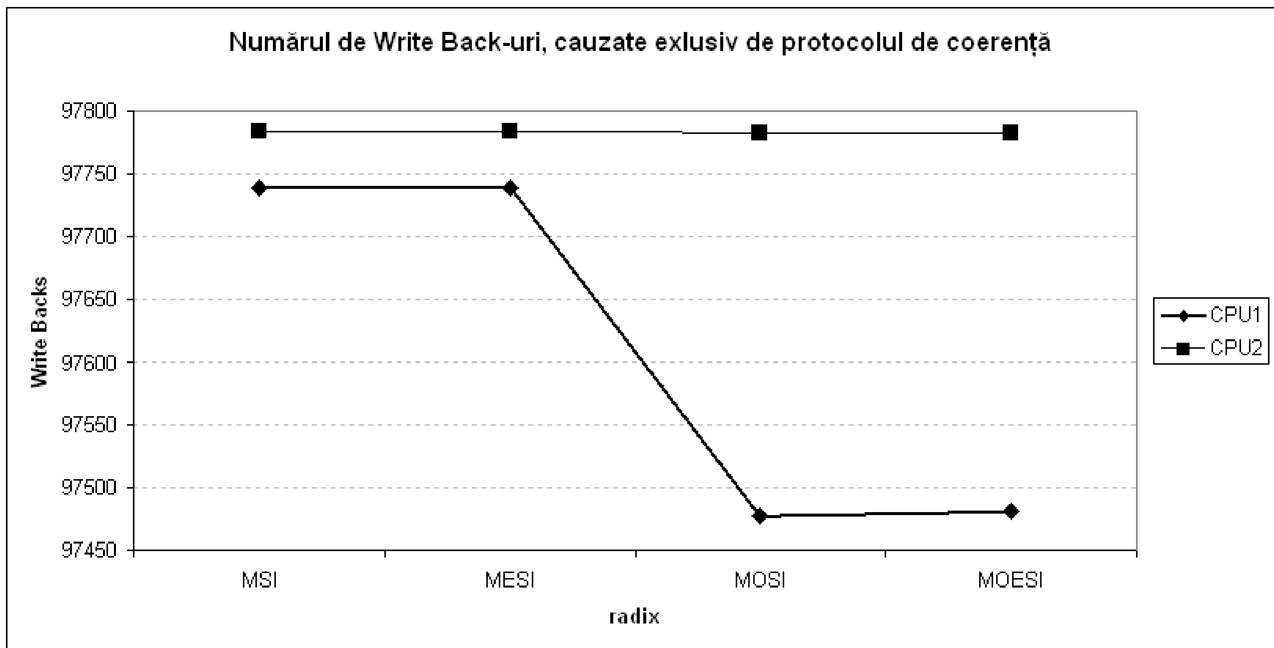
Acest grafic ilustrează care este timpul de comunicație pe bus, pe arhitectura cu 2 procesoare, în funcție de protocolul de coerență utilizat. În medie aritmetică, se constată o îmbunătățire a acestui timp, cu circa 0,1 ms, de la MSI (122,4) şi continuând progresiv către MESI (122,3), MOSI (122,2) şi MOESI (122,1).

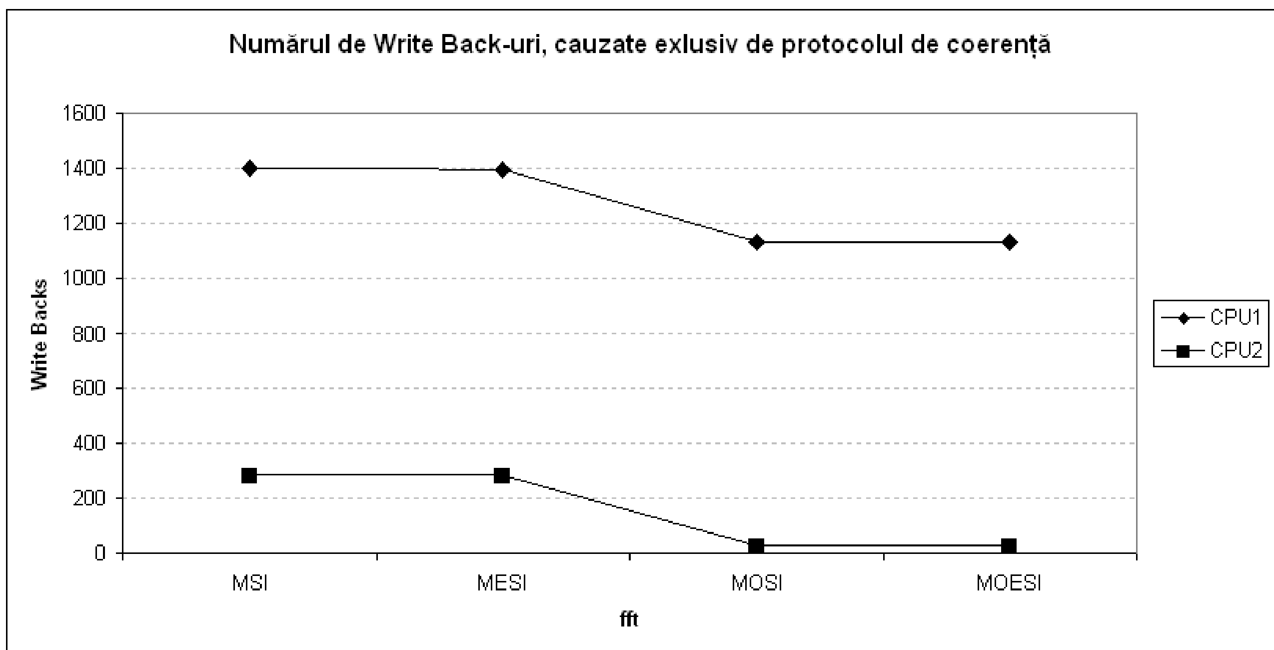


Următorul grafic ilustrează care este gradul de încărcare al celor 2 procesoare, în materie de număr de instrucțiuni executate de fiecare nucleu în parte. Se poate remarca din nou faptul că blackscholes şi radix sunt aplicații puternic paralelizate, la fel de altfel ca şi înmulțirea de matrice. Remarcăm de asemenea şi cât de puțin profită o sortare de tip quicksort de cele două procesoare. Practic, pe acest caz, rularea quicksort ar dura circa 80% din cât ar dura pe numai un procesor!. O metodă de sortare precum merge sort ar dura numai circa 55% din timpul pe un sistem uniprocesor.

