Misurare le performance delle caches

Il processore viene messo in stall (per diversi motivi) principalmente per gli **accessi in memoria**; infatti, se non si riesce a trovare un dato all'interno delle cache di primo livello (istruzione da eseguire, blocco di memoria) sono costretto ad andare alla memoria esterna (RAM), durante il quale il processore va messo in stall, perchè l'accesso in memoria è estremamente lento.

I cicli di clock persi in **stall** sono dati da:
`[ (numero di accessi in memoria) / (programma) ] x miss rate x penalità di miss`

Esempio

Supponiamo di avere una cache istruzioni con **miss rate del 2%** ed una cache dati con **miss rate del 4%**.
La penalità di miss è pari a **100 cicli di clock del processore**.
Il CPI base (ideale) è pari a 2, inoltre il 36% delle operazioni sono di tipo **load/store**.

- Numero di cicli persi per 100 cicli sulla cache istruzioni= `0.02 x 100 = 2` (prendiamo in esame **tutte le istruzioni**)
- Numero di cicli persi per 100 cicli sulla cache dati = `0.04 x 100 x 0,36 = 1.44` (prendiamo in esame il **36% delle istruzioni (accesso ai dati)**)

Il CPI effettivo diventa `CPI = 2 + 2 + 1.44 = 5.44`; capiamo quindi che in realtà la CPU è più lenta, per via dei search miss.

Tempo di accesso medio

Bisogna tenere conto del **tempo di hit**, ovvero il tempo per prelevare dalla cache (anche quando è presente il blocco che mi serve).

Il tempo medio corrisponde a: `AMAT = Hit time + Miss rate x Miss Penalty`

Esempio

Abbiamo un clock pari a **1ns**, l'hit time è di **1 ciclo**, il miss penalty è di **20 cicli** ed il miss rate è del **5%**.

`AMAT = 1 + 0.05 x 20 = 2ns` quindi, ci servono mediamente 2 cicli di clock per istruzione.

Performance summary

Più aumenta la prestazione della CPU (pipelining abbondante, parallelismo di istruzioni, clock altissimo, ecc.) più le CPU diventano veloci. Si confrontano però con un **tempo di accesso ai dati** (RAM) che negli anni è sceso di poco (sempre lento). Questo significa che più si va avanti, più la miss penalty diventa significativa.

Mondo reale

Inizialmente il sistema di cache, fisicamente, era esterno al processore. Questo significa che sulla MOBO erano presenti dei chip di RAM statica che facevano da cache. A seconda della MOBO il sistema funzionava più o meno bene.

Attualmente le cache sono **integrate all'interno della CPU**.

🏁 1:10

Direct Mapped Cache - fregatura

Potrebbe succedere, che il programmatore (stupidamente) utilizza delle variabili che per caso sono mappate sullo stesso indirizzo della cache, ed in questo modo si ottiene che un indirizzo e laltro esce, un grande numero di volte, perchè entrambi servono al programma per funzionare. Quindi, la cache può anche essere vuota, ma le variabili utilizzate sono mappate sullo stesso blocco, che vengono scambiati (facendo una lunga sequenza di search miss) continuamente (perdendo molto tempo).

La disposizione intelligente sarebbe quella di poter disporre con libertà gli indirizzi su tutta la cache, senza dover sovrascrivere inutilmente dei dati che potrebbero ancora servirmi.

Caches associative - full associative

Rimuovere questi vincoli, ci permette **in teoria** di avere delle prestazioni migliori. Il sistema ideale è avere una cache che sia **completamente associativa**: a differenza del cache direct mapped, dove ogni blocco va in una posizione prefissata, sulla base dei bit di mezzo, in questo tipo di cache ogni blocco può essere posto in modo arbitrario.

