Architetture Parallele

- Introduzione
- Multithreading
- Multiprocessori (sistemi a memoria condivisa)
- Multicomputers (sistemi a scambio di messaggi)

1

Parallelismo Esplicito

- Abbiamo finora visto vari modi per cercare di sfruttare il parallelismo implicito presente nelle istruzioni macchina di un programma.
- Questi diversi modi hanno un presupposto comune: il programmatore concepisce il programma che ha scritto per risolvere un certo problema come una sequenza di istruzioni che verranno *eseguite una dopo l'altra dalla* CPU.
- Non sa (o non è tenuto a sapere) come verranno manipolate queste istruzioni dal compilatore (ILP statico) o una volta entrate nella CPU (ILP dinamico)

2

Parallelismo Esplicito

- Se il programma gira troppo lentamente, si può adottare una CPU più veloce, e magari riscrivere meglio il programma, ma se tutto ciò è già stato fatto?
- L'unica possibile soluzione è ricorrere ad una architettura dotata di più unità computazionali, nella speranza che almeno parte della soluzione del problema possa essere parallelizzata.
- Ora il programmatore sa dell'esistenza di una architettura parallela, e può scegliere una soluzione algoritmica in grado di sfruttare opportunamente questa architettura.

3

Limiti del Parallelismo Esplicito

- Del resto, molti problemi hanno una soluzione che è naturale ottenere con un insieme di programmi che girano in parallelo fra loro.
- Tuttavia, (eccetto che in casi molto molto particolari) l'incremento di prestazioni (il cosiddetto **speed-up**) che si può ottenere usando un insieme di CPU su cui far girare in parallelo i vari programmi è **meno che lineare rispetto al numero di CPU (o core) disponibili**.
- Infatti, i programmi che girano in parallelo dovranno prima o poi sincronizzarsi per mettere in comune i dati elaborati da ciascuno; oppure sarà necessaria una fase comune di inizializzazione prima di far partire i vari programmi in parallelo: c'è comunque sempre una parte di lavoro che non può essere svolta in parallelo a tutte le altre operazioni.

4

Limiti del Parallelismo Esplicito

- Consideriamo ad esempio il seguente comando di compilazione:
  - gcc main.c function1.c function2.c –o output
- Supponiamo di lanciare la compilazione su una macchina monoprocessore, e che ci vogliano:
  - 3 secondi per compilare main.c
  - 2 secondi per compilare function1.c
  - 1 secondo per compilare function2.c
  - 1 secondo per linkare gli oggetti (main.o, function1.o, function2.o)
- per un totale di 7 secondi.

5

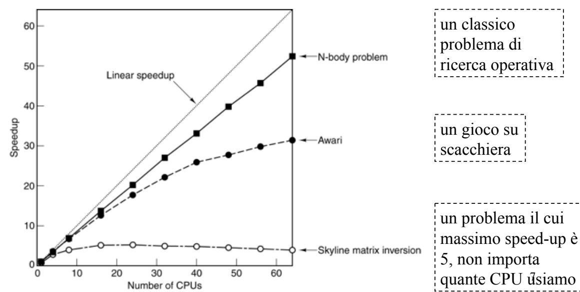
Limiti del Parallelismo Esplicito

- se abbiamo a disposizione 3 CPU, i tre sorgenti possono essere compilati in parallelo per generare i corrispondenti moduli oggetto.
- Quando gli oggetti sono stati prodotti, possono essere linkati assieme usando uno dei tre processori.
- Ma l'operazione di linking può essere eseguita solo dopo che tutti e tre i file oggetto sono stati prodotti, ossia solo dopo 3 secondi.
- Il tempo necessario per generare l'output sarà quindi 3+1=4 secondi, contro i 7 secondi del sistema monoprocessore, per uno speed-up di 7/4 = 1.75, pur avendo usato un numero triplo di processori.
- Se anche tutte le compilazioni richiedessero un secondo, lo speed-up sarebbe di 4/2 = 2.

