Appunti di Calcolatori Elettronici
Concetti generali sulla memoria cache

Introduzione

Storicamente, le **CPU** sono state sempre più veloci delle **memorie**: quando le memorie sono migliorate, lo hanno fatto anche le CPU, mantenendo la distanza. Ciò comporta, in concreto, una conseguenza importante: dopo che la CPU ha emesso una *richiesta di memoria*, deve rimanere inattiva per un tempo rilevante, aspettando la risposta dalla memoria. Tipicamente, la richiesta della CPU avviene in un ciclo di clock e poi devono passare due o tre cicli successivi prima di ottenere i dati. Questo rallenta troppo l'esecuzione complessiva, per cui è il caso di porvi rimedio.

E' importante capire che il "problema" è di natura economica e non tecnologia: infatti, gli ingegneri sanno bene come costruire memorie veloci quanto le CPU, ma sanno anche che esse sarebbero molto costose, tanto che dotare un calcolatore di anche un solo megabyte di tali memorie sarebbe improponibile. Per questo motivo, si è scelto di avere solo una piccola quantità di memoria veloce (eventualmente espandibile a cura dell'utente) ed una grande quantità di memoria lenta.

Esistono varie tecniche per combinare una piccola quantità di memoria veloce con una grande quantità di memoria lenta, in modo da ottenere, allo stesso tempo e con un costo moderato, (quasi) la velocità di quella veloce e la capacità di quella grande. La memoria piccola e veloce è detta **cache** (che deriva dal francese "cacher", ossia "nascondere") ed è sotto il controllo del *microprogramma*. Vogliamo allora descrivere come si usano e come funzionano le *memorie cache*.

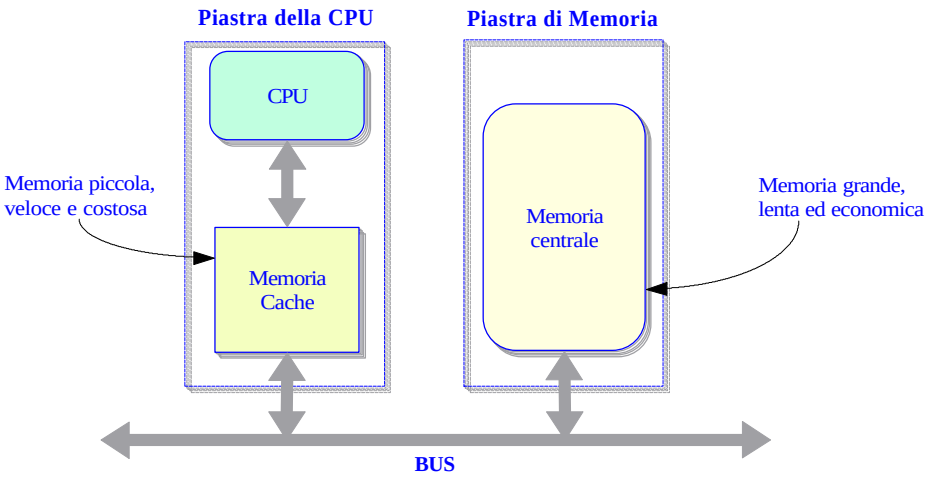
Il principio di localizzazione

Ormai da diverso tempo è stato accertato che i **programmi** non accedono al loro codice in modo completamente casuale: se capita un *riferimento di memoria* ad un dato indirizzo A, è molto probabile che il riferimento successivo sia nelle vicinanze di A. Ne è una prova lo stesso programma:

- ad eccezione dei *salti* e delle *chiamate di procedura*, le istruzioni sono prelevate da *locazioni* consecutive in memoria;
- inoltre, la maggior parte del tempo di esecuzione di un programma viene usata per i *cicli*, in cui quindi vengono eseguite un numero limitato di istruzioni in maniera ripetitiva.