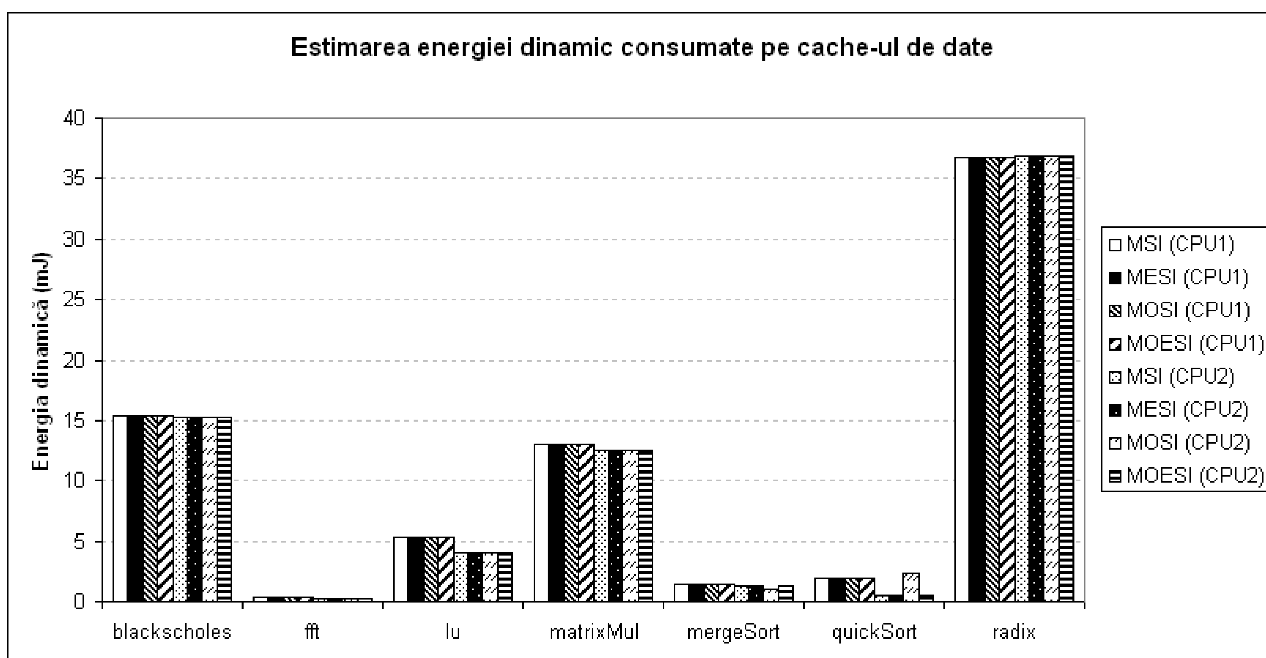


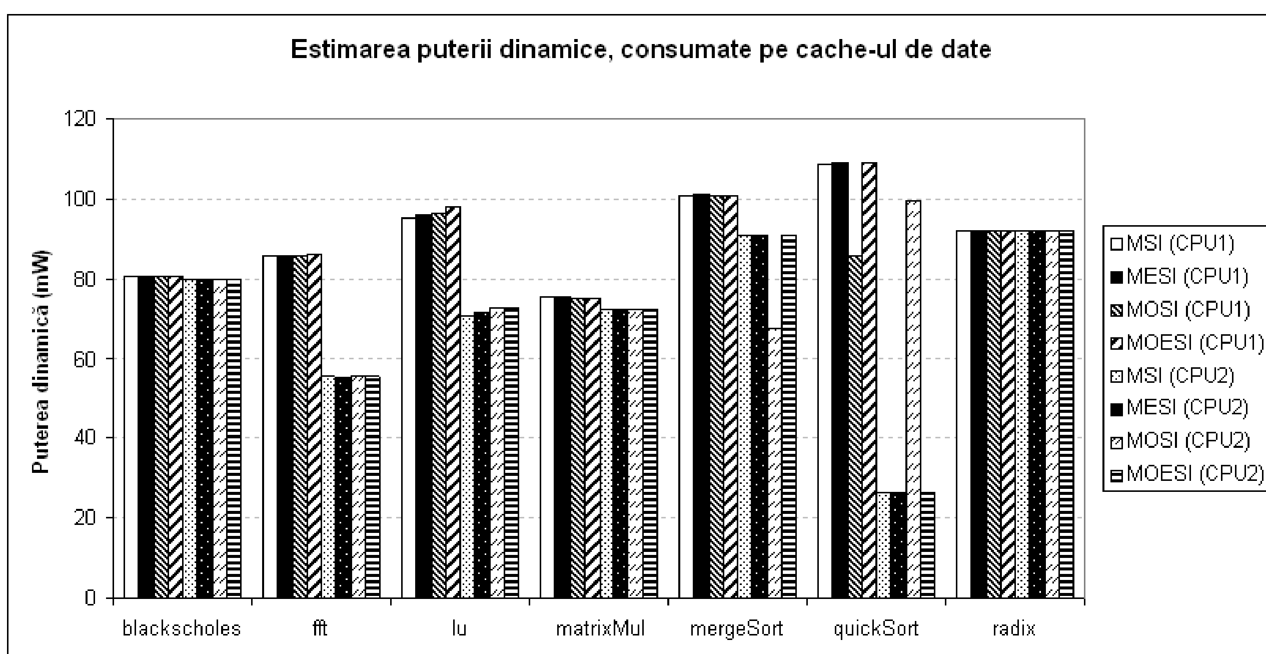
Ne-am propus de asemenea să evaluăm şi influența protocolului de coerență asupra ratei de miss a memoriei cache de date, pe ambele procesoare. Se poate observa că aceasta variază aproape insesizabil pe benchmarkurile puternic paralelizabile. Benchmarkurile LU şi FFT sunt mai puțin paralelizabile dar arată un lucru interesant: pe nucleul 1, protocoalele MESI şi MOESI au dat rezultate puțin mai bune, iar nucleul 2, protocoalele MOSI şi MOESI. Analizând şi rezultatele de la matrix mul, am putea trage concluzia că, cel puțin pe aceste simulări, protocolul MOESI a oferit cea mai mică rată de miss, fapt explicabil dacă ținem cont de însemnătatea stării Owned.



