Multiprocesoare simetrice: coerenţa cache-urilor
Seria: arhitectura modernă a calculatoarelor

Mihai Budiu — mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

octombrie 1998

Subiect: cum se implementează coerenţa cache-urilor în maşinile multiprocesor

Cunoştinţe necesare: cache, programare în limbaj maşină, noţiuni elementare despre arhitectura calculatoarelor

Cuvinte cheie: multiprocesoare simetrice, coerenţă, cache, atomat finit

Cuprins

În ultimele numere din PC Report am văzut o serie de articole deosebit de interesante despre arhitectura internă a unor procesoare moderne (Pentium, AMD K6, etc.). Articolele erau bine documentate, şi prezentau o cantitate impresionantă de informaţii într-un spaţiu mai curînd restrîns.

Ca atare mi-am propus să scriu o întreagă serie de articole numite generic “arhitectura modernă a calculatoarelor” în care să explic pe îndelete noţiunile esenţiale care apar în proiectarea acestor bijuterii tehnologice. Voi încerca pe rînd să “demistific” noţiuni ca “superscalar”, “pipelined”, “staţie de rezervare”, “paralelism la nivel de instrucţiune”, “multiprocesoare simetrice”, “redenumirea regiştrilor”, “forwarding”, “protocol de coerenţă”, “predicţia salturilor”, “execuţie speculativă”, “prefetching”, etc.

Adevărul este că în ultimii 10 ani arhitectura calculatoarelor a suferit nişte modificări uriaşe. În urmă cu 20 de ani gama de putere şi preţ ale calculatoarelor avea extreme foarte îndepărtate; PCul era o jucărioară în comparaţie cu staţiile de lucru, care la rîndul lor erau puse în umbră de “mainframes”; la vîrful ierarhiei tronau supercalculatoarele. Cu fiecare an trecut, extremele s-au apropiat simţitor; practic toate firmele care făceau numai supercalculatoare au dat faliment[1], iar diferenţele între puterea de calcul a unui PC şi a unei staţii de lucru s-au redus enorm; idei care se foloseau pe vremuri în proiectarea supercalculatoarelor sunt acum în mod comun implementate în procesoarele PCurilor de pe biroul fiecăruia. Complexitatea design-urilor şi ingeniozitatea soluţiilor sunt extrem de surprinzătoare pentru cel care nu a rămas la curent cu evoluţia domeniului fie şi pentru puţină vreme.

Complexitatea aparatelor este uriaşă, iar viteza cu care apar inovaţiile depăşeşte cu mult viteza cu care pot tasta eu texte, aşa că nu voi putea face decît o ilustrare a conceptelor esenţiale, care sper să poată da o idee generală despre anatomia unui calculator contemporan.

Prima tema pe care mi-am ales-o are de-a face cu arhitectura calculatoarelor cu mai multe procesoare; din spectrul larg de implementări (despre care vorbesc pe scurt în secţiunea 2) voi ataca o singură opţiune, dar care este dominantă economic: cea a multiprocesoarelor simetrice cu memorie partajată bazate pe o magistrală comună.

1 Paralelism şi performanţă

De vreo douăzeci de ani încoace se prezice moartea paradigmei arhitecturale numită “von Neumann”, a calculatoarelor cu un singur procesor şi o unitate de memorie. Unii experţi se încăpăţînează să prezică plafonarea performanţelor sistemelor bazate pe astfel de arhitecturi, iar fabricanţii de sisteme se încăpăţînează să contrazică prin sistemele construite predicţiile sumbre. În trecere vreau să observ că numele “arhitectură von Neumann”, folosit adesea peiorativ pentru a indica o arhitectură depăşită, nu este întru totul potrivit; John von Neumann, care a murit în 1956, a fost conştient pe deplin de importanţa paralelismului; a şi scris nişte cărţi foarte interesante pe tema asta; von Neumann însă era un inginer extraordinar (de fapt era profesor de matematică la Princeton, cu o diplomă în inginerie chimică de la universitatea din Budapesta, dar nu o să ne incomodeze astfel de distincţii din a-l trata drept “inginer”), şi a realizat că pentru a rezolva problemele ele trebuie adesea simplificate, sparte în bucăţi mai mici care sunt apoi tratate independent. Starea catastrofală a componentelor electronice disponibile pe vremea aceea (relativ la ziua de azi) făcea proiectarea şi întreţinerea unui sistem atît de complex ca un calculator o sarcină supraomenească, aşa că von Neumann a făcut problema tractabilă şi a rezolvat-o cu atît de mult succes încît şi acum, după 50 de ani, organizarea propusă de el este dominantă.

[1] La ora asta supercalculatoare mai fabrică numai Intel, IBM, Silicon Graphics — care a cumpărat în 1987 cea mai celebră firmă de supercalculatoare, Cray Research — Hitachi şi dacă nu mă înşel Fujitsu.

Oricum, fie că se plafonează sau nu performanţele unui microprocesor, să ai mai multe nu poate să strice prea tare, nu? Unde-s mai multe capete se gîndeşte mai bine. Că lucrurile nu stau întotdeauna aşa reiese din următoarea întrebare: "dacă un ins sapă o groapă în 60 de secunde, 60 de inşi sapă o groapă într-o secundă?" E (aproape) clar că nu orice poate fi rezolvat în paralel cu eficacitate. Asupra consecinţelor acestui enunţ, care sunt exprimate şi de celebra lege a lui Amdahl, vom reveni în alte articole.