Questa osservazione per cui i riferimenti alla memoria, fatti in un intervallo breve, tendono ad usare una piccola frazione della memoria totale è noto come **principio di localizzazione** ed è alla base di tutti i *sistemi di cache*. L'idea generale è la seguente: quando viene fatto un riferimento ad una parola, questa viene trasferita dalla memoria grande e lenta nella cache, in modo da essere accessibile velocemente per le eventuali e probabili successive utilizzazioni.

Una architettura molto usata della CPU, della cache e della memoria centrale è illustrata nella figura seguente:



Architettura tipica di un calcolatore con memoria cache: quest'ultima è generalmente localizzata sulla piastra della CPU

Per renderci conto dell'opportunità di usare una cache, possiamo fare il seguente semplicissimo discorso: supponiamo che una parola venga letta o scritta *k* volte in un breve intervallo, durante l'esecuzione di un dato programma; se tale parola si trova inizialmente nella memoria centrale e, alla prima richiesta, viene portata nella cache, il calcolatore avrà bisogno di un solo riferimento per la memoria lenta (il primo) e k-1 riferimenti per la memoria veloce. Quanto maggiore è *k* (ossia, sostanzialmente, quanto più risulta verificato il principio di localizzazione), tanto migliori saranno le prestazioni globali, proprio perché tanto maggiore è il numero di volte in cui viene coinvolta la memoria veloce al posto della memoria lenta.

Possiamo formalizzare ulteriormente questo calcolo, introducendo due tempi: **tempo di accesso alla cache** (che indichiamo con **C**) e **tempo di accesso alla memoria centrale** (che indichiamo con **M**). Sulla base di questi tempi, possiamo calcolare il cosiddetto **rapporto di successo** (simbolo **H**), ossia la frazione di tutti i riferimenti che si possono soddisfare con la cache: nell'esempio che abbiamo fatto poco fa, risulta evidentemente

$$H = \frac{K-1}{K}$$

Il duale di questa quantità è il cosiddetto **rapporto di fallimento**, pari evidentemente a

$$1-H = 1-\frac{K-1}{K} = \frac{1}{K}$$

Possiamo inoltre calcolare il **tempo medio di accesso** alla generica parola, nel modo seguente:

$$\text{tempo medio di accesso} = C + (1-H)\times M = C + \frac{M}{K}$$

Questa formula esprime chiaramente il fatto che il tempo medio di accesso è la somma del tempo di accesso alla cache e del tempo di accesso alla memoria, pesato però quest'ultimo per (1-H) o, ciò che è lo stesso, per 1/K. Di conseguenza, quanto più H tende ad 1, tanto più i riferimenti a memoria possono essere soddisfatti dalla cache e quindi il tempo totale di accesso si avvicina a quello della cache. Al contrario, se $H \to 0$, è quasi sempre necessario un riferimento alla memoria centrale, per cui il tempo di accesso si avvicina adesso a C+M: il tempo iniziale C serve a controllare che la parola richiesta sia in cache e, una volta verificato che non è così, serve un tempo M per accedere alla memoria centrale.

