Perchè avere uno scheduling dinamico?

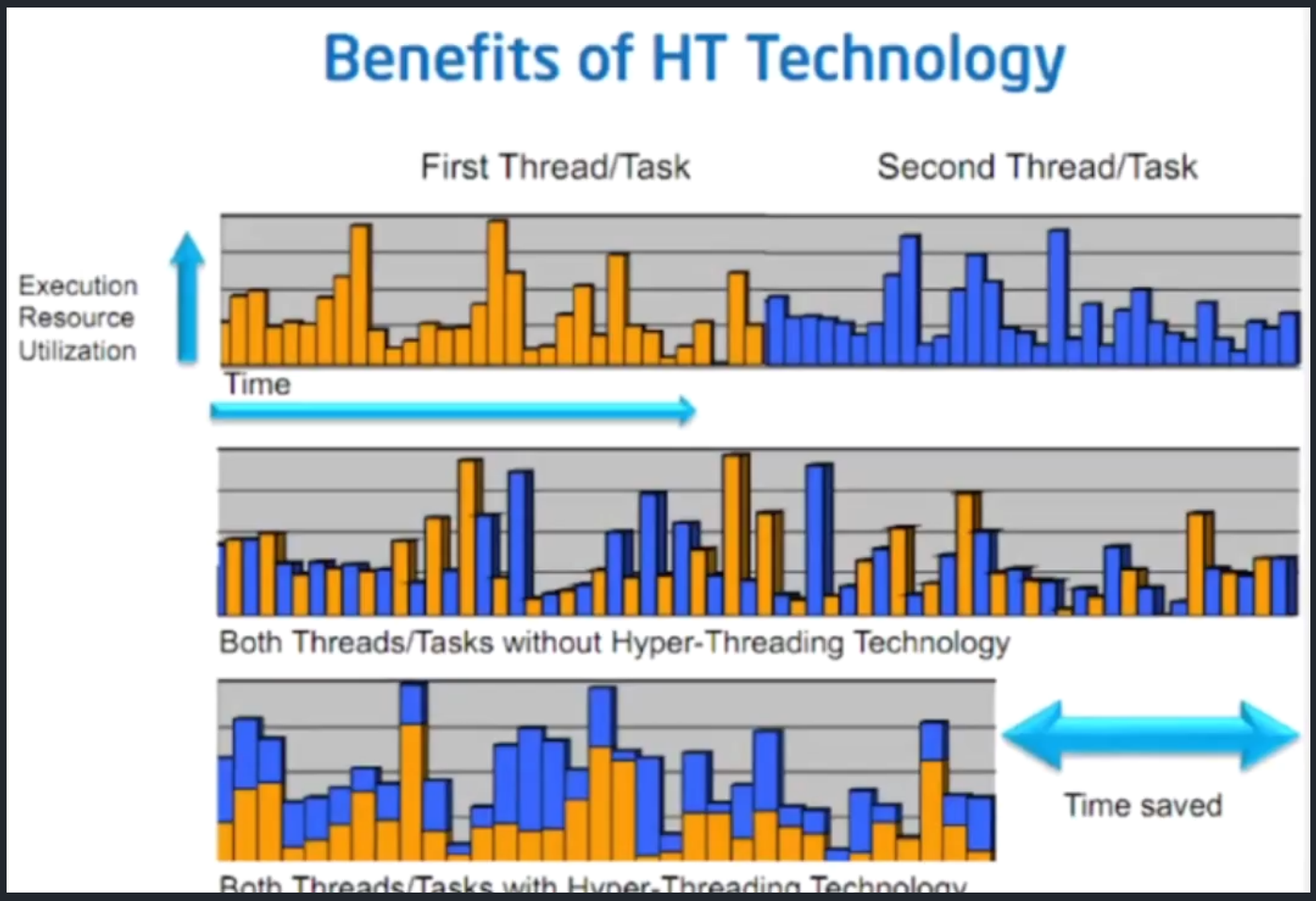
Non tutti gli stalli sono prevedibili, quindi a volte è necessario provvedere quando il codice è già stato compilato ed è in esecuzione

Il multiple issue funziona?

A fronte di un utilizzo intenso di tecnologia (fisicamente) molto spesso la logica dei programmi fa sì che ci siano delle dipendenze che limitano il parallelismo finale. In pratica non si riesce a sfruttare tutto l'hardware a disposizione.

Un problema tipico è quello dell'**aliasing dei puntatori**; in C posso accedere ad una variabile in memoria chiamata "x" o con il nome della variabile o con il **puntatore alla variabile**. Al primo sguardo possono sembrare due cose diverse, ma in realtà puntano alla stessa variabile.