Oricum, anumite activităţi se pot într-adevăr face relativ repede dacă avem mai multe resurse computaţionale la-ndemînă. De exemplu, într-o întreagă suită de articole despre sistemele de operare, am observat că sistemele de operare moderne permit executarea simultană a mai multor programe printr-un mecanism simplu numit time-sharing, care constă în executarea întreţesută a grupe de instrucţiuni din programe diferite (adică programul 1 merge 10 milisecunde, după care programul 2 10 milisecunde, etc.). Dacă avem mai multe procesoare, atunci putem executa cu adevărat mai multe programe deodată: cîte unul pe fiecare procesor.

2 Multiprocesoare

Există mai multe feluri de maşini cu mai multe procesoare, dar noi ne vom ocupa de unul singur în acest articol. Celelalte variante merită desigur menţiune; să le-o dăm.

Putem împărţi multiprocesoarele în două mari categorii, care diferă din modul în care se accesează memoria în care se află programele şi datele.

- Prima variantă pune toate procesoarele la un loc, ca utilizatoare ale unei singure memorii. Aceste sisteme se numesc cu memorie partajată (*shared memory*).

  Există două feluri mari de sisteme cu memorie partajată:

  - Sisteme cu acces uniform la memorie (UMA, *uniform memory access*): dacă oricare ar fi sursa şi destinaţia unui cuvînt de date accesat timpul este cam acelaşi. Noi vom vorbi în acest articol despre o clasă de maşini UMA numite multiprocesoare simetrice (SMP: *Symmetric Multi-Processors*). Vom vedea imediat de ce se numesc aşa.
  - A doua variantă se numeşte NUMA: de la ne-uniform. În astfel de sisteme fiecare procesor are propria lui memorie la care poate ajunge repede, iar accesul la memoria altuia ia ceva mai mult timp.

- În fine, avem sistemele la care un procesor nu poate accesa nicicum memoria altuia, iar comunicaţia între procesoare se face prin mesaje (*message passing*).

Trebuie spus că tipul în care se încadrează o maşină depinde foarte mult de nivelul la care o privim. De pildă dacă luăm o colecţie de staţii de lucru, la nivel hardware acestea categoric comunică prin mesaje. Pe de altă parte putem instala pe ele un nivel software (cum ar fi de pildă sistemul de operare Mach), care folosind tehnici de memorie virtuală permite maşinilor să partajeze un acelaşi spaţiu de adrese virtual, astfel încît ce scrie o maşina la adresa 5 celelalte pot citi de la adresa 5; în cazul ăsta maşinile au ceea ce se numeşte memorie distribuită partajată (DSM: *Distributed Shared Memory*), care este un tip de NUMA.

Un alt exemplu despre fragilitatea clasificării: chiar maşinile cărora acest articol le este dedicat, multiprocesoarele simetrice, sunt categorisite drept maşini UMA. Dar vom vedea că fiecare procesor dintr-un sistem SMP are de fapt un cache (o memorie locală mică), la care accesul este mult mai rapid decît la memoria globală a sistemului; la acest nivel maşina arată de fapt ca o NUMA...

O maşină cu mai multe procesoare se numeşte "simetrică" dacă toate procesoarele au roluri egale: fiecare procesor poate executa orice tip de cod. Prin contrast, maşinile asimetrice dedică anumite procesoare anumitor treburi, cum ar fi executarea codului întreruperilor, executarea codului sistemului de operare, etc. Pe noi ne vor interesa procesoarele simetrice pentru că sunt cele mai reprezentate în maşinile comerciale; există şi PCuri cu 2 sau 4 procesoare Pentium.

În fine, vom mai disocia calculatoarele cu memorie partajată în două categorii distincte, după felul în care ajung la memorie. Tipul care ne va interesa, şi din care cele mai multe maşini SMP fac parte, este cel al maşinilor bazate pe o magistrală comună (bus): toate procesoarele pun cererile de acces la memorie pe o singură sîrmă. Vom vedea că acesta este un factor crucial. PCurile SMP sunt de acest tip, ca şi cele mai multe maşini comerciale (staţiile ULTRASparc, SGI Challenge, etc.)

Pe de altă parte, există maşini care au o reţea de interconectare *interconnection network* între procesoare şi (de obicei multiplele) module de memorie.

```
sistem bazat pe magistrala comuna     sistem bazat pe retea de interconectare
     -----------                           ------------------
     | memorie |                           |    retea de    |
     -----------                           | interconectare |
          ||                               ------------------
     ===========   magistrala                |    |    |
     |    |    |                           ---- ---- ----
    ---- ---- ----                         |m1| |m2| |m3| memorie locala
    |c1| |c2| |c3| cacheuri                ---- ---- ----
    ---- ---- ----                           |    |    |
     |    |    |                           ---- ---- ----
    ---- ---- ----                         |c1| |c2| |c3| cacheuri
    |p1| |p2| |p3| procesoare              ---- ---- ----
    ---- ---- ----                           |    |    |
                                           ---- ---- ----
                                           |p1| |p2| |p3| procesoare
                                           ---- ---- ----
```

Diferenţa esenţială este de *scalabilitate*: este extrem de greu de construit o maşină bazată pe bus cu multe procesoare, din cauză că, aşa cum e uşor de imaginat, de de la un anumit număr de procesoare încolo bus-ul devine factorul care limitează viteza de comunicare.