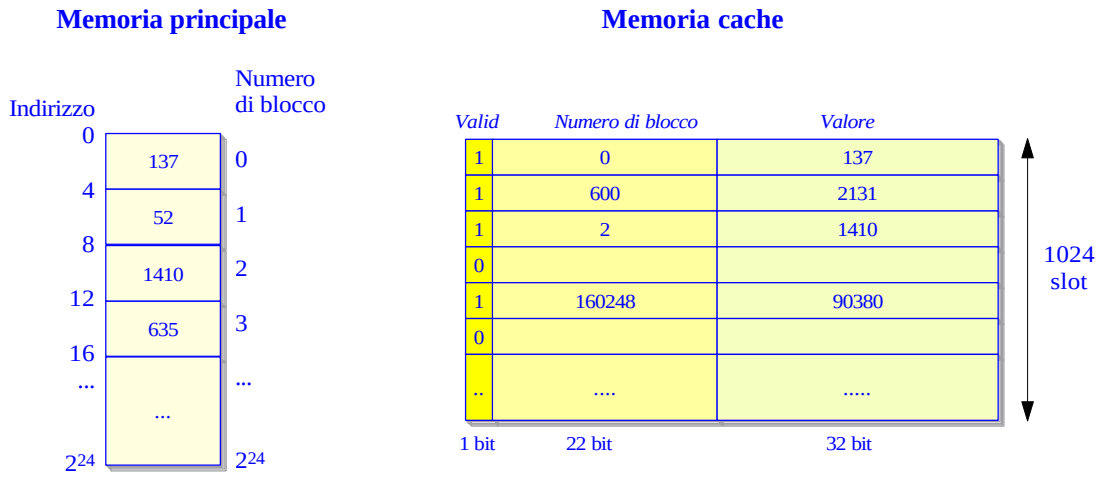
In alcuni sistemi, per velocizzare il funzionamento, si possono cominciare in parallelo l’accesso alla memoria e la ricerca nella cache: in tal modo, se la ricerca nella cache è infruttuosa, il ciclo di memoria è stato già attivato. Tuttavia, una strategia di questo tipo richiede che si possa fermare la ricerca in memoria quando si ha successo (in inglese **hit**) nella cache, il che rende l’hardware più complicato. In ogni caso, l’algoritmo di base per cercare nella cache e cominciare (o fermare) l’accesso alla memoria centrale, a seconda del risultato della ricerca nella cache, è gestito dal microprogramma.

Organizzazioni delle memorie cache

Si usano due diverse organizzazioni fondamentali della cache, oltre ad una terza che è una forma di “ibrido” delle prime due.

Supponiamo, per tutti i tre tipi, che la memoria principale sia di 2^m byte (dove m è quindi la lunghezza degli indirizzi di memoria) e che essa sia divisa (solo concettualmente) in **blocchi** consecutivi, ciascuno di b byte, per un totale chiaramente di $2^m/b$ blocchi. Così facendo, ogni blocco ha un indirizzo che è un multiplo di b (dove b è generalmente una potenza di 2).

Il primo tipo di cache è quella cosiddetta **completamente associativa**, composta da un certo numero di **slot** (detti anche *linee*), ciascuno dei quali contiene tre elementi: un blocco, il numero identificativo di tale blocco ed un bit (chiamato *Valid*) che indica se lo slot è attualmente in uso oppure no. La figura seguente illustra un esempio di questa organizzazione:



Esempio di schema di cache completamente associativa con 1024 slot (affiancata ad una memoria centrale con blocchi da 4 byte)

A sinistra è riportato un "pezzo" di memoria principale, suddivisa in blocchi da 4 byte ciascuno; a destra è invece riportato un "pezzo" di una memoria cache completamente associativa, in cui ogni slot ha la struttura descritta prima. Si considera, in particolare, una cache con 1024 slot ed una memoria di 2^{24} byte divisi in 2^{22} blocchi da 4 byte ciascuno. Nella cache completamente associativa, l'ordine degli elementi è casuale.

Quando il calcolatore viene inizializzato, tutti i bit *Valid* sono posti a 0, in modo da indicare che, al momento, non ci sono elementi della cache "validi", cioè utilizzati. Supponiamo adesso che la prima istruzione del programma in esecuzione si riferisca alla parola di 32 bit situata all'indirizzo 0:

- il microprogramma controlla tutti gli elementi della cache, cercandone uno valido contenente il blocco numero 0;
- non trovando il suddetto blocco (la cache è vuota), emette una richiesta sul bus, per prelevare la parola 0 dalla memoria;
- sistema quindi tale parola nel blocco 0 e rende valido quest'ultimo.

A questo punto, se la parola dovesse risultare nuovamente necessaria, potrà essere prelevata direttamente dalla cache, eliminando la necessità di una operazione sul bus.

