Laborator 10 - Memoria Cache

În laboratoarele precedente a fost studiată arhitectura calculatoarelor din prisma execuției instrucțiunilor în procesor, urmând ca în cadrul acestui laborator să se abordeaze mecanismele de funcționare și implementare ale memoriei rapide cache. După cum s-a discutat în laboratorul 4, la partea de pipeline, execuția unei instrucțiuni presupune în cele mai multe cazuri și aducerea sau scrierea operanzilor din sau în memorie. Pentru ca aceste operații să nu impună întârzieri prea mari, procesoarele folosesc memorii cache, mai rapide decât memoria principală, însă cu o capacitate mult mai mică (figura 1).

1. Principiul memoriei cache

Pentru început, vom introduce memoria cache în contextul general al unui sistem de calcul.

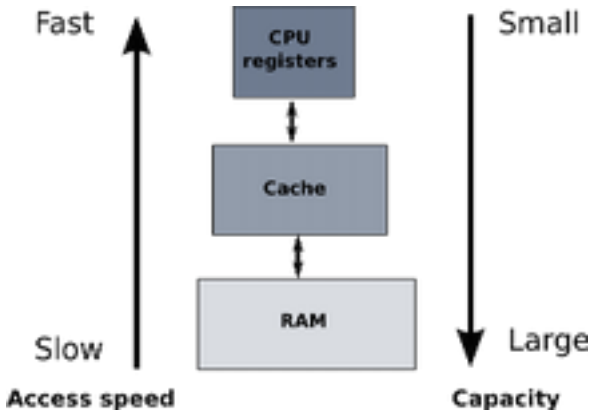

Figura 1. Ierarhizarea memoriei în sistemele de calcul

Prin adăugarea memoriei cache în ierarhia de memorii, interfața dintre procesor și memoria principală rămâne neschimbată și, dintr-un punct de vedere funcțional, accesul se realizează ca și în cazul în care nu există cache. Ideea de bază este că, din moment ce cache-ul este mai mic decât memoria principală, el stochează o submulțime din conținutul acesteia. Deoarece cache-ul este implementat cu o tehnologie mai rapidă (SRAM) decât memoria principală (DRAM), accesul la conținutul cache-ului este mai rapid.

Memoria cache este organizată pe *linii*, fiecare linie conținînd date de la adrese consecutive din memoria principală (în cadrul memoriei principale ne referim la acestea ca fiind *blocuri* de date), asa cum este ilustrat în figura 2. O memorie cache fiind mult mai mică decat memoria principală, poate menține doar o parte din datele acesteia la un moment dat, și de aceea avem nevoie de mecanisme care să controleze aducerea și evacuarea frecventă a liniilor acesteia, tehnici ce vor fi prezentate în cadrul acestui laborator.

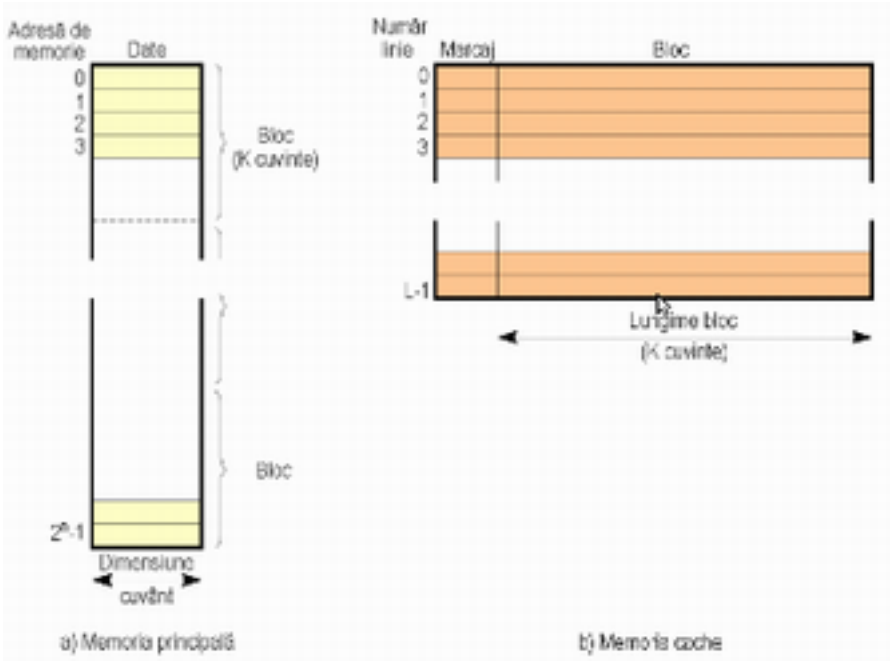

Figura 2. Organizarea pe linii a memoriei cache. Unei linii de cache ii corespunde un bloc de date de adrese consecutive din memoria principală

Ca rezultat al **localității** fluxului de adrese, cache-ul reduce încărcarea restului ierarhiei de memorie deoarece poate fi administrat să rețină mulțimea curentă de date și instrucțiuni. Atunci când procesorul încearcă citirea unui cuvânt din memorie, se testează dacă respectivul cuvânt se află în memoria cache. În caz afirmativ, cuvântul este furnizat unităţii centrale de execuție. În caz contrar, se încarcă în memoria cache un bloc al memoriei principale, constând dintr-un număr fix de cuvinte, iar apoi cuvântul este returnat unităţii centrale. Se cunoaşte că programele nu accesează memoria în mod complet aleator. Dacă se face o referire la o anumită adresă este probabil că următoarea referire la memorie va fi în vecinătatea acestei adrese. În spaţiul adreselor de memorie, câteva regiuni au o probabilitate ridicată de a fi accesate, câteva au o probabilitate moderată, iar celelalte au o probabilitate foarte mică de a fi accesate în viitorul apropiat. O regiune care are o probabilitate înaltă este cea corespunzătoare contorului de program actual, deoarece este probabil să se execute următoarea instrucţiune din secvenţa de instrucţiuni. Alte regiuni care au o probabilitate mare de a fi accesate sunt cele care conţin datele active, procedurile şi punctul

de întoarcere dintr-o procedură.