6

Limiti del Parallelismo Esplicito

- In generale, diversi problemi computazionali hanno una diversa risposta (formalmente: un diverso *speed-up*) al tentativo di risolverli distribuendo il lavoro su più CPU (Tanenbaum, Fig. 8.47):



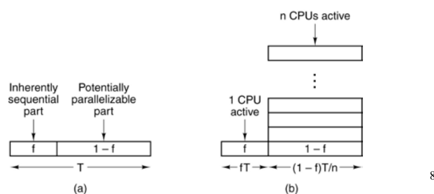
un classico problema di ricerca operativa

un gioco su scacchiera

un problema il cui massimo speed-up è 5, non importa quante CPU usiamo

Limiti del Parallelismo Esplicito

- Di fatto, qualsiasi programma è composto (anche) di una parte di operazioni sequenziali che non possono essere in ogni caso parallelizzate. Cerchiamo di formalizzare il problema:
- Sia P un programma che gira in tempo T su un processore, con f = la frazione di T dovuta a codice sequenziale e (1-f) la frazione di T dovuta a codice parallelizzabile (Tanenbaum, Fig. 8.48):



8

Limiti del Parallelismo Esplicito

- Allora, il tempo di esecuzione dovuto alla parte parallelizzabile, passa da (1-f)T a (1-f)T/n se sono disponibili n processori.
- lo speed-up che si ottiene è poi dato dal tempo di esecuzione su un'unica CPU diviso per il tempo di esecuzione su n CPU:

$$\text{speed-up} = \frac{fT + (1 - f)T}{fT + (1 - f)T/n} = \frac{n[fT + (1 - f)T]}{nfT + (1 - f)T}$$

$$= \frac{nT}{T(1 + nf - f)} = \frac{n}{1 + (n-1)f} =$$ **legge di Amdahl**

9

Limiti del Parallelismo Esplicito

- La legge di Amdahl ci dice quindi che possiamo ottenere uno speed-up perfetto, pari al numero di CPU usate, solo per f = 0.
- Ad esempio, se vogliamo avere uno speed-up di 80, usando 100 CPU, quale frazione f della computazione originale può essere sequenziale?
- 80 = 100 /(1 + 99f); f = 20/(80 x 99) = 0.0025252525
- Ossia, circa solo lo 0.25% del tempo di computazione originale può essere dovuto a codice sequenziale.
- Possiamo in qualche modo applicare la legge di Amdhal anche alle architetture single core che implementano qualche forma di ILP e il multiple issue?

10

Limiti del Parallelismo Esplicito

- All'atto pratico, i principali problemi del parallelismo esplicito sono sostanzialmente due: uno software e uno hardware.

1. **La quantità limitata di parallelismo presente nei programmi**, o per lo meno la quantità di parallelismo che si è in grado di esplicitare, e quindi di sfruttare.

- Lo studio di algoritmi paralleli è infatti un campo di ricerca tutt'ora molto attivo, proprio per le sue potenzialità.

2. **Gli elevati costi delle comunicazioni tra processori e memoria**, che possono aumentare enormemente il costo di un cache miss o di una sincronizzazione tra processi che girano su CPU diverse.

- Questi costi dipendono ovviamente dal tipo di architettura adottata e dal numero di CPU coinvolte, ma in generale sono molto più elevati che in un sistema monoprocessore (sono invece un po' mitigati nei proessori multi-core, di cui parleremo nel prossimo capitolo).

11

Parallelismo Esplicito

- Naturalmente, per molte applicazioni avere uno speed-up meno che lineare, ad esempio di n usando 2n processori è comunque più che accettabile, visto il costo sempre più basso delle CPU.
- Inoltre, l'aumento della potenza computazionale non è l'unica motivazione che porta a progettare architetture formate da più CPU:

a) La presenza di più processori aumenta l'affidabilità del sistema: se anche uno si guasta, altri possono svolgere il lavoro al suo posto.

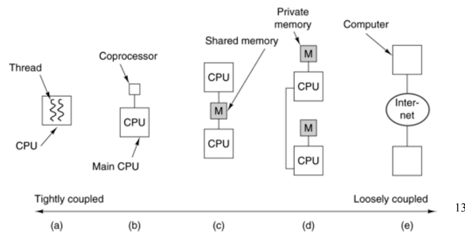
b) Servizi che per loro natura sono forniti su ampia scala geografica, devono essere implementati con una architettura distribuita. Se il sistema fosse centralizzato in un unico nodo, l'accesso di tutte le richieste a quest'unico nodo costituirebbe probabilmente un collo di bottiglia in grado di rallentare enormemente il servizio fornito.