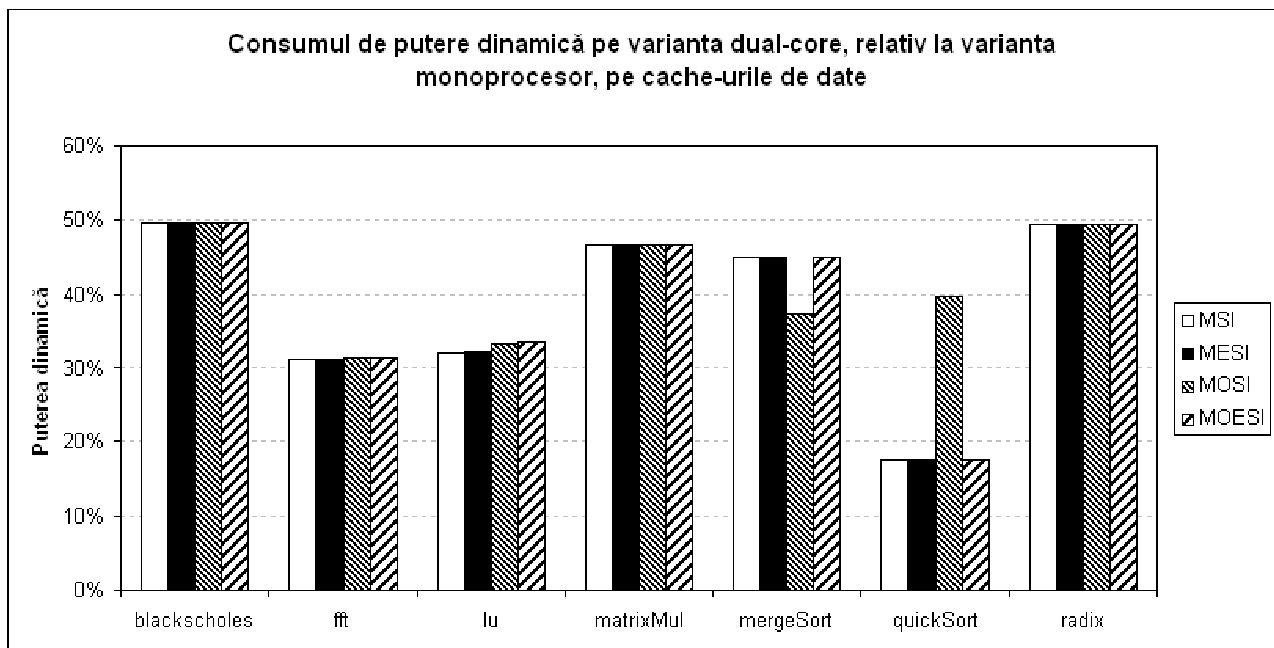


Cele două grafice de mai sus arată de altfel care este influența stării Owned asupra numărului de scrieri în memoria principală (Write Back), dictate de protocolul de coerență. Se observă că acest număr scade, atât pentru radix, cât şi pentru fft.





Graficele de mai sus arată o estimare a consumului de energie şi putere dinamică, pe cache-ul de date al ambelor procesoare, în funcție de protocolul de coerență utilizat. Din nou, benchmarkurile puternic paralelizate sugerează valori similare, pentru toate cele 4 protocoale de coerență, iar lu, fft şi mergesort puțin mai mare cu MOESI (protocolul cu cele mai multe stări), MSI (protocolul cu cele mai puține stări) fiind protocolul care consumă cea mai mică putere dinamică.



Acest ultim grafic ilustrează cum creşte consumul de putere dinamică, pe varianta cu două procesoare, relativ la arhitectura monoprocesor. În vreme consumul de energie a crescut în general cu un procent de sub 5%, se poate observa o similaritate mare, care există între creşterea procentuală a puterii dinamic consumate pe cache-urile de date în varianta biprocesor relativ la varianta cu un singur procesor şi accelerarea (primul grafic prezentat în cadrul acestei secțiuni cu rezultate) care se obține în ceea ce priveşte viteza de execuție a benchmarkurilor, în varianta paralelă, relativ la varianta secvențială. Se poate astfel remarca că, benchmarkuri foarte bine paralelizate, care încarcă aproximativ la fel (50% - 50%) cele două procesoare, cum sunt blackscholes şi radix, generează aproape o dublare a consumului de putere pe cache-urile de date.

