Gerarchia delle memorie - continuo

Quando si parla di **cache**, si parla di **blocchi di cache o linee di cache**; tipicamente si prende dati dalla **memoria più lenta** e la si porta in quella **più veloce**. Quando il processore non trova un dato all'interno della memoria più veloce, la va a ricercare all'interno di quella più lenta e ne fa una copia in cache (mem veloce).

Tecnologia delle memorie

- **SRAM** Il tipo di memoria più costosa, ma estremamente costosa; viene realizzata con i famosi **flip flop** e con i **transistors**.
- **DRAM** Questo è il tipo di memoria usata nella RAM comune, molto più lenta della SRAM ma molto più veloce dei dischi.
- **Dischi magnetici** Uno dei tipi di memoria più lenta, molto utile per immagazzinare dati di grandi dimensioni.

Tecnologia DRAM

I dati vengono salvati sottoforma di **condensatori** (quindi abbastanza lenti da caricare) ed un singolo **transistor** viene usato per accedere ad una carica. Questo tipo di memoria deve essere periodicamente **ricaricato (refreshed)** proprio perchè i condensatori tendono a scaricarsi.
A questo tipo di ram si accede per **righe e colonne**, quindi ha una struttura a matrice.

Memoria flash

A differenza dei dischi di tipo magnetico che hanno delle testine meccaniche, questo tipo di memoria è realizzata con una tecnologia completamente elettronica e non ci sono parti meccaniche. Sono particolarmente piccole e richiedono una minore potenza (consumano meno), ma hanno un prezzo più elevato dei dischi.

NOR flash:

Quando abbiamo bisogno di una memoria più simile alla RAM, ad esempio in un piccolo sistema embedded, la memoria di tipo NOR è quella giusta. Con queste memorie possiamo effettuare accessi di tipo random, sono memorie particolarmente piccole e sopratutto costose.

NAND flash:

Questo tipo di memoria è quella che viene utilizzata all'interno degli **SSD**; questa tecnologia è meno costosa ma il tipo di accesso è **a blocchi**, non posso quindi leggere a singolo byte. Di conseguenza, per modificare anche un piccolo file, dobbiamo **riscrivere interamente il blocco**.

Se con questa memoria accediamo ai files in maniera **sequenziale** andiamo molto più veloce di un disco; se invece accediamo ai dati in maniera random l'accesso è molto più lento.

Inoltre, con questo tipo di memoria non possiamo scrivere sempre alla stessa locazione di memoria, proprio perchè dopo migliaia di accessi alla stessa locazione di memoria, questa si deteriora.

Settori dei dischi

I settori dei dischi erano da sempre di 512 bytes, ma negli ultimi anni si sta provando di creare dei settori più grandi, che permetterebbero la velocizzazione dell'accesso a files grandi.

Ogni settore tiene traccia di:

- ID del settore
- Dati (i famosi 512 bytes)
- Codice di correzione errori
- Campi di sincronizzazione

Come si accede ad un settore, ad esempio, su un hard disk?

- Inizialmente bisogna attendere, qualora il disco fosse in utilizzo da qualcun altro.
- Spostare le testine (fisicamente)
- Elettronicamente viene selezionata la testina che si utilizzerà per leggere (siccome abbiamo un **pettine di testine**)
- Attendere che nella rotazione si arrivi al settore che ci serve

Problemi di performance sui dischi

Se il sistema riesce a tenere i dati tutti "più o meno" nella stessa zona, non c'è il bisogno di spostare la testina tantissime volte; siccome abbiamo delle "mooving parts", come in tutti i sistemi meno si muovono meglio è. Bisogna quindi tentare di "raggruppare" e quindi leggere/scrivere in modo quanto più sequenziale possibile.

Non sempre è possibile farlo però; questo è dovuto al semplice fatto che il disco potrebbe essere occupato già da altri dati (di altri processi ecc), e quindi il sistema è costretto a scrivere "spezzettando" i dati. Questo problema verrà visto meglio nella parte di Sistemi Operativi, con il problema della Frammentazione.

Utilizzo delle cache

E' abitudine comune dei dischi (sopratutto quelli di fascia bassa) di utilizzare delle cache. Questo perchè se un disco ha degli RPM particolarmente lenti, la cache può aiutare a compensare.

Memoria cache

In generale una cache è un sistema posto tra **CPU e RAM**. Solitamente i processori moderni hano 3 livelli di cache, ma per ora facciamo finta che ce ne sia solo uno.

Nel momento in cui il processore fa riferimento ad un indirixxo Xn, questo indirizzo va portato dalla RAM al processore. Per utilizzare la cache, dobbiamo inzialmente capire se esso **è presente nella cache**, inoltre, se è presente, dobbiamo capire **dove è**.

Dobbiamo quindi trovare un sistema che mi permetta di capire **a partire dall'indirizzo che mi serve** qual è il posto all'interno della cache dove devo cercare.
Il metodo più semplice è quello della mappatura diretta:

Mappatura Diretta della cache

In pratica bisogna effettuare una **mappa** che ci dica per ogni indirizzo di memoria in quale zona della cache è salvato. Il modo più semplice per effettuare il mapping, è proprio attraverso il modo **diretto**.

Sulla base degli indirizzi di memoria, ogni indirizzo ha una **locazione prefissata nella cache**. Quindi ogni locazione di memoria va a finire ad un indirizzo della cache **solo grazie al suo indirizzo**.

Basta quindi effettuare il calcolo: *indirizzo del blocco* **modulo** *blocchi nella cache*.

Il problema di questa implementazione, però, è che se un modulo viene mappato su un indirizzo della cache già utilizzato, devo rimuovere il precedente e salvare il nuovo.

Come vengono mappati gli indirizzi?

Siccome gli indirizzi in ram sono composti da una stringa di bit molto lunga, ci basta semplicemente prendere le 3 cifre meno significative dell'indirizzo, invece di effettuare l'operazione modulo (che tra le tante cose porta via anche del tempo).

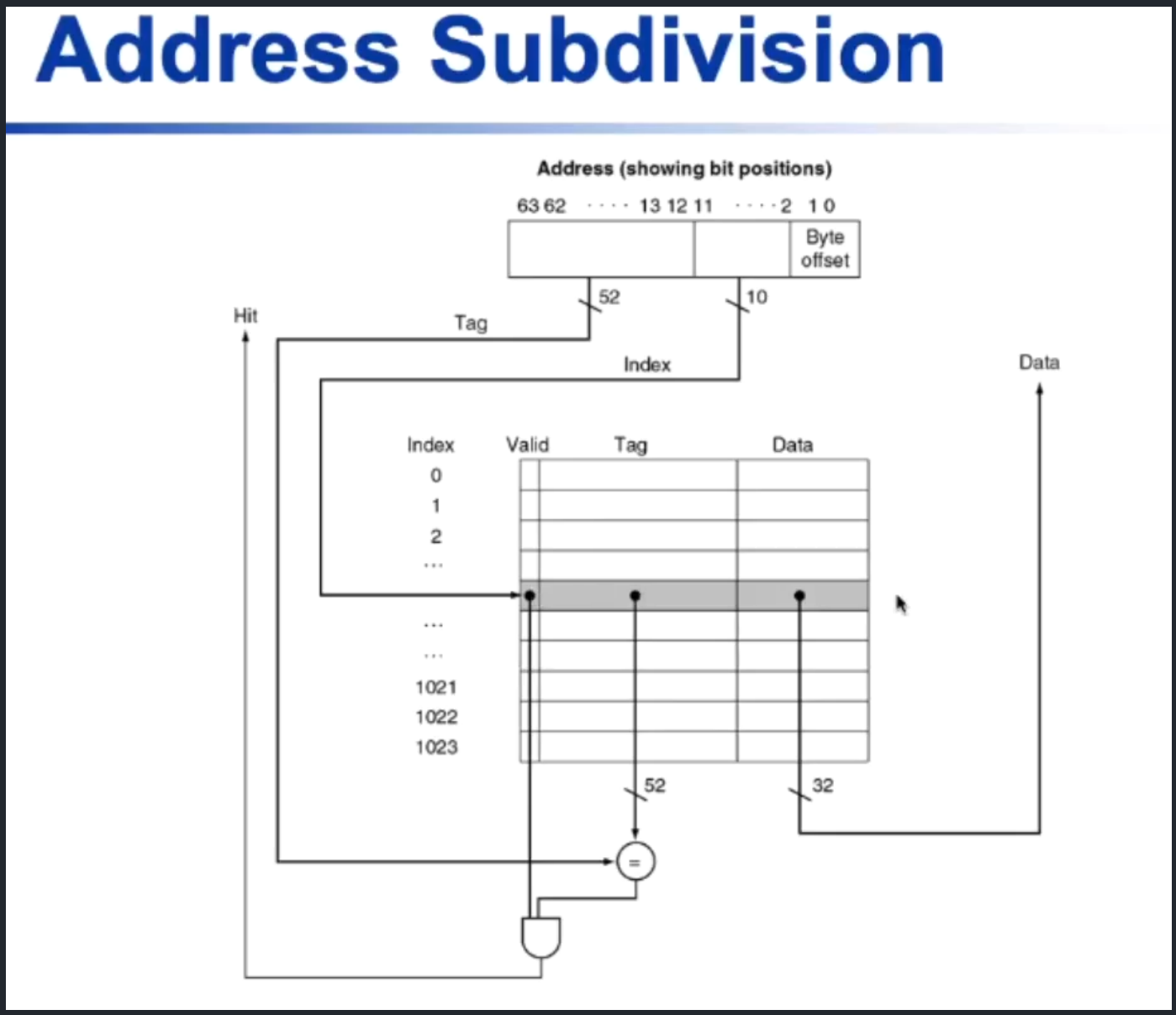
ad esempio: `00101 -> viene mappato su 101`.

A questo punto, abbiamo i lower bits, ma **come capiamo a quale locazione in RAM fanno riferimento?** Ci basterà semplicemente tenere traccia anche dei due bit più alti dell'indirizzo, sotto la voce **tag**. Inoltre abbiamo anche dei **bit di validità**, che ci dicono se l'indirizzo è inizializzato o meno.

Se quando andiamo a cercare nella cache un indirizzo tramite i suoi **lower bits** e lo troviamo, andiamo a controllare anche i **tag**; se i due tag sono diversi, vuol dire che dovremo andare a pescare nuovamente in memoria e **sovrascrivere** nella cache il valore nuovo.

Piccola precisazione

Il mapping, come abbiamo detto, va fatto in relazione agli slot disponibili in ram. Se nell'esempio precedente utilizzavamo solo i 3 bit più bassi per mappare l'indirizzo, vuol dire che la cache aveva solo **8 slots**, addressabili su 3 bit (0-7). Quindi, se abbiamo una cache di 1024 posizioni, ci serviranno 2^{10} bit per rappresentarli, e quindi prenderemo i 10 bit più bassi dell'istruzione.



Come si vede nell'immagine, la ricerca dell'indirizzo nella cache, ci da risultato vero solo nel momento in cui:

1. I bit **valid** ci dicono che l'indirizzo è presente
2. il **tag** coincide con quello dell'indirizzo da trovare (si usa un **comparator**)

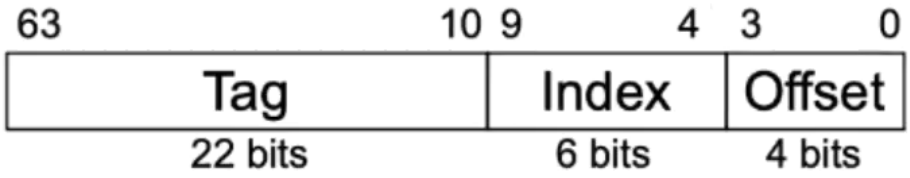
Ovviamente, il tutto va **implementato in hardware**.

Esempio: blocco più grande

Con un blocco realistico, abbiamo una cache di 64 blocchi e 16 bytes per blocco.

Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



- Essendo ogni blocco da 16 byte, servono **4 bits** per trovare la posizione del byte all'interno del blocco.
- Successivamente, abbiamo 64 blocchi di cache -> ci servono 2^6 bits, quindi ci servono 6 bit per trovare l'indice.
- Siccome avevamo un indirizzo a 32 bit, 4 se ne vanno per l'offset, 6 per l'index e quindi ci rimangono **22 bits** per il **tag**.

Write through

Supponiamo che si voglia **scrivere in memoria**; se quando si scrive in memoria aggiorniamo il dato solo sulla cache senza accedere alla RAM, ci troviamo in uno stato inconsistente. Questo significa che la cache contiene il valore aggiornato, ma nella ram è ancora presente il valore vecchio.

Supponiamo di aver incrementato una variabile x nella cache, avremo quindi il valore aggiornato nella cache ma non nella memoria effettiva. La prima idea sarebbe quella di scrivere sia nella cache che nella RAM, ma il problema è che la scrittura su memoria esterna potrebbe richiedere un numero esorbitante di cicli di clock. Quindi andare a scrivere sulla memoria esterna **non è fattibile**.

La tecnica

La tecnica che invece viene usata è proprio quella del **write through**: dopo aver scritto nella cache, i dati che vanno aggiornati anche nella RAM vanno posti in un **buffer di scrittura** e mentre la CPU lavora ad altro, c'è un altro componente che svuota il buffer e **sincronizza in un secondo momento la RAM**. Per qualche istante, il dato aggiornato è solo all'interno della cache, e sulla RAM arriva qualche istante più tardi.

Il problema, come al solito, è il fatto che il buffer prima o poi si riempie; quando esso è pieno, è come se non ci fosse, e quindi il processore **è costretto a fermarsi, come se il buffer non esistesse**.

La tecnica - migliorata

La tecnica giusta, invece, è quella di "favorire l'inconsistenza dei dati". Sembra strano ma funziona: quando il blocco di dati all'interno della cache (ancora non aggiornato sulla RAM) deve essere sovrascritto da un dato nuovo, solo a quel punto verrà **aggiornato nella RAM**. In questo modo non abbiamo più la sincronizzazione "in tempo reale" tra cache e RAM, ma dobbiamo aggiornare i

dati **solo una volta**, e quindi risparmiamo tempo.

Per implementare questa tecnica, abbiamo bisogno di un ulteriore bit detto **dirty bit**, che ci dice che il dato nella cache è stato modificato; questo serve anche a non salvare in memoria i dati, qualora questi non fossero stati modificati nel frattempo (e risparmiamo ancora più tempo).

Write allocation

Quando devo effettuare un'operazione di **store**, ma nella cache non viene trovato il blocco da scrivere, come mi comporto?
Dipende da come è progettata la cache, ma le soluzioni sono due:

Prima soluzione

Caricare i dati nella memoria cache (fare un fetch) e poi scriverci

Seconda soluzione

Scrivere direttamente in memoria.