12

Parallelismo Esplicito

- A grandi linee, distinguiamo tre tipi di architetture parallele esplicite (Tanenbaum, Fig. 8.1):

1. multi-threading (a)
2. Sistemi a memoria condivisa (b,c)
3. Sistemi a memoria distribuita (d,e)



13

Multithreading

- Introduzione
- Fine-grained Multithreading
- Coarse-grained Multithreading
- Simultaneous Multithreading
- Multithreading nei processori Intel

14

Multi-Threading

- Una CPU multithreaded non è, in senso stretto, una architettura parallela, in quanto il multithreading è realizzato appunto con un'unica CPU, ma porta già il programmatore a concepire e sviluppare le sue applicazioni come formate da un insieme di programmi che possono girare virtualmente in parallelo: i thread, appunto.
- Se questi programmi vengono fatti girare su una CPU che supporta il multi-threading potranno sfruttarne al meglio le caratteristiche architetturali.
- Nota: il multi-threading è nato su CPU single core, ma ora le CPU sono formate ciascuna da 2 o più core, ciascuno dei quali implementa il multi-threading. Nel seguito di questo capitolo per CPU continuiamo ad intendere un processore single core.

15

Multi-Threading

- L’idea del multithreading nasce dalla constatazione di un problema di fondo presente in qualsiasi CPU pipelined: un cache miss produce una “lunga” attesa necessaria per recuperare l’informazione mancante in RAM. Se non c’è un’altra istruzione indipendente da poter eseguire, la pipeline va in stall.
- Una soluzione per non sprecare inutilmente cicli di clock in attesa del dato mancante è il multithreading: permettere alla CPU di gestire più peer-thread allo stesso tempo: se un thread è bloccato la CPU ha ancora la possibilità di eseguire istruzioni di un altro thread, in modo da tenere le varie unità funzionali comunque occupate.
- Però, perché non è possibile estendere facilmente l’idea all’esecuzione di thread appartenenti a processi diversi?

16

Multi-Threading

- Per implementare il multithreading, la CPU deve poter gestire lo stato della computazione di ogni singolo thread.
- Ci devono quindi essere almeno un Program Counter e un set di registri separato per ciascun thread.
- Inoltre, il thread switch deve essere molto più efficiente del process switch, che richiede di solito (essendo effettuato almeno in parte a livello software) centinaia o migliaia di cicli di clock
- Esistono due tecniche di base per il multithreading:

1. fine-grained multithreading
2. coarse-grained multithreading

- NB: nel seguito semplifichiamo il termine *peer-thread* con *thread.* Inoltre, **ci concentriamo inizialmente su processori “single-issue”**

Fine-grained Multi-Threading

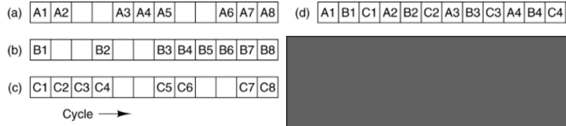
1. **Fine-grained Multithreading:** lo switching tra i vari peer-thread avviene ad ogni istruzione, indipendentemente dal fatto che l’istruzione del thread in esecuzione abbia generato un cache miss.

- Lo “scheduling” tra le istruzioni dei vari peer-thread avviene secondo una politica round robin, e la CPU deve essere in grado di effettuare lo switch praticamente senza overhead, che altrimenti sarebbe inaccettabile
- Se vi è un numero sufficiente di peer-thread, è possibile che ve ne sia sempre almeno uno non in stall, e la CPU può essere mantenuta sempre attiva.

18

Fine-grained Multi-Threading

- (a)-(c) tre threads con i relativi stall (i riquadri vuoti).
  (d) Fine-grained multithreading. Ogni riquadro rappresenta un ciclo di clock, e assumiamo per semplicità che ogni istruzione possa essere completata in un ciclo di clock, a meno di stall.
  (Tanenbaum, Fig. 8.7)



- In questo esempio, con 3 thread si riesce a mantenere la CPU sempre occupata, ma che succede se lo stall di A2 dura 3 o più cicli di clock?