Atunci când vorbim de îmbunatățirile de performanță rezultate din folosirea cache-ului, vom folosi următorii termeni:

● ***Cache hit*** - generat atunci când pentru a accesa date din memoria principala le luăm din cache.

● ***Cache miss*** - generat atunci când datele nu sunt în cache, fiind aduse procesorului direct din memoria principală.

- *cache miss obligatoriu* – cauzat de primul accesul la o zonă de memorie.
- *cache miss de capacitate* – cauzat de capacitatea limitată a cache-ului.
- *cache miss conflictual* – cauzat de înlocuirea unui element din cache cu unul nou adus.
-

- ***Fetch*** - aducerea unui bloc de cuvinte din memoria principala într-o **linie de cache**. *Atenție!* În cache încărcăm linii care conțin mai multe cuvinte, nu putem încărca doar un anumit cuvânt. Aceasta structură a cache-ului este descrisă și în figura 2.

- ***Principiul localității*** - așa cum am exemplificat mai sus, se referă la accesul programului la date din aceeași zonă. Există două tipuri de localitate a datelor:
  - **localitatea spațială** - se referă la faptul că datele aflate *adiacent în memorie* au o probabilitate mare de a fi folosite împreună; organizarea pe linii a cache-ului face ca atunci când un cuvânt este accesat, întreaga linie care conține acel cuvânt și cuvintele adiacente să fie încărcată în cache, astfel accesele la cuvintele adiacente vor fi servite din cache.
  - **localitatea temporală** - se referă la faptul că datele *accesate recent* au o probabilitate mare de a fi accesate din nou în viitorul apropiat; cache-ul mărește performanța sistemului deoarece stochează datele accesate recent de program într-o memorie de mare viteză de unde pot fi reaccesate rapid.

- Performanța programului este cu atât mai mare cu cât **timpul mediu de acces l**a memorie este mai mic.

$$\overline{T_{access}} = T_{cache} + f_{MISS} * T_{main\,mem}$$

unde T_{cache} este timpul de acces la cache, iar $T_{main\,mem}$ este timpul de acces la memoria principală, iar f_{MISS}este factorul de cache miss

$$f_{MISS} = \frac{nrCacheMiss}{nrAccese}$$

$$f_{HIT} = \frac{nrCacheHit}{nrAccese}$$

Condiția ca memoria cache să îmbunătățească performanța unui sistem sistem este:

$$f_{HIT} > \frac{T_{cache}}{T_{main\,mem}}$$

Pentru proiectarea memoriei cache trebuie luate în considerare o serie de mecanisme pentru scrierea datelor, citirea datelor și evacuarea liniilor, mecanisme care afectează performanța și complexitatea cache-ului. Ne referim astfel la :

- *Politici de fetch*
- *Politici de scriere*
- *Funcția de mapare*
- *Politici de evacuare*

Ca un exercițiu, puteti folosi programe de genul CPU-Z http://www.cpuid.com/softwares/cpu-z.html care va permit, printre alte informații de sistem, și vizualizarea informaţiilor despre memoriile cache din calculatoarele voastre.

2. Politici de Fetch

Atunci când datele de care procesorul are nevoie nu se regăsesc în cache, ele sunt aduse în cache pe baza politicii de fetch.

Să considerăm exemplul unei memorii cache cu 1024 de octeți, organizați pe 256 de linii cu câte 4 cuvinte. Procesorul emite o cerere de încărcare din memorie de la adresa 2, care se mapează direct în prima linie a cache-ului (cuvintele de la 0 la 3). Cache-ul nu conține pe prima linie datele asociate cu adresa 2, deci trebuie executat un fetch, fiind adus tot blocul din memorie care conține acel cuvânt. Problema care se pune este în ce ordine vor fi aduse cuvintele din memoria principală. Există două opțiuni:

- ***cuvântul critic mai întâi (critical word first)*** – presupune o încărcare a cuvintelor în ordinea 2,3,0,1. Cuvântul de la adresa de care avem nevoie este încărcat primul. Această tehnică micșorează miss-penalty (timpul de acces la memorie pentru a face fetch la date în cazul unui cache-miss) în cazul liniilor de dimensiuni mai mari. În exemplul dat, diferența de performanță este neglijabilă.
- ***cuvântul natural mai întâi (natural word first)*** – presupune o încărcare a cuvintelor în ordinea 0,1,2,3. Adresa de la începutul liniei este încărcată prima.

Chiar dacă politica *critical word first* prezintă avantajul aducerii cuvântului necesar mai repede în procesor, sporind viteza acestuia de lucru, oferă totuși dezavantajul unui mecanism de gestiune a cache-ului mai complicat.

Pre-fetching

O altă tehnică de optimizare a timpului de acces la memorie, este prin aducerea în avans a datelor din memorie în cache prin tehnici de **pre-fetching**. Prin anticiparea accesării datelor putem reduce numărul de cache-miss însă algoritmul/euristicile pe baza căruia se face speculația trebuie să nu polueze cache-ul cu date ce nu sunt folosite, sau sunt folosite rar.