5.3. Concluzii şi dezvoltări ulterioare

Rezultatele simulărilor prezentate anterior scot în evidență faptul că există o clară creştere de performanță, atunci când se foloseşte un sistem multicore, comparativ cu unul uniprocesor. Această creştere de performanță este însă condiționată de un aspect extrem de important: gradul de paralelism al aplicațiilor. Numai acele aplicații bine şi cât mai mult paralelizate vor duce la performanțe optime şi vor reuşi să profite la maximum de o arhitectură de calcul care beneficiază de mai multe microprocesoare. Quick Sort este una dintre cele mai rapide metode de sortare, dar acest lucru începe să dispară pe arhitecturile multicore deoarece metoda aceasta de sortare nu se scalează uniform pe n procesoare, aşa cum reuşeşte să o facă, prin natura algoritmului utilizat, metoda de

sortare Merge Sort.

De asemenea, am mai putut observa faptul că creşterea complexităţii protocolului de coerenţă, adăugarea, pornind de la MSI a stărilor Exclusive şi Owned au adus un plus de performanţă, care s-a concretizat printr-o scădere a timpului de comunicaţie pe bus (iar comunicaţia este unul din factorii cheie care contribuie la performanţa globală a arhitecturilor cu memorie partajată).

Cu toate acestea, complicarea protocolului de coerenţă pare să ducă la o uşoară creştere a consumului de putere al memoriilor cache.

Pentru obţinerea rezultatelor de mai sus, am folosit un simulator pus la dispoziţie în cadrul mediului UNISIM [UNI08]. Cu toate că simulatorul a fost proiectat pentru a cuprinde o arhitectură multicore, au mai fost necesare multe modificări pentru a se ajunge la o arhitectură multicore, cu memorie partajată simetric. Spre exemplu, busul nu făcea o distincţie în ceea ce priveşte semantica mesajelor primite de la procesoare, iar operaţiile de Lock şi Unlock, cele care permit intrarea unui proces într-o secţiune critică, erau implementate greşit, superficial. Acestea sunt însă doar două exemple care ridică însă o întrebare firească: este UNISIM un mediu de dezvoltare fezabil? În mod cert am putut ajunge la concluzia că UNISIM este încă în dezvoltare instabil, dar, pe de altă parte, cred că ideile şi principiile pe care cercetătorii din cadrul UNISIM doresc să le impună sunt demne de urmat. Mă refer la modularitate, la reutilizabilitatea simulatoarelor dezvoltate. În acest fel simulatoarele vor fi mai uşor de întreţinut, nu va mai fi necesar să existe persoane care să fie nevoite să aibă o înţelegere, completă, a tuturor componentelor care intră în componenţa unui simulator. Tot cei de la UNISIM propun simularea la nivel de sistem, adică se pune un accent nu doar pe partea hardware ci şi pe partea software. Ambele componente majore trebuie să conlucreze pentru a putea beneficia de un întreg sistem care să poată fi testat. Aşa cum am arătat şi prin simulatorul dezvoltat, software-ul, Sistemul de Operare, poate sprijini programarea paralelă: algoritmul lui Lamport este un algoritm de excluziune mutuală care rezită la nivelul S.O. şi care permite realizarea operaţiilor de Lock şi Unlock fără a fi necesare instrucţiuni atomice la nivelul procesorului.

Simulatorul dezvoltat foloseşte metodologia de modelare la nivel de tranzacţii (TLM), care oferă un avantaj major în cadrul utilizării unui simulator: viteză de execuţie. Desigur, câştigul de viteză creează o pierdere la nivelul acurateţii de simulare, dar acest dezavantaj nu este atât de mare: beneficiul major al acestei metodologii este acela că îţi dă de ales: rămâne la latitudinea dezvoltatorilor de simulatoare şi emulatoare să decidă cât de multă acurateţe vor să aibă modelul, unde este aceasta de fapt necesară. Merită subliniat faptul că TLM nu te restricţionează în ceea ce

priveşte gradul de acurateţe al modelului şi până la urmă, probabil că aici îi stă adevărata valoare: proiectantul are posibilitatea a balansa viteza modelului cu acurateţea acestuia după cum doreşte. Poate crea un model foarte simplu, de tip Untimed TLM, pe care îl are deci mult mai repede, în cadrul ciclului de dezvoltare. Apoi, acel model poate fi complicat progresiv, incremental (Timed TLM), devenind mai complex, mai precis, putând ajunge să fie chiar la fel de precis ca un model proiectat la nivel de tact.

În plus, modelarea la nivel de tranzacţii poate fi făcută cu succes cu ajutorul SystemC, de altfel un standard IEEE, care şi-a dovedit utilitatea şi valoarea în industrie. SystemC este fără îndoială benefic în dezvoltarea sistemelor dedicate, dar utilitatea lui am putut-o observa şi în dezvoltarea simulatorului folosit în cadrul acestei lucrări: poate fi dată, cred eu, ca exemplu în acest sens, modalitatea prin care am implementat sincronizarea la barieră.

În privinţa dezvoltărilor ulterioare ale acestui simulator, considerăm în primul rând că ar trebui dezvoltat simulatorul, generalizat suficient încât să permită conectarea la bus a mai mult de 2 procesoare, pentru a putea observa care sunt limitările unei arhitecturi multicore cu memorie partajată.

Un alt aspect care merită urmărit se referă la ierarhiile de memorii cache. Pentru a putea avea un protocol de coerenţă de tip write-invalidate care să funcţioneze pe sisteme de memorii cache cu 2 sau mai multe nivele, este necesar ca operaţiile de invalidare să se propage vertical, de sus în jos şi de jos în sus, pentru ca toate copiile unei locaţii de memorie, din toate cache-urile aflate în ierarhie, să fie invalidate, respectându-se astfel coerenţa.