19

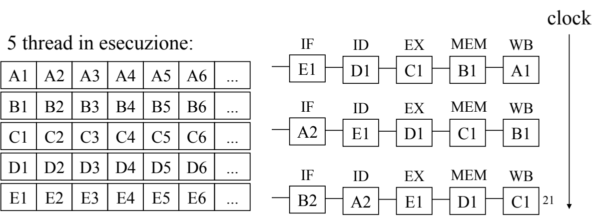
Fine-grained Multi-Threading

- Lo stalling della CPU può essere dovuto ad un cache miss, ma anche ad una true data dependence, o a un branch: le tecniche di ILP dinamico che abbiamo visto non garantiscono sempre che lo stall della pipeline possa essere evitato.
- Invece, col fine-grained multithreading, in una architettura pipelined, se:
  - la pipeline ha *k* stage,
  - ci sono almeno *k* peer-thread da eseguire,
  - e la CPU è in grado di eseguire uno switch tra thread ad ogni ciclo di clock
- allora non ci può essere mai più di una istruzione per thread nella pipeline in un dato istante, non si verificano problemi di dipendenze, e la pipeline non va mai in stall (in realtà dovremmo fare una assunzione in più…).

20

Fine-grained Multi-Threading

- Fine-grained multithreading in una CPU con pipeline a 5 stadi: non ci sono mai due istruzioni dello stesso thread contemporaneamente nella pipeline. Se le istruzioni possono essere eseguite out of order, allora anche in presenza di cache miss può essere possibile mantenere la CPU in piena attività.



21

Fine-grained Multi-Threading

- A parte la necessità di riuscire ad implementare il context switch tra thread in maniera molto efficiente, lo scheduling fine-grained fra i thread ad ogni istruzione fa sì che un thread venga rallentato anche quando potrebbe proseguire la sua esecuzione perché non sta generando alcun stall.
- Inoltre, ci potrebbero essere meno thread che stage della pipeline (anzi, è questo il caso più frequente), per cui può non essere così facile mantenere la CPU sempre occupata.
- Tenendo presente questi problemi, un approccio complementare viene adottato nel multithreading coarse-grained (a “grana grossa”)

22

Coarse-grained Multi-Threading

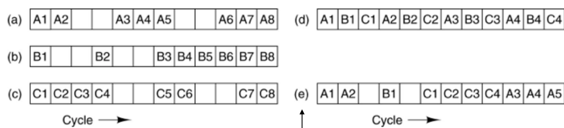
2. **Coarse-grained Multithreading**: lo switch avviene solo quando il thread in esecuzione genera uno stall, provocando così lo spreco di un ciclo di clock.

- A questo punto viene effettuato lo switch ad un altro thread. Quando anche questo thread genererà uno stall, verrà schedulato un terzo thread (o eventualmente si tornerà al primo) e così via.
- Questo approccio spreca potenzialmente più cicli di clock del fine-grained, perché comunque lo switch avviene solo quando si è verificato lo stall.
- ma se ci sono pochi thread attivi (anche solo due), questi possono già essere sufficienti per tenere la CPU sufficientemente occupata.

23

Coarse e Fine-grained Multi-Threading

- (a)-(c) tre threads con i relativi stall (i riquadri vuoti).
  (d) Fine-grained multithreading.
  (e) Coarse-grained multi-threading (Tanenbaum, Fig. 8.7)



in questa figura c’è un errore?

24

Coarse e Fine-grained Multi-Threading

- Nell'esempio del lucido precedente, il fine-grained multithreading sembra funzionare meglio, ma non sempre è così.
- In particolare, non è pensabile che lo switch tra thread possa avvenire senza alcuna perdita di tempo.
- Allora, se le istruzioni dei vari thread non generano stall molto frequentemente, uno scheduling coarse-grained può essere anche più vantaggioso di uno fine-grained, in cui comunque ad ogni ciclo di clock si paga l'overhead del context switch tra peer-thread (tale overhead è molto limitato, ma comunque non è nullo).

25

Coarse e Fine-grained Multi-Threading

- Coarse e Fine grained multi-threading non vi fanno pensare a due concetti fondamentali studiati nel corso di Sistemi Operativi?

