Cache-uri

Mihai BUDIU
mihaib@pub.ro,budiu@cs.cornell.edu,
http://www.cs.cornell.edu/Info/People/budiu/budiu.html

ianuarie 1997

Subiect: principiul cache-ului şi aplicaţii

Cuvinte cheie: cache, memorie virtuală, memorie asociativă, politică de înlocuire, localitate

Cunoştinţe necesare: arhitectura elementară a calculatoarelor

Cuprins

1 Ideea 2

2 Numele 2

3 Eficienţa şi localitatea 3

4 Variaţiuni 4

5 Algoritmul 6

6 Incarnaţii 10

6.1 Cache-ul de disc 10
6.2 Cache-ul Microprocesorului 11
6.2.1 Cache-ul cu adresare directă (direct mapped) 11
6.2.2 Cache-ul cu adresare asociativă (fully associative) 13
6.2.3 Cache-ul parţial asociativ (set-associative) 14
6.3 Cache-ul shell-ului 15
6.4 Serverele de Nume ("Domain Name Servers") 15
6.5 Clienţii de WWW 16
6.6 Memoria Virtuală 16
6.7 "Translation Lookaside Buffer" (TLB) 18

În acest articol îmi propun să disec unul dintre cele mai simple dintre "aparatele" folosite de calculatoare. Voi încerca să arăt că există foarte multe varietăţi de cache, dar că toate se bazează pe aceeaşi idee foarte simplă: ţine la-ndemînă ceea ce foloseşti des. Voi încerca de asemenea să arăt că fiecare calculator foloseşte cache-ul într-o mulţime de instanţe, şi că impactul lui asupra eficienţei este extrem de mare.

1 Ideea

Cuvîntul "cache" este extrem de familiar celor care lucrează cu calculatoare; chiar şi cînd vrei să cumperi un calculator ţi se spune: "120MHz, cache de 256Kb, etc, etc". Cuvîntul este împămîntenit în forma asta, şi cum nu cunosc o traducere rezonabilă, voi continua să-l folosesc astfel. Voi desluşi puţin mai tîrziu sursa numelui (care este un cuvînt franţuzesc, dar folosit ca atare de întreaga comunitate anglofonă a calculatoriştilor, ceea ce e se întîmplă foarte rar). Pronunţia la rîndul ei variază: este sau "caşe" (franţuzism), sau "cheiş" (citire americană a cuvîntului).

Ce este un cache?

Mai curînd decît un dispozitiv concret putem spune despre cache că este un "principiu". Există foarte multe feluri de cache, unele construite din hardware special, altele care sunt doar programe. Toate însă se bazează pe aceeaşi idee:

Noi avem acasă un dulap (nu prea mare, dar la-ndemînă) şi o debara (ceva mai mare, dar ce dezordine înauntru!). Primăvara şi toamna se produce o mutare masivă de haine: hainele groase o iau într-o direcţie, dislocuindu-le pe cele subţiri. Decizia este naturală, chit că ia o zi sa faci mutarea: după aia ai tot ce-ţi trebuie la-ndemînă pentru juma' de an. O dificultate se iveşte cînd dai peste o iarnă extrem de caldă, sau o vară prea rece: trebuie să scurmi în debara după ceva haine şi cu alt prilej decît cu schimbarea sezonului.

Acesta este un exemplu tipic de cache.

În calculatoare lucrurile stau tot aşa: am la dispoziţie două memorii, una ieftină, mare (pentru că-ţi permiţi să cumperi), dar cam lentă, una mică şi scumpă, dar rapidă. (Cel mai bine e sa fii şi bogat şi sănătos, dar nu se poate întotdeauna.) În plus mai am şi o cantitate mare de date de păstrat, aşa că trebuie să le ţin în memoria lentă [debaraua], care e suficient de încăpătoare. Din păcate durează prea mult să iau şi să pun lucruri acolo: nu e practic să am numai debara; trebuie şi un dulap (memoria rapidă): doar nu o să pun hainele musafirilor în debara, nu? Memoria rapidă [dulapul] este cache-ul. Din păcate nu încape totul în memoria rapidă, aşa că sunt forţat ca atunci cînd pun ceva în ea, să scot altceva în memoria lentă. Treaba este avantajoasă atîta vreme cît lucrurile aduse în memoria rapidă sunt folosite frecvent, deci nu trebuie să le tot mut. Figura 1 arată cum stau lucrurile.

Observaţi ca scopul meu nu este nici să am debara, nici să am dulap, ci să pot să pun undeva hainele, şi să le pot manipula suficient de uşor. Dulapul şi debaraua sunt doar *unelte*, nu scopuri. Dacă debaraua ar fi suficient de la-ndemînă, n-aş mai folosi dulapul deloc.

Atenţie la analogie, care nu merge pînă la capăt: în calculatoare conţinutul memoriei se copiază foarte uşor, ceea ce nu este adevărat despre paltoane. În calculatoare, cînd "mut" ceva, de fapt fac o copie; acel ceva rămîne şi în memoria-sursă (locul de unde iau).

2 Numele

Abstract vorbind: vreau să implementez două operaţii pentru stocarea de date: *citeşte* şi *scrie*. Pot să citesc şi să scriu din memoria lentă, şi totul merge cum trebuie, dar prea lent. Pentru a

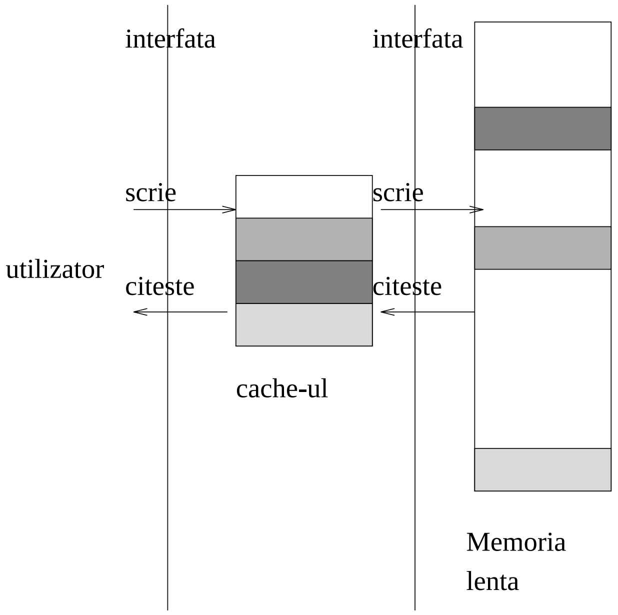

Figure 1: Principiul cache-ului

spori eficienţa, copiez parte din memoria lentă în cea rapidă, şi încerc să citesc şi să scriu acolo. Cînd nu găsesc ce-mi trebuie, operez din nou în memoria lentă.

