Proprietà di Località.

Un concetto fondamentale nei calcolatori moderni è quello di Località.

✳ Località nel Tempo. Un programma ha la proprietà di località nel tempo se, dopo aver generato un indirizzo, dopo un po' di tempo genera nuovamente lo stesso indirizzo. Insomma il codice punta sempre alla stessa cella di memoria.

✳ Località nello Spazio. Un programma ha la proprietà di località nello spazio se, dopo aver generato un indirizzo, dopo un po' di tempo genera un indirizzo adiacente al primo

→ Memoria Cache

Viene usata per aumentare la velocità di esecuzione dei programmi, e soprattutto di Refresh di una RAM dinamica. Infatti quando la CPU prova a interagire con la RAM in fase di Refresh, non ci riesce, e deve aspettarne la fine.

Perciò introduciamo questa nuova memoria più veloce e sempre disponibile per la CPU.

Il nostro obiettivo è di prendere una parte della RAM dinamica, copiarla nella Cache e far sì che la CPU acceda alle copie nella Cache, in quanto di accesso più veloce, piuttosto che ai riferimenti nella RAM. Fisicamente la Cache può essere realizzata in vari modi:

✳ **Completamente Associativa**: si basa sul concetto di divisione in tag e data (come nelle memorie associative). Nel Tag dobbiamo inserire l'indirizzo della RAM, nei dati inseriamo il valore della cella. Per sfruttare in modo efficace la proprietà di località nel tempo, basta inserire nella Cache il dato a cui è stato fatto riferimento, dunque se non c'è l'indirizzo, lo devo aggiungere. Se c'è già posso terminare l'operazione di accesso alla Cache senza accedere alla RAM. Per la proprietà di località nello spazio invece, nei dati non dobbiamo mettere il contenuto di una sola cella, ma il contenuto di più celle di indirizzo adiacente.

Ciò mi porta a definire il concetto di Linea di Cache, un concetto simile alle pagine della memoria virtuale, ma in parti molto più piccole.

Se considero 4 celle da 4byte consecutive, e le considero una sola linea, i loro indirizzi saranno uguali se non per gli ultimi 2 bit di indirizzamento (00,01,10,11). A questo punto abbiamo un sotto insieme di bit di indirizzo che inseriremo nel campo tag, dunque nel tag ci saranno i 30 bit più significativi comuni a tutte le celle di memoria della stessa linea. Abbiamo un indirizzo lungo, di cui la maggior parte è il tag e l'ultima parte sono i dati, che distinguono i singoli byte all'interno della stessa parola, dunque possiamo creare linee di lunghezza ragionevole da implementare nella Cache completamente associativa.

Se la ricerca associativa ha successo (Hit), il dato cercato dalla CPU è nella linea, altrimenti no (Miss).

Nel caso siano piene tutte le linee di Cache, dobbiamo svuotarne una. Per scegliere la linea da eliminare, vediamo le proprietà di Località in tempo e spazio, e vediamo che esistono altri bit di controllo delle linee, come il bit di accesso, che si comporta similmente ai bit di stato (che abbiamo visto nell'alu e nell'lru)

→ Lettura e Scrittura nella Cache.

La CPU può fare sia lettura che scrittura sulla linea di Cache. Se fa lettura, abbiamo un dato presente in RAM che, quando serve, va copiato nella Cache, per averne un accesso più immediato. La scrittura invece è

più difficile da gestire. Intanto come sappiamo, per fare una scrittura dobbiamo prima eseguire una lettura. La gestione della scrittura è complicata perché se modifichiamo un valore nella Cache, esso non sarà più uguale al corrispondente valore nella RAM. Il nostro obiettivo è di mantenere una consistenza tra i due valori di A, e per farlo è stata ideata una serie di metodi: ✳ Write Through. Una Cache con questo algoritmo implementato velocizza solo la lettura, e non la scrittura, dunque abbiamo la stessa velocità di un sistema senza Cache, in quanto abbiamo una doppia copia in Cache e RAM. Se dobbiamo fare un numero alto di scritture non è conveniente. ✳ Write Back. Una Cache con questo algoritmo non ha come obiettivo il ripristino della consistenza tra RAM e Cache, poiché, mantenendo questa inconsistenza fra dati, fa sì che la scrittura duri lo stesso tempo della lettura, in quanto non c'è accesso alla RAM per sovrascrivere i dati in RAM con quelli nella Cache ogni volta che si verifica una modifica. Per supportare il Write Back, occorre aggiungere un bit di modifica, che vale 1 quando c'è stata una modifica nella Cache ma non nella RAM, dunque quando c'è inconsistenza, 0 altrimenti. Questo bit permette dunque di ricordare che c'è inconsistenza, e ci permette di andare a ristabilire questa inconsistenza il più tardi possibile, ovvero quando dobbiamo cancellare la linea di Cache per far posto ad un'altra. In questo caso, l'eliminazione della linea di Cache porta a considerare anche il bit di modifica, in modo da eseguire una copia da linea di Cache a RAM prima di cancellarla, se il bit è a 1, dunque se c'è stata una modifica, mentre viene semplicemente sostituito senza copia in RAM quando il bit di modifica è a 0. Se voglio sostituire una linea dunque, ne cercherò in primis una con bit di modifica a 0, insomma una linea non usata e non sovrascritta, poiché possiamo sostituirla immediatamente senza eseguire la copia nella RAM. Vediamo quindi che tra Write through e Write back il più prestante è il secondo, con tempi di accesso molto inferiori uguali per lettura e scrittura, in modo che il rallentamento della RAM non influisca quasi mai. Tuttavia il primo è più semplice, non ha il bit di modifica e non prende in considerazione altri fattori per la sostituzione di una linea, è meno costosa a livello hardware, è meno spaziosa e consuma meno corrente, dunque generalmente ha prestazioni inferiori ma è più economica.

