Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesina di Laurea Triennale

TECNICHE DI PARALLELISMO NELL'HARDWARE

Laureando: Filippo Longo

Relatore: Sergio Congiu

26/03/2010

Anno Accademico 2009/2010

Indice

1. Introduzione

Si può definire Parallelismo, quella proprietà di cui godono due fenomeni, avvenimenti o situazioni paralleli, che presentano cioè corrispondenze reciproche o che avvengono contemporaneamente, ma senza interferenze.[1]
In questa trattazione si considera il Parallelismo temporale delle operazioni svolte da un calcolatore, individuando così l'argomento che va sotto il nome di Elaborazione Parallela: una forma di calcolo nella quale più elaborazioni vengono svolte simultaneamente, sfruttando il principio che grandi problemi spesso possono essere divisi in sottoproblemi più piccoli, questi ultimi risolti in concorrenza.[2]
Si ricorda che un insieme di attività sono concorrenti quando vengono eseguite simultaneamente, e possono insorgere interazioni (dipendenze) tra le stesse.

L'obiettivo del parallelismo consiste dunque nel miglioramento delle prestazioni di elaborazione, tramite l'aumento del numero di operazioni compiute in un intervallo di tempo.

Esistono diverse tecniche di parallelismo, che utilizzano implementazioni software oppure hardware, e non necessariamente le une escludono le altre: possono infatti venire impiegate metodologie di entrambi i tipi in una stessa macchina, al fine di raggiungere risultati migliori.
Nel seguito si porrà l'accento sulle strategie che fanno uso di hardware, accennando rapidamente le tecniche software laddove sia opportuno.

---

[1] Sabatini, Coletti. Dizionario della Lingua Italiana.

[2] Wikipedia (http://en.wikipedia.org/wiki/Parallel_computing) , Almasi & Gottlieb (1989). Highly Parallel Computing.

2. Tipologie di Parallelismo

Sono due le categorie principali di parallelismo sfruttato dai calcolatori:
il *parallelismo a livello di istruzione* (Instruction-level parallelism, *ILP*), che è la possibile sovrapposizione dell'esecuzione di istruzioni, valutabili in modo parallelo nel tempo, e il *parallelismo a livello di thread* (Thread-level parallelism, *TLP*), che sfrutta l'indipendenza tra *flussi* (*thread*) di esecuzione distinti.
Un thread possiede le istruzioni e i dati (il cosiddetto "*stato*", che comprende anche i registri, il PC ecc…) necessari al proprio funzionamento e può essere un programma a sé stante, oppure un processo facente parte di un programma parallelo, che risulta essere composto da un insieme di processi (talvolta il termine thread viene associato solo a quest'ultimo caso).

Esaminando queste tipologie, si può notare come l'ILP sia un parallelismo di tipo *implicito*, "trasparente" al programmatore, che può continuare a concepire il programma come una sequenza di operazioni eseguite una dopo l'altra e non deve preoccuparsi di rendere evidente il parallelismo tra le istruzioni, poiché è il processore che sopperisce a tale compito di individuazione; inoltre, essendo a livello di istruzioni è un parallelismo definito ad *un basso livello* nelle applicazioni.
D'altro canto il TLP è *esplicito*, in quanto deriva dall'uso di più flussi di esecuzione, che sono palesemente paralleli per loro natura ed è un parallelismo che si presenta ad *alto livello* in un'applicazione.

All'interno di un *blocco elementare* (basic block: una sequenza di codice che non presenta salti in ingresso tranne il salto iniziale e salti in uscita tranne il salto finale) la quantità di ILP è bassa poiché è facile che le istruzioni dipendano

tra loro; per ottenere migliori prestazioni è necessario servirsi dell'ILP tra più blocchi di base.

Il procedimento più comune è adoperare un altro importante tipo di parallelismo: il *parallelismo a livello di ciclo* (loop-level parallelism), che deriva dall'indipendenza fra le iterazioni di un ciclo.

Per accrescere l'ILP disponibile, si sovrappongono istruzioni provenienti da iterazioni differenti e quindi prive di vincoli tra loro; le tecniche che eseguono questa conversione da livello di ciclo a livello di istruzione, operano tramite uno "*srotolamento*" del ciclo (*loop unrolling*), sia staticamente per mezzo del compilatore, che duplica un certo numero di volte il ciclo e sistema di conseguenza il codice alla fine dell'iterazione, sia dinamicamente grazie all'hardware (vedi scheduling dinamico).