"Caché", (cu accent ascuţit pe ultimul e), înseamnă în limba franceză "ascuns". Aşa este şi memoria rapidă: este un dispozitiv care poate să fie, sau poate să lipsească; absenţa lui se va remarca numai printr-o viteză mai scăzută, dar nu printr-o reducere a funcţionalităţii (adică toate operaţiile pe care le pot face cu cache-ul, le pot face şi fără el).

În limbaj tehnic asta înseamnă că un cache oferă aceeaşi interfaţă lumii exterioare ca interfaţa pe care el o foloseşte; nu oferă nici o funcţie suplimentară (pentru că atunci mi-aş da seama cînd l-aş scoate că lipseşte ceva), ci doar creşte eficienţa.

Figura 1 arată acest lucru, indicînd interfeţele, care constau din două proceduri: una pentru scriere şi una pentru citire.

Cu toate că nu face mare lucru, vom vedea că arhitectura internă a cache-ului nu este banală, şi că pentru a folosi eficient spaţiul mic pe care-l are, trebuie să-şi dea ceva osteneală.

3 Eficienţa şi localitatea

În mod evident, un cache nu este util în orişice situaţie: să ne imaginăm o ţară cu o climă foarte capricioasă, în care fiecare zi este altfel decît cea precedentă. În cazul ăsta aproape niciodată nu am avea în dulap hainele necesare, şi ar trebui să le mutăm din debara şi înapoi. Dulapul mai mult ar complica treaba decît ar ajuta, pentru că pe lîngă faptul că nu oferă hainele de care avem nevoie, mai trebuie să avem grijă şi de el.

La fel este şi în calculatoare: un cache este util *numai* dacă anumite informaţii sunt folosite frecvent şi mult. Atunci acele informaţii merită păstrate înauntru. Din fericire, experimental s-a constatat că acest lucru este foarte adesea adevărat. Această observaţie poate fi formulat în felurite moduri; unul dintre ele este "principiilor localităţii (locality principles)". Avem două feluri de localitate:

- localitate *spaţială*: dacă ai nevoie de nişte date, în curînd vei avea nevoie probabil de date aflate în apropierea lor în memorie;
- localitatea *temporală*: cînd ai pus mîna pe nişte date, foarte adesea le vei folosi de mai multe ori.

Acestea sunt doar *observaţii*, dar se potrivesc destul de bine la programele de calculator, aşa cum funcţionează ele în calculatoarele moderne (poate în viitor lucrurile se vor schimba). Validitatea observaţiilor ne permite să folosim cache-uri. Asta nu înseamnă că nu există programe care folosesc prost cache-urile; dimpotrivă, dacă vreţi puteţi scrie destul de uşor un astfel de program (este o metodă eficientă de a încetini calculatorul de la locul de muncă). Programele obişnuite însă nu se comportă aşa.

Eficienţa unui cache se măsoară în procentajul de "nimereli" H (*hit ratio*): din 100 de accese, cîte găsesc datele în cache? Opusul acestei valori este *miss ratio* M (rateuri). Procentajele astea se măsoară rulînd o grămadă de programe şi făcînd media. Avem desigur, dacă socotim 33% ca 1/3:

$$H = 1 - M$$

Dacă timpul de citire din cache este Tc ("hit time"), iar timpul pe care îl pierdem cînd ratăm este Tm ("miss time") atunci putem măsura timpul mediu de acces la memoria cu cache cu următoarea formulă:

$$T = Tc \times H + Tm \times M$$

Observaţi că timpul unei ratări (Tm) nu este neapărat egal cu timpul de citire din memoria lentă (Tl), pentru că în cazul unei ratări, întîi trebuie să ne dăm seama dacă datele sunt în cache, iar dacă nu sunt să accesăm memoria lentă.

Cache-ul va fi eficient dacă $T < Tl$. Altfel mai bine fără el.

H depinde de mărimea cache-ului: pentru un cache de mărimea memoriei lente (caz limită), toate datele pot fi ţinute în memoria rapidă, şi vom avea $H = 1$. Pentru un cache de mărime 0, $H = 0$, pentru că niciodată datele nu se găsesc în el. Relaţia între mărimea cache-ului, a memoriei lente şi H nu este o linie dreaptă, ci creşte repede la început (figura 2). Din cauza asta un cache relativ mic ca mărime are o importanţă mare ca eficienţă.

Eficienţa depinde şi de raportul dintre Tc şi Tm; în anumite cazuri Tm este de ordinul a $10000 \times Tc$, deci chiar un H mic poate să însemne mare lucru.