În cazul instrucțiunilor, acestea sunt executate de cele mai multe ori consecutiv (excepții apar în cazul instructiunilor de salt) și de aceea există o mare probabilitate ca după instrucțiunile dintr-un bloc K de cuvinte (linie cache) procesorul să execute și instrucțiunile stocate în blocul următor *K+1*. Prin *pre-fetching* se aduce din memorie acest bloc și se mapează pe una din liniile din cache, înainte de apărea o cerere de la procesor pentru vreo adresă din bloc.
În cazul datelor, pre-fetching-ul este mai complicat și, de obicei, are performanțe bune când blocurile de date sunt prelucrate secvențial, de exemplu prelucrari de vectori și matrici.

Metodele de pre-fetching pot fi implementate atât la nivel software, prin optimizări făcute de către compilatoare, cât și la nivel hardware prin analiza adreselor de cache-miss folosind memory reference patterns, sau analiza adreselor instrucțiunilor (pentru cazul cache-ului de instrucțiuni). O scurtă descriere a principalelor metode este făcută aici.

3. Politici de Scriere

Politicile de scriere în cache se referă la fluxul de date dinspre procesor spre memoria cache și spre memoria principală. Astfel, diferențiem două situații:

1. Datele aflate deja în cache trebuie actualizate (*write-hit policies*).
2. Datele care trebuie scrise nu se află în cache (*write-miss policies*).

Pentru primul caz problema ce trebuie adresată de politicile de scriere este cea a **păstrării consistenței dintre memoria principală și memoria cache**. Cele două opțiuni sunt:

- ***write-through:*** Când procesorul emite o cerere de scriere, un cache write-through actualizează datele din cache, executând, eventual, operația de fetch (dacă datele nu se aflau în cache), dar actualizează și datele din memoria principală.
    - Avantaj: implementare simplă, asigură ușor consistența dintre memorii: dacă memoria cache și cea principală au mereu același conținut, ele sunt consistente.
    - Dezavantaj: atunci când datele sunt actualizate des și la intervale scurte de timp, devine ineficient accesul repetat la memoria principală pentru efectuarea operațiilor de scriere, ducând la scăderea performanței.
- ***write-back:*** Când procesorul emite o cerere de scriere, un cache write-back actualizează datele din cache, însă nu actualizează imediat și pe cele din memorie.
    - Avantaj: prin acest mecanism se elimină traficul inutil dintre cache și memoria principală.
    - Dezavantaje: duce la creșterea complexității hardware-ului, fiind necesară reținerea (marcarea) acelor linii care au devenit inconsistente cu memoria principală. Pentru a marca aceste linii (murdare), se include un indicator suplimentar pe fiecare linie. În momentul evacuării, dacă linia este murdară va fi scrisă în memoria principală. În cazul sistemelor multicore în care avem cache-uri individuale pentru fiecare core/ procesor, care partajează memoria principală, păstrarea consistenței devine mai dificilă, necesitând mecanisme și hardware și mai complex (discutat în secțiunea 6).

În cazul în care datele ce vin de la procesor nu se află în cache, avem următoarele opțiuni pentru **implementarea scrierii în cache**:

- ***write-allocate:*** se alocă o linie în cache pentru datele respective.
- ***no-write-allocate:*** nu se alocă o linie în cache pentru datele respective.
- ***fetch-on-write:*** se aduce linia în cache din memoria principala (sau dintr-un cache de la un nivel superior); implicit se face și alocarea liniei.
- ***no-fetch-on-write:*** dacă se foloseste și optiunea write-allocate atunci linia este scrisă doar în cache și invalidată, altfel cu no-write-allocate datele sunt actualizate direct în memoria principală (*write-around*).

Folosirea opțiunii write-allocate poate duce la poluarea cache-ului.

4. Funcția de mapare

O caracteristică de bază a memoriei cache este funcţia de mapare (de translatare), care atribuie unui bloc din memoria principală o locaţie din memoria cache. Se pot utiliza trei tehnici:

- mapare directă
- mapare asociativă
- mapare asociativă pe seturi

Aceste alternative se vor examina printr-un exemplu. Se consideră o memorie cache de 1024 (1K) octeţi. Datele se transferă între memoria principală şi memoria cache în blocuri de câte 8 octeţi. Aceasta înseamnă că memoria cache are 128 de linii de câte 8 octeţi fiecare. Memoria principală constă din 64K octeţi, fiecare octet fiind direct adresabil printr-o adresă de 16 biţi. Se poate considera că memoria principală constă din 8K blocuri de câte 8 octeţi fiecare.

Deoarece există un număr mai mic de linii ale memoriei cache faţă de numărul blocurilor memoriei principale, este necesar un algoritm pentru plasarea blocurilor memoriei principale în liniile memoriei cache. În plus, este necesar un mijloc de a determina care bloc al memoriei principale ocupă la un moment dat o linie a memoriei cache.

4.1 Memorii cache mapate direct

În acest caz, abordarea este simplă – fiecare a 2^n-a adresă este mapată pe același cuvânt din cache (figura 5). Este o dependență unu-mai multe, deoarece memoria principală este mai mare și există mai multe locații care se mapează exact pe același cuvânt din cache. Mecanismul implementat va trebui să distingă între acestea.

