MEMORIA

GERARCHIE DI MEMORIA

La memoria è divisa in più livelli che sono dal più vicino al più distante al CPU, per tempi di accesso:
memoria cache, SRAM, capienza (MB), più veloce (decimi di ns), costosa
memoria principale, DRAM, capienza 2-4 GB, più lenta (decine di ns), economica

Gerarchizzare la memoria serve perché i dati già usati della memoria principale vengono salvati in cache per essere utilizzati in modo più veloce.
Il principio è sfruttare la dimensione della memoria principale e la velocità della cache.
Questo passaggio è necessario visto il trend attuale, conosciuto come “collo di bottiglia del sistema Von Neumann”, in cui, per limiti costruttivi, le memorie crescono rapidamente di dimensione ma non altrettanto in velocità, in particolare non mantengono il rapporto con la crescita di velocità dei processori.

Principio di località

L'efficienza della gerarchia di memoria è garantita dal principio di località
temporale: un dato riferito da un programma viene con molta probabilità riferito in un breve intervallo di tempo.
spaziale: dati vicini vengono generalmente riferiti entro un breve intervallo di tempo.
Quindi se per dati consideriamo una word, questa viene ritrovata in cache o perché è stata riferita da poco o perché è vicina ad un'altra appena riferita in cache e quindi nello stesso blocco di cache.

Acceso alla cache

Hit Il blocco cercato è stato trovato
Miss Il blocco non è stato trovato, occorre portarlo in cache
Hit rate % di hit rispetto agli accessi totali
Miss rate % di miss rispetto agli accessi totali
Hit time tempo speso per trovare un dato che è in cache
Miss penalty tempo speso per ricopiare il dato dalla memoria principale in cache, dopo un miss

MAPPING

Una locazione di memoria in cache è associata ad un'altra in memoria principale in base ad un indirizzo che si ottiene considerando la dimensione della cache ed il numero di blocchi in cache

block size (Byte)
#cache blocks

l'Indirizzo si compone, a partire dai bit meno significativi di

byte offset = log_2 block size

I bit esclusi l'offset sono detti Block Address e sono

index = log_2 #cache blocks
tag = indirizzo – index – byte offset (bit più significativi restanti!)
valid, bit di validità 0 o 1 (segnala se il blocco è in uso o meno)

Quando si accede ad una locazione di memoria principale verifica se all'**index** corrispondente in cache c'è bit valid = 1 (in caso contrario è miss) e se c'è memorizzato il corrispettivo **tag** (altrimenti è miss per conflitto), usa del dato memorizzato il particolare byte **byte offset** (hit).

Cache di 1 Byte, detta “indirizzamento al Byte” proprio perché il blocco è 1B.
Non c'è il byte offset, infatti $\log_2 2^0 B = 0$.
Cache > 1 B, detta “indirizzamento al blocco”.
Il dato viene cercato all'interno di un blocco, in cui occupa il byte di posizione pari all'offset.

Passare dall'indirizzo di un byte alla posizione in cache

Indirizzo al blocco = indirizzo del byte / block size
l'offset è il resto della divisione, cioè
offset = indirizzo del byte % block size
index = indirizzo al blocco % #cache blocks

RICERCA IN CACHE

Dato un indirizzo di memoria questo può restituire un hit o un miss, in particolare darà
miss
se è la prima volta che si accede a quell'indirizzo
index con valid = 0

miss per conflitto
se l'indirizzo di memoria coincide con un altro già occupato in cache:
index con valid = 1, tag cercato != tag trovato

hit
se non è la prima volta che si accede al dato e questo è in cache
index con valid = 1, tag cercato = tag trovato

NB: l'offset serve solo per capire a quale byte del dato in memoria accedere, non per determinare un miss o un hit.
In seguito verranno definiti con più precisione i casi di miss (regola delle 3 C).

DIMENSIONI CACHE

Avendo block size e #cache blocks si può calcolare la quantità di dati memorizzabili in cache:

Size data cache = block size * #cache blocks

La dimensione complessiva della cache è più grande perché include il bit di validità e i bit di tag:

Size cache = Size data cache + (valid + tag) * #cache blocks

Considerazione sul Block Size
All'aumentare della dimensione del blocco aumenta la località spaziale, cioè la probabilità di trovare un byte adiacente all'interno dello stesso blocco, quindi riduciamo il miss rate. Di contro aumenta il miss penalty, ma non in modo proporzionale al miss rate, perché maggiore è il blocco, più tempo ci si impiega a trasferirlo in caso di miss. Aumentare la dimensione del blocco conviene ma oltre un certo limite la tendenza a ridurre il miss rate si inverte. Infatti se aumentiamo i blocchi li riduciamo anche di numero ottenendo che troppi indirizzi siano mappati con l'index di uno stesso blocco, che verrà sostituito con un miss per conflitto prima di essere sfruttato del tutto.

ASSOCIATIVITÀ

La cache vista finora ha un'associatività di tipo
Diretta = blocco memoria → preciso blocco cache

L'associatività diretta è vantaggiosa nel caso di memorie di grandi dimensioni, perché essendoci molto spazio il miss rate è basso, l'hit time invece è veloce perché si conosce l'indirizzo a cui cercare il dato richiesto.

La memoria può essere anche
(completamente) associativa = blocco memoria → primo blocco libero in cache

In questo modo si riduce il miss rate, perché non ci possono essere conflitti, ma si aumenta l'hit time perché un dato va cercato in tutta la cache prima di essere trovato.

Il tipo di associatività che si usa per la cache è un compromesso tra i primi due, ovvero
Associativa su insiemi = blocco memoria → preciso gruppo, primo blocco libero
All'interno di un gruppo (set) identificato in maniera diretta, scrivi il blocco in modo associativo su più vie.

MAPPING per cache associative a più vie

index = log_2 #set
con
set = #cache blocks / #vie

MEMORIA E PRESTAZIONI

Finora abbiamo calcolato il tempo di esecuzione senza tener conto del tempo speso per accedere alla memoria. Il tempo di esecuzione ideale è calcolato come cicli di esecuzione per T, periodo di clock. Per calcolare il tempo di esecuzione reale si deve tener conto che i cicli per istruzione CPI aumentano di numero quando si ottiene un miss in cache, in percentuale l'aumento **miss rate**, riguarda:

Instruction miss rate	miss che si verificano quando si esegue il fetch di un'istruzione generica
data miss rate	miss che si verificano quando si accede alla memoria per prelevare un dato, istruzioni lw, sw.

In entrambi questi casi si effettua un numero di cicli in più detto **miss penalty**

*CPI miss = (% instruction miss + % data miss * % instr. lw,sw) * miss penalty*

*Texe reale = IC * CPI reale * T = IC * (CPI ideale + CPI miss) * T*

Rispetto al tempo si può anche scrivere

*Texe di stallo = cicli di stallo * T = IC * CPI miss * T*
Texe reale = Texe ideale + Texe di stallo

L'accesso alla memoria influenza solo CPI, non IC e T quindi

Speed up = Texe reale / Texe ideale = CPI reale / CPI ideale

Miss penalty e latenza
Il numero di cicli di stallo da subire in caso di miss è legato al tempo di accesso, latenza ad una memoria. Considerando che un ciclo impiega un tempo T, periodo di clock, abbiamo che

Miss penalty = Tempo latenza / T

Miss penalty e Bandwidth
Il miss penalty è legato alla larghezza della memoria, che determina quante informazioni si possono trasferire contemporaneamente, data transfer e dalla quantità di informazioni a cui accede contemporaneamente tramite un bus clockato, data access.

*Miss penalty = address + data transfer * Byte / larghezza mem + data access * Byte / larghezza bus*
Bandwidth = Byte trasferiti / miss penalty per trasferirli

Esempio: una memoria impiega 1 ciclo di address, 1 di data transfer, 15 di data access.
Memoria di una word con bus che accede ad una word per volta (4B)
Byte da trasferire = 16 B
Miss penalty = 1 + 1 * 16 / 4 + 15 * 16 / 4 = 65 cicli
Bandwidth = 16 B / 65 cicli = 0.25 B / ciclo

Cache a più livelli
Per subire un miss penalty minore rispetto a quello legato alla memoria si utilizza un sistema di cache a livelli. Tra la prima cache, di primo livello e la memoria vengono interposti uno o più livelli di altre cache con un miss penalty minore. Nel caso di una cache a due livelli, la cache di secondo livello risolve la maggiorparte dei miss della prima cache riducendo il miss rate della cache di primo livello nei confronti della memoria e più in generale il CPI miss.
Esempio cache a due livelli, L1, L2, memoria:
mr = miss rate generico per dati e istr., mp = miss penalty, mr ridotto = $mr_{L1} * mr_{L2}$

Tutti i miss rate accedono alla cache L2, di questi un po' accederanno anche in memoria
*CPI reale = CPI ideale + $mr_{L1} * mp_{L2}$ + mr ridotto * $mp_{memoria}$*

o detto in altri termini, una parte di miss rate accede solo alla cache L2, un'altra accede alla cache L2 e alla memoria
*CPI reale = CPI ideale + (mr_{L1} – mr ridotto) * mp_{L2} + mr ridotto * (mp_{L2} + $mp_{memoria}$)*

La cache secondaria, essendo utilizzata solo nei casi di miss della cache di primo livello, non è molto veloce, né ha un alto grado di associatività, ha un miss rate molto più lato rispetto alla cache di primo livello, compensato in parte dal fatto di essere di dimensioni maggiori.

WRITE TROUGTH E WRITE BACK

Quando si modifica un dato già presente in cache bisogna porre attenzione alla relazione con il corrispettivo dato in memoria, le due tecniche usate sono

write trought
Sovrascrivi il blocco sia in cache che in memoria principale. Siccome scrivere in memoria principale è dispendioso in termini di tempo, il blocco viene prima scritto in una memoria tampone, write buffer, e da lì trasferito poco a poco mentre si svolgono le operazioni successive (in maniera asincrona).

write back
Sovrascrivi il blocco in cache e, solo nel momento in cui dev'essere rimpiazzato da un altro, scrivilo in memoria. Occorre un nuovo bit nell'indirizzo di cache, detto dirty, che segnala se il blocco è stato modificato o meno.

CLASSIFICAZIONE DEI TIPI DI MISS (Regola delle 3 C)

Certi
Il blocco a cui si accede dev'essere portato in cache in un blocco libero (valid 0).

Conflitti
Il blocco deve sovrascriverne un altro perché mappato con lo stesso indice (valid 1). Non si verifica con cache completamente associativa perché in questo caso finché c'è un blocco libero si utilizza quello (se sono tutti occupati si parla di miss per capacità).

Capacità
Il blocco deve sovrascriverne un altro e la cache è piena.

MEMORIA VIRTUALE

Cos'è e perché si usa
La memoria virtuale consiste nel considerare la memoria principale una cache del disco. I programmi vengono compilati con indirizzi virtuali, che puntano ad un blocco (in questo caso detto pagina), presente nel disco, che vengono tradotti in indirizzi fisici solo quando vi si accede.
Le pagine in memoria a loro volta saranno salvate nella cache principale in ordine di accesso.

efficienza perché solo le parti del programma in uso vengono salvate in memoria
sicurezza un programma accede ad uno specifico spazio di indirizzamento fisico quindi non sovrascrivere mai dati in memoria che potrebbero comprometterne un altro
più spazio i programmi nel complesso possono superare lo spazio della memoria fisica

Implementazione considerando la latenza del disco
- pagine grandi (località spaziale) per generare meno miss, detti page fault
- write back
- LRU (Least Recently Used) sostituzione delle pagine meno recenti calcolata dal SO
- associatività completa tra disco e memoria, mapping registrato nella PT, page table in modo da non dover accedere al disco e cercare la rispettiva pagina in tutta la memoria.

Ogni programma al momento dell'avvio crea la propria page table in memoria e si ricava uno spazio di indirizzamento fisico in memoria ed uno di indirizzamento virtuale nel disco.

Page Table
Gli ultimi bit di un indirizzo virtuale sono l'offset, i primi sono il virtual page number (abbr. vpn). Il numero di bit di vpn determina il numero massimo di indirizzi virtuali possibili, che è 2^{vpn} e corrisponde alle righe della PT.
Quando si accede ad un indirizzo virtuale si verifica in PT che la riga relativa al vpn abbia valid 1, se è così si preleva il physical page number salvato nella riga, gli si aggiungo i bit di offset per ottenere il physical address, cioè l'indirizzo fisico in cui è allocata la pagina che stiamo cercando.

Page fault
Si ha quando in PT troviamo valid 0, è l'unico miss possibile perché i miss per conflitto non ci sono in caso di associatività completa.
Interviene il sistema operativo (vedi in seguito) per portare la pagina dal disco in memoria in un indirizzo fisico libero. Potrebbe essere che tutti gli indirizzi fisici siano occupati perché gli indirizzi virtuali sono di numero maggiore. In questo caso si verifica in PT quale pagina è stata riferita meno di recente, grazie ad 1 bit di reference che viene azzerato periodicamente e avvalendosi di calcoli lato sw. Si sostituisce la pagina candidata in memoria cambiando di conseguenza le righe in PT, nel caso la pagina fosse anche in cache viene eliminata dal sistema operativo perché contiene la pagina ora sostituita. Nel caso ulteriore in cui la pagina è stata modificata, ha bit di dirty = 1, prima di essere sostituita va aggiornata nel disco (write back).

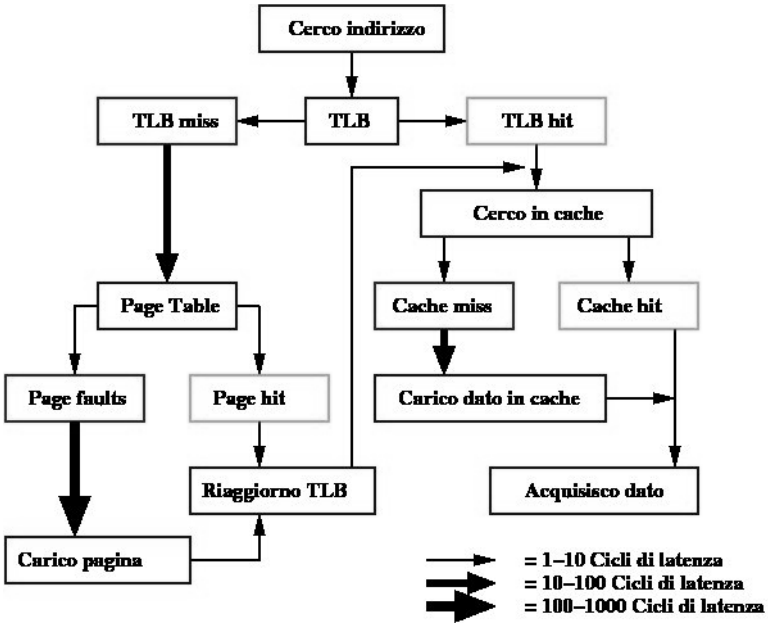
TLB
Per velocizzare la ricerca nella PT, questa adotta una cache detta TLB, Translation Lookaside Buffer, che contiene gli indirizzi riferiti più di recente in PT.

Sequenza TLB, PT, cache, memoria
Si accede ad un indirizzo virtuale.
Nel caso in cui il corrispettivo indirizzo fisico sia salvato in TLB, c'è sicuramente in PT e quindi la pagina è in memoria, si controlla direttamente se la pagina è in cache o meno ed eventualmente la si carica in cache.
Nel caso abbiamo un TLB miss ma l'indirizzo è mappato in PT si provvede ad aggiornare TLB e si cerca nuovamente la pagina in cache, che può esserci o meno, eventualmente la si carica in cache.
Nel caso abbiamo un TLB miss ed un page fault in PT, la pagina viene copiata dal disco in un indirizzo fisico. L'indirizzo della memoria può essere libero, quindi non in cache, oppure sostituito ad un altro esistente (comunque eliminato dalla cache). Dopo aver copiato la pagina si aggiornano PT e TLB e si cerca, come da sequenza, il dato in cache, con la certezza di ottenere un cache miss e di doverlo recuperare dalla memoria.
Da un page fault abbiamo sempre un cache miss. Il contrario invece è falso, perché se siamo arrivati alla ricerca in cache, sappiamo che il dato è già in memoria, basta copiarlo dalla memoria in cache.

Casi possibili di accesso ad un indirizzo virtuale

TLB hit			→	cache hit
TLB hit			→	cache miss
TLB miss	→	PT hit	→	cache hit
TLB miss	→	PT hit	→	cache miss
TLB miss	→	PT page fault	→	cache miss

Memoria virtuale e tempi di latenza



Page fault e Sistema Operativo
Caso di miss in PT, ovvero indirizzo virtuale con valid 0. Il processore interrompe il processo in corso e salva il valore del PC, solleva l'eccezione di page fault al SO che gli consente di passare in kernel mode.
Il SO salva lo stato del processo (registri generali, page table register), determina l'indirizzo fisico in cui scrivere la pagina e comincia la lettura della pagina dal disco. Durante la lettura, che dura migliaia di cicli di clock, il SO ripristina lo stato di un altro processo pronto ed assegna al processore l'esecuzione di tale processo. Al termine della lettura ripete il processo che aveva causato il page fault, che ora troverà la pagina.

Nel caso gli indirizzi fisici siano tutti occupati il SO dovrà scegliere quale pagina rimpiazzare (LRU), nel caso in cui debba sostituire una pagina che è stata modificata deve prima copiarla su disco (write back). Quando sostituisce una pagina deve porre a valid 0 la riga in cui era mappata in PT e fa la stessa cosa per TLB e cache, nel caso la pagina fosse riferita anche lì.

Indirizzo virtuale con cui vengono compilati i programmi
virtual address = virtual page number, offset
offset = log page_size

PT
#righe PT = $2^{\text{virtual page number}}$
riga PT = valid, dirty, reference, physical page number
*PT size = #righe PT * riga PT*

Indirizzo fisico in cui viene salvata la pagina
physical address = physical page number, offset

TLB
virtual page number = tag, index

index = log #set TLB
#set = #blocks (o ingessi) / #ways

riga TLB = valid, dirty, reference, tag, physical page number

Cache
physical address = tag, index, offset

index = log #set cache
#set = #blocks (o ingessi) / #ways
offset = log block_size (in Byte)

riga cache = valid, tag, pagina

NB in TLB e cache, se di tipo diretto, considerare #ways = 1