4 Variaţiuni

Pe lîngă dimensiuni şi timpi de acces, există o sumedenie de detalii prin care cache-urile diferă între ele, datorate faptului că mediile de stocare a informaţiei nu se comportă chiar la fel. Iată unele dintre posibilele diferenţe:

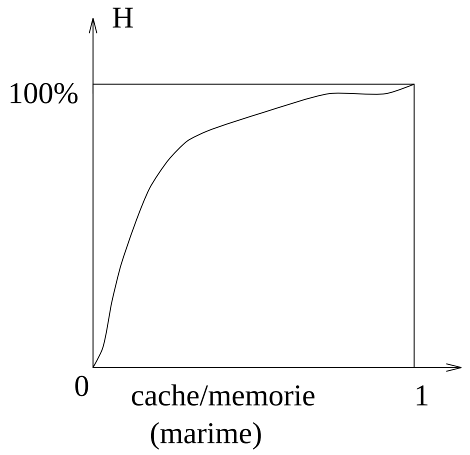

Figure 2: Performanţa cache-ului

1) Mărimea blocului de date: cîteodată este mai economic să transporţi mai multe date deodată din memoria lentă; cache-urile aduc atunci mai mult decît li se cere (un calup) şi păstrează totul, în ideea (sugerată de principiul de localitate spaţială) că şi vecinii obiectului accesat vor fi căutaţi în curînd. Unitatea de transfer între cache şi memoria lentă e numită *bloc*.

2) Politica de înlocuire: după o vreme cache-ul se va umple cu date, şi totuşi altele vor trebui aduse. Decizia despre care date trebuie scoase afară este foarte importantă pentru eficienţă; ea este *politica de înlocuire* (replacement policy). Există o sumedenie de politici; iată-le doar numite pe unele; orice carte de sisteme de operare va descrie cele mai multe dintre ele: politica aleatoare (random), politica circulară (round robin), politica "cel mai rar folosit" (least frequently used), politica "primul intrat – primul ieşit" (first in, first out), politica "cel mai demult folosit" (least recently used), politica "setul de lucru" (working set), politica "optimă" (optimal), politica ceasului, politica celei de-a doua şanse (second-chance).

3) Politica de scriere: odată cu prezenţa unui cache, datele efectiv devin duplicate: există o copie în cache. Cînd se fac scrieri, care dintre copii trebuie modificată? Una sau amîndouă? În funcţie de circumstanţe există varii răspunsuri la această întrebare.

4) Metoda de identificare: cînd vreau ceva din memoria lentă indic *adresa* la care acel obiect se găseşte. Principiul transparenţei (faptul că interfeţele sunt identice) cere ca in cache datele să fie căutate tot după această adresă; dar cum cache-ul este mic, adresele din memoria externă (= memoria lentă) nu reprezintă adrese şi în memoria rapidă. Cum găsesc datele? Răspunsul este dat de *metoda de identificare* şi strîns legat de politica de înlocuire, pentru că datele trebuie căutate acolo unde puteau fi aduse.

5) Timpul de viaţă al informaţiei: dacă dintre copiile pe care le am (una în cache şi alta în memorie) una se schimbă? Care este cea bună după aceea? Ce trebuie să fac cu cealaltă? Cu ce ocazie trebuie făcută schimbarea?

5 Algoritmul

Schema generală de funcţionare a unui cache este descrisă la un nivel abstract de următorul program în C. Aceasta este doar schema generală; în mod normal codul ar trebui să mai facă o grămadă de teste de eroare.

Singurul punct mai interesant este calculul adreselor: dacă împart fişierul în blocuri de mărime `MARIME`, octetul cu adresa `a` se va afla în blocul cu numărul `a / MARIME` în fişier (ATENŢIE: nu şi în cache!) şi va fi octetul cu numărul `a % MARIME` în acel bloc. Adresa de început a blocului va fi `a - a % MARIME`.

De asemenea, împărţirea în proceduri, constantele globale, nu sunt întotdeauna perfect alese; am încercat să păstrez cît mai clară structura esenţială.

```
typedef unsigned char BYTE;

#define DA 1
#define NU 0

#define BLOCURI 20
#define MARIME 1024

struct bloc {
   unsigned int liber:1;      /* e folosit acest bloc? */
   unsigned int modificat:1;  /* e modificat acest bloc? */
   unsigned int adresa;       /* datele din fisier intre adresa si adresa+MARIME-1 */
   BYTE date[MARIME];         /* informatia */
};

typedef int blocnr;

struct cache {
   struct bloc blocuri[BLOCURI];
   int blocuri_libere;
   FILE * fisier;
};

/* Metoda de identificare implementata de 'cauta' */
extern blocnr cauta(struct cache *c, unsigned int adresa);

/* Politica de inlocuire implementata de 'alege_bloc' */
extern blocnr alege_bloc(struct cache *c, unsigned int adresa);

/* metodele de citire/scriere in fisier */
extern void
citeste_din_fisier(FILE *f, int pozitie, char * buffer, int cantitate);
extern void
```

```
scrie_in_fisier(FILE *f, int pozitie, char * buffer, int cantitate);

static int contine(struct cache * c, blocnr bloc, unsigned int adresa)
/* DA daca blocul indicat contine adresa respectiva */
{
   if (c->blocuri[bloc].liber) return NU;
   if ((c->blocuri[bloc].adresa <= adresa) &&
       (c->blocuri[bloc].adresa + MARIME > adresa)) return DA;
   else return NU;
}

static blocnr aduce(struct cache * c, unsigned int adresa)
/* gaseste blocul care contine aceasta
   adresa, sau aduce datele cu
   aceasta adresa intr-un bloc */
{
   blocnr bloc;

   bloc = cauta(c, adresa);  /* caut adresa indicata in cache */
   if (! contine(c, bloc, adresa)) {
        if (c->blocuri_libere > 0) {
           bloc = alege_bloc_liber(c);
           c->blocuri[bloc].liber = NU;
           c->blocuri_libere -= 1;
       } else {
           bloc = alege_bloc(c);
           if (c->blocuri[bloc].modificat) {
               scrie_in_fisier(c->fisier,           /* fisier */
                          c->blocuri[bloc].adresa,  /* offset in fis. */
                          c->blocuri[bloc].date,    /* datele */
                          MARIME);                  /* octeti */
               c->blocuri[bloc].modificat = NU;
           }
       }

       /* am facut rost de un bloc in cache;
          trebuie sa aducem blocul cu octetul respectiv
          de pe memoria externa */
       c->blocuri[bloc].adresa = adresa - adresa % MARIME;
       citeste_din_fisier(c->fisier,                /* fisier */
                          c->blocuri[bloc].adresa,  /* offset in fis. */
                          c->blocuri[bloc].date,    /* datele */
                          MARIME);                  /* octeti */
       c->blocuri[bloc].modificat = NU;
```

```
   }
   return bloc;   /* acesta este blocul cu datele */
}

void scrie(struct cache * c, unsigned int adresa, BYTE valoare)
   /* scrie un octet folosind cache-ul */
{
   blocnr bloc;

   bloc = aduce(c, adresa);
   c->blocuri[bloc].date[adresa % MARIME] = valoare;
   c->blocuri[bloc].modificat = DA;
}

BYTE citeste(struct cache * c, unsigned int adresa)
   /* citeste un octet folosind cache-ul */
{
   blocnr bloc;

   bloc = aduce(c, adresa);
   return c->blocuri[bloc].date[adresa % MARIME];
}
```