Se vor calcula două componente utilizate în accesarea conținutului memoriei cache:

$$A_{word} = A \bmod C_{words}$$

$$A_{line} = [A \operatorname{div} C_{words}] \bmod C_{lines} = C_{blocuri\ in\ memorie} \bmod C_{lines}$$

Valoarea A_{line} identifică linia pentru adresa A, în timp ce valoarea A_{word} identifică exact cuvântul din linie. Dacă C_{lines} și C_{words} sunt alese ca puteri ale lui 2, calculele sunt facile – deplasări de biți.

Pentru a face distincția între blocuri de memorie mapate pe aceeași linie, în memoria cache fiecare linie primește o etichetă. Apare, deci, nevoia unui calcul suplimentar:

$$A_{tag} = [A \operatorname{div} C_{words}] \operatorname{div} C_{lines}$$

Să considerăm un caz generic de memorie cache cu $C_{lines}= 2^m$ și $C_{words}= 2^k$. Adresa A transmisă de procesor este pe $n > m + k$ biți. Atunci, figura urmatoare reprezintă cele **trei componente ale adresei** care trebuie calculate – A_{word}, A_{line} și A_{tag}.

Astfel, adresele cuvintelor din memoria principală vor fi interpretate după formatul prezentat în figura 3, pentru a putea fi mapate pe locații din cache.

Figura 3. Formatul adresei cuvintelor pentru maparea directă

Procedeul prin care se **verifică existența unui cuvânt în cache** este următorul (figura 4):

1. Fiind dată o adresă A, cache-ul calculează cele trei componente.
2. Inspectează valoarea $C[A_{line}]_{valid}$.
3. Dacă nu are conținut valid, cuvântul nu este rezident și are loc un cache-miss.
4. Dacă linia are conținut valid, se inspectează valoarea $C[A_{line}]_{tag}$ și se compară cu A_{tag}.
5. Dacă cele două etichete nu se potrivesc, are loc un cache-miss.
6. Dacă etichetele se potrivesc, are loc un cache-hit și cuvântul este rezident în $C[A_{line}]$ $data[A_{word}]$

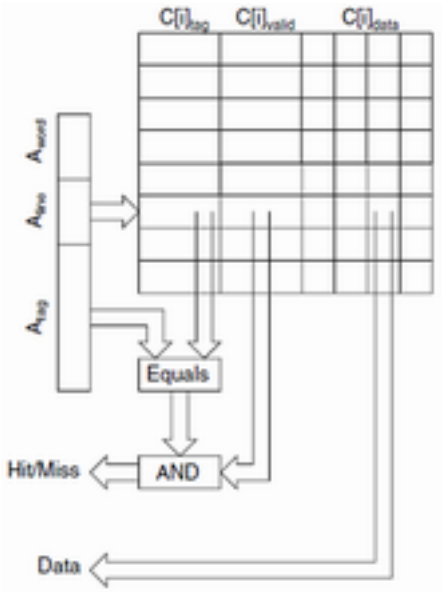

Figura 4. Diagrama procedeului de lucru pentru memoria cache mapată direct

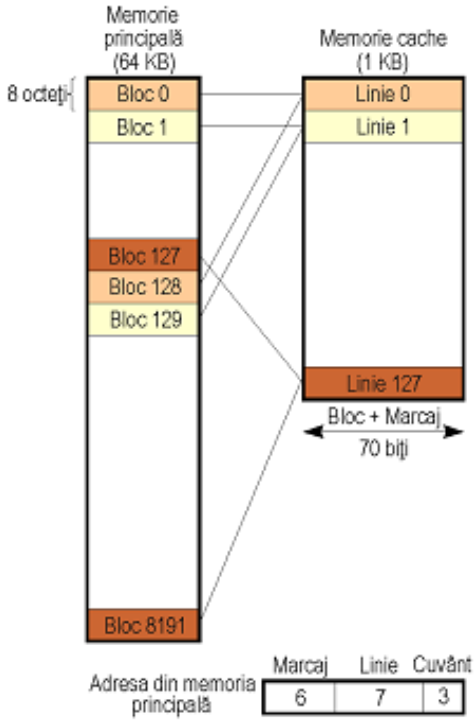

Figura 5. Exemplu de mapare directă pentru un cache de 1K și memorie principală de 64K

4.2 Memorii cache complet-asociative

O memorie mapată direct nu poate avea decât o linie ca alegere pentru o anumită adresă; acest fapt poate duce la fenomenul numit interferență cache sau contenție de cache. Pentru un caz în care două adrese sunt mapate pe aceeași linie și procedeul de evacuare are loc repetat datorită accesării alternative a celor două adrese, se folosește sintagma ”*fluxul de acces deteriorează cache-ul*”. Este vorba despre o situație în care performanțele memoriei cache sunt degradate semnificativ deoarece trebuie adusa din memorie și evacuată foarte des aceeasi linie de cache.

O abordare pentru rezolvarea acestei probleme este utilizarea ideii de asociativitate. Un cache asociativ permite ca o locație de memorie să fie mapată în mai multe linii de cache; un cache complet-asociativ permite unei locații de memorie să fie mapată pe orice linie. Din moment ce acum datele necesare ar putea fi oriunde în cache, căutarea lor necesită hardware mai complex. Totuși, memoriile cache asociative pot avea un factor de miss mult mai mic decât cele mapate direct.

Din moment ce un bloc de cuvinte poate fi mapat pe oricare linie din cache (figura 8), nu se mai poate identifica indexul liniei prin intermediul adresei. De aceea formatul folosit de către maparea complet asociativa este cel descris în figura 6.

Figura 6. Formatul adresei cuvintelor pentru maparea complet asociativă

Valoarea A_{word} ce identifică cuvintele dintr-un bloc (linie) este construită, ca și la memoria cache mapată direct, din ultimii k cei mai puțin semnificativi biți. Valoarea A_{tag}, care reprezintă identificatorul (eticheta) blocului de cuvinte este dată de restul biților.