3. Parallelismo a livello di Istruzione

Per sfruttare l'ILP si possono seguire due approcci distinti: Hardware, mediante l'uso tecniche dinamiche durante l'esecuzione dei programmi, e Software, che impiega strategie di compilazione. Entrambi sono fondati sul pipelining.

Il Pipelining

Il *Pipelining* è una tecnica implementativa che sfrutta il parallelismo esistente tra le azioni richieste per l'esecuzione delle istruzioni, al fine di ridurre il tempo complessivo di elaborazione, o allo scopo di aumentare il throughput: numero di istruzioni eseguite per ciclo di clock (notare che il pipelining non riduce il tempo necessario all'esecuzione di un'istruzione, poiché il beneficio ricade sulla somma totale del tempo impiegato per eseguire un insieme di istruzioni).

Consta nella suddivisione dell'esecuzione di un'istruzione in più *fasi* (o *stadi),* ciascuna delle quali impiega un ciclo di processore; i diversi stadi operano in contemporanea, eseguendo differenti parti di diverse istruzioni, e permettendo così di inserire nel *pipeline* (*tubatura*), una nuova istruzione ad ogni ciclo.

La lunghezza del ciclo di clock è determinata dalla fase più lenta, poiché tutti gli stadi devono procedere simultaneamente, e questo pone in luce l'importanza di bilanciare gli stadi del pipeline affinché alcuni di essi non debbano rimanere inattivi troppo a lungo, mentre attendono che lo stadio più lento sia completato.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figura 1 : Semplice pipeline a 5 stadi. Ad ogni ciclo di clock viene immessa un'istruzione nel pipeline, in questo modo a regime e in presenza di condizioni ideali, il pipeline permette di raggiungere un numero di CPI pari a 1 (un ciclo di clock per istruzione). I 5 stadi del pipeline vengono indicati con le rispettive abbreviazioni: Instruction Fetch, Instruction Decode, Execution, Memory Access, Write Back.

L'unità di calcolo (o *data path*, "percorso sui dati"), viene partizionata in settori distinti (costituiti da un insieme di unità funzionali), ognuno dei quali provvede al funzionamento di uno stadio (si vedrà in seguito come risolvere i casi in cui le stesse unità funzionali, servano a due stadi differenti nel medesimo ciclo).