Codul de mai sus implementează două proceduri: `scrie` şi `citeste`. Acestea manipulează structura de date pusă la dispoziţie pentru a memora blocuri. Procedura centrală este `aduce`. Ea caută blocul în cache, dacă nu e scoate unul afară pentru a face loc, şi aduce blocul în locul disponibil.

Cei cinci parametri sunt vizibili în cod astfel:

1) Mărimea blocului: constanta `MARIME`

2) Politica de înlocuire: este realizată de funcţia `alege_bloc()`, pe care n-am scris-o. Această funcţie va arăta diferit în funcţie de politica dorită. De exemplu, pentru a implementa o politică aleatoare, corpul ei, foarte simplu, este:

```
#include <stdlib.h>  /* pentru rand() */

blocnr alege_bloc(struct cache *c, unsigned int adresa)
{
   return rand() % BLOCURI;
}

blocnr alege_bloc_liber(struct cache *c)
  /* chemata numai cind EXISTA blocuri libere; gaseste unul */
{
```

```
   blocnr b;

   for (b=0; b < BLOCURI; b++) if (c->blocuri[b].liber) return b;
}
```

3) Politica de scriere este în implementarea de mai sus cea numită *write-back*; scrierile se produc doar în copia din cache a datelor. [Opusa ei este *write-through*: cînd am de scris ceva scriu simultan şi în memoria lentă. Cîteodată aceste două politici se suplimentează cu regula: *no-load-on-write-miss*: cînd scrierea *nu* găseşte datele în cache nu le mai încarcă, ci doar scrie pe disc.]

4) Metoda de identificare este de asemenea neprecizată; ea este treaba funcţiei `cauta()`. Cea mai simplă implementare posibilă este probabil căutarea liniară:

```
blocnr cauta(struct cache *c, unsigned int adresa)
  /* gaseste blocul care contine aceasta adresa */
{
   blocnr b;

   for (b=0; b < BLOCURI; b++)
     if (contine(c, b, adresa)) return b;

   /* daca nu l-am gasit pot da orice rezultat;
      oricum se verifica din nou */
   return 0;
}
```

5) Datele rămîn în cache-ul acesta pînă sunt scoase afară din lipsă de spaţiu. De obicei cache-urile conţin şi o procedură care le "goleşte", astfel (numele tradiţional este `sync`, pentru că procedura "sincronizează" cache-ul cu memoria lentă):

```
void goleste(struct cache *c)
  /* salveaza tot */
{
   blocnr b;

   for (b=0; b < BLOCURI; b++) {
      if (!c->blocuri[b].modificat) continue;
      scrie_in_fisier(c->fisier,                /* fisier */
                      c->blocuri[b].adresa,     /* offset in fis. */
                      c->blocuri[b].date,       /* datele */
                      MARIME);                  /* octeti */
      c->blocuri[b].modificat = NU;
   }
}
```

Cache-ul este aproape complet; Mai trebuie şi o procedură de iniţializare; ceva de genul:

```
void initializeaza(struct cache *c, FILE * fisier)
{
   blocnr b;

   c->fisier = fisier;
   for (b=0; b < BLOCURI; b++)
      c->blocuri[b].liber = DA;
   c->blocuri_libere = BLOCURI;
}
```

Puţine proceduri mai trebuie scrise pentru a putea fi testat; `scrie_in_fisier()` şi `citeste_din_fisier()` se pot scrie foarte simplu:

```
#include <stdio.h>

void citeste_din_fisier(FILE *f, int pozitie,
                        char * buffer, int cantitate)
{
    fseek(f, (long)pozitie, SEEK_SET);
    fread(buffer, 1/* octet */, cantitate, f);
}

void scrie_in_fisier(FILE *f, int pozitie,
                     char * buffer, int cantitate)
{
    fseek(f, (long)pozitie, SEEK_SET);
    fwrite(buffer, 1, cantitate, f);
}
```

6 Incarnaţii

Încheiem acest articol cu o listă (incompletă) de aplicaţii ale cache-urilor.

6.1 Cache-ul de disc

Orice sistem de operare modern (mai puţin MS-DOS) are un cache de disc. (Chiar şi pentru MS-DOS există `smartdrive` sau `ncache` de la Norton). Cache-ul de disc este probabil una din cele mai mari surse de eficienţă într-un sistem de operare. Acesta se datoreşte faptului că diferenţa între timpul de acces la disc şi cel la memorie este uriaşă (timpul de acces al unei memorii este de circa 60-70 de nanosecunde, adică $60 \cdot 10^{-9}$, iar timpul de acces al unui disc este de ordinul a 10 milisecunde, adică $10 \cdot 10^{-3}$). Cache-ul de disc este o structură de date care conţine un vector de blocuri de mărime egală. Discul este la rîndul lui împărţit în blocuri de aceeaşi dimensiune. Cînd utilizatorul cere un octet de pe disc, blocul care conţine acel octet este încărcat în cache, eventual scoţînd un alt bloc afară.