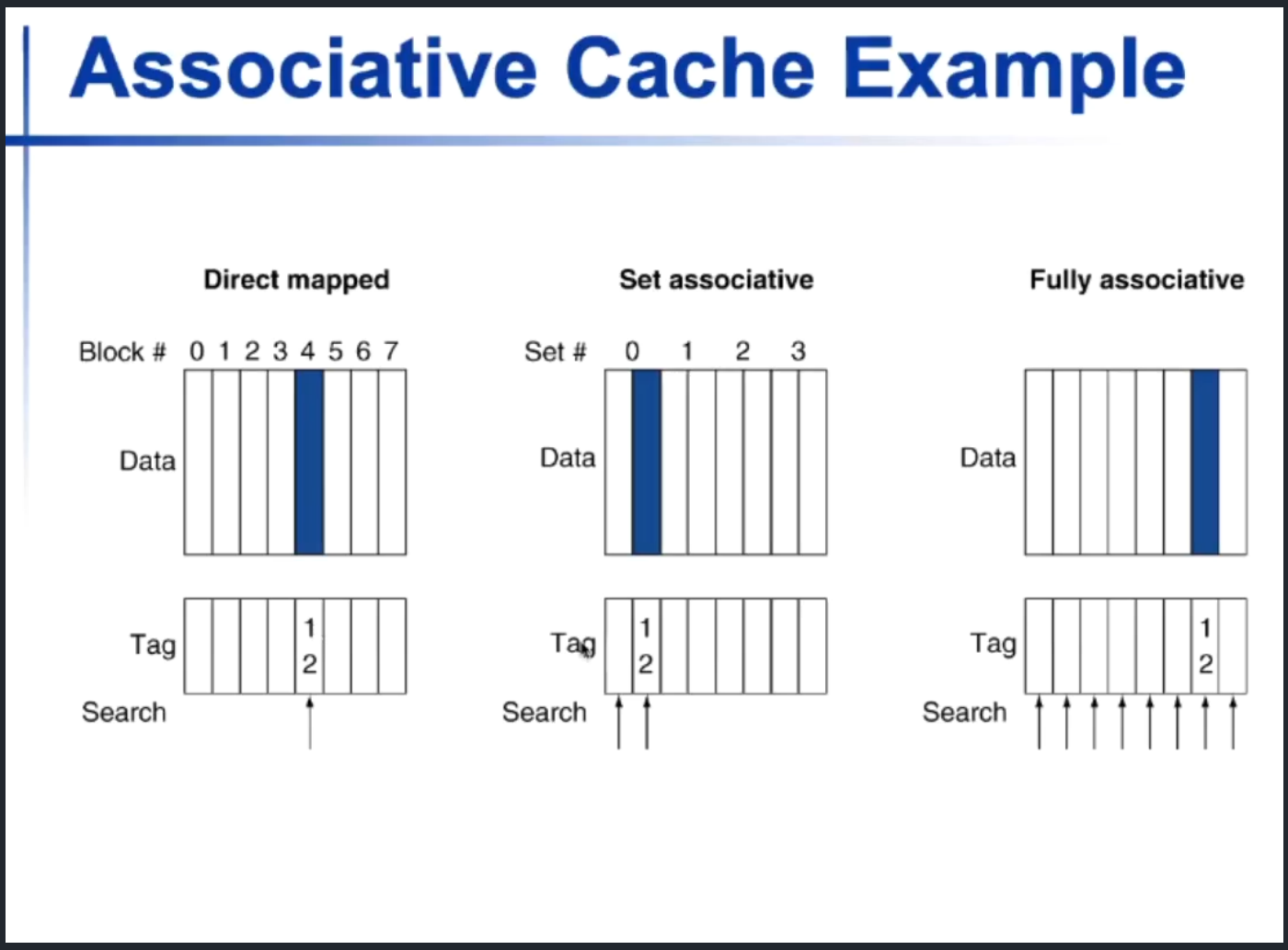
Il problema, però, è che a questo punto non so più dove andare a cercare i vari tag! Dovremmo quindi andare a cercare (Sequenzialmente? non sia mai) tra tutti i tag presenti nella cache. Per avere una ricerca fattibile, **in un colpo solo**, dobbiamo avere **un comparatore per ogni tag**, in modo che, se presente, l'unico comparator ritorni true (sempre hardware).

###Soluzione fattibile n-associative

Si crea una cache di tipo **n-way set associative**, ovvero una cache associative ad n vie. Questo tipo di cache, che vengono utilizzate tuttora nei sistemi moderni, sono composte di vari insiemi dove vengono posti i tag. Abbiamo un comparatore per ogni set, in modo tale che possiamo confrontare ad un certo numero di vie, in modo da avere delle prestazioni simili a quelle fully associative ma riducendo il numero dei comparatori (pe risparmià sordi).

Come calcolare il set dove mettere un dato?

La tecnica è sempre la stessa, ovvero usando i bit meno significativi. Usiamo sempre l'operazione modulo, ma questa volta non sugli entry della cache, ma sul **numero di sets**.



Dall'immagine notiamo che con la **direct mapped** andiamo direttamente nella locazione della cache che ci interessa; questo però non ci permette di mappare più valori sulla stessa posizione.

Con la **fully associative** dobbiamo controllare (in hardware, molto costoso) su **tutti i blocchi della cache**, mentre con la **set associative** (soluzione adottata) cerchiamo solo **nel set**.

Politica di rimpiazzo

- **Direct Mapped** non c'è bisogno di una politica perchè il valore mappato è direttamente quello da rimpiazzare
- **Sett associative** se c'è un entry non valido c'è una posizione libera che si può usare. Se invece non c'è un entry valido, dobbiamo scegliere tra quelli contenuti nel set, quindi c'è bisogno di una **politica di rimpiazzo** .

Strategie usate

Visto che l'obbiettivo è quello di **ridurre al minimo i cache miss**, dovremmo essere in grado di sapere **in anticipo** quello che non mi servirà per più tempo. Il problema, come al solito, è che non sappiamo a priori cosa ci servirà a cosa no.

Solitamente si tenta di utilizzare una strategia chiamata **LRU**, ovvero **least recently used**; visto che non so nel futuro cosa mi servirà per più tempo, facciamo l'ipotesi contraria, ovvero di avere gli entry **meno usati recentemente**, quindi se una entry non è usata da molto tempo, molto probabilmente continuerà a non servirmi nell'immediato futuro.

Risultato inaspettato

Facendo degli opportuni studi, è emerso che scegliere l'entry da sostituire in maniera **random**, si ottengono grossomono le stesse prestazioni. Quindi, a questo punto, conviene risparmiare hardware e sceglierlo a caso.

---

fine lezione 17