După traducerea adresei pentru a realiza operația de fetch, trebuie să analizăm problema găsirii liniei în momentul inițierii unei operații de citire. Datorită modului în care s-a depus în cache linia mapată, este necesar a efectua o căutare în întregul său cuprins. Pentru că memoria cache trebuie să fie foarte rapidă, o căutare liniară (sau chiar binară) nu este o opțiune; trebuie investigate toate liniile în paralel, cea ce mărește complexitatea hardware a logicii asociate cache-ului, ducând și la costuri mai mari.

Procedeul prin care se **verifică existența unui cuvânt în cache** este următorul (figura 7):

1. Fiind dată o adresă A, cache-ul calculează cele două componente.
2. Se inspectează toate valorile C_{tag} și se compară cu valoarea A_{tag}.
3. Dacă nici una din etichete nu se potriveste, are loc un cache-miss.
4. Dacă o etichetă se potrivește, se inspectează valoarea $C\{A_{tag}\}_{valid}$.
5. Dacă linia nu are conținut valid, cuvântul nu este rezident și are loc un cache-miss.
6. Dacă linia este validă, are loc un cache-hit și cuvântul este rezident în $C\{A_{tag}\}_{data[Aword]}$

Obs: $C\{A_{tag}\} = C[i]$, unde $C[i]_{tag} = A_{tag}$

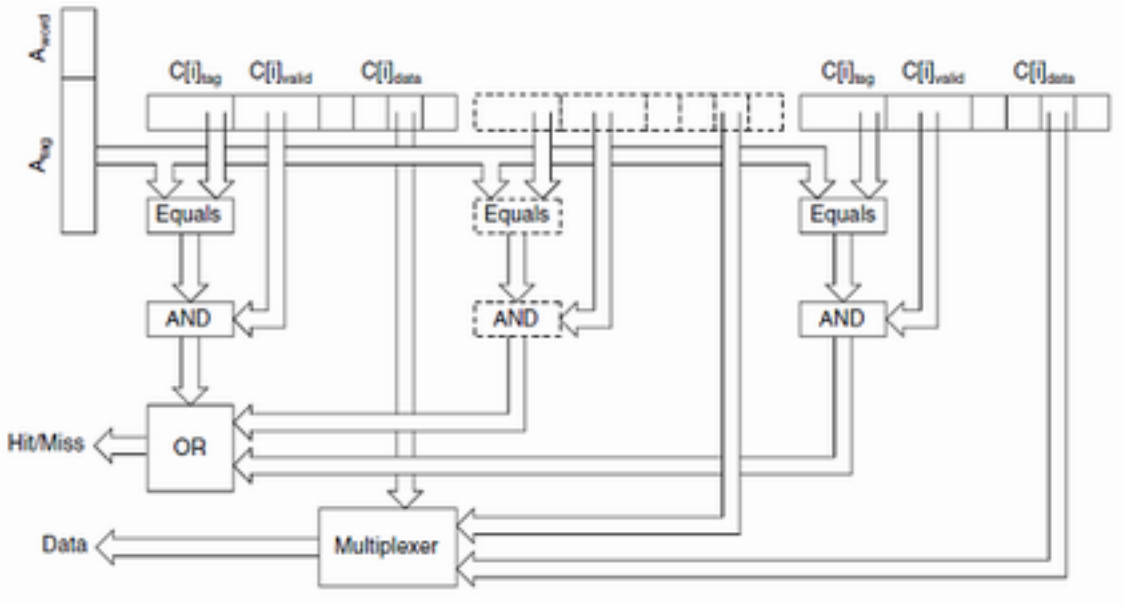

Figura 7. Procedeul prin care se verifică existența unui cuvânt în cache pentru maparea complet asociativă

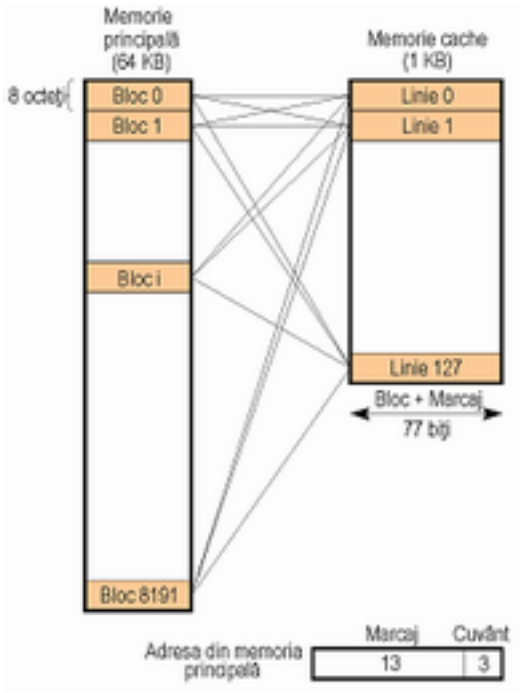

Figura 8. Exemplu de mapare complet asociativă pentru un cache de 1K si memorie principală de 64K

4.3 Memorii cache asociative pe mulțimi (N-way set-associative)

Memoriile cache asociative pe mulțimi îmbină simplitatea relativă a memoriilor cache mapate direct cu metodele de evitare a contenției de la memoriile cache complet-asociative, ducând la costuri rezonabile și un factor de hit bun. Ideea de bază este aceea a secționării memoriei cache în C_{set} seturi, fiecare set având N linii:

$$N = C_{lines} \text{ div } C_{set}$$

În locul mapării unei adrese la o linie, se face maparea la un set, iar în interiorul setului căutarea unei linii se face într-o manieră complet-asociativă.
Ca și în cazurile anterioare, este necesară calcularea anumitor valori pentru poziționare în memoria cache și identificare unică.