Din cele 5 puncte de vedere indicate anterior, un cache de disc are următoarele caracteristici:

1) Mărimea blocului: Blocuri mari (512 octeţi – 8 Kb).

2) Politica de înlocuire: Politica de înlocuire este de obicei: dă-l afară pe cel pe care nu l-am folosit cel mai mult timp (Least Recently Used).

3) Politica de scriere: Cache-urile de disc sunt în general "write-back". Asta înseamnă că atunci cînd se *scrie* pe disc, modificările sunt făcute doar în cache. Ele sunt mutate pe disc doar cînd blocul respectiv este dat afară, sau cînd acest lucru este cerut explicit de utilizator.

4) Metoda de identificare: Pentru a găsi un bloc în cache se folosesc algoritmi de *hash*, care sunt foarte eficienţi (nu ne putem permite să intrăm în detalii; orice carte elementară de algoritmi descrie hash-urile).

5) Timpul de viaţă al informaţiei din cache: Pentru a preveni catastrofele, sistemele de operare "golesc" (scriu toate blocurile modificate) din cache pe disc periodic (de ex. la 30 de secunde).

6.2 Cache-ul Microprocesorului

Un microprocesor la 200 de Megahertzi (un Pentium pro, de pildă) are un ciclu de instrucţiune de $1/(200 \cdot 10^6) = 5$ nanosecunde. O instrucţiune poate dura un număr variabil de cicluri, între 1 şi cîteva zeci. Executarea unei instrucţiuni înseamnă: citirea ei din memorie, decodificarea, executarea, memorarea rezultatelor. Dacă accesul la memorie durează 60 de nanosecunde atunci la fiecare citire procesorul trebuie să piardă 12 cicluri! Din cauza asta între microprocesor şi memoria RAM principală se pune un cache construit din memorie rapidă, cu timp de acces de 5–10 nanosecunde.

Cîteodată designerii pun chiar mai mult decît atît: două nivele de cache între procesor şi RAM: un nivel ceva mai lent, dar mai mare (pentru un PC între 64Kb şi 512Kb de obicei), şi un cache construit chiar în microprocesor, de ordinul a 1-10Kb, mult mai rapid.

Aceste cache-uri se implementează folosind hardware specializat.

1) Mărimea blocului: blocurile sunt mici 1 – 16 octeţi.

2) Politica de înlocuire,**4)** Metoda de identificare:

Există două clase mari de cache-uri de microprocesor, şi una intermediară. Ele diferă prin locurile din cache în care un octet din memoria externă poate fi plasat. Cele două mari varietăţi sunt: cache-ul cu adresare directă, în care locul fiecărui octet este unul şi precis calculat, şi cache-ul asociativ, în care un octet din memoria externă poate fi plasat în orice loc din cache.

6.2.1 Cache-ul cu adresare directă (direct mapped)

De obicei chiar structura adresei este folosită la căutare. Figura 3 arată cum este plasat un anume bloc în cache: biţii de la sfîrşitul adresei blocului dau şi posibila poziţie a blocului

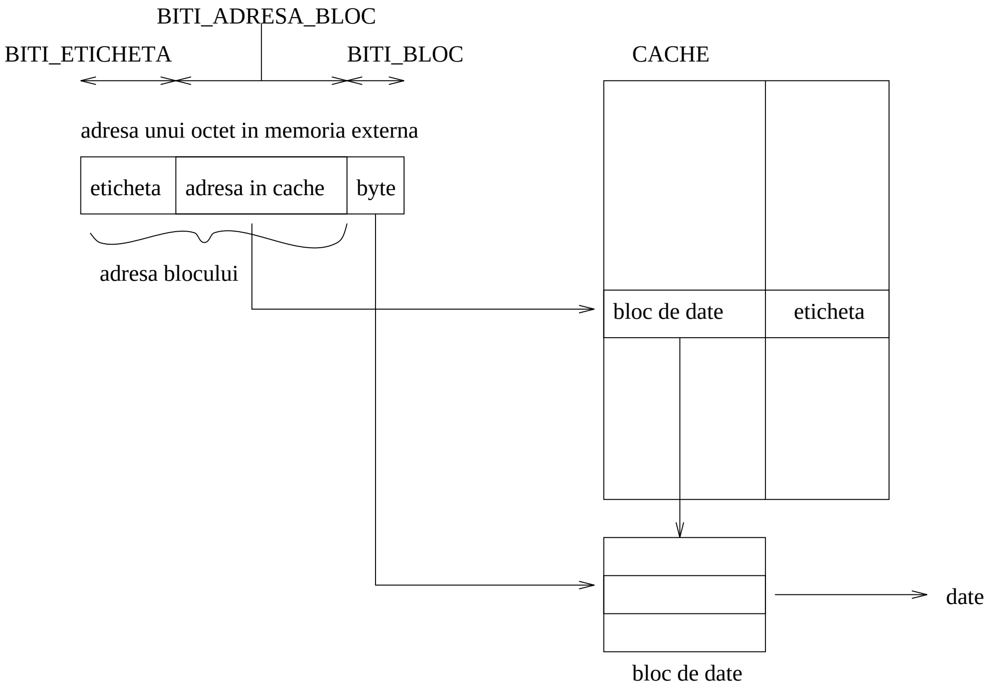

Figure 3: Cache-ul cu adresare directă

în cache. Biţii din începutul adresei blocului constituie verificarea dacă blocul este cel aflat în cache (mai multe blocuri candidează pentru aceeaşi poziţie; cel care se află înauntru este indicat prin *etichetă* (tag)).

În fine, ultimii biţi din adresă indică poziţia octetului în blocul de date.