26

Medium-grained Multi-Threading

3. **Medium-grained multithreading**: una via di mezzo tra il fine- e il coarse- grained multithreading consiste nell'eseguire lo switch tra thread solo quando quello in esecuzione sta per eseguire una istruzione ***che potrebbe generare uno stall di lunga durata***, come ad esempio una load (che potrebbe richiedere un dato non in cache), o un branch (perchè?).

- L'istruzione viene avviata, ma il processore esegue in ogni caso lo switch ad un altro thread. In questo modo, non si spreca neanche il ciclo di clock dovuto all'eventuale stall della load eseguita (come accadrebbe col multithreading coarse-grained).

27

Multi-Threading

- Ma come si fa a sapere a quale thread appartiene una qualsiasi istruzione nella pipeline?
- Nel caso del fine-grained MT, l'unico modo è di attaccare un *thread identifier* ad ogni istruzione, ad esempio l'ID univoco associato a quel thread all'interno dell'insieme di peer threads a cui appartiene.
- Nel coarse-grained MT si può adottare la stessa soluzione, oppure si può anche svuotare la pipeline ad ogni thread switch: in questo modo le istruzioni di un solo thread alla volta sono nella pipeline, e quindi si sa sempre a quale thread appartengono.
- Naturalmente, ciò ha senso solo se lo switch avviene ad intervalli molto maggiori del tempo necessario a svuotare la pipeline.

28

Multi-Threading

- Infine, le istruzioni dei vari thread in esecuzione devono essere (per quanto possibile) tutte contemporaneamente nella cache delle istruzioni, altrimenti ogni context switch tra thread produce un cache miss, e si perde qualsiasi vantaggio ad usare i thread.

29

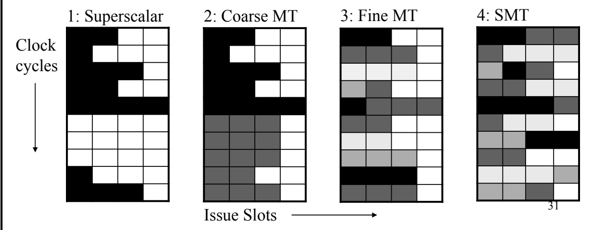
Simultaneous Multi-Threading e Multiple Issue

- Le moderne architetture superscalari, multiple issue e a scheduling dinamico della pipeline danno la possibilità di sfruttare contemporaneamente il parallelismo insito nelle istruzioni di un programma (ILP) con il parallelismo insito in un insieme di peer threads: il thread level parallelism (TLP):
- **ILP + TLP = Simultaneous Multi-Threading (SMT)**
- La ragione per implementare l'SMT risiede nell'osservazione che le moderne CPU multiple-issue hanno più unità funzionali di quante siano mediamente sfruttabili dal singolo thread in esecuzione.
- Sfruttando il register renaming e lo scheduling dinamico, istruzioni appartenenti a thread diversi possono essere eseguite insieme.

30

Simultaneous Multi-Threading

- nell'SMT, ad ogni ciclo di clock vengono avviate più istruzioni, potenzialmente appartenenti a thread diversi, aumentando così l'utilizzo delle varie risorse della CPU (H-P5, Fig. 3.28: ogni slot di un certo colore rappresenta una istruzione di un thread).



31

Simultaneous Multi-Threading

1. Nelle CPU superscalari senza multithreading, il multiple issue può venire vanificato dalla mancanza di sufficiente parallelismo tra le istruzioni di ogni thread, e/o da un lungo stall (ad esempio un cache miss su L3) che lasciano l'intero processore idle.
2. Nel coarse-grained MT, i lunghi stall sono mascherati dal passaggio ad un altro thread, ma la mancanza di parallelismo tra le istruzioni di ciascun thread limita il grado di utilizzo delle risorse della CPU (ad esempio, non possono essere usati tutti gli slot di issue disponibili)
3. Anche nel fine-grained MT, la mancanza di ILP in ciascun thread limita di utilizzo delle risorse della CPU
4. SMT: le istruzioni appartenenti a thread diversi sono (quasi) certamente indipendenti, se possiamo lanciarle assieme aumentiamo il grado di utilizzo delle risorse sella CPU.

32

Simultaneous Multi-Threading

- Anche nell'SMT comunque, non si riesce sempre ad avviare il massimo numero possibile di istruzioni per ciclo di clock, a causa del numero limitato di unità funzionali disponibili e di stazioni di prenotazione disponibili, della capacità della cache di istruzioni di fornire le istruzioni dei vari thread, e del numero di thread presenti.
- E' evidente, che l'SMT può essere adeguatamente supportato solo se sono disponibili registri rinominabili in abbondanza.
- Inoltre, nel caso di una CPU che supporti la speculazione, sarà importante avere (almeno logicamente) un ROB distinto per ogni thread, in modo da gestire in modo indipendente l'operazione di commit.

33

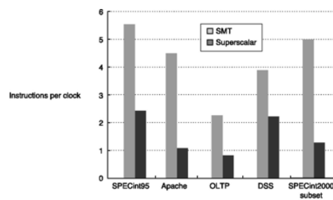
Simultaneous Multi-Threading

- Costruire un processore in grado di sfruttare a pieno l'idea dell'SMT è sicuramente un problema complesso, ma conviene?
- Una simulazione: consideriamo una CPU superscalare e multithreaded con le seguenti caratteristiche:
  - Pipeline a 9 stage
  - 4 unità floating point general purpose
  - 2 ALU
  - possibilità di eseguire fino a 4 load o store per ciclo
  - 100 interi + 100 FP registri rinominabili
  - fino a 12 commit per ciclo
  - 128k + 128k cache di primo livello
  - 16MB cache di secondo livello
  - BTB da 1k
  - 8 contesti per l'SMT (8 PC, 8 blocchi di registri architetturali)
  - fino a 8 istruzioni lanciate per ciclo di clock, da 2 diversi contesti

34

Simultaneous Multi-Threading

- Questo ipotetico processore ha leggermente più risorse di un moderno processore reale, e soprattutto è in grado di gestire fino ad 8 thread contemporaneamente. Ma la capacità di gestire l'SMT sarebbe effettivamente vantaggiosa? Ecco i risultati per alcuni benchmark di applicazioni concorrenti, in termini di di numero di istruzioni completate per ciclo di clock (H-P3, Fig. 6.46):



35

© 2003 Elsevier Science (USA). All rights reserved.

Simultaneous Multi-Threading

- Domanda: quando parliamo di multithreading, pensiamo di solito ad un insieme di peer thread le cui istruzioni possono essere contemporaneamente in esecuzione in una CPU multithreaded.
- Avrebbe senso pensare di eseguire contemporaneamente le istruzioni appartenenti a processi diversi?
- Dovremmo aggiungere qualche risorsa particolare?

36

Il Multi-Threading Intel

- Il multithreading è stato introdotto dalla Intel nello Xeon già nel 2002, e successivamente nel Pentium 4 nella versione a 3,06 GHz con il nome di **hyperthreading**. Il nome è altisonante, ma in realtà viene supportata l'esecuzione di due thread in modalità SMT.
- Secondo quanto dichiarato dalla Intel, i progettisti avevano visto che per aumentare ulteriormente le prestazioni della CPU, il multithreading era la soluzione più semplice: per avere un secondo thread in esecuzione che sfruttasse le risorse della CPU che altrimenti rimanevano inutilizzate era sufficiente aumentare la dimensione dell'area della CPU del 5%.
- Secondo i benchmark Intel, questo permette di aumentare le prestazioni della CPU di circa il 25% -- 30%.

37

Il Multi-Threading Intel

- Dal punto di vista del Sistema Operativo, un processore multithreaded appare come un doppio processore, in cui le due CPU condividono cache e RAM: se due applicazioni possono girare indipendentemente e condividono lo stesso spazio di indirizzamento, possono essere eseguite in parallelo nei due thread.
- Ad esempio, un programma di editing di filmati, può permettere di specificare dei filtri da applicare ad ogni frame di un filmato. Il programma può allora essere fatto di due thread che manipolano rispettivamente i fotogrammi pari e dispari del filmato, e i due thread possono lavorare in parallelo.

38

Il Multi-Threading Intel

- Siccome due thread possono usare contemporaneamente la CPU, occorre adottare delle strategie per fare in modo che entrambi i thread possano utilizzare le varie risorse della CPU.
- Intel individua 4 diverse strategie per la condivisione delle risorse tra due thread.
- **Duplicazione delle risorse.** Ovviamente, alcune risorse devono essere duplicate, per poter gestire i due thread: c'è bisogno di due program counter e di due tavole di mappatura dei registri visibili al livello ISA sui registri rinominabili (in modo da avere due set di registri indipendenti per i due thread). E' questa duplicazione responsabile dell'aumento del 5% della superfice del processore.