Chiaramente, con il procedere dell'esecuzione, un numero sempre crescente di elementi della cache risulteranno contrassegnati come validi. Nel caso in cui il programma usi meno di 1024 parole per il programma e per i dati, si verifica la situazione ideale, dato che tutto il programma ed i suoi dati appariranno alla fine nella cache, per cui esso sarà eseguito ad alta velocità, senza bisogno di accessi alla memoria centrale attraverso il bus. E' chiaro, però, che questa è una situazione poco realistica; è più probabile, invece, che il programma ed i suoi dati richiedano più di 1024 parole: in questo caso, si arriverà ad un certo punto in cui tutta la cache risulterà piena e bisognerà eliminare vecchi elementi per far spazio a quelli nuovi. La decisione di quale elemento eliminare è di importanza critica e deve tra l'altro essere presa molto velocemente (pochi nanosecondi): alcune macchine prendono uno slot a caso, mentre invece generalmente sono usati algoritmi più sofisticati.

La caratteristica importante della cache completamente associativa è che ogni slot contiene il numero identificativo del blocco contenuto oltre al blocco stesso.

Quando si presenta un indirizzo di memoria, il microprogramma deve compiere quindi due operazioni:

- calcolare il numero del blocco corrispondente all’indirizzo specificato;
- controllare l’elemento della cache che ha quel numero di blocco, se presente.

Mentre la prima operazione è molto semplice, la seconda non lo è affatto. Per evitare una ricerca sequenziale sulla cache, quest’ultima possiede un hardware speciale, che confronta ogni elemento simultaneamente con il numero di blocco calcolato. Questo evita di usare un ciclo del microprogramma, ma rende anche più costosa la cache.

Per ridurre tale costo, fu allora inventato un diverso tipo di cache, detta **a mappa diretta** (o anche *ad indirizzamento diretto*): in questo caso, si evita la fase di ricerca del blocco, mettendo ogni blocco in uno slot il cui numero può essere facilmente calcolato dal numero del blocco. In altre parole, dato un blocco, esiste un unico slot della cache in cui può essere sistemato e quindi il controllo sulla presenza o meno del blocco in cache andrà effettuato su quell’unico slot. Il numero di tale slot può essere ad esempio calcolato nel modo seguente: si fa la divisione tra il numero del blocco ed il numero degli slot e si considera il resto di tale divisione. Brevemente, si scrive

(numero del blocco) *modulo* (numero di slot)

Ad esempio, con blocchi di 4 byte (cioè una parola) e con 1024 slot nella memoria, il numero dello slot per la parola che si trova all’indirizzo A è

$$\frac{A}{4} \text{ modulo } 1024$$

dove ovviamente A/4 è il numero del blocco in cui si trova la parola il cui indirizzo è A.

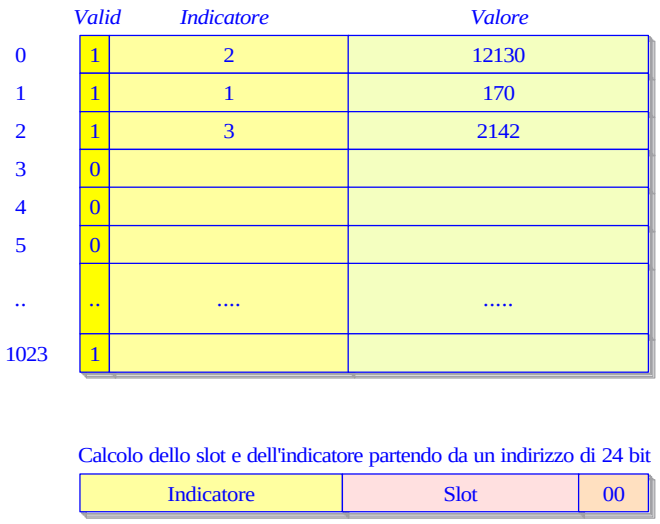
Nel nostro esempio, lo slot 0 della cache è destinato alle parole che hanno indirizzo 0, 4096, 8192, 12288 e così via; lo slot 1 è invece destinato alle parole che hanno indirizzo 4, 4100, 8196, 12292 e così via.

La cache a mappa diretta elimina dunque il problema della ricerca di un blocco, ma crea un nuovo problema: bisogna infatti capire come indicare quale, delle diverse parole che corrispondono ad un dato slot, lo stia effettivamente occupando. Ad esempio, poco fa abbiamo detto che lo slot 0 può accogliere le parole ad indirizzo 0, 4096, 8192, 12288 e così via. Il modo per indicare quale di queste sia effettivamente presente nello slot è di porre una parte dell'indirizzo di tali parole nella cache, introducendo un apposito campo **Tag** (*indicatore*): tale campo viene così a contenere quella parte di indirizzo che non può essere calcolata partendo dal numero dello slot.