Marele avantaj al schemei directe este că dată fiind adresa, poziţia în cache a blocului este unic determinată, şi nu trebuie făcută nici o căutare. Politica de înlocuire nu există din acelaşi motiv: nu poţi alege în ce loc să aduci un bloc. Din cauza asta funcţia de căutare şi cea de înlocuire sunt identice.

În termenii codului anterior un exemplu ar fi:

```
#define BITI_BLOC        4   /* 2^4 octeti in bloc */
#define BITI_ADRESA_BLOC 8   /* 2^8 blocuri in cache */
#define MASCA_BLOC    0xff   /* un nr format din BITI_ADRESA_BLOC biti 1 */
#define BITI_ETICHETA    4

#define MARIME (1 << BITI_ETICHETA)     /* marimea unui bloc */
#define BLOCURI (1 << BITI_ADRESA_BLOC) /* nr blocuri */

#define BITI_ADRESA (BITI_BLOC + BITI_ADRESA_BLOC + BITI_ETICHETA)
  /* marimea adresei */
```

```
blocnr alege_bloc(struct cache *c, unsigned int adresa)
{
   return (adresa >> BITI_BLOC) & MASCA_BLOC;
}

blocnr cauta(struct cache *c, unsigned int adresa)
{
   return alege_bloc(c, adresa);
}
```

(Pentru eficienţă putem rescrie şi structura blocului, să păstreze numai eticheta în loc de adresă, şi sa facă toate calculele numai cu biţi; de exemplu:)

```
int contine(struct cache * c, blocnr bloc, unsigned int adresa)
/* DA daca blocul indicat contine adresa respectiva */
{
   unsigned int diferenta;

   if (c->blocuri[bloc].liber) return NU;
   diferenta = adresa ^ c->blocuri[bloc].adresa;
        /* sau exclusiv între adrese */
   diferenta >>= BITI_BLOC;
        /* arunc adresa octetului */
   if (diferenta == 0)
        /* toti bitii identici (rezultat 0) => acelasi bloc */
        return DA;
   else return NU;
}
```

Acest fel de operaţii se implementează foarte repede în hardware.

6.2.2 Cache-ul cu adresare asociativă (fully associative)

Cache-ul cu adresare asociativă se bazează pe un dispozitiv hardware foarte simpatic, care se numeşte *memorie asociativă* (din cauza prezenţei ei îşi capătă cache-ul numele).

O memorie obişnuită oferă două operaţii: (a) dîndu-se o adresă, citeşte conţinutul şi (b) dindu-se o adresă şi o valoare scrie această valoare acolo.

Pe lîngă aceste operaţii o memorie asociativă mai oferă încă una: dîndu-se o valoare, poate spune la care adresă se găseşte ea. O memorie asociativă nu este tehnologic greu de construit, însă este un dispozitiv relativ costisitor.

Un cache asociativ foloseşte o memorie asociativă pentru a memora *adresele externe* ale blocurilor care corespund fiecărui bloc din cache.

Un bloc poate acum ocupa orice poziţie în cache; cînd este căutat memoria asociativă spune unde se află.

Politica de înlocuire va fi însă ceva mai complicată, oricare din schemele înşirate fiind un candidat.

6.2.3 Cache-ul parţial asociativ (set-associative)

Putem să ne imaginăm un cache parţial asociativ ca o colecţie de mai multe cache-uri directe care lucrează în paralel. Fie `k` numărul de astfel de cache-uri directe. (un astfel de cache se numeşte "associative on k ways" — asociativ pe `k` direcţii).

Ideea este simplă: cînd caut o adresă folosesc adresare directă în *toate* cele `k` cache-uri directe simultan. Dacă blocul se găseşte într-unul am rezolvat problema. Daca nu, aleg unul dintre ele pentru înlocuire. Numele este de "parţial asociativ", pentru că plasamentul în cele `k` blocuri posibile este oricare, ca la un cache asociativ.

Să revenim la discuţia privind cache-urile microprocesoarelor.

3) Politica de scriere, **5)** Timpul de viaţă al informaţiei din cache: Dacă mai multe microprocesoare sunt legate la aceeaşi memorie, există riscul ca fiecare să facă modificări în propriul cache, obţinînd astfel rezultate eronate, aşa cum arată şi figura 4. Din cauza asta cache-ul se face adesea "write-through": toate modificările se fac simultan în memorie şi în cache. Cache-urile monitorizează modificările făcute în memorie de celelalte cache-uri şi invalidează copiile datelor pe care le posedă şi care au fost modificate. (Un astfel de cache se numeşte "snooping cache": cache care trage cu urechea, să vadă dacă altcineva nu modifică memoria externă.)

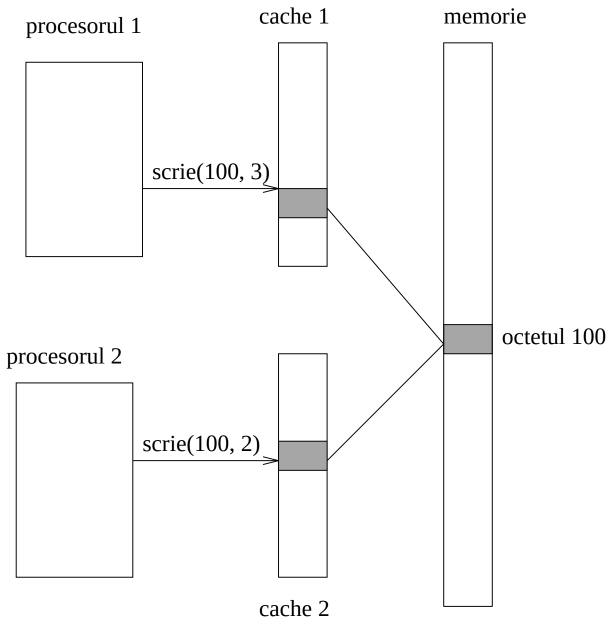

Figure 4: Acces prin două cache-uri

6.3 Cache-ul shell-ului

În Unix (şi MS-DOS) o comandă tastată shell-ului (programului `command.com`) este adesea numele unui fişier. Acest fişier este căutat de shell într-o listă de directoare numită "path" (cărare). De pildă, dacă tastez `ls` (`dir` în DOS), shell-ul caută un fişier cu numele `ls` pe rînd (de la dreapta la stînga) în directoarele indicate de variabila `PATH` (o posibilă valoare este:
`/sbin:/usr/sbin:/bin:/usr/bin:/usr/bin/X11:/usr/local/bin:.`) Cum găseşte un fişier executabil numit `ls`, îl execută. Shell-ul va găsi pe `ls` în directorul `bin` şi va ţine minte acest lucru.