$$A_{word} = A \bmod C_{words}$$

$$A_{set} = [A \operatorname{div} C_{words}] \bmod C_{set}$$

$$A_{tag} = [A \operatorname{div} C_{words}] \operatorname{div} C_{set}$$

Valoarea A_{word} identifică indexul cuvântului în cadrul blocului (liniei), A_{set} identifică indexul setului, iar A_{tag} este eticheta blocului. Formatul adresei cuvintelor pentru maparea set asociativă este prezentat în figura 9.

Figura 9. Componentele adresei unui cuvânt pentru maparea set-asociativă

Procedeul prin care se **verifică existența unui cuvânt în cache** este următorul (figura 10):

1. Fiind dată o adresă *A*, cache-ul calculează cele trei componente.
2. In cadrul setului A_{set} se inspectează toate valorile $C[A_{set}]_{tag}$ și se compară cu valoarea A_{tag}.
3. Dacă nici una dintre etichetele din cadrul setului nu coincide cu A_{tag} are loc un cache-miss.
4. Dacă o etichetă se potrivește, se inspectează valoarea $C[A_{set}]\{A_{tag}\}_{valid}$.
5. Dacă linia nu are conținut valid, cuvântul nu este rezident și are loc un cache-miss.
6. Dacă linia este validă are loc un cache-hit și cuvântul este rezident în $C[A_{set}]\{A_{tag}\}_{data[Aword]}$

Obs: $C[A_{set}]\{A_{tag}\} = C[A_{set}][i]$, unde $C[A_{set}][i]_{tag} = A_{tag}$

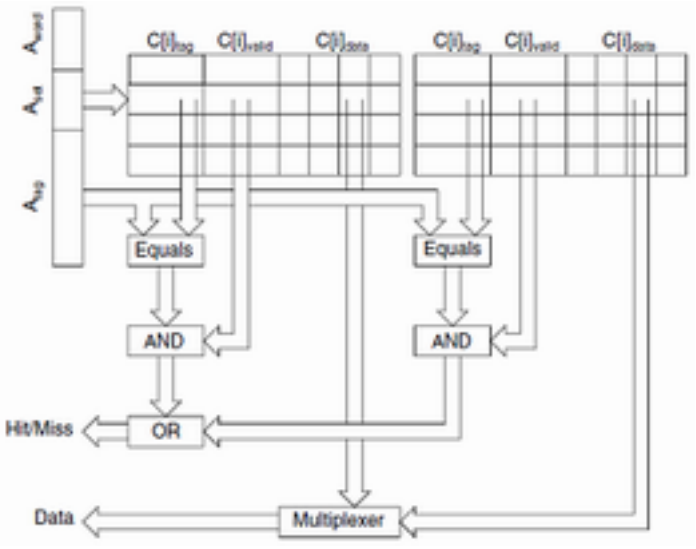

Figura 10. Procedeul prin care se verifică existența unui cuvânt în cache pentru maparea set asociativă

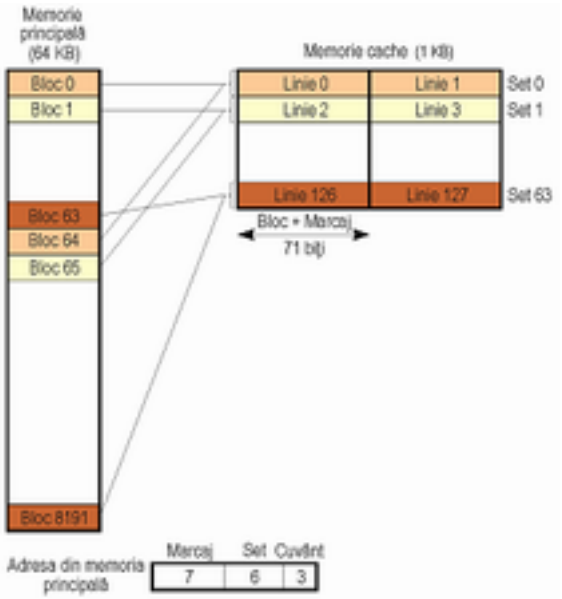

Figura 11. Exemplu de mapare set asociativă pentru un cache de 1K cu 2 linii pe set și memorie principală de 64K

Pentru a înlocui o linie în cadrul unui segment, vom urma una dintre politicile de evacuare. Am putea selecta LRU. Pentru că *N* este de dimensiuni reduse (2, 4, 8), este fezabilă alegerea unei politici de evacuare foarte eficiente precum LRU. Această opțiunea este neviabilă pentru memoriile

complet asociative datorită faptului că acolo există prea multe posibilități pentru a permite o realizare eficientă și acceptabilă ca preț.

4.4 Politici de evacuare a liniilor de cache

Pentru construcția mecanismului de traducere a adreselor, trebuie avut în vedere, mai întâi, modul în care determinăm pe ce linie punem noile adrese aduse din memorie. Cum în majoritatea cazurilor cache-ul este plin, trebuie făcut loc noilor cuvinte prin evacuarea datelor dintr-o linie de cache înapoi în memorie.