Facciamo un esempio:

- consideriamo una istruzione all'indirizzo 8192 e supponiamo che essa sposti una parola dall'indirizzo 4100 all'indirizzo 12296;
- il numero del blocco corrispondente a 8192 si ottiene dividendo per 4 (ossia per la dimensione dei blocchi nel nostro esempio): si ottiene 2048;
- dal numero del blocco possiamo poi risalire al numero dello slot che lo può accogliere: facendo 2048 modulo 1024, si ottiene 0. Lo stesso risultato si sarebbe ottenuto usando i 10 bit meno significativi del numero 2048, ossia del numero del blocco. Essendo tale numero di 22 bit, i restanti 12 bit (che in questo caso contengono il valore 2) andranno a costituire il tag.

Nella figura seguente è riportata la cache dopo che tutti e tre gli indirizzi sono stati elaborati nel modo appena descritto:



Esempio di cache a mappa diretta, con 1024 slot di 4 byte ciascuno. Nella parte inferiore è mostrato il procedimento di calcolo dello slot e dell'indicatore partendo da un indirizzo di 24 bit

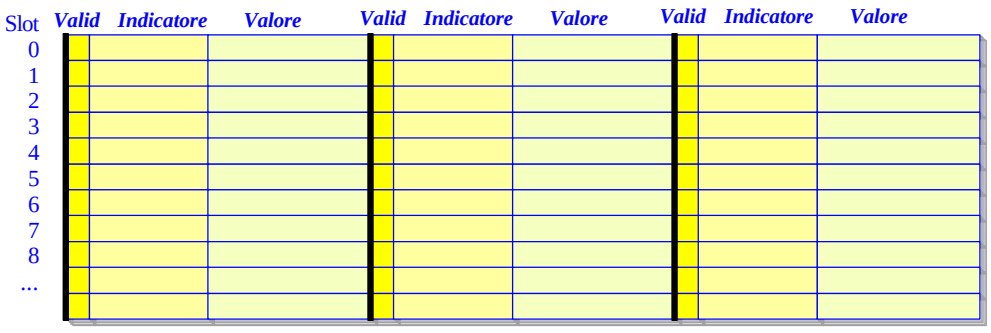
Questa figura mostra anche (nella parte inferiore) come si divide l’indirizzo di memoria (da 24 bit, avendo supposto una memoria da 2^{24} byte):

- i due bit meno significativi sono sempre a 0, dato che la cache lavora sempre con blocchi interi e questi sono multipli della dimensione del blocco, che nel nostro esempio è di 4 byte;
- segue il numero di slot da 10 bit;
- infine, il numero dell’indicatore (12 bit) preso direttamente dalla cache.

Una soluzione di questo tipo è anche facilmente implementabile, dato che si può agevolmente costruire un hardware che estrae direttamente il numero dello slot e dell’indicatore da qualunque indirizzo di memoria.

Il fatto che diversi blocchi di memoria possono far riferimento allo stesso slot della cache può comunque provocare dei problemi. Ad esempio, supponiamo che una istruzione di spostamento sposti una parola dall’indirizzo 4100 all’indirizzo 12292 (al posto di 12296 come supposto prima); entrambi questi indirizzi si riferiscono allo slot 1: infatti, 12292/4 fornisce il numero di blocco 3073 e questo numero, diviso per 1024 slot, fornisce un resto pari a 1. In questa situazione, a seconda della microprogrammazione, l’ultimo calcolato finirà nella cache, mentre l’altro verrà eliminato; non si tratta ovviamente di un evento disastroso, ma esso contribuisce comunque a rallentare le prestazioni, tanto più quante più sono le parole in uso che si riferiscono allo stesso slot.