Simulatorul dezvoltat permite în momentul de faţă numai varierea latenţei busului, dar este la fel de importantă şi lăţimea de bandă a acestuia. De fapt, lăţimea de bandă a magistralei este cea care devine insuficientă atunci când la bus se conectează tot mai multe procesoare.

Nu în ultimul rând, ar merita măsurat separat care este timpul pe care operaţiile de sincronizare (lock, unlock, bariere) îl ocupă. Cât de mult timp se pierde cu sincronizarea proceselor? Desigur, există aplicaţii paralele care comunică foarte puţin şi există altele care comunică mai mult, dar cu siguranţă merită măsurat cam cât timp este nevoit un procesor să aştepte, să-şi stagneze execuţia, din simplul motiv că aşteaptă un răspuns de la un alt procesor.

Bibliografie

[ACM88] *Reevaluating Amdahl's Law,* Communications of the ACM 31(5), 1988, pg. 532–33

[AMD07] *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, Advanced Micro Devices, Septembrie 2007, pg. 167-169

[Amd67] Amdahl G., *The validity of the single processor approach to achieving large-scale computing capabilities,* Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., Aprilie 1967, AFIPS Press, pg. 483–485

[Asa06] Asanovic K. et al., *The Landscape of Parallel Computing Research: A View from Berkeley,* University of California, Berkeley. Technical Report No. UCB/EECS-2006-183, 18 Decembrie 2006

[Ber66] Bernstein A., *Program Analysis for Parallel Processing*, IEEE Trans. on Electronic Computers, EC-15, Octombrie 1966, pg. 757–62

[Bie08] Bienia C., Kumar S., Singh J., Li K., *The PARSEC Benchmark Suite: Characterization and Architectural Implications*, Princeton University Technical Report TR-811-08, January 2008

[Bur04] Burt G., *Linux System Call Table*, Linux Assembly Tutorial (http://docs.cs.up.ac.za/programming/asm/derick_tut), 2004

[Cul99] Culler D., Singh J., Gupta A., *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999

[Ega07] Egan C., *An Introduction to Multiprocessor Systems* , Lecture Notes, Sibiu, 2007

[Fly04] Flynn L., *Intel Halts Development of 2 New Microprocessors,* The New York Times, 8 mai 2004

[Fre06] *MPC755 RISC Microprocessor Hardware Specifications* , Freescale Semiconductor Inc., Rev. 8, februarie 2006

[Gar06] Garg M., *Sysenter Based System Call Mechanism in Linux 2.6*, http://www.manugarg.com, 2006

[Ghe05] Ghenassia F., *Transaction-Level Modeling With SystemC. TLM Concepts and Applications for Embedded Systems*, Editura Springer, 2005

[Gir08] Girbal S., *Performing Native system calls for user-level simulator*, saitul UNISIM (https://unisim.org/site/os/native_syscalls/start), 2008

[Hen02] Hennessy J, Patterson D., *Computer Architecture: A Quantitative Approach*, ediția a 3-a, Morgan Kaufmann, 2002

[Hil06] Hill M., *Thread-Level Parallelism and Transactional Memory* , ACACES 2006. 23 – 29 iulie 2006, L'Aquila, Italia

[Jon07] Jones T., *Kernel command using Linux system calls*, IBM Developer Works (http://www-128.ibm.com/developerworks/linux/library/l-system-calls/), 2007

[Lam74] Lamport L., *A New Solution of Dijkstra's Concurrent Programming Problem,* Communications of ACM, august 1974, volumul 17, nr. 8, pg. 454 – 456

[Lam79] Lamport L., *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, C-28,9, Septembrie 1979, pg. 690–691

[Pat04] Patt Y., *The Microprocessor Ten Years From Now: What Are The Challenges, How Do We Meet Them?* Distinguished Lecturer talk at Carnegie Mellon University, Aprilie 2004

[PCM07] *MPP Definition,* PC Magazine, 2007

[Per07] Pérez G., Mouchard G., Carloganu A., *UNISIM Methodology for Transaction Level Modeling*, Conferința HiPEAC'07, 2007

[Pow00] *PowerPCTM Microprocessor Family: The Programming Environments for 32-Bit Microprocessors* , IBM, 2000

[Rab96] Rabaey J. M., *Digital Integrated Circuits*, Prentice Hall, 1996, pg. 235

[Roo00] Roosta, Seyed H. *Parallel processing and parallel algorithms: theory and computation*, Springer, 2000, pg. 114

[She05] Shen J., Lipasti M., *Modern Processor Design: Fundamentals of Superscalar Processors,* McGraw-Hill Professional, 2005, pg. 561

[Sys08] http://systemc.org

[UNI08] http://unisim.org

[Vin00] Vințan L., Florea A., *Microarhitecturi de procesare a informației*, Editura Tehnica,

Bucuresti, 2000

[Web08] *What is clustering?*, dicționarul Webopedia, 2008

[Woo95] Woo S., Ohara M., Torrie E., Singh J., Gupta A., *The SPLASH-2 Programs: Characterization and Methodological Considerations* , Proceedings of the 22nd Annual International Symposium on Computer Architecture, iunie 1995, pg. 24-36

[Zah03] Zahran M., On cache memory hierarchy for chip-multiprocessor, ACM SIGARCH Computer Architecture News, ACM, New York, SUA, martie 2003