39

Il Multi-Threading Intel

- **Partizionamento vero e proprio delle risorse.** alcune risorse hardware sono partizionate rigidamente tra i due thread. In altre parole, ogni thread può usare esattamente la metà della risorsa. E' questo il caso della coda delle microistruzioni (le uops) che attendono di essere instradate alle varie stazioni di prenotazione, e del ROB (chiamato "retirement queue" nella terminologia Intel).

- Questa forma di partizionamento può ovviamente generare una sottoutilizzazione delle risorse gestite in questo modo, nel caso in cui un thread non usi tutta la sua parte di risorsa, che potrebbe essere sfruttata dall'altro thread.

40

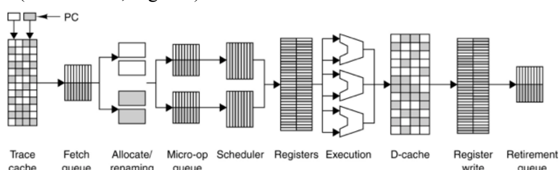
Il Multi-Threading Intel

- **Condivisione vera e propria delle risorse.** La risorsa hardware è completamente condivisa. Il primo thread che si impossessa della risorsa la usa, e l'altro thread deve attendere il suo turno.
- Questa forma di gestione delle risorse risolve il problema di una risorsa idle mentre c'è un thread che vorrebbe usarla. Ovviamente nasce il problema opposto: un thread potrebbe venire rallentato dal fatto che l'altro occupa completamente la risorsa.
- Per questa ragione, nei processori Intel le uniche risorse completamente condivise sono quelle presenti in abbondanza, per cui si ritiene che non si possano verificare problemi di "starvation" per un thread, ad esempio le cache lines.

41

Il Multi-Threading Intel

- **Condivisione controllata delle risorse (threshold sharing).** In sostanza, un thread può utilizzare la risorsa in modo dinamico, ma solo fino ad un certo massimo, in modo che ne rimanga sempre una parte (che può essere però meno della metà) all'altro thread.
- Ad esempio, lo scheduler che invia le uops alle varie stazioni di prenotazione è gestito in questo modo.
- La condivisione delle risorse nella pipeline del Pentium 4 (Tanenabum, Fig. 8.9).



Il Multi-Threading Intel

- E' evidente che la condivisione dinamica con limite massimo delle varie risorse richiede un monitoraggio a run-time di tale risorse, e quindi dell'hardware aggiuntivo, e quindi dell'overhead computazionale.
- Inoltre, possono sorgere dei problemi nell'uso delle risorse totalmente condivise. Una di queste è la memoria cache. la piena condivisione ne semplifica la gestione, ma cosa accade se ciascun thread ha bisogno di almeno 3/4 delle cache lines per poter essere eseguito efficientemente?
- Un numero molto alto di cache miss, che non si verificherebbe se un solo thread alla volta fosse fatto girare (oppure, sarebbe più efficiente usare un multithreading coarse-grained?)

43

CPU Multi-Threading

- Intel ha abbandonato la tecnologia hyperthreading nel passaggio ai nuovi processori dual core (ricordate che sono basati su una versione aggiornata della microarchitettura P6, che non supporta il multithreading).
- L'hyperthreading è stato poi reintrodotto a partire dal 2008 con l'architettura Nehalem.
- Alcuni altri processori che implementano il multithreading sono:
- IBM Power 7 (2010) che implementa l'SMT con quattro thread contemporaneamente attivi;
- I processori UltraSPARC, da T2 in poi, supportano il fine-grained multithreading con 8 thread per core. L'UltraSPARC T1 implementa 4 thread per core.