✳ **Corrispondenza Diretta**: In questo caso il funzionamento è analogo alla Cache completamente associativa, ma la Cache è implementata con una memoria statica. Diminuendo i bit di Tag, introduciamo un altro sotto insieme di bit di indirizzamento che vengono usati per indirizzare una delle linee di Cache. Supponendo di voler realizzare una Cache a corrispondenza diretta con 128 linee di Cache, separo i 7 bit (per avere 128 combinazioni, una per ogni linea) che uso per indirizzare una linea dagli altri 20 di Tag. Ora ho che un certo indirizzo può finire solo in una particolare linea di Cache, dunque se prendo un indirizzo a caso, senza specificare il Tag e gli ultimi 5 bit, vediamo che esso si riferirà ad una determinata linea di Cache, organizzata come una RAM. Inoltre una linea che nei 7 bit di linea codifica il valore 8, sarà nella posizione 8. Quali vantaggi ha questa memoria? È una soluzione meno costosa, ma esegue sempre una ricerca associativa, in quanto confronto Tag con una parte dei bit di indirizzo, dunque per fare questo confronto ho bisogno di un comparatore da 20bit in questo caso. Tuttavia nel caso di memoria associativa dovrà avere 128 comparatori di uguaglianza per 128 Tag, in modo da confrontare. Qui si fa un solo confronto, se ha successo è Hit, altrimenti è Miss. È una notevole semplificazione dal punto di vista Hardware, costa molto meno di una Cache completamente associativa. L'algoritmo di sostituzione qui, nel caso di corrispondenza diretta, non esiste, ma è l'indirizzo mandato dalla CPU che determina quale linea sostituire, in questo caso la 8. Anche se ci fossero altre linee vuote, noi comunque dobbiamo liberare la linea a cui si riferiscono i 7bit.

✳ **Associativa a Insiemi.** Essa è una via di mezzo tra le due Cache viste in precedenza, poiché prende più Cache a Corrispondenza Diretta e le considera come singoli Insiemi/livelli di Associatività, che manterranno la proprietà di essere a Corrispondenza Diretta, ma che uniti formeranno una sola Cache. Dal punto di vista pratico la Cache Associativa ad Insiemi è la migliore, e ci permette di dosare la quantità di Insiemi di Associatività da inserire nella Cache a seconda del risultato che voglio ottenere. Se voglio avere più prestazioni aumento la Associatività, se voglio ridurre il costo riduco il numero di Insiemi/livello di associatività, per trovare un compromesso.

✳Cache in serie. Se prendiamo ad esempio due Cache di dimensioni diverse e le mettiamo in catena, possiamo arrivare per gradi alla RAM. Se nella Cache singola non trovo subito l'elemento che cerco allora dovrò accedere alla RAM, ma se avessi una struttura a più Cache, posso averne una seconda più grande che conterrà più informazioni, anche quelle che eventualmente non sono contenute nella prima Cache. Una volta trovata un'informazione che sta nella seconda Cache e non nella prima, il tempo di accesso sicuramente è più grande di quello nella Cache primaria, ma sicuramente sarà enormemente minore rispetto a quello che impiegheremmo per accedere a tutta la RAM. Possiamo avere molteplici livelli di Cache, sempre costruiti in modo che più ci avviciniamo alla CPU, più la Cache dovrà essere piccola e veloce, il più associativa possibile, con un protocollo di consistenza Write through meno complesso, mentre quelle vicine alla RAM saranno più grosse e più lente, tendenzialmente a corrispondenza diretta se vicine alla RAM e organizzate in protocollo Write back per essere più complessi ma veloci, in modo da contenere tanti dati senza essere troppo costose, per avere un compromesso in tutte le Cache

PIPELINE.

L'organizzazione a pipeline serve a costruire una cpu efficiente e veloce.