Hyper Threading



I progettisti di intel si accorgono di produrre dei processori molto potenti, ma le dipendenze all'interno del programma non riescono a lanciare in esecuzione tanta roba quanto si vorrebbe, e quindi impegnare tutte le pipe disponibili. L'idea è quella di prendere pipes da più programmi diversi, proprio perchè l'hardware è abbastanza potente da gestire più processi (programmi), in modo da tenere impegnato tutto l'hardware disponibile. AMD chiama questa tecnica **multi threading**.

Un processore in grado di eseguire istruzioni da due processi diversi, e quindi abilitato all'hyper threading, ha **due unità di fetch** (invece di uno), che prendono istruzioni da due programmi separati.

Nell'immagine si vede chiaramente che ci sono dei "tempi morti" in cui il processore è poco attivo; con la tecnica del HT, possiamo **sovrapporre** l'esecuzione di due processi in modo da compensare i tempi morti di un processo con l'esecuzione di un altro.

Power Efficiency

Inizialmente i processori non consumavano molta energia, ma avevano anche una potenza di calcolo molto bassa. Basta guardare **l'i486 del 1989** che non utilizzava **il multiple issue** ed aveva un singolo core.

Il famoso **Intel Pentium pro**, ha un aumento importante della frequenza di clock, dovuto in parte anche al numero di stadi di pipeline e l'utilizzo del multiple issue.

Con l'inizio degli anni 2000, è iniziata la cosiddetta "guerra dei gigaherz", dove Intel ed AMD si combattevano il mercato a colpi di frequenze di clock, arrivando addirittura a 3,6 GHz con l'intel Pentiun 4, dove il numero di stati pipeline erano addirittura 31, con uno spessore **dell'issue** di 3; il processore era ancora a singolo core.

Fallace

Il pipelining è semplice?

No, l'idea di base del pipelining è semplice, ma l'implementazione è molto complicata.

Il pipelining è indipendente dalla tecnologia?

Anche in questo caso no. Il pipelining richiede una tecnologia avanzata (molti transistors), inoltre richiede tecniche di predizione avanzate.

Pitfalls

Se il set di istruzioni è progettato male, il pipelining diventa più difficile; per "progettato male", si intende un set di istruzioni **difficile**.

Se una sola istruzione esegue molte operazioni, è molto difficile far avvenire tutte le operazioni in sequenza su una pipeline; per far funzionare il pipelining con un set di istruzioni complesse, abbiamo bisogno di un **overhead hardware** notevole.

Considerazioni finali

Il pipelining sicuramente ha un forte impatto sul **throughput finale**, mentre la **latenza** non viene ridotta, anzi, dura di più.

Inoltre, il pipelining è affetto da una serie di problemi tra cui **gli hazards**, che possono essere di tipo strutturale, sui dati, e di controllo.

🏁 1:04 04-15 fine capitolo 4

Capitolo 5: gestione della memoria

Principio di località

Un programma in esecuzione riceve una fetta di memoria, ma essa non viene utilizzata tutta in modo uniforme; per la logica del funzionamento dei programmi, essi tendono ad accedere solo **ad una piccola parte del loro spazio indirizzi**.

- **Località temporale**: il programma tende ad accedere più volte sempre alla stessa locazione di memoria; abbiamo un programma composto da area dati ed area istruzioni, ed entrambe le aree sono poste nello stesso spazio indirizzi. Supponiamo di accedere ad una variabile x (non posta in un registro, ma in memoria) e continuo ad incrementarne il valore (magari in un loop). E' ben difficile che una variabile in un programma venga usata una sola volta.

Sono finito su una locazione di memoria dove era posta un'istruzione da eseguire; ritornerò su quella locazione di memoria? Se il programma è dotato di un **loop**, è inevitabile.

Quindi: **è probabile che gli elementi acceduti recentemente saranno acceduti nuovamente**.

- **Località spaziale**: poniamo il caso di aver letto da una locazione di memoria, quanto è probabile la lettura di altre locazioni di memoria molto vicine alla prima? **Per quanto riguarda i dati**, è molto molto probabile (accesso sequenziale, ad esempio gli arrays).
  Per il codice di un programma, **è praticamente certo;** questo perchè le istruzioni di un programma sono sequenziali.

Probabilmente abbiamo già intuito dove si vuole arrivare: se questi dati che visitiamo **molto molto** spesso, li salviamo in una memoria più veloce ed accessibile, riusciamo a velocizzare il tutto.

Usiamo questo principio a nostro vantaggio

Gerarchia delle memorie

Grazie alla gerarchia delle memorie, possiamo suddividere i dati in diverse unità di archiviazione; possiamo, ad esempio, salvare i dati molto grandi sulla memoria più grande e meno costosa possibile (ad esempio i dischi), mentre più un dato è utilizzato frequentemente, più è salvato su una memoria veloce (ad esempio cache e registri).

Le cache sono realizzate con una tecnologia di tipo SRAM, molto pià veloce della DRAM utilizzata per le RAM (che utilizza dei condensatori); sono entrambe memorie volatili.

---

fine lezione 15