1 Lezione del 07-03-25

Riprendiamo il discorso della memoria cache. Avevamo che questa è montata fra la CPU e lo spazio di memoria. Più propriamente, questa si trova fra la CPU e il bus.

In questo, può vedere non solo le operazioni sulla memoria, ma anche sullo spazio di I/O. In questo caso, però, dovrà ovviamente comportarsi sempre in maniera *read-through* e *write-through*, quindi effettivamente disattivarsi e lasciare che il processore interagisca direttamente con l'I/O.

Questo è dovuto al fatto che allo spazio di I/O potrebbero accedere e modificare dati dispositivi esterni alla CPU (le interfacce), operazione che invaliderebbe immediatamente qualsiasi cosa venga scritta in memoriaa cache.

Inoltre, ogni operazione di lettura può comportare di per sé un aggiornamento delle interfacce, che comporterà un aggiornamento della memoria, motivo per cui un operazione di caching sarebbe superflua.

Operazione simile varrà effettuata per la memoria video (che non sta nello spazio di I/O). Questa facoltà verra realizzata dalla cache attraverso, probabilmente, *maschere* o *tabelle*.

1.0.1 Cache associative ad insiemi

Avevamo visto come il difetto principale della cache ad indirizzamento diretta è quello delle *collisioni*. Presentiamo un metodo, quello delle **cache associative ad insiemi**, che risolve il problema permettendo di allocare più cacheline allo stesso indirizzo.

Duplichiamo quindi la struttura vista per la cache ad indirizzamento diretto (qui 2 volte), e sruttiamo le uscite hit/miss delle singole memorie delle etichette per pilotare un multiplexer con in ingresso le linee dati delle memorie di cache corrispondenti.

In questo caso a letture allo stesso indice le cache potranno rispondere diversamente (magari la prima in miss e la seconda in hit), e il processore vedrà ritornarsi il dato corretto (in questo caso quello della seconda).

Compito di scegliere quale cache sfruttare nel caso di collisioni è quello del **controllore** di cache (nella cache ad indirizzamento diretto non c'era scelta). La scelta migliore possibile sarebbe quella di scegliere la cacheline al cui i accederà più tardi nel futuro (per mantenere i dati immediatamente utili nella cache).

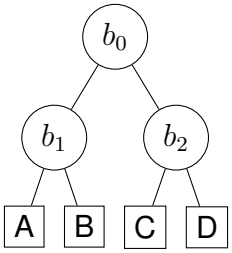
Chiaramente, visto che non si può prevedere il futuro (o almeno non lo possono fare né la CPU né il controllore di cache), occorre adottare un euristica. Una di queste euristiche è la politica **LRU** (*Least Recently Used*), dove si sceglie la cacheline al quale non si accede da più tempo.

Per realizzare tale politica si sfrutta una memoria, che chiamiamo R. Con solo due vie, basterà memorizzare su R l'ultima via usata, e quella su cui scriviere sarà immediatamente l'altra.

Con più di due vie sarebbe necessario mantenere l'ordine degli accessi, cio per n vie ricordare informazione necessaria a controllare $n!$ diverse possibilità. Nella pratica, però, conviene usare politiche approssimate.

1.0.2 Pseudo-LRU dell'80486

Vediamo una di queste politiche approssimate, che gestiva 4 vie attraverso 3 bit $b_0 0$, b_1 e b_2. Si usava un albero binario per la selezione di una delle vie, disposto come:



dove i valori 1 sono i rami a destra, e viceversa i valori 0 sono i rami a sinistra.

In fase di rimpiazzamento, si sceglie la via seguendo l'albero. In fase di accesso, si modificano i b_i in modo da portare la via a cui si è fasso accesso in fondo all'ordinamento che si ottiene visitando l'albero. L'errore può essere dato dal fatto che la via che si trova nello stesso gruppo della via a cui si è fatto accesso potrebbe trovarsi ad un indice più alto del necessario, visto che si abbassa cumulativamente l'intero gruppo aggiornando b_0.