Pe de altă parte, maşinile cu reţea de interconectare pot creşte mai bine, pînă la sute sau chiar mii de procesoare (maşina IBM care l-a bătut pe Kasparov, un SP2, e bazată pe o reţea care interconectează pînă la o mie de procesoare).

Trebuie spus că multiprocesoarele sunt încă departe de a fi panaceul universal în a rezolva probleme computaţionale dificile, mai ales din cauză că scrierea de programe pentru astfel de maşini este extrem de dificilă. Oricum, cercetarea în domeniu este în continuare febrilă, şi o mulţime de schimbări revoluţionare aşteaptă probabil la cotitură.

3 Cache-uri

Chiar dacă avem un singur procesor, viteza de răspuns a unei memorii DRAM este de vreo 15 ori mai mică decît viteza cu care procesorul doreşte date şi instrucţiuni, iar disparitatea de viteză tinde să crească. Din cauza aceasta fiecare procesor modern este echipat cu o memorie SRAM rapidă, plasată foarte aproape de procesor, numită cache. (SRAMul este mult mai scump decît DRAMul; pentru acest motiv nu înlocuim tot DRAMul cu SRAM.) Despre cache am scris cel puţin două articole în PC Report: unul în martie 1997 şi unul luna trecută, în octombrie 1998. Dacă nu aveţi revistele sunteţi bineveniţi să luaţi articolele din pagina mea de web. Cu toate astea vom vedea că subiectul nu a fost epuizat, pentru că miezul acestui articol tot despre cache-uri va vorbi.

Ideea de bază este relativ simplă: programele au nevoie foarte rar de o cantitate mare de date sau instrucţiuni simultan; ele tind să refolosească de mai multe ori o cantitate relativ mică de date. Această proprietate, verificată empiric, se numeşte “localitate”. Din cauza asta un cache este util: dacă cuprinde toate datele necesare, sau măcar o mare parte dintre ele, atunci eficienţa creşte simţitor, pentru că timpul mediu de acces este redus simţitor.

Procesoarele dintr-un sistem SMP nu fac excepţie: fiecare este echipat cu un cache (sau chiar două, de mărimi diferite, unul fiind mai mare şi mai lent).

3.1 Problema coerenţei

Toate bune şi frumoase, aparent: în loc să punem un procesor pe magistrală punem două, şi maşina merge de două ori mai repede. Hopa, să nu ne grăbim. O grămadă de probleme urîte apar de îndată.

Să luăm un exemplu simplu, în care două procesoare accesează aceeaşi adresă de memorie. Iată o posibilă evoluţie în timp:

	Instrucţiune		cache		
Timp	Procesor 1	Procesor 2	Procesor 1	Procesor 2	Memorie
0					mem[5]=3
1	read 5		cache[5]=3		mem[5]=3
2		read 5	cache[5]=3	cache[5]=3	mem[5]=3
3	write 5, 4		cache[5]=4	cache[5]=3	mem[5]=3
4		write 5,2	cache[5]=4	cache[5]=2	mem[5]=3

Avem acum nu mai puţin de 3 valori diferite pentru adresa 5: un 4 în cache-ul procesorului 1, un 2 la procesorul 2 şi 3 în memoria însăşi. Noi am pus procesoarele la aceeaşi memorie tocmai ca să poată comunica mai uşor; dacă vroiam izolare foloseam două calculatoare. Care este acum de fapt conţinutul real al adresei 5? Cache-urile au devenit *ne-coerente*.

Trebuie menţionat că problema coerenţei nu apare numai în sistemele SMP; ea apare la o mulţime de nivele în proiectarea sistemelor, oricînd avem de-a face cu *replicarea* (copierea) informaţiei; ea apare de exemplu cînd avem de-a face cu replicarea unor servere de web, sau cu sisteme de fişiere în reţea (Network File System, Andrew File System, etc), sau în oricare sistemele cu un singur procesor atunci cînd un alt controler de magistrală (de pildă discul) accesează memoria prin acces direct (*DMA: Direct Memory Access*).

Problema aceasta apare de îndată ce cineva poate modifica o copie a datelor; problema este că celelalte copii vor avea valori diferite. Din cauza asta trebuie luate măsuri speciale ca celelalte valori să fie aduse la zi imediat. Orice întîrziere se poate solda cu efecte bizare pentru programator, care se aşteaptă ca efectele execuţiei unei instrucţiuni să fie imediat vizibile.

Deşi problema pare relativ simplă, există zeci de soluţii foarte diferite calitativ, fiecare cu avantajele şi dezavantajele ei. Ele diferă prin complexitatea implementării în hardware, prin gradul de scalabilitate (cît de bine se pot implementa pentru maşini cu mai multe procesoare), prin modul în care se manifestă pentru programator, prin gradul de performanţă pe care îl oferă.