Politicile cele mai cunoscute pentru evacuarea unei linii sunt:

- ***Aleator***: Folosind un generator pseudo-aleator de numere, se alege un număr la întâmplare, indiferent de adresă și de ceea ce se găsește în cache. Este o variantă necostisitoare, dar poate suferi dacă se generează o secvență nedorită, care duce la contenție.
- ***LRU (Least Recently Used)***: De fiecare dată când o linie este accesată, este mutată în capul listei de utilizare. Când memoria cache trebuie să încarce date noi, este selectată linia de la coada listei; raționamentul este că aceasta, nemaifiind folosită de ceva timp, cel mai probabil nu va mai fi folosită. Pentru memorii cache de dimensiuni mari întreținerea unei liste LRU este o opțiune neviabilă.
- ***LFU (Least Frequently Used)***: Linia care conține date accesate de cele mai puține ori este înlocuită.
- ***FIFO (First In, First Out)*** sau ***LRR (Least Recently Replaced)***: Linia care conține datele cele mai vechi este înlocuită. Deosebirea față de LRU este că o linie poate fi eliminată chiar dacă ea a fost accesată des pentru a fi citită.

5. Organizarea memoriei cache

5.1 Ocolirea cache-ului

Uneori este util/atractiv să se realizeze acces direct la memoria principală. Pentru a permite acest lucru, proiectarea sistemului de calcul are două opțiuni. În primul rând, s-ar putea furniza instrucțiuni explicite (sau un mod al procesorului) care întreprinde accesul fără a trece prin cache; acest procedeu se numește ocolirea memoriei cache (***cache bypass)***. În al doilea rând, s-ar putea furniza un mecanism de oprire a memoriei cache. Unele procesoare permit ca acest lucru să se realizeze la pornire, urmând a funcționa fără cache. Alte procesore permit o comutare dinamică on/off a memoriei cache. Ocolirea memoriei este un procedeu riscant datorită creșterii șanselor de apariție a inconsistențelor între cache și memoria principală.

5.2 Memorii cache divizate prin utilitate

Conceptul de divizare prin utilitate, de separare a datelor de instrucțiuni este folosit și în ceea ce privește organizarea memoriei cache a procesoarelor, însă a fost introdus de arhitecturile de tip Hardvard. Spre deosebire de arhitectura von Neumann, care impune o singură magistrală și memorie pentru date și instrucțiuni, în arhitectura Harvard se face o separare clară a acestora, memoria de date fiind separată de cea de instrucțiuni (de multe ori și de tipuri diferite: flash și SRAM; cum e în cazul microcontrollerelor). La arhitectura Harvard accesul la memorii se face pe magistrale separate, astfel procesorul putând să citească o instrucțiune și să citească/scrie date în același timp. Datorită acestui avantaj de rapiditate adus de arhitectura Harvard, s-a recurs la arhitectura Harvard modificată (**Modified Harvard Architecture**) întâlnită în majoritatea procesoarelor actuale (x86, ARM etc). Astfel, procesorul poate accesa datele și instrucțiunile pe magistrale separate și din cache-uri separate, pe modelul Harvard, însă spațiul de adresare este comun, accesul la memoria principală fiind făcut după modelul von Neumann, iar instrucțiunile putând fi privite ca date la acest nivel (se pot face programe care modifică codul la runtime).

Unul din avantajele existenței unui **cache de date separat de cel de instrucțiuni** este cel al diferitelor forme de localitate specifice fiecăreia (vedeți exemplul de la pre-fetching din secțiunea 2). Astfel, se folosesc două memorii cache, una pentru date, alta pentru instrucțiuni. Această implementare pare mai sigură și pentru că separarea accesărilor poate micșora șansele de poluare și de contenție în cache. Un alt avantaj ar fi că se pot implementa anumite optimizări ale logicii hardware a cache-urilor. Spre exemplu, memoria cache de instrucțiuni poate fi mai simplă decât cea de date. Aceasta nu va avea nevoie de logica necesară operațiilor de scriere deoarece, în cele mai multe cazuri se efectuează doar citirea de instrucțiuni, iar în situațiile rare de ”self-modifying code” instrucțiunile noi se vor scrie direct în memoria principală.

5.3 Memorii cache divizate pe niveluri

Este posibilă stabilirea unei ierarhii și între memoriile de tip cache. În particular este atractiv să facem distincția între cache de pe același chip cu procesorul (*on-chip cache*) și cache din afara chip-ului. Diferența dintre ele este evidentă prin dimensiune, *off-chip cache* este mai mare decât memoria de pe chip. În lipsa unei interfețe complicate de comunicare și aflându-se la distanță mică de procesor, memoria on-chip este mai rapidă. Pe baza distanței față de procesor se identifică memoria cache de nivel 1 (**L1 cache**) – cu dimenisiunea cea mai mică, memoria cache de nivel 2 (**L2 cache**) - mai mare și memoria cache de nivel 3 (**L3 cache**). În funcție de arhitectura

procesorului unul sau mai multe core-uri pot partaja același cache și numărul de niveluri poate varia în funcție de costurile și utilitatea acestuia.

Ideea din spatele implementării este că L1 cache se va ocupa de majoritatea accesărilor memoriei, într-un timp foarte scurt, fiind separată în memorie cache de date și memorie cache de instrucțiuni. Cele pe care nu le poate deservi vor fi preluate de celelalte niveluri de cache, în locul memoriei principale. Pentru a avea o oarecare eficiență, acest mecanism se bazează pe valabilitatea principiului localității. Există două filozofii de construcție a memoriei cache pe niveluri:

1. ***cache exclusiv*** – liniile evacuate din cache L1 sunt trimise în cache L2 și așa mai departe. Aducerea liniilor în memoria cache L1 se face direct, deci rapid.
2. ***cache inclusiv*** – liniile din cache L1 sunt prezente mereu în toate nivelele de cache. Evacuarea liniilor este mai rapidă.