Per cache più grandi si sfruttano sempre algoritmi ad albero di questo tipo, magari tagliando i rami più bassi per lasciare spazio a scelte completamente casuali.

Notiamo poi che le memorie cache di questo tipo incontrano sempre difficoltà quando si fanno accessi ciclici ad indici che si ripetono con un modulo con il numero di vie diverso da zero: ad esempio se si leggono ciclicamente 5 indirizzi che corrispondono allo stesso indice, la cache non riuscirà mai a mantenere tutti e 5 in una delle cacheline delle vie, e quindi ogni accesso comporterà un miss.

1.0.3 Livelli di cache

Nei processori moderni si hanno solitamente più livelli di cache (3 o 4), che crescono in dimensioni e associatività più si vanno a disporre "lontano" dal processore e "vicini" alla RAM. Le cache di livello più basso saranno quindi più veloci ma più piccole, mentre le cache di livelo alto saranno più lente ma più grandi.

Il controllore di cache provvederà a gestire i livelli di cache, effettuando gli accessi controllando a partire dal livello più basso (più veloce) per arrivare al livello più basso, fino alla RAM.

1.1 Interruzioni

La limitazione principale del processore studiato finora è che il flusso di controllo è completamente determinato dal progamma in esecuzione. Attraverso il meccanismo dell'interruzione, il sistema definisce $e_1, ..., e_n$ **eventi**, e il programmatore $r_1, ..., r_n$ **routine** per la gestione di tali eventi. Da qui in poi il processore continua ad eseguire il suo normale flusso di controllo, ma monitorando in qualche modo lo stato di questi eventi. Nel caso uno degli eventi e_i effettivamente si verifichi, la CPU provvederà a sospendere il flusso di controllo attuale e ad eseguire la routine r_i.

Un esempio classico dell'utilità di un meccanismo di questo tipo è dato dalle fasi di stampa che avevamo definito per dispositivi come le stampanti: attraverso l'approcio visto finora dovremmo controllare periodicamente un certo registro di stato per verificare la possibilità di scrivere un nuovo dato in un certo registro di buffer. Questo occupa la CPU con operazioni inutili, che potrebbe saltare se fosse la stampante stessa ad avvertirla di quando è pronta a ricevere un nuovo dato.

L'idea di base è quella di avere una nuova operazione da svolgere in fase di esecuzione di un instruzione da parte della CPU, dopo l'esecuzione dell'istruzione stessa.

Ad esempio, potremmo riportarci un bit di valdiità, `READY`, da parte della stampante, e controllarlo ad ogni istruzione per la chiamata di una routine di stampa. La chiamata sarà semplicemente un aggiornamento condizionato a RIP, con scrittura del contenuto attuale di RIP in pila (che è compatibile con le regole di chiamata dei sottoprogrammi a cui siamo abituati).

Un problema di questo approccio potrebbe essere che, se il bit che segnala l'evento non si aggiorna immediatamente, la CPU andrà in un ciclo continuo di arresto dell'esecuzione e inizio di una routine. Una soluzione potrebbe essere dotare della CPU di una *rete di accettazione* della richiesta: il bit di segnalazione dell'evento va in un generatore di impulsi che setta un SR flip-flop. A questo punto la CPU risponde (livello hardware, nella nuova fase di esecuzione appena descritta) con un segnale di reset nel momento in cui riesce a rilevare l'evento e spostarsi nella routine.

In verità la situazione è più complicata: ad esempio potremmo voler ignorare nuovi eventi quando stiamo già cercando di soddisfarne uno. Per questo i processori x86 prevedono un apposito flag, il flag **IF** (*Interruption Flag*), che determina se le nuove interruzioni dovranno essere soddisfatte o meno. Il processore può essere quindi configurato per attivare automaticamente il flag IF in fase di risposta ad una richiesta di interruzione. Per effettuare il corretto ritorno, si usa la funzione `IRETQ`, che ripristina, oltre ad altre cose, lo stato dei flag (che era stato salvato in pila).