Per ovviare a questo problema, si può adottare una estensione della cache a mappa diretta, tesa a poter memorizzare più blocchi in ciascuno slot. Si ottiene così la *cache associativa ad insiemi*, detta anche **cache set-associativa ad N vie**, dove N sono i blocchi memorizzabili in ciascuno slot:



Esempio di cache set associativa a 3 vie (3 blocchi per ogni slot)

In effetti, sia la cache completamente associativa sia quella ad indirizzamento diretto sono casi particolari della cache set-associativa:

- se usiamo un unico slot, tutti gli elementi della cache si trovano nello stesso slot e dobbiamo perciò distinguerli con indicatori, dato che fanno riferimento allo stesso indirizzo. Otteniamo perciò nuovamente una cache completamente associativa;
- se invece prendiamo N=1 (1 blocco per ogni slot), allora abbiamo nuovamente la cache ad indirizzamento diretto.

Dal punto di vista di vantaggi e vantaggi dei vari tipi di cache, possiamo citare i seguenti:

- le cache ad indirizzamento diretto sono semplici ed economiche da costruire e presentano inoltre un tempo di accesso minore, dato che lo slot corrispondente si può trovare semplicemente indicizzando la cache (usando una porzione dell'indirizzo come indice);
- al contrario, la cache associativa ha un numero maggiore di successi per un dato numero di slot, in quanto non si verificano mai *conflitti*: non può infatti succedere che k parole importanti non si possano mettere nella cache contemporaneamente perché hanno la sfortuna di far riferimento allo stesso slot.

Il progettista deve decidere non solo il numero degli slot, ma anche la dimensione dei blocchi. Nei nostri esempi, abbiamo sempre usato blocchi da 4 byte, ma sono usate anche altre dimensioni. Il vantaggio di usare una grande dimensione per i blocchi è che risulta meno gravoso prelevare un blocco di 8 parole rispetto ad 8 blocchi ciascuno di 1 parola, specialmente se il bus permette il trasferimento di blocchi. C'è però anche lo svantaggio che non tutte le parole di un dato blocco possono essere necessarie, nel qual caso parte del prelevamento è sprecato.

Gestione delle scritture in una cache

Un altro importante problema, nella progettazione di una cache, è quello della gestione delle **scritture**. Si usano solitamente due strategie:

- nella prima, detta **scrittura in avanti**, quando si scrive una parola nella cache, la si registra immediatamente anche nella memoria. In tal modo, si assicura che gli elementi della cache siano sempre uguali ai corrispondenti elementi della memoria;
- nella seconda, detta invece **copia all’indietro**, la memoria non viene aggiornata tutte le volte che la cache viene alterata, ma solo quando bisogna eliminare un elemento dalla cache per far posto ad un altro. In questo caso, è chiaramente necessario un bit per ogni elemento della cache, che indichi se l’elemento è stato modificato oppure no durante la sua permanenza in cache: solo in caso affermativo, prima di eliminarlo esso andrà registrato in memoria.

Ci sono una serie di conseguenze legate ad una ed all’altra scelta:

- la scrittura in avanti provoca evidentemente un maggiore traffico nel bus rispetto alla copia all’indietro;
- viceversa, se una CPU comincia un trasferimento di I/O dalla memoria al disco e la memoria non è “aggiornata”, allora dati non corretti saranno scritti sul disco. E’ ovvio che una simile evenienza viene comunque sempre evitata, al prezzo però di una maggiore complessità del sistema;
- se il rapporto tra letture e scritture risulta molto alto, può essere più conveniente l’uso della scrittura in avanti, tollerando il traffico sul bus, che comunque subirebbe dei picchi saltuari (in occasione appunto delle scritture). Se invece ci sono molte scritture, allora il traffico potrebbe essere eccessivo e sarebbe perciò preferibile la scrittura all’indietro; non solo, ma sarebbe anche opportuno che il microprogramma “pulisca” l’intera cache (ossia la copi interamente nella memoria) prima di una eventuale operazione di I/O.

Autore: **Sandro Petrizzelli**
e-mail: sandry@iol.it
sito personale: http://users.iol.it/sandry