6. Memorii cache pentru sisteme multi-procesor

În sistemele de calcul moderne se întâlnesc în mod uzual mai multe procesoare, fiecare cu mai multe nuclee. Figura 12 prezintă un exemplu de schemă bloc a unui procesor multicore (arhitectura Intel i7 sau AMD Phenom - diferă dimensiunile cache-urilor).

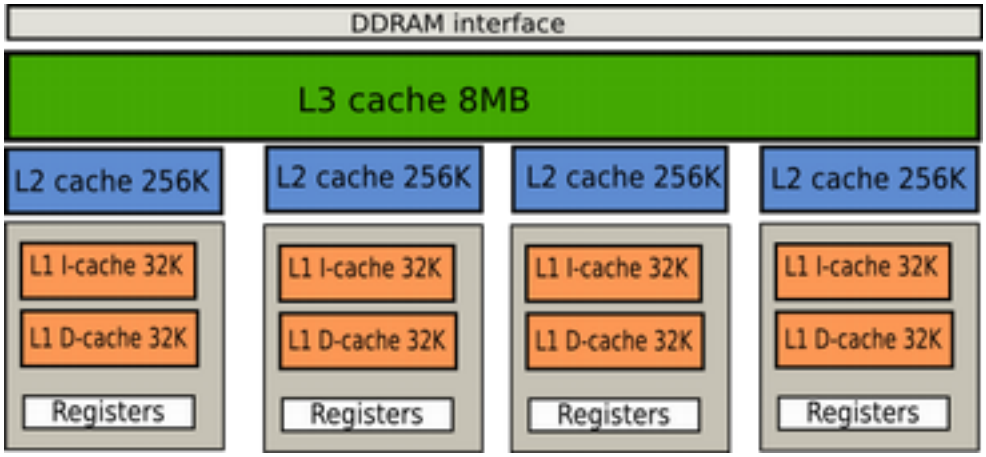

Figura 12. Ierarhia de memorii cache într-un procesor cu 4 nuclee

În imagine sunt prezentate cele patru nuclee ale procesorului. Nucleele au un cache L1 pentru date, unul pentru instrucțiuni și un cache L2 unificat. Chiar dacă nucleele au propriul set de regiștrii, cache L1 și L2, ele partajează memoria cache L3 și memoria principală.

Această structură aduce pe lângă beneficii și noi probleme. Astfel, în sistemele multi-procesor simetrice, memoriile cache ale diferitelor procesoare nu pot lucra independent. Toate procesoarele trebuie să vadă, la un anumit moment de timp, exact aceleași date în memorie. Menținerea acestei uniformități a vederii memoriei poartă numele de ***"asigurarea coerenței memoriei cache"***.

Pentru cazul cel mai întâlnit, al memoriilor cache de tip *write-back*, dacă un procesor ar analiza doar conținutul propriului cache nu ar vedea liniile murdare din cache-urile celorlalte procesoare. O soluție deloc eficientă ar fi furnizarea accesului la cache-ul unui procesor către celelalte procesoare. O altă soluție, mai eficientă, are la bază detectarea de către celelalte procesoare/ nuclee a intenției unui procesor/nucleu de a scrie în propriul cache. Dacă s-a detectat o operațiune de scriere în cache-ul altui procesor, celelalte procesoare își vor marca linia respectivă ca invalidă. Când vor dori să acceseze acea linie, va avea loc o operație de fetch din memoria principală.

Memoriile cache avansate oferă suport pentru tehnici de snooping. Când un procesor cere o linie murdărită de alt procesor, procesorul care a murdărit acea linie detectează ("snoops in on the other processors") cererea și transmite linia direct în cache-ul primului procesor, eventual scriind-o și în memoria principală în proces.

Pentru menținerea coerenței memoriei cache s-au dezvoltat numeroși algoritmi, cel mai important fiind ***MESI (Modified, Exclusive, Shared, Invalid)***. Numele acestuia provine de la cele patru stări în care se poate afla o linie din cache, cele mai importante tranzițiile dintre acestea fiind date de următoarele condiții:

- o linie murdară este prezentă în cache-ul unui singur procesor – cel care a murdărit-o
- copii curate ale aceleiași linii se pot regăsi în oricât de multe cache-uri

Partajarea memoriei principale și a cache-ului în sistemele multicore și multiprocesor reprezintă o problemă complexă și necesită o tratare separată, de aceea accentul acestui laborator nu este pus pe această tematică. Mai multe informații puteți să găsiti în articole de pe site-urile din domeniu [1] [2] sau în articole academice.

Bibliografie

[1] J. Hennessy, D. Patterson. *Computer Architecture: A quantitative approach*, 4th ed.
[2] D.Page. *A Practical Introduction to Computer Architecture*, Springer, 2009
[3] W. Stallings. *Computer Organization & Architecture,* Pearson Education, Prentice Hall, 2003
[4] N. Jouppi. *Cache Write Policies and Performance*, 1991
[5] S. Berg. *Cache Prefetching*, 2002. pdf download
[6] Software Techniques for shared cache multi-core systems: http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/, last accessed 16.11.2011
[7] Effective Use of the Shared Cache in Multi-core Architectures: http://drdobbs.com/embedded-systems/196902836, last accessed 16.11.2011
[8] CPU-Z http://www.cpuid.com/softwares/cpu-z.html