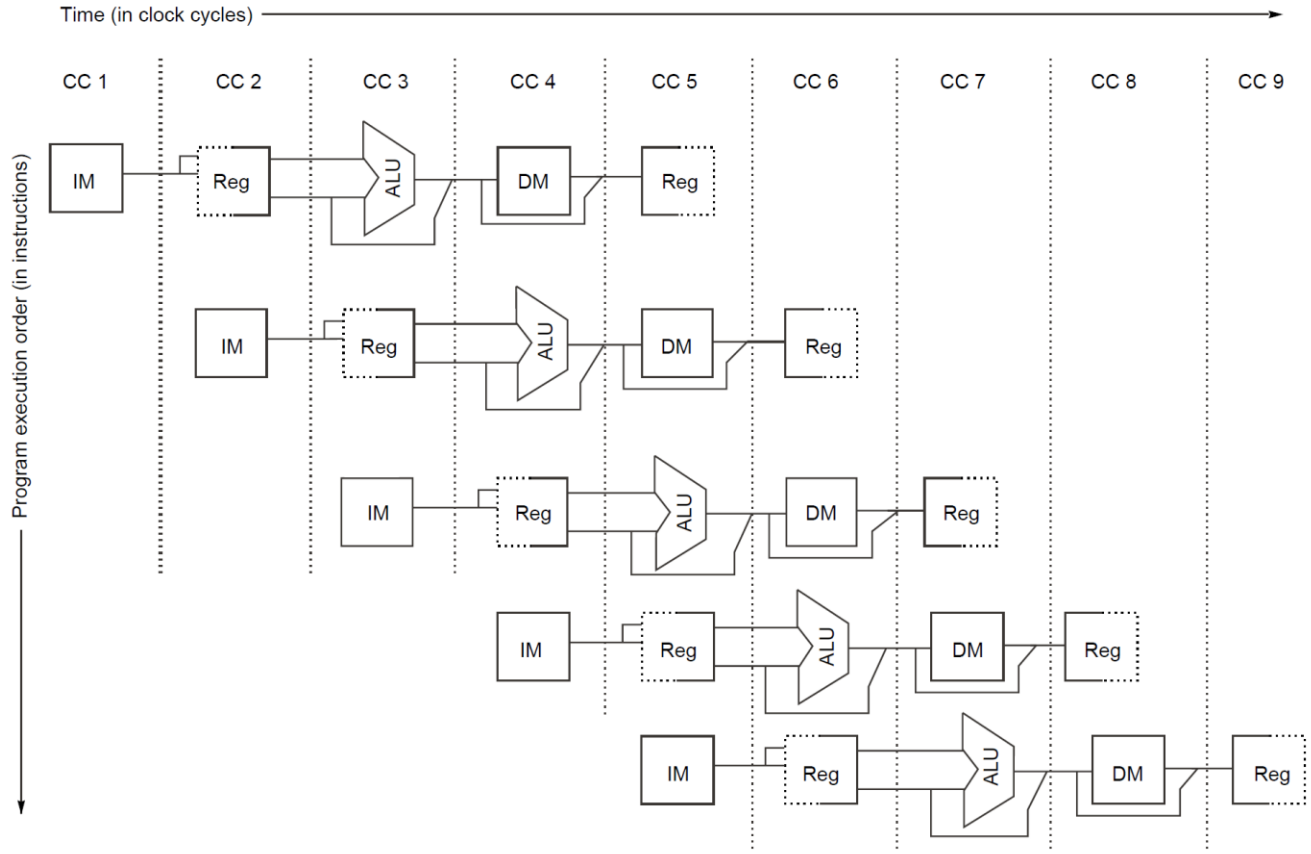

Figura 2 : Il Pipeline visto come una serie di data path (unità di calcolo) sovrapposti temporalmente. Sono evidenziate le porzioni di data path coinvolte nel funzionamento di uno stadio: IM = Memoria delle Istruzioni, DM = Memoria dei Dati, CC = Ciclo di Clock.

Se gli stadi del pipeline sono bilanciati e ci si trova in presenza di condizioni ideali, il tempo per istruzione nel processore con pipeline è uguale a:

$$\frac{\text{Tempo per istruzione della macchina senza pipeline}}{\text{Numero di stadi del pipeline}}$$

Ovvero il miglioramento delle prestazioni (il cosiddetto *speedup*: il rapporto tra i tempi medi per istruzione del processore senza pipeline e del processore con pipeline), equivale esattamente al numero di stadi.

Come già sottointeso dalle prime due figure utilizzate, in questa esposizione si prende come esempio una architettura RISC (Reduced Instruction Set Computer), la cui semplicità permette di implementare facilmente il pipelining e porre in evidenza i suoi concetti chiave.

Accorgimenti e modifiche alle risorse hardware presenti

Poiché le istruzioni vengono sovrapposte nell'esecuzione, deve essere assicurato che due operazioni diverse (cioè appartenenti ciascuna ad una distinta istruzione) non utilizzino le stesse risorse hardware dell'unità di calcolo, nel medesimo ciclo di clock.

Nel semplice caso del pipeline a 5 stadi (presentato in Fig. 1), si può osservare come le fasi ID e WB utilizzino entrambe l'archivio dei registri (Fig. 2) (l'archivio dei registri è l'unità funzionale che racchiude i registri e il controllo necessario al loro funzionamento).

Ciò potrebbe causare problemi qualora due diverse istruzioni, come la prima e la quarta nel caso della figura, tentassero nello stesso ciclo di clock di accedere ai registri, una per scrivere dei risultati, l'altra per leggere degli operandi;

come si vedrà, questo è un esempio di conflitto strutturale, nel caso specifico risolvibile facilmente tramite la convenzione che le scritture avvengano tutte nella prima metà del ciclo di clock, mentre le letture nella seconda metà.

Come indicato in Fig. 2 dalle abbreviazioni IM e DM, le memorie per le istruzioni e i dati vengono separate (tramite l'adozione di cache dati e cache istruzioni distinte) per eliminare la situazione conflittuale tra un prelievo di istruzione, IF, e un accesso alla memoria dei dati, MEM che possono avvenire nello stesso ciclo di clock.

Sempre con riferimento al sistema di memoria, nel processore con pipeline, rispetto alla versione che ne è sprovvista (e a parità di ciclo di clock), deve essere fornita un'ampiezza di banda superiore secondo un fattore pari al numero di stadi, detto anche profondità del pipeline (nel nostro esempio, cