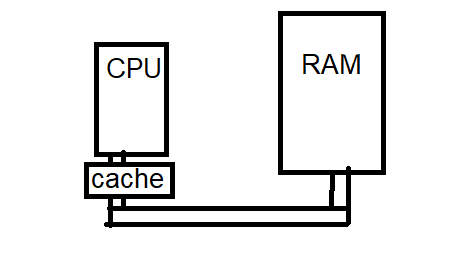
Abbiamo visto una delle due applicazioni principali delle memorie associative, il TLB. Oggi vediamo la seconda applicazione più importante: la memoria cache.

In generale la velocità di un processore è molto di elevata di quella di una memoria RAM, poiché è una memoria dinamica, il che le da dei tempi di accesso più lenti rispetto ai registri in memoria statica del processore. Mentre la Cpu prende pochi nanosecondi per accedere ai registri, per accedere alla RAM servono qualche decina/un centinaio di nanosecondi. Per via di questa disparità si introducono le memorie cache.

  
Le memorie Cache sono interposte tra il processore e il BUS dati, sono realizzate con una tecnologia simile a quella del processore, inoltre sono molto più vicine fisicamente al processore della RAM.

Le memorie cache usano quindi la tecnologia statica, che però ha lo svantaggio di non poter essere usata per grandi memorie (infatti le cache sono più piccole delle RAM). L’obiettivo della cache è quella di conservare delle porzioni/sottoinsiemi del contenuto della memoria RAM, che variano a seconda dei bisogni del processore: questo causa un sostanziale risparmio di tempo nell’esecuzione dei programmi.

I principi su cui si basa la memoria Cache sono i principi di località: nel tempo e nello spazio. Dunque, se il processore sta usando una certa pagina di memoria è probabile che nel prossimo futuro continui ad usarla, così come è probabile che userà celle vicine all’ultima usata.

Linee di cache(?) cerca su google.

Il trasferimento dei dati dalla RAM alla Cache viene gestito in maniera completamente indipendente dalla cache stessa. Essa è costituita da linee di Cache, in cui una porzione è dedicata al campo Tag (tipica delle memorie associative) e diversi (uno, due o più) campi per i valori contenuti nelle celle memoria a cui punta.

Supponiamo di avere una memoria RAM di 2^30 byte organizzate in parole da 32 bit. In totale ci sono 2^28 celle di memoria, i cui indirizzi vanno espressi su 28 bit. Utilizziamo però 27 bit per il campo Tag, in quanto prendiamo due celle per linea di cache.

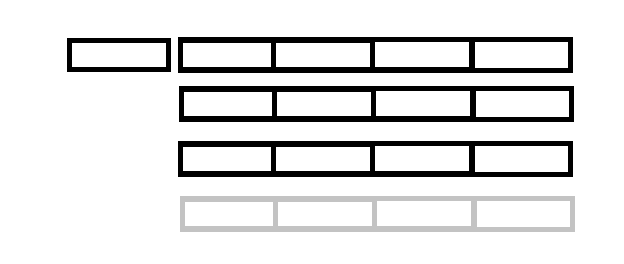
Ipotizziamo che il processore abbia generato l’indirizzo 1000. Il controllore della cache andrà a guardare due celle: quella di lunghezza 1000 e quella di lunghezza 1001. A questo punto il contenuto della cella 1000 viene salvato nella prima cella data, mentre quello della cella 1001 viene salvato nella seconda cella data.

Nel nostro esempio abbiamo aggregato due celle consecutive in una linea di cache, tuttavia potremmo anche aumentarla a 4. In questo caso la dimensione del campo tag si ridurrebbe a 26 bit. All’aumentare della dimensione della linea di cache sfrutto maggiormente la località nello spazio. Tuttavia la località nel tempo viene invece sfruttata utilizzando un algoritmo di sostituzione (LRU = less recently used).

Abbiamo nella cache una certa quantità di linee (ad esempio 8), che possono essere usate in maniera indipendente l’una dall’altra. Esse si riempiono una per volta copiando n celle della RAM per linea di cache. Quando si è arrivati ad utilizzare tutte le linee di cache e il processore chiede un indirizzo diverso, l’algoritmo LRU prevede che ci sia una sostituzione: la linea usata per ultima viene svuotata e viene riempita dal gruppo di celle di cui il processore ha bisogno.

Tutte queste operazioni sono realizzate a livello hardware: per questo, onde evitare che diventi tutto troppo complicato, si evita di applicare con rigore l’algoritmo LRU (perché richiederebbe troppi bit e troppa energia per contare il tempo di inutilizzo di tutte le linee di cache e poi confrontarli). L’algoritmo si implementa in versione approssimata, riducendo la rappresentazione del tempo di ultimo accesso a un solo bit, che (parte al valore 1 al momento del caricamento della linea, che) viene resettato a 0 da un timer di tanto in tanto e che viene reimpostato a 1 quando la linea viene utilizzata nuovamente. Così la Cache decide di sostituire una qualsiasi delle sue linee che abbia il valore 0 a quel bit. Non è un meccanismo perfetto (può capitare che la cache sostituisca una linea che il processore stava usando perché per sfortuna i bit si sono resettati un istante prima), tuttavia è un metodo molto migliore rispetto alla sostituzione casuale.

Un secondo modo per ridurre la complessità circuitale e ridurre i costi è quello di cercare alternative all’uso di una memoria associativa (perché ogni linea richiede non solo un campo tag ma anche un circuito comparatore). Quindi, vediamo l’organizzazione interna della cache:

* la prima alternativa è quella completamente associativa (ogni linea ha il suo campo tag e il suo circuito comparatore, la ricerca avviene su tutte le linee contemporaneamente)
* la seconda è la cache a corrispondenza diretta (c’è un solo campo tag, che vale per TUTTE le linee di cache; in questo caso il campo tag è ridotto e nel nostro esempio di prima conterrebbe meno di 26 bit, i bit rimanenti vengono usati per determinare una linea all’interno del nostro dispositivo Cache. Se per esempio ci sono 16 linee, gli ultimi 6 bit vengono usati per determinare una delle linee e la posizione nella linea). Facciamo per esempio che il processore voglia accedere all’indirizzo RAM 65, che in binario è 001000001, esso corrisponde alla seconda cella della prima riga della cache (poiché gli ultimi due bit sono 01 e i successivi quattro sono 0000). Questo metodo fa sì che a ciascun indirizzo possa corrispondere UNA SOLA linea di cache, quindi se lo si vuole caricare bisogna sostituire necessariamente quella linea anche se le altre non sono state usate.  
    
  Il vantaggio è che richiede un solo Tag e un solo circuito comparatore, tuttavia diventa più simile a una RAM in miniatura, quindi è un po’ più lenta e sfrutta meno bene il principio di località.  
  Questo è il polo opposto rispetto alla cache completamente associativa: molto meno costosa ma certo meno efficiente.
* La via di mezzo è la memoria Cache associativa ad insiemi. Essa si basa sui livelli di associatività.  
  Immaginiamo di avere una cache a 256 linee, ognuna delle quali ha un campo tag e 4 celle. Se la memoria fosse completamente associativa avremmo un livello di associatività di 256, e altrettanti comparatori. A un livello di associatività 1, avremmo un solo tag e un solo comparatore, e in questo caso avremmo 2 bit meno significativi per la cella di memoria all’interno della linea (così come per la 256) ma poi avremmo però altri 8 bit in cui è presente l’informazione del numero di linea, lasciando i restanti per il contenuto vero e proprio del Tag.   
  Una via di mezzo è per esempio il livello di associatività 16. Questo significa che ci sono 16 comparatori (e quindi la cache è divisa in 16 parti). Ciascuna delle 16 parti in cui è suddivisa la cache viene trattata come una Cache a Corrispondenza Diretta. C’è quindi un iniziale confronto tra tutti e 16 i Tag e l’indirizzo di memoria e se uno ha un riscontro positivo (notato da una funzione OR) allora la Cache procede a trattare l’informazione presente nella linea e cella corrispondente. Questo è un compromesso perché costruendo tante piccole sotto-cache a corrispondenza diretta il tempo da loro impiegato è molto piccolo.

In genere nel realizzare una Cache non si utilizza la realizzazione completamente associativa (per cache di decenti dimensioni) per via degli elevati costi di quest’ultima.

È abbastanza intuitivo ottimizzare un sistema usando più memorie cache. Infatti è così, c’è addirittura una vera e propria gerarchia di memorie cache, con memorie di livello 1, di livello 2 ecc.

I vantaggi di questa memoria cache su più livelli è quello di cercare di arrivare gradualmente alla memoria RAM dinamica passando attraverso memorie cache a velocità diverse. Si parte da un processore a memoria molto piccola ma estremamente veloce. C’è poi un primo livello di Cache la cui memoria è più grande del processore, ma non gigantesca, ed è di velocità paragonabile al processore (realizzata con una tecnologia quasi completamente associativa). Poi c’è una cache di secondo livello, molto più grande della cache di primo livello, ma decisamente più lenta in quanto la memoria associativa è a insiemi ma a livello di associatività basso. C’è infine la RAM, molto più grande ma più lenta. Idealmente la prima memoria Cache ha dimensione di pochissimi kilobyte, se il processore trova lì i dati che necessita, stonks, altrimenti va a cercare nella cache di secondo livello, dove la probabilità di trovare il dato è più alta ma la velocità rimane comunque molto più alta rispetto alla RAM. Ci possono essere anche più livelli di memorie cache (non è strano trovare un terzo livello).

Bisogna ricordarsi, infine, che tra queste memorie cache e la RAM da qualche parte deve essere inserita anche l’MMU (si trova sicuramente dopo il primo livello e probabilmente dopo il secondo). Il vantaggio di avere la cache prima dell’MMU è quello di evitare il più possibile l’utilizzo del meccanismo di traduzione (specie se l’indirizzo non è ancora presente nel TLB). Quindi i primi due livelli di cache funzionano, come il processore, a indirizzi virtuali. Un eventuale terzo livello potrebbe stare dopo L’MMU e in questo caso il campo Tag funzionerebbe a indirizzo fisico.

Per passare dall’esecuzione di un programma a quella di un altro una delle cose che bisogna fare è svuotare la memoria cache.

Veniamo ora alle operazioni di lettura e scrittura all’interno della cache. Se quando voglio accedere in lettura il dato non è presente nella cache il gestore della cache riconosce la mancanza dell’indirizzo e legge l’indirizzo nella RAM, importando il dato. In questo caso dopo l’accesso in lettura abbiamo due copie del dato (uno in RAM e uno in cache), e se usiamo più livelli possiamo avere ancora più copie (2, 3 o 4). Quando però il processore vuole effettuare delle operazioni di scrittura ci sono più valori tra i quali si deve scegliere quali modificare. Per questo motivo si introduce il protocollo di consistenza. Ce ne sono di due tipi diversi ( o meglio quelli sono quelli che vediamo noi):

* Il più semplice è il Write-Through: quando il processore chiede la scrittura essa avviene sia all’interno della memoria cache, sia all’interno della memoria RAM (si aggiornano quindi tutte le copie e si fa aspettare il processore finché non termina l’operazione). Il vantaggio in questo caso è quello di mantenere la consistenza dei dati, lo svantaggio è che il processore rallenta consistentemente quando effettua un’operazione di scrittura (quindi è come se la cache non ci fosse a migliorare l’efficienza). La giustificazione che si porta tale protocollo è che se si fa l’analisi del comportamento dei programmi reali il numero delle operazioni di scrittura è più piccolo delle operazioni di lettura (in particolare non superano il 25% del totale degli accessi alla memoria RAM). Ergo il rallentamento del protocollo Write-Through si applica solo a ¼ delle istruzioni di accesso.
* L’alternativa è il protocollo Write-Back. Durante la scrittura essa viene considerata terminata quando essa è stata aggiornata nella memoria Cache (senza preoccuparsi della RAM). Per una certa quantità di tempo, quindi, la consistenza non è mantenuta: la copia presente in cache non è più uguale all’originale presente in RAM. Il momento in cui si aggiorna la cella RAM è invece l’ultimo momento utile: cioè quando la cella di memoria cache sta per essere svuotata/sovrascritta. Ricordarsi di ciò è responsabilità del controllore della memoria Cache. Questo velocizza l’operazione di scrittura ma rallenta la sostituzione delle linee di cache. Questa modalità di funzionamento è certamente più efficacie, però richiede una maggiore complessità circuitale, è quindi più costoso da realizzare.

Cosa ha senso mettere al primo livello/al secondo ecc.? Tipicamente al primo livello si preferisce avere dispositivi semplici (ma veloci, almeno in lettura), quindi in genere è realizzata con il protocollo Write-Through. Questo specialmente se è presente un secondo livello e in maniera particolare se (come ci si aspetta) questo presenta un protocollo del tipo Write-Back (per non rallentare troppo le operazioni di scrittura del processore).

Quindi in generale i livelli superiori di Cache sono più complicati e sono di dimensioni superiori, quindi si dovrebbe presentare meno frequentemente (rispetto al primo livello) il problema della sostituzione della linea di Cache (che è ciò che fa rallentare il Write-Back).

C’è un altro piccolo problema da tenere in conto, legato alla realizzazione di sistemi con più processori. Un sistema moderno è basato su processori multi-core. Vi è quindi più di una CPU, ognuna delle quali ha la propria gerarchia di cache. In genere ogni cache è dotato della sua personale cache di primo livello, solo a livelli più alti le cache sono condivise. Un processore a più core può quindi causare ulteriori problemi di consistenza: per esempio se un core modifica una cella di memoria e un altro prova poi ad accedervi. Nell’esempio proposto ci è una differenza tra il valore salvato nelle cache di alto livello/RAM e quello presente nelle cache di primo livello dei processori.   
Questo è un tipo di problema è molto più complicato da risolvere.