44

Caso di studio: SUN UltraSPARC T1

- I processori UltraSPARC implementano il fine-grained multi-threading, e sono specificamente focalizzati sullo sfruttamento del parallelismo a livello di thread (thread-level parallelism), anziché sullo sfruttamento dell'ILP (andate a rivedere le caratteristiche del Sun T3 sui lucidi dell 'ILP dinamico parte 2)
- Il Sun T1 è stato introdotto nel 2005, è formato da 8 cores ognuno dei quali gestisce 4 threads. La pipeline è single issue a 6 stadi. Di fatto è identica alla pipeline della MIPS vista a inizio corso, con uno stadio in più per il thread switching.
- Lo switch tra thread avviene ad ogni ciclo di clock, saltando quei thread che sono idle a causa di un cache miss o perché il codice corrispondente è in stato di waiting.

45

Caso di studio: SUN UltraSPARC T1

- Il processore è quindi idle solo se tutti quattro i thread sono idle. Le Load e i branch di un thread producono un ritardo di tre cicli di clock che può essere mascherato solo se ci sono altri thread attivi.
- Ecco qui sotto altre caratteristiche del Sun T1 (H-P5, Fig. 3.29)

Characteristic	Sun T1
Multiprocessor and multithreading support	Eight cores per chip; four threads per core. Fine-grained thread scheduling. One shared floating-point unit for eight cores. Supports only on-chip multiprocessing.
Pipeline structure	Simple, in-order, six-deep pipeline with three-cycle delays for loads and branches.
L1 caches	16 KB instructions; 8 KB data. 64-byte block size. Miss to L2 is 23 cycles, assuming no contention.
L2 caches	Four separate L2 caches, each 750 KB and associated with a memory bank. 64-byte block size. Miss to main memory is 110 clock cycles assuming no contention.
Initial implementation	90 nm process; maximum clock rate of 1.2 GHz; power 79 W; 300 M transistors; 379 mm^2 die.

46

Caso di studio: SUN UltraSPARC T1

- Se consideriamo un singolo core, il CPI ideale di ciascun thread sarebbe 4, a significare che quel thread consuma un ciclo di clock ogni 4. Il CPI del core sarebbe invece 1. Ecco invece quali sono i CPI per tre diversi benchmark (H-P5, Fig. 3.33):

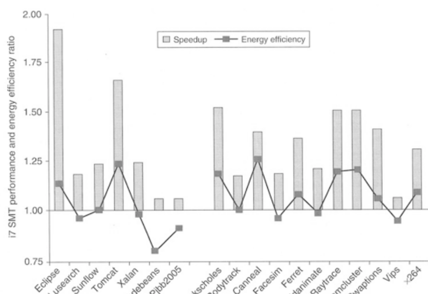
Benchmark	Per-thread CPI	Per-core CPI
TPC-C	7.2	1.80
SPECJBB	5.6	1.40
SPECWeb99	6.6	1.65

- Questi valori sono comparabili a quelli di molti cores del 2005 che implementavano un ILP dinamico "aggressivo". La presenza di 8 cores (contro i 2 o 4 di altre CPU) rendeva la T1 superiore a molti altri processori dell'epoca, in particolare per le applicazioni intere.

47

Caso di studio: Intel Core i7

Ecco lo speed-up ottenuto su uno dei core di una CPU i7 nel passare da un solo thread a 2, per diversi benchmarks. Il dato "Energy efficiency" ci dice se l'introduzione del multi-threading è vantaggiosa dal punto di vista dell'energia consumata. Un valore superiore a 1.0 significa che il multithreading riduce il tempo di esecuzione più di quanto aumenti i consumi. (H-P5, Fig. 3.35)



Osservazioni conclusive

- E' improbabile che nel futuro si riuscirà a sfruttare ulteriormente l'ILP per aumentare il potere computazionale dei processori.
- In questi ultimi anni, il multiple issue è rimasto pressoché costante, mentre è aumentato sensibilmente il numero di transistors usati per le cache, per la duplicazione dei cores, e per il multi-threading, che ormai sembra una tecnica consolidata su cui si focalizzano i costruttori (H-P5, Fig. 3.47)

	Power4	Power5	Power6	Power7
Introduced	2001	2004	2007	2010
Initial clock rate (GHz)	1.3	1.9	4.7	3.6
Transistor count (M)	174	276	790	1200
Issues per clock	5	5	7	6
Functional units	8	8	9	12
Cores/chip	2	2	2	8
SMT threads	0	2	2	4
Total on-chip cache (MB)	1.5	2	4.1	32.3