Operaţia de căutare este costisitoare, implicînd multe accese la disc. Din cauza asta, de îndată ce un fişier a fost reperat, shell-urile inteligente ţin minte unde şi nu mai fac a doua oară căutarea. A doua oară cînd voi tasta `ls`, shell-ul se va duce direct în `bin`, fără să mai caute. Acesta este tot un cache: în loc să acceseze memoria lentă (discul) shell-ul se uită în structura de date pe care a construit-o.

(Puteţi verifica acest lucru: dacă mutaţi un fişier executabil din locul unde se afla într-un alt director din `path`, shell-ul nu îl va mai găsi, pentru că nu mai caută lista de directoare o a doua oară. Shell-ul `bash` afişează conţinutului cache-ul intern la comanda `hash`.)

1) Mărimea blocului: Un shell ţine minte de obicei numai fişierele care au fost folosite; "blocul" este perechea nume-loc.

2) Politica de înlocuire: Cel mai puţin frecvent folosite nume pot fi şterse.

3) Politica de scriere, **5)** Timpul de viaţă al informaţiei din cache: După cum am văzut, de obicei shell-ul nu dă atenţie modificărilor; el ţine minte locul unui fişier chiar dacă acest a fost mutat. De obicei shell-ul are o comandă prin care i se spune explicit să uite tot ce a învăţat despre fişiere (la shell-ul `csh` comanda este `rehash`).

4) Metoda de identificare: Pentru a găsi locul unui fişier foloseşte funcţii de hash.

6.4 Serverele de Nume ("Domain Name Servers")

Cache-urile sunt extrem de utile în reţelele de calculatoare, în care memoria lentă este un calculator aflat la distanţă. Un exemplu foarte interesant în acest context (dar nu singurul posibil) este cel al serverelor de nume.

Ca să facem dintr-o poveste lungă una scurtă, fiecare calculator este identificat printr-o adresă numerică (de pildă 141.85.128.1). Din cauză că astfel de adrese sunt greu de ţinut minte, fiecăruia i se atribuie şi o adresă "simbolică" (de pildă `pub.pub.ro`). Asta simplifică problema oamenilor, dar nu pe a calculatoarelor, pentru că atunci cînd cineva indică o adresă simbolică trebuie găsită adresa corespunzătoare numerică. Adresele simbolice sunt aranjate (ca şi numele directoarelor, numai că se scriu de la dreapta la stînga) într-o ierarhie: pentru a afla adresa lui `pub.pub.ro`, trebuie să aflăm adresa (numerică) a calculatorului care ştie totul despre `ro`, să-l întrebăm pe acesta cine este cel care ştie totul despre `pub.ro`, care ne va zice la rîndul lui cine este despre `pub.pub.ro`. Treaba asta cere timp (secunde preţioase).

Din cauza asta, de îndată ce a aflat o pereche de adrese simbolică–numerică, un calculator o păstrează într-un cache local, în ideea că o va mai folosi. Trucul merge pentru că adresele se schimbă foarte rar.

1) Mărimea blocului: Blocul este o pereche (nume simbolic–adresa).

2) Politica de înlocuire: LRU este un candidat bun.

3) Politica de scriere: Această informaţie este practic "read-only": un utilizator nu poate schimba adresa unui calculator; nu se pune problema scrierii. (Cu alte cuvinte memoria lentă este "read-only").

4) Metoda de identificare: Identificarea se face tot cu funcţii de hash.

5) Timpul de viaţă al informaţiei din cache: Pentru a permite adaptarea la schimbări de adrese, fiecare pereche de adrese are un "timp de viaţă", de ordinul săptămînilor. După ce timpul de viaţă expiră, informaţia este ştearsă din cache. În acest fel, dacă adresa unui calculator se schimbă, celelalte vor afla acest lucru la un moment dat.

6.5 Clienţii de WWW

Dacă aţi folosit Netscape (sau Internet explorer sau Mosaic), aţi observat că la apăsarea butonului "back", care vă mută la documentul pe care l-aţi vizionat înaintea celui curent, afişarea se face mult mai repede decît prima oară cînd l-aţi văzut. Explicaţia este simplă: clientul de web (Netscape, etc.) face pe discul local un cache în care ţine minte ultimele documente pe care le-aţi văzut (în directorul `.netscape/cache` de obicei. Pune acolo în fişiere toate paginile recente, figurile aduse şi ce-o mai fi. Asta pentru că (de obicei) e mult mai rapidă citirea de pe discul local decît transferul din reţea.

1) Mărimea blocului: Unitatea de memorare (blocul) este în acest caz fişierul (fie el pagina html, figură gif sau document postscript).

2) Politica de înlocuire: Probabil LRU.

3) Politica de scriere: În principiu clientul de web poate doar citi datele; din cauza asta politica de scriere nu există.

4) Metoda de identificare: Identificarea prezenţei unui document în cache se face după URL-ul lui (Universal Resource Locator). În lume nu pot exista două documente cu acelaşi URL. Căutarea este probabil o simplă căutare într-un vector ne-ordonat, care conţine lista URL-urilor prezente în cache, combinată cu un hash parţial după prima literă (uitaţi-vă la directorul indicat).