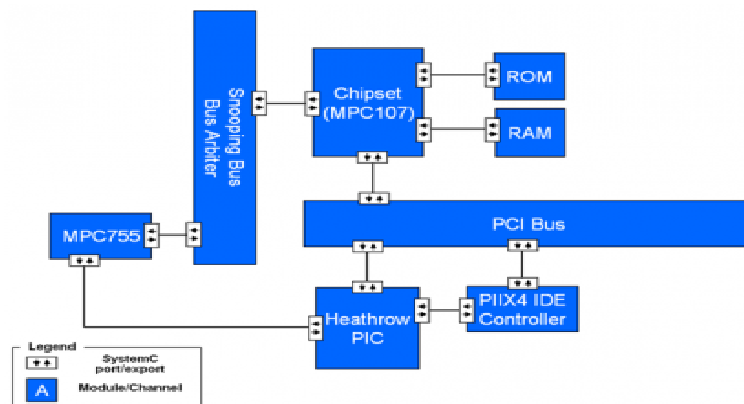
ANEXA

Instalarea simulatorului TLM PowerPC755 (snapshot august 2007)

Descriere

Acest simulator corespunde arhitecturii Mac G3 şi include:

- microprocesor MPC755;
- memorii;
- chipset MPC107;
- PIC (Programmable Interrupt Controller);
- PIIX4 IDE controller;
- Discuri IDE;
- Framebuffer display.



Simulatorul este capabil să booteze sistem de operare Linux, pentru PowerPC şi să ruleze majoritatea benchmarkurilor SPEC 2006.

Două versiuni ale simulatorului sunt disponibile:

- **ppcemu**: simulator al procesorului PowerPC 755, care nu include niciun periferic şi nu simulează sistemul de operare. Apelurile sistem sunt transmise sistemului de operare gazdă.
- **ppcemu-system**: simulator al procesorului PowerPC 755, care conţine întreaga arhitectură prezentată mai sus şi simulează sistemul de operare. Apelurile sistem sunt rulate în cadrul simulatorului şi se accesează periferice virtuale.

  Ambele variante ale simulatorului au fost realizate respectând metodologia TLM.

Instalarea simulatorului pe sistem de operare Windows XP™

Pentru instalarea acestui simulator pe sistem de operare Windows XP™, se utilizează emulatorul de Linux numit **Cygwin**. Acesta este disponibil la adresa http:// cygwin.com.
Simulatorul este disponibil la adresa https://unisim.org/site/simulators:tlm:mac-

g3:snapshot_2007_08.

Precizare: înainte de a începe efectiv cu prezentarea paşilor necesari pentru instalarea acestui simulator, merită menționat faptul că s-a preferat ca paşii realizați să fie mici, clari, implicând o abordare cât mai naturală, pornind de la instrucțiunile de instalare oferite de autorii acestui simulator. Paşii necesari instalării simulatorului puteau fi comprimați şi se puteau chiar omite unele etape, prin rezolvarea implicită a acestora. S-a considerat însă a fi mult mai utilă o abordare în mai multe etape, deoarece lămureşte mai bine de ce s-au făcut acele modificări şi în plus, prezintă probabil toate deosebirile care apar față de modul de instalare propus pe saitul Unisim.

Cerințe

Pentru compilarea şi instalarea acestui simulator sunt necesare următoarele unelte de dezvoltare:

- g++ (>= 4.0 recomandat)
- automake (>= 1.9.6 recomandat)
- autoconf (>= 2.61 recomandat)
- bison (2.3 recomandat) sau berkeley YACC (1.9 recomandat)
- flex (2.5.4 recomandat)
- libncurses-devel (5.5 recomandat)
- libSDL-devel (1.2.8 recomandat)
- boost (1.34.0 recomandat)
- boost-devel (1.34.0 recomandat)
- libreadline-devel (5.2 recomandat)
- libxml2 (2.6.28-2 recomandat)
- libxml2-devel (2.6.28-2 recomandat)

Exceptând libSDL-devel, toate uneltele de mai sus sunt disponibile via Cygwin (pachetul Devel). Aşadar, pachetele de mai sus (mai puțin SDL) se pot instala uşor folosind setup.exe, adică în aceeaşi manieră în care se instalează şi Cygwin.

Precizări:

1. în momentul redactării acestui document versiunea boost existentă în Cygwin era 1.33.1, care însă s-a folosit cu succes în locul versiunii recomandate (1.34.0).
2. în continuare se presupune că Cygwin este instalat în C:\cygwin.

Simulatorul se bazează pe SystemC, versiunea 2.2, care poate fi găsit aici: http://www.systemc.org (este necesară o înregistrare gratuită). SystemC este un limbaj de descriere de nivel înalt bazat pe

C++, similar cu VHDL sau Verilog, dar fiind mai complex decât acestea deoarece permite o descriere la nivel de sistem, nu doar la nivel hardware.
Urmează procesul de instalare al uneltelor necesare simulatorului. O serie de fişiere necesare sau utilte procesului de instalare există pe grupul ACAPS_UNISIM, aflat la adressa http://groups.yahoo.com/group/acaps_unisim/, secţiunea **Files > Docs > Install notes**. În continuare se vor face implicit referiri la această locaţie.

Instalarea SDL

Simple DirectMedia Layer este o librărie multimedia independentă de platformă, realizată pentru a furniza acces de nivel jos, pentru audio, tastatură, maus, joystick, hardware video 2D (framebuffer video), dar şi 3D, via OpenGL.
Pentru instalare, trebuie descărcate două arhive, care se găsesc la adresele http://www.libsdl.org/release/SDL-1.2.11-win32.zip şi http://www.libsdl.org/release/SDL-devel-1.2.11-mingw32.tar.gz
Cele două fişiere se află în secţiunea compilare Unisim, de pe ACAPS_UNISIM.
Se dezarhivează **SDL-devel-1.2.11-mingw32.tar.gz** în directorul home al contului Linux (ex.: C:\cygwin\home\admin). Folosind comanda cd, se navighează în directorului tocmai creeat (sdl-1.2.12), după care se dă comanda
make install-sdl prefix=/cygdrive/c/cygwin/usr
(vezi instalare_sdl.png din secţiunea compilare Unisim)
Dacă nu funcţionează comanda de mai sus se recomandă folosirea comenzii
make native
Procesul de instalare SDL ar trebui sa decurga fara probleme, având pachetele enumerate mai sus instalate în Cygwin. Pentru informaţii suplimentare cu privire la instalarea SDL este recomandată citirea fişierului README.
SDL-1.2.11-win32.zip conţine un singur fişier, SDL.dll, care trebuie copiat în C:\cygwin\usr\bin. Dacă s-a utilizat comanda de instalare nativă, copierea acestui fişier nu mai este necesară.

Instalarea SystemC

Pentru instalarea systemC, pe sistem de operare Windows, se foloseşte o versiune modificată a acestuia, care se regăseşte la secţiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08 > systemc-2.2.0-PATCHED.zip. Această versiune modificată a fost obţinută prin

aplicarea unui patch, ce se poate găsi în depozitul (repository) SVN al Unisim, directorul projects/systemc (**https://unisim.org/svn**), folosind pentru autentificare utilizator guest şi nicio parolă.

Se dezarhivează fişierul într-o locaţie oarecare pe hard disk, în directorul systemc-2.2.0-PATCHED. Totodata, se creează un director numit systemc-2.2.0-PATCHED-install, locaţia fiind la alegere. În acest director va fi instalat systemC.

Se deschide cygwin şi se navighează către directorul tocmai creeat, folosind comanda cd, după care se execută comenzile de mai jos.

Precizare: fişierul configure probabil nu va fi bun (a se face distincţie faţă de configure.in, care este un alt fişier). Vor apare erori atunci cand se dă comanda configure). Dacă este aşa, se recomandă redenumirea acestuia în _configure şi rularea comenzii autoconf, din cygwin. Folosind autoconf, se regenerează fişierul configure. Un file-compare între _configure şi noul fişier configure poate evidenţia modificările făcute. Se poate apela autoconf --help pentru detalii privind unealta autoconf (acest document nu se doreşte a fi un tutorial de comenzi Linux).

Se navighează în systemc-2.2.0-PATCHED-install/include/sysc/kernel şi se modifică fişierul sc_constants.h, înlocuind 0x10000 cu 0x50000 (linia 57). Fişierul gata modificat este disponibil în secţiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08.

Modificarea de mai sus semnifică schimbarea dimensiunii prestabilite a unei stive folosită de SystemC. Necesitatea realizării acestei modificări se poate vedea numai la sfârşitul instalării simulatorului, când se încearcă testarea acestuia. Dacă modificarea de mai sus nu se realizează, orice încercare de utilizare a simulatorului se va termina cu neîndeplinerea următoarei aserţiuni: *m_stack_size > (2 ** pagesize). Creşterea aşadar a dimensiunii stivei rezolvă această problemă.

Tratarea problemei împreună cu soluţia evidenţiată mai sus pot fi găsite la adresa http://www.ti.cs.uni-frankfurt.de/pipermail/systemc-ams/2006-July/000061.html.

Se rulează apoi comenzile următoare

```
mkdir objdir
cd objdir
../configure --prefix=/cygdrive/f/facultate/diploma/systemc/systemc-2.2.0-PATCHED-install
CXX=g++
make
make install
cd ..
rm –rf objdir
```

unde f/facultate/diploma/systemc/ reprezintă locaţia unde s-a creat directorul systemc-2.2.0-

PATCHED-install.

După instalare, se înlocuieşte fişierul systemc.h din systemc-2.2.0-PATCHED-install/include cu cel aflat în secțiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08 (modificarea constă în comentarea liniilor 175-181 folosind //).

O ultimă modificare mai trebuie realizată: se navigheză în **C:\cygwin\lib\gcc\i686-pc-cygwin\3.4.4\include** şi se creează fişierul **float.h**, care va avea exact acelaşi conținut ca şi fişierul **values.h**.

Instalarea simulatorului

Ne aflăm în etapa în care avem toate uneltele necesare simulatorului. Urmează un set de modificări privind configurarea procesului de instalare.

- Se navigheză în directorul **C:\cygwin\usr\include\boost-1_33_1**. Tot conținutul de aici se copiază sau se mută într-un director nou, numit **boost**. Procesul de compilare va căuta într-un subdirector boost, nu direct în boost-1_33_1. Este posibil ca, dacă se foloseşte versiunea 1.34.0 de boost, acest pas să nu mai fie necesar.
- Se navigheză în directorul **C:\cygwin\usr\include**. Aici, se creează directorul **libxml**, în care se copiază tot directorul **libxml2**, împreună cu conținutul său, existent în calea evidențiată mai sus. Şi această modificare se face tot pentru ca procesul de instalare să poată regăsi biblioteca libxml2.
- Se dezarhivează conținutul fişierului powerpc755-tlm-snapshot-2007-08.tar.gz, locația fiind la alegere. Rezultă directorul **PowerPC755-TLM-snapshot-2007-08**, care conține simulatorul ce va fi instalat.
- Se navighează către locația unde s-a dezarhivat simulatorul (folosind cygwin), după care se apelează comanda de configurare:

./configure --prefix=/cygdrive/f/facultate/diploma/unisim/TLM-PowerPC --with-systemc=/cygdrive/f/facultate/diploma/systemc/systemc-2.2.0-PATCHED-install

  f/facultate/diploma/unisim/TLM-PowerPC reprezintă locul în care va fi instalat simulatorul (directorul TLM-PowerPC nu trebuie creeat pentru că va fi creeat automat)

  /f/facultate/diploma/systemc/systemc-2.2.0-PATCHED-install reprezintă calea către SystemC instalat.

- După ce configurarea se termină, rezultă, în directorul în care s-a dezarhivat simulatorul, fişierul Makefile, care va suferi 3 modificări:

  - se modifică calea către libxml2, linia 5 devine:

  LIBXML2_PATH := /cygdrive/c/cygwin/usr/include/libxml2

- se adaugă

  BOOST_PATH := /cygdrive/c/cygwin/usr/include/boost-1_33_1

  pe linia imediat următoare celei care specifică calea către systemC
- se modifică linia 56 în

  cd $(FULLSYSTEM_PATH); ./configure --prefix=$(INSTALL_PATH) --with-libxml2=$(LIBXML2_PATH) --with-systemc=$(SYSTEMC_PATH) --with-simfloat=$(INSTALL_PATH)/include --with-genisslib=$(INSTALL_PATH)/bin/genisslib --disable-armemu --with-boost=$(BOOST_PATH)

  (s-a adăugat practic --with-boost=$(BOOST_PATH) la comanda de configurare, din target-ul make_fullsystem)

Toate aceste modificări se regăsesc în fişierul PATCH-Makefile din ACAPS_UNISIM, secţiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08

- Se navighează în directorul **PowerPC755-TLM-snapshot-2007-08\TLM-PowerPC\src\tlm\pci**, unde se modifică fişierele **pci_ide_module.hh** şi **pci_net_module.hh**. Modificarea constă în eliminarea cuvântului template din declararea PciConfigData. Cele două fişiere se găsesc gata modificate la secţiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08.
- Se navighează în directorul **PowerPC755-TLM-snapshot-2007-08\TLM-PowerPC\src\plugins\os\linux** şi se modifică headerul **cpu_linux_os_interface.hh**, prin înlocuirea structurii stat64 cu stat. Explicaţia acestei modificări constă în faptul că cygwin nu lucrează cu stat64. Mai multe detalii cu privire la această modificare sunt disponibile aici: http://www.cygwin.com/faq/faq.programming.html#faq.programming.stat64. Aceeaşi modificare se face asupra fişierelor **arm_linux_os.hh** şi **arm_linux_os.cpp** din subdirectorul arm (al locaţiei evidenţiată mai sus), dar şi **asupra powerpc_linux_os.hh** şi **powerpc_linux_os.cpp** din subdirectorul powerpc. Cele 5 fişiere gata modificate se află la secţiunea compilare Unisim > PowerPC755-TLM-snapshot-2007-08.
- Acum se poate porni instalarea simulatorului.

  Din cygwin se navighează către directorul care conţine simulatorul şi se apelează comenzile specifice procesului de instalare GNU:

  make (~ 45-50 minute; depinde evident de maşina pe care se face instalarea, dar s-a dorit evidenţierea faptului că acest proces durează mai mult**)**

  make install (~ 4-5 minute)

  (comanda configure am rulat-o deja la pasul 4)

  Dacă totul este în regulă, se va obţine mesajul Compile successful la finele execuţiei make şi

mesajul Instalation successful, după ce comanda make install se termină.

Se pot rula apoi cele două teste, care verifică dacă instalarea s-a făcut corect:

make test-user

make test

Prima comandă testează varianta de simulator **ppcemu**, iar a doua rulează varianta **ppcemu-system** care va boota un nucleu de sistem de operare Linux, pe arhitectura simulată.

Este recomandat ca primul test să fie cel pt. ppcemu, pentru că durează mult mai puțin. Pentru detalii cu privire la ce trebuie să se obțină în urma celor două teste, se recomandă să se viziteze https://unisim.org/site/simulators:tlm:mac-g3:snapshot_2007_08.

Instalarea simulatorului pe sistem de operare Linux (Ubuntu 7.10, Gutsy)

Instalarea acestui simulator se poate face mult mai uşor pe un sistem de operare Linux. S-a folosit pentru instalare Ubuntu Gutsy.

Pachetele necesare simulatorului, prezentate la începutul acestui document (secțiunea Cerințe), pot instalate uşor folosind Synaptic. Se foloseşte comanda

apt-get install <nume_pachet>

Mai multe informații cu privire la această comandă sunt disponibile aici: http://www.debian.org/doc/manuals/apt-howto/ch-apt-get.en.html#s-install

Exemplu de utilizare: **sudo apt-get install libsdl-dev** (sudo se foloseşte ca prefix al comenzii deoarece sunt necesare drepturi de administrator).

SystemC se instalează simplu, folosind versiunea disponibilă pe saitul producătorului. Nu se fac modificări asupra versiunii (aşa cum sunt cele făcute pentru instalarea sub Windows) ci se rulează direct comenzile de instalare.

De asemenea nu se face absolut nicio modificare asupra fişierelor din cadrul simulatorului. Cu alte cuvinte, pentru instalarea simulatorului sub Linux, trebuie doar instalare pachetele necesare şi apoi rulat procesul se instalare.

În loc de concluzie

Se recomandă instalarea acestui simulator pe un sistem de operare Linux deoarece procesul de instalare este mult mai simplu pe Linux decât pe Windows şi durează mai puțin timp (aproximativ 25% - 35% din timpul de instalare sub Windows).

Pentru orice probleme legate de procesul de instalare al acestui simulator puteți trimite un email la

radu_ciprianro@yahoo.com cu rugămintea ca întrebările adresate să fie referitoare la probleme care nu sunt incluse în acest document, la potențiale greşeli, scăpări…

Dacă aveți nelămuriri legate de instalarea cygwin, lucrul cu comenzi Linux, aveți la dispoziție o sinteză foarte bună în cartea profesorului Adrian Florea (Universitatea "Lucian Blaga" Sibiu), intitulată "Predicția dinamică a valorilor în microprocesoarele generației următoare", editura Matrix Rom, Bucureşti, 2005, paginile 96-144, unde se prezintă instalarea setului de simulatoare SimpleScalar, în contextul unei abordări mai ample, Metodologia de simulare.