Potremmo diminuire la durata dei singoli Ck, ma sarebbe costoso a livello hardware, dunque possiamo invece diminuire i Ck necessari per l'esecuzione di un'istruzione.

Organizzazione a Pipeline, più efficiente rispetto alla MVN, che si basa sul concetto di far svolgere le singole fasi di Fetch, Decode ed Execute separatamente a dispositivi specializzati. L'idea qui è di prendere alcuni Registri del Data Path e staccarli da quelli usati dall'ALU, per andare a formare tre dispositivi:

✳ Unità di fetch. Per fare il Fetch dovrà avere IR e PC, leggere da RAM o Cache l'istruzione e memorizzarla nell'IR, per poi incrementare il PC.

✳ Unità di Decode. Decodifica il valore memorizzato in IR, in modo da preparare l'istruzione per la fase di Execute.

✳ Unità di Execute. Eseguiamo l'istruzione nei Ck che le servono, e poi svuotiamo l'unità per prepararla ad una nuova istruzione.

Nelle MVN (macchine di Von Neumann) succedeva che finché era in corso una delle 3 fasi, le altre due non venivano eseguite. Invece in un sistema a Pipeline, avendo specializzato 3 dispositivi diversi per ogni fase, possiamo, in 1 Ck, svolgere, su istruzioni diverse, le 3 fasi in contemporanea. Infatti, tranne che in casi di istruzioni particolari, come Branch, o all'avvio del sistema, non appena abbiamo l'Esecuzione Sequenziale di un programma, ovvero una serie di codici situati in celle adiacenti, le tre unità lavoreranno simultaneamente, e alla fine di 1Ck, ognuna di esse avrà finito il proprio compito, e passerà alla prossima istruzione. In particolare l'istruzione che era nella fase di Fetch passerà all'unità di Decode, quella nella fase di Decode passerà alla Execute, e quella alla Execute sarà stata terminata, e l'unità di Fetch prenderà in

carico una nuova istruzione adiacente a quella che ha appena mandato all'unità di Decode. Con questa tecnica di ottimizzazione siamo riusciti a parallelizzare le 3 fasi.

Tuttavia abbiamo ipotizzato finora l'esecuzione di programma unicamente sequenziali, senza eventuali salti. In tal caso abbiamo un conflitto perché, ogni volta che viene eseguita un'istruzione di salto, avremo nella Unità di Decode l'istruzione successiva al salto, e in quella di Fetch l'istruzione ancora successiva, nessuna delle quali dev'essere eseguite. Per questo motivo, ogniqualvolta il sistema si accorge di star eseguendo un Branch, deve liberare la pipeline, svuotare le Unità di Decode e di Fetch, e ripartire con la fase di Fetch dell'istruzione a cui siamo saltati. In questo momento dunque Unità di Decode e di Execute rimangono inattive, in attesa dell'istruzione di cui stiamo facendo il Fetch.

Dividiamo le istruzioni di salto in

✳ Salto non Condizionale. In questo caso il Salto viene eseguito sempre. Viene individuato alla fine della fase di Decode, e ciò permette di modificare immediatamente il PC e iniziare con 1Ck di anticipo l'istruzione all'indirizzo in cui stiamo per saltare, in modo da iniziare il Fetch della nuova istruzione quando l'istruzione di salto si trova nella fase di Execute.

✳ Salto Condizionale. In questo caso viene effettuato un controllo su una determinata condizione, che, se verificata, permette l'esecuzione del Salto. Qui non possiamo fare il controllo durante la fase di Decode, perché dobbiamo aspettare che quella sia l'ultima istruzione da eseguire per poter controllare le condizioni del salto.

Un'altra soluzione può essere quella del Salto Ritardato, che consiste nel memorizzare l'istruzione non dove vogliamo eseguire il salto, ma subito prima, in modo che quando facciamo il Decode di questa istruzione, riusciamo già a identificare che si tratta di un salto ritardato, la passiamo alla fase di Execute, e nel mentre possiamo anche fare il Decode dell'istruzione situata immediatamente dopo quella di salto, il Fetch di quella dopo, e via dicendo, poiché vogliamo che queste istruzioni vengano eseguite prima del salto. Insomma questo salto viene "congelato", in modo da essere eseguito dopo un certo Delay Slot, ovvero un ritardo prestabilito, in modo da permettere alle istruzioni ancora da eseguire di essere eseguite prima del salto. L'unica problematica che sorge è che più il Delay Slot è alto, più cresce la probabilità che l'istruzione che vogliamo eseguire prima del salto ritardato sia a sua volta un salto, che quindi andrebbero in conflitto col salto ritardato. Perciò viene introdotta l'istruzione NOP, che banalmente non fa nulla, e dunque permette di perdere 1Ck nel caso l'istruzione successiva di un Salto ritardato sia un salto a sua volta, cosicché venga eseguito solo il Salto Ritardato

(quindi questa non è una vera e propria soluzione ottimale, ma più una "pezza")