Interesant este că soluţiile cele mai economice din punct de vedere al impactului administrativ (overhead) cer colaborarea explicită a programatorului şi oferă o vedere puţin bizară asupra memoriei. În acest articol vom vedea însă o soluţie relativ "simplă".

4 Un protocol bazat pe ascultarea magistralei

Acum vom vedea cum multiprocesoarele simetrice bazate pe o magistrală comună reuşesc să sincronizeze cache-urile. Faptul că avem o singură magistrală pentru toate procesoarele este crucial. Iată de ce: orice transfer de date între memorie şi un cache se va petrece pe magistrală. Magistrala este din punct de vedere electric un singur circuit, conectat la toate cache-urile. Astfel, atunci cînd procesorul 1 aduce cuvîntul 5 din memorie în cache-ul său local, transferul se desfăşoară pe magistrală şi toată lumea vede acest lucru. Celelalte procesoare vor şti deci că o copie a cuvîntului 5 se află la procesorul 1, aşa că atunci cînd vor dori să o modifice vor negocia cu acesta.

Pentru că fiecare cache se uită la magistrală şi atunci cînd alţii discută, acest gen de protocoale se numesc "snoopy bus-based". "To snoop" înseamnă "a-şi băga nasul în treburile altora", adică exact ceea ce se întîmplă.

Ca să înţelegem avantajul schemei, să ne gîndim la schema alternativă, care foloseşte o reţea de interconectare. Cînd procesorul P1 ia ceva din memorie, această tranzacţie nu este vizibilă pentru nimeni altcineva. Asta înseamnă că P2 o să trebuiască să afle acest lucru (de la memorie) abia cînd încearcă să acceseze acelaşi obiect, şi abia după aceea poate să iniţieze negocierea cu P1.

Perfect; avem un mecanism extrem de ieftin pentru a strînge informaţii despre plasamentul datelor (trasul cu urechea). Acum mai trebuie să punem la punct un protocol care să folosească această informaţie.

Problema centrală sunt scrierile; atunci cînd vrem să modificăm o valoare, ca să garantăm consistenţa, trebuie să fim siguri că nu mai există alte copii care după modificare vor avea valori diferite. (Soluţia de a modifica toate copiile simultan pune mari dificultăţi tehnice.) Ce vom face: atunci cînd vrem să scriem la adresa 5, vom face în aşa fel încît nimeni altcineva să nu mai aibă o copie a datelor de acolo. Din cauză că am făcut "snoop" ştim precis cine are copii; ceea ce avem de făcut este să rugăm pe toţi aceştia să *invalideze* copiile lor, scoţîndu-le din cache. Cînd nimeni nu mai are o copie, luăm noi una, pe care o putem modifica apoi în deplină siguranţă.

5 Detaliile protocolului

Chiar dacă lucrurile par simple, pînă la o implementare completă şi corectă mai sunt mulţi paşi de făcut, şi chiar multe alegeri. Noi o să dăm o anumită posibilă implementare, şi o să indicăm pe alocuri unde sunt posibile variaţiuni. Vom fi relativ completi în detalii, în aşa fel încît un arhitect să poată implementa în mod neambiguu din descrierea noastră un sistem.

Tipul de cache: fiecare procesor va avea un cache *write-back*. Asta înseamnă că atunci cînd procesorul scrie date, valoarea modificată rămîne în cache şi nu este trimisă spre memorie. (Varianta alternativă, *write-through* ar utiliza magistrala, şi aşa supra-încărcată, la fiecare scriere, pentru a trimite datele în memorie.)

Bus master: În fiecare clipă un singur cache va avea dreptul de a folosi magistrala. Dacă două procesoare simultan găsesc datele în cache-urile proprii, atunci ele pot continua execuţia în paralel. Dacă însă amîndouă vor să acceseze memoria, atunci unul dintre ele va trebui să aştepte pînă celălalt termină. Regula după care se decide cine are dreptul să folosească magistrala se numeşte *arbitrarea magistralei*; există mai multe protocoale pentru asta, dar, pentru a nu ne îneca în detalii, nu o să discutăm despre cum se face asta.

Copiile datelor: Fiecare informaţie din memorie se poate afla simultan în mai multe cache-uri, dacă nu este modificată; o informaţie care are mai multe copii *nici nu poate fi modificată* (e "read-only"). Dacă cineva vrea să modifice o valoare, trebuie să obţină o copie *exclusivă*. Odată obţinută o astfel de copie, după modificare valoarea este marcată ca "murdărită" (*dirty*).

Propagarea copiilor: Dacă un cache doreşte să obţină o copie, o să facă o cerere pe magistrală. Memoria va oferi datele dacă acestea sunt "curate", altfel ele vor veni de la cache-ul posesor. Cînd nişte date "murdare" sunt copiate la un alt procesor, ele sunt copiate şi în memorie, şi devin apoi "curate".

5.1 Semnalele

Să facem o schemă care arată cum vor comunica procesorul, cache-ul şi magistrala. Fiecare săgeată arată direcţia în care informaţia este transportată.

```
                              cache
 ________      adresa     __________________              | m |
|        |------------->|adresa stare date|  operatie     | a |
|        |     date     |-----------------|------------->| g |
|procesor|<------------>|  1     curat   3 |   adresa     | i |
|        | scrie/citeste|  2     murdar  2 |------------->| s |
|        |------------->|  3    invalid  - |    date      | t |
|        | gata / blocat|                  |<------------>| r |
|________|<-------------|__________________|              | a |
                                                          | l |
                                                          | a |
```

5.2 Starea blocului

Aşa cum indică şi desenul anterior, fiecare bloc dintr-un cache va fi într-una din următoarele stări:

Curat: valoarea din bloc este validă şi nu poate fi modificată de posesor; aceeaşi valoare se află în memoria principală, şi poate şi în alte cache-uri;

Murdar: blocul este modificat de procesorul curent, şi nu există nici o altă copie a sa; valoarea din memorie nu este corectă;

Invalid: acest bloc nu se găseşte de loc în cache.

5.3 Operaţiile pe magistrală

Cînd un procesor devine master de magistrală, comanda lansată de acesta va fi pusă pe magistrală şi observată de toată lumea. Vom presupune că din clipa în care un procesor devine master de magistrală el rămîne astfel pînă cînd tranzacţia dorită de el a fost terminată. Această presupunere este destul de restrictivă pentru sistemele reale, în care timpul de reacţie al memoriei poate fi de sute de cicli; un sistem real complicat va permite executarea operaţiilor în două faze diferite, cerere-răspuns (astfel de operaţii se numesc *split-transaction*: tranzacţie despicată, din motive evidente). Din păcate protocoalele cu tranzacţii despicate devin extrem de complicate, mai ales dacă vrem să ne asigurăm de corectitudinea lor.

Pe magistrală se poate afla una din următoarele cereri:

Citeşte: procesorul vrea să citească o copie ne-exclusivă;

Scrie: unicul procesor care posedă un bloc murdar trimite valoarea lui spre memorie (sau/şi un alt cache);

Invalidează: masterul vrea să modifice blocul indicat; pentru aceasta roagă pe toată lumea care are o copie să o invalideze, iar pe eventualul posesor exclusiv să scrie valoarea copiei sale.

5.4 Comunicaţia

Vom folosi următorul tabel cu 7 linii pentru a indica semnalele care circulă de la/spre un cache:

Cine iniţiază operaţia	procesor / bus
Operaţia cerută	citire / scriere / invalidare
Este blocul cerut în cache	da / nu
Operaţia executată pe bus	citire / scriere / nimic / invalidare
Datele aflate pe bus	adresa
Modificarea în cache-ul local	adresa
Ce face procesorul local	citeşte / scrie / blocat

Primele trei rînduri descriu o cerere care soseşte la un cache; următoarele 4 rînduri indică cum acea cerere este executată de către un cache. Figurile care urmează conţin astfel de tabele, dar numai coloana a doua.

5.5 Automatul finit al bus-master-ului

Dificultatea proiectării unui astfel de sistem este mare din cauză că trebuie să descriem ce face fiecare părticică avînd la dispoziţie numai informaţia vizibilă acelei părticele. Noi, ca proiectanţi, putem "vedea" sistemul în ansamblul său, dar un cache nu poate să vadă decît sîrmele cu care e conectat; cacheul 1 nu are nici cea mai mică idee despre conţinutul lui 2 (înafară de ceea ce a inferat pe parcurs).

Modul în care se descrie comportarea fiecărei entităţi este folosind *un automat finit.* Noi vom construi automatul finit al unui cache, care arată exact ce acţiune trebuie acesta să facă ca răspuns la fiecare stimul venit din afară.

Un automat finit este o colecţie de *stări*, în care circuitul se poate afla la un moment dat. Voi indica în desene stările prin cerculeţe. Între stări există săgeţi numite *tranziţii.* Fiecare tranziţie este etichetată cu o tabelă ca cea de mai sus; primele rînduri indică cererile făcute asupra cache-ului, fie că ele vin din partea procesorului local, fie dinspre bus, de la masterul de magistrală; ultimele rînduri indică reacţia cache-ului.

În realitate, fiecare cuvînt din cache are un astfel de automat finit; noi vom arăta cum se comportă automatul finit asociat cuvîntului din cache care trebuie modificat de tranzacţia curentă. Pentru a simplifica expunerea, vom desena două automate finite, unul care este valabil pentru procesorul master de magistrală, şi unul care este valabil pentru celelalte procesoare. Fiecare procesor va acţiona după una din reguli, depinzînd de starea lui la acel moment de timp.

Figura 1 arată automatul asociat unui master de magistrală. Automatul este destul de complicat, şi are 10 tranziţii posibile, numerotate în figură (sunt doar 5 săgeţi, dar unele tranziţii folosesc aceeaşi săgeată). Le vom analiza una cîte una. Rîndurile albe din tabele indică faptul că nu contează ce se află acolo, sau că acolo nu se află nimic.

Cititorul care este plictisit de minuţiozitatea detaliilor din desen şi text va trebui cu siguranţă să-şi caute o meserie înafara proiectării de calculatoare; modelul pe care îl prezentăm aici este de fapt o simplificare pedagogică substanţială; ceea ce se construieşte în realitate în procesoarele pe care le folosim în fiecare zi este cu mult mai complicat. Voi spune cîteva cuvinte mai jos despre încrederea pe care o putem avea în astfel de design-uri.

Pe de altă parte uneltele pe care le prezint aici pentru a descrie comportarea sunt chiar cele folosite de către designerii profesionişti: automate finite cu reguli detaliate de tranziţie.

Iată explicaţia tranziţiilor, în ordine crescătoare a dificultăţii:

Tranziţia numărul 3: Procesorul (linia 1 din tabelul nr. 3) doreşte să citească (indicat de linia 2) de la adresa x. Cuvîntul este prezent în cache (linia 3), şi este "curat" (starea curentă este "curat"). Rezultatul este foarte simplu: nici o acţiune nu este iniţiată pe magistrală (liniile 4, 5 sunt goale în tabel); cuvîntul este imediat servit procesorului, care termină operaţia de citire (linia 7).

Tranziţia 8: Procesorul doreşte să citească de la adresa x, care este în cache, şi este marcată "murdară". Citirea se poate face imediat, fără a implica magistrala.

Tranziţia 7: Procesorul vrea să scrie la adresa x, iar cuvîntul de la adresa x este prezent şi este deja murdar. Scrierea se poate face imediat, fără a implica magistrala.

Primele trei tranziţii descrise mai sus sunt foarte simple, pentru că datele sunt imediat disponibile şi pentru că nu trebuie recurs la magistrală. Iată acum trei situaţii care se pot

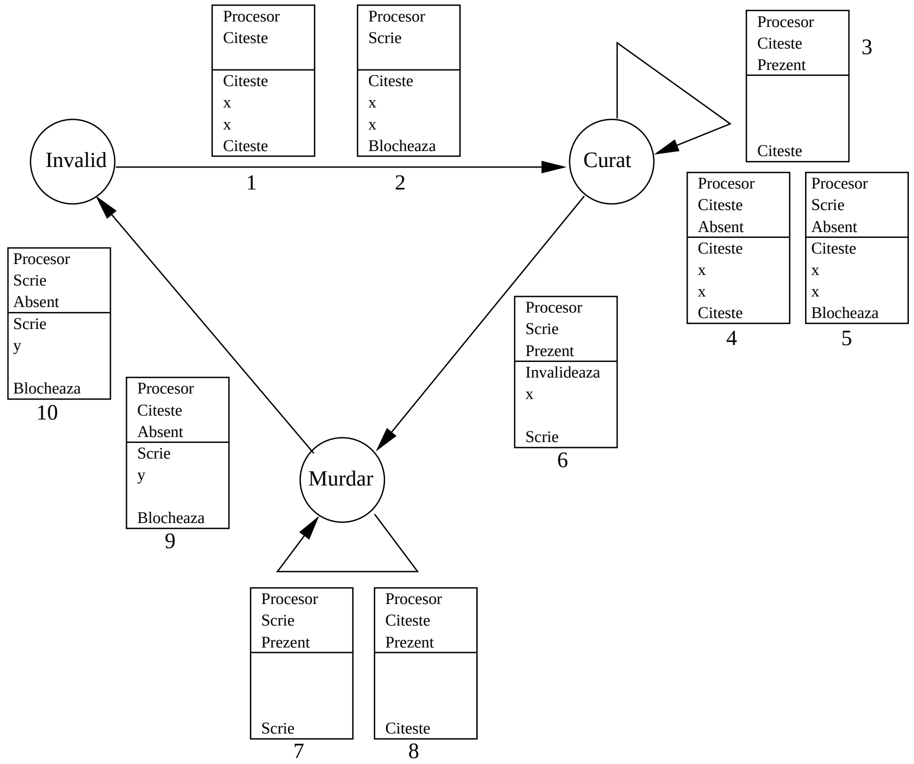

Figura 1: Acţiunile unui master de magistrală atunci cînd procesorul vrea să opereze asupra blocului x iar în cache se află blocul y.

rezolva cu cîte o tranziţie fiecare, dar care au nevoie de magistrală:

Tranziţia 1: Procesorul doreşte să citească de la adresa x. În cache obiectul de la adresa x este marcat ca fiind invalid. În cazul asta pe magistrală este lansată comanda de citire (linia 4 din tabel) de la adresa x (linia 5). Modelul nostru presupune că datele de la adresa x vor fi trimise imediat de cel care le posedă (memoria sau un alt cache), deci ele vor fi încărcate în cache (linia 6); simultan ele vor fi trimise la procesor, care termină astfel operaţia de citire (linia 7). Pentru că procesorul are de acum o copie a valorii, blocul este marcat ca "curat" (tranziţie de la Invalid la Curat).

Tranziţia 4: Procesorul vrea să citească x, dar în cache în acel loc se află altceva (linia 3), care este curat. Pentru că acea valoare este curată, ea este imediat ştearsă din cache (există o copie în memorie), pe bus este pusă o cerere pentru x (linia 5); presupunem că x vine imediat. x este apoi încărcat în cache, şi procesorul termină citirea.

Tranziţia 6: Vreau să scriu în cuvîntul de la adresa x, am o copie a lui, dar nu sunt singurul. Atunci pun o cerere de invalidare (linia 4) a adresei x (linia 5) pe magistrală; în acest fel toţi posesorii lui x sunt rugaţi să-l invalideze (ştiu că nimeni nu are o copie murdară). Toată lumea va reacţiona corect la această cerere, aşa că putem modifica valoarea lui x în cache-ul local. Valoarea ca atare devine "murdară".

Şi astea trei situaţii au fost relativ simple, datorită modelului nostru care nu foloseşte “split transaction” (am presupus că datele cerute vin imediat; dacă durează mai mulţi cicli, practic procesorul master şi bus-ul sunt blocate pînă cînd datele aşteptate de master ajung la destinaţie).

Operaţiile rămase sunt mult mai încîlcite, şi cer o *succesiune de mai multe tranziţii pentru a fi îndeplinite*. Presupunem, din nou pentru a simplifica, că acelaşi cache rămînă master de magistrală pînă termină operaţia începută, chiar daca asta implică mai multe transferuri pe magistrală.

Iată nişte operaţii care cer cîte *două tranzacţii* pentru a fi executate (pe bus trebuie să circule două valori diferite). Procesorul master este deci blocat timp de două operaţii.

Tranziţia 5 (+6): Vreau să scriu în cuvîntul de la x, dar nu-l am. Atunci fac doi paşi: întîi obţin o copie a cuvîntului (tranziţia 5), după care continui cu tranzacţia 6, invalidînd celelalte copii şi scriind valoarea. Tranziţia 5 va bloca procesorul (linia 7) pînă se termină tranziţia 6 (abia atunci pot să fac ceea ce mi s-a cerut).

Tranziţia 9 (+1): Vreau să citesc la adresa x, dar locul e ocupat de cuvîntul de la y, care e şi murdar pe deasupra. Atunci, în tranziţia 9, pun acest cuvînt pe bus, şi rog memoria să-l copieze. Blochez procesorul. După ce am scris cuvîntul, marchez linia ca invalidă şi execut tranziţia 1, pentru a citi cuvîntul cerut.

Tranziţia 2 (+6): Vreau să scriu la adresa x, dar nu am nici o copie a lui x (e marcată “invalidă”). Atunci în primul rînd (tranziţia 2) obţin o copie a lui x ca la o citire (linia 4) obişnuită, după care pot face scrierea (tranziţia 6).

În fine, avem o situaţie, cea mai neplăcută, cînd pentru a face ceea ce procesorul ne cere, trebuie să facem nu mai puţin de 3 tranziţii! Situaţia apare cînd vreau să scriu într-un cuvînt, dar am în acel loc un altul murdar.

Tranziţia 10 (+1, +6): Încep prin a goli cuvîntul murdar cu tranziţia 10; marchez linia “invalidă”. Apoi fac tranziţia 1, pentru a obţine o copie “curată”, şi apoi fac tranziţia 6, pentru a invalida celelalte copii şi marchez copia curentă ca murdară.

5.6 Automatul finit al unui cache non-master

Bravo! Dacă aţi ajuns pînă aici sunteţi foarte răbdători, sau aţi sărit suficient de multe paragrafe.

Ce-a rămas este mult mai simplu: descrierea acţiunilor unui cache “sclav” (care nu e master de magistrală). Schema este mult mai simplă, pentru că un astfel de cache nu poate folosi de loc magistrala, cu o singură excepţie: cînd primeşte o cerere de citire a unui bloc murdar, în care caz trebuie să-l pună pe magistrală pentru master. Toată schema este în figura 2.

Tranziţia 4: bus-ul vrea o anumită adresă pe care nu o avem; nu facem nimic;

Tranziţia 3: bus-ul vrea o adresă pe care o avem, dar care e curată; atunci memoria o să satisfacă această cerere, deci iarăşi nu facem nimic;

Tranziţia 2: bus-ul vrea o adresă pe care nu o am; nu fac nimic;

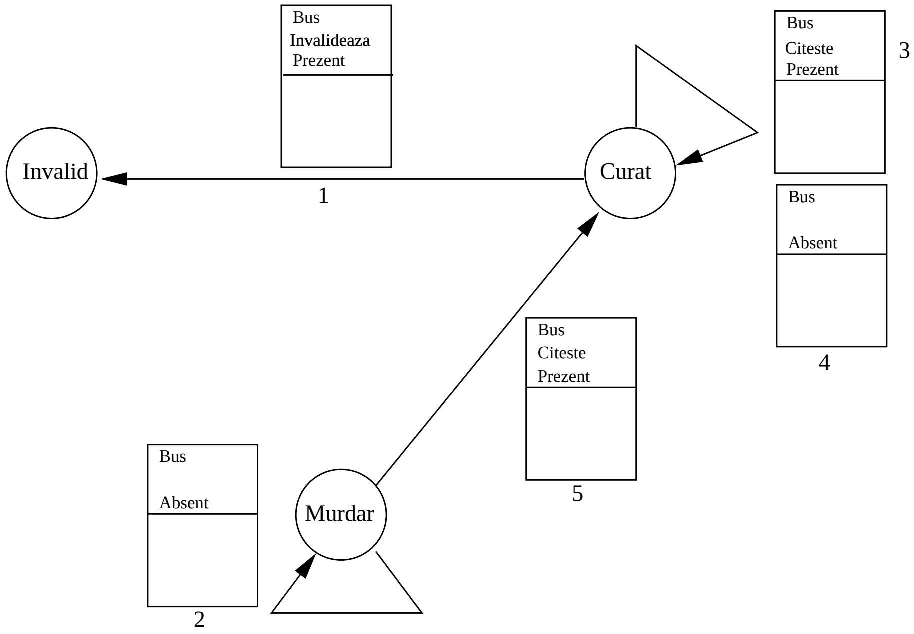

Figura 2: Acţiunile unui sclav.

Tranziţia 1: primesc o cerere de invalidare a unui bloc curat pe care îl am: atunci marchez blocul ca invalid;

Tranziţia 5: bus-ul vrea să citească un bloc pe care îl am murdar; atunci pun valoarea pe bus, memoria va copia această valoare. Eu o marchez curată.

6 Corectitudine

Perfect; am specificat fiecare acţiune în funcţie de circumstanţe suficient de detaliat pentru a putea fi implementată. Dar cum de ştim că protocolul este corect?

De pildă poate părea ciudat că în ultima diagramă nu e nici o săgeată din starea "murdar" care să aibă în tabel o invalidare. Este cumva o omisiune? Nu, pentru că aşa cum am proiectat protocolul, un cache poate lansa o invalidare numai după ce are o copie a datelor, deci ele nu pot fi simultan murdare altundeva.

Inutil de spus că verificarea unor sisteme reale este o treabă extrem de grea; modelul nostru este simplificat într-o mulţime de privinţe. Ceea ce e într-un Pentium este de multe ori mai complicat decît modelul prezentat (Pentium are cache-uri ne-blocante, şi dacă procesorul nu primeşte nişte date pe care le-a cerut, el poate cere altele; datele pot sosi într-o ordine diferită de cea a cererilor!).

Marile companii de hardware (ex. Intel, IBM) folosesc pentru a verifica corectitudinea unor astfel de scheme tot calculatoarele. Pe de o parte se folosesc simulări şi teste complete: sistemul este implementat ca un program, care apoi este testat pe toate combinaţiile posibile de ordini de evenimente care pot apărea. Treaba asta este extrem de complicată, pentru că numărul de teste creşte enorm cu complexitatea circuitului, ajungînd rapid la valori impractice chiar pentru cele mai performante calculatoare.

O altă metodă este cea de a folosi scule speciale pentru a verifica corectitudinea; o ramură specială a ştiinţei calculatoarelor se numeşte "model theory" şi se ocupă cu metode de a analiza sisteme finite extrem de complicate folosind ecuaţiile care descriu traiectoria sistemului. Un automat finit este o descriere suficient de precisă pentru a face o analiză exactă posibilă. Metodele se bazează pe teorii matematice destul de sofisticate, pentru a reduce substanţial spaţiul căutărilor (logici temporale, diagrame binare de decizie, simulare şi bisimulare, abstracţie, etc.). Din păcate metodele practice sunt cam cu 10 ani în urma tehnologiei: nici nu se pune problema de a verifica exhaustiv circuitele contemporane, chiar şi cu scule atît de sofisticate.

7 Un exemplu: SGI Challenge

Iată caracteristicile tehnice ale unei maşini reale care foloseşte tehnicile prezentate: SGI Challenge; acesta este cel mai mare multiprocesor bazat pe bus, cu 36 de procesoare MIPS R4400, o adevărată minune tehnologică.

Procesoare	pînă la 36
Memorie	pînă la 16 Gb
Lăţime bus (date)	256 biţi
Lăţime bus (adrese)	40 biţi
Rata de transfer	1.22 Gb/sec
Tranzacţii	model "split"
Frecvenţa magistralei	50Mhz
Penalizarea pentru un rateu în cache	>164 cicli

Costul unei operaţii care nu găseşte datele în cache este extrem de substanţial, datorită complexităţii protocoalelor (în articolul anterior din PC Report am văzut ca la un Pentium uniprocesor durata este de ordinul a 13 cicli de procesor). Din cauza asta implementarea unor protocoale de coerenţă eficiente este extrem de importantă, dar performanţa obţinută datorită unui număr ridicat de procesoare compensează costul ridicat al execuţiei fiecăruia.

Voi încheia acest articol aici, deşi îmi propusesem să mai atac şi alte subiecte; rezultatul însă pare destul de consistent, aşa că o să amîn pentru alte ocazii discuţii despre subiecte subtile cum ar fi: alte protocoale de coerenţă, sau impactul modului în care e implementat un protocol de coerenţă pentru performanţa anumitor construcţii din programele utilizatorilor. Vă îndemn să studiaţi cu atenţie diagramele de mai sus. Chiar dacă conceptele sunt pe alocuri simplificate, scenariile sunt toate plauzibile. În definitiv e foarte plăcut să înţelegi sculele cu care lucrezi, nu?