5) Timpul de viaţă al informaţiei din cache: Dacă informaţia pe care o citiţi cu clientul de web se schimbă la sursă, clientul nu-şi va da seama, şi vă va arăta tot ce are în cache. Puteţi să-l forţaţi să recitească datele de la sursă apăsînd pe butonul "reload".

6.6 Memoria Virtuală

Memoria virtuală este un mecanism prezent în orice sistem modern de operare. Prin acest mecanism se pot executa programe mai mari decît încap în memorie. Ideea este de a ţine programele pe disc,

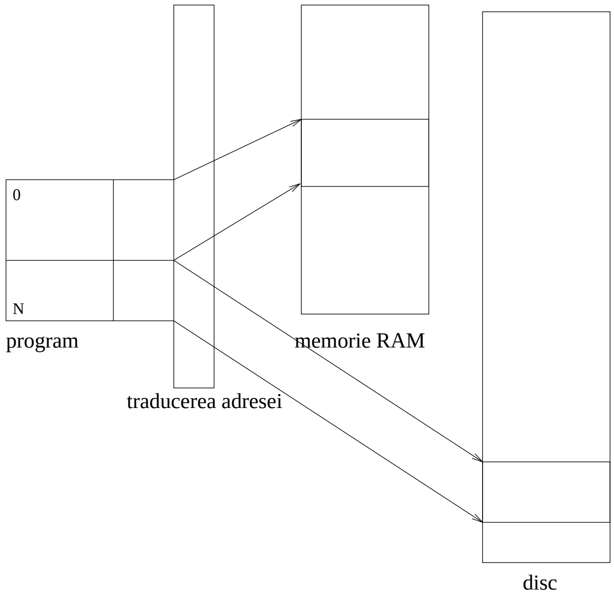

Figure 5: Memoria virtuală

şi de a aduce în memoria RAM a calculatorului numai partea de program care tocmai se execută. Figura 5 ilustrează acest lucru.

Sună familiar, nu? Este tocmai comportarea unui cache, dacă stăm să ne gîndim bine! Tot ce am discutat despre cache-uri se aplică şi în acest caz.

1) Mărimea blocului: Blocurile sunt numite în acest caz pagini, şi au dimensiuni tipice de 1Kb–8Kb (la Pentium 4Kb).

2) Politica de înlocuire: Politica cea mai des folosită este o variantă a politicii "setului de lucru" (WS), care pe engleză are împămîntenit numele de WSCLOCK. Pe scurt politica funcţionează astfel:

- la primul ticăit de ceas toate paginile aflate în memorie se marchează ca fiind ne-atinse;
- procesele sunt lăsate să meargă o vreme; fiecare pagină pe care o folosesc este marcată (cu ajutorul unui hardware special);
- la următorul ticăit de ceas toate paginile încă ne-atinse sunt date afară; celelalte formează "setul de lucru". Paginile care vor fi necesare în viitor vor fi aduse în locurile eliberate de cele date afară. Apoi se reia algoritmul de la început.

3) Politica de scriere: Write-back.

4) Metoda de identificare: Identificarea poziţiei în RAM a unei adrese virtuale se face folosind hardware specializat (Unitatea de Management a Memoriei) care ţine "tabele de pagini". Tabelele de pagini descriu pentru fiecare pagină de adrese virtuale pagina fizică ce-i corespunde. (Tabelele arată cîteodată ca nişte arbori, dar deja am aluneca prea departe cu detaliile).

5) Timpul de viaţă al informaţiei din cache: Paginile pot fi modificate numai cînd se află în memoria fizică (RAM).

6.7 "Translation Lookaside Buffer" (TLB)

Din păcate nu cunosc nici o traducere a termenului, aşa că îl voi folosi cu numele englezesc. TLB este un dispozitiv folosit pentru a implementa memoria virtuală.

Memoria virtuală se bazează pe *traducerea* fiecărei adrese pe care un program o generează într-o altă adresă. (Revedeţi figura 5). Traducerea se face căutînd într-o tabelă mare, care ţine pentru fiecare adresă virtuală adresa din RAM (sau de pe disc) corespunzătoare. Problema este că tabela însăşi trebuie să fie stocată în memorie.

În felul acesta la fiecare acces la memorie se fac două accese: unul în care se caută în tabelă, şi unul în care se iau datele de la adresa indicată de tabelă. Vedeţi şi figura 6.

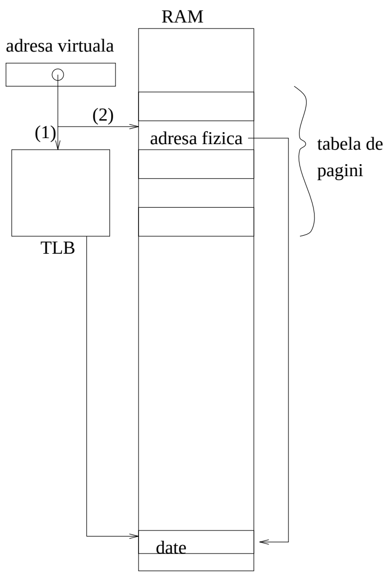

Figure 6: Traducerea adresei

Pentru că nu are sens să faci două accese pentru unul singur, se construieşte un cache cu cele mai folosite adrese virtuale şi traducerile lor. Acest cache se numeşte TLB.

Adresarea se face atunci astfel: întîi se încearcă drumul (1) din figură, căutînd adresa în TLB. Doar dacă nu e acolo căutăm şi în tabela de pagini, pe varianta (2).

1) Mărimea blocului: Un bloc este o pereche de adrese virtuală–fizică.

2) Politica de înlocuire: LRU probabil.

3) Politica de scriere, **5)** Timpul de viaţă al informaţiei din cache: Cînd translatarea virtual-fizic se schimbă (de pildă cînd o pagină este înlocuită), informaţia din TLB nu mai este corectă. Soluţia este de obicei ştergerea întregului conţinut al TLB.

4) Metoda de identificare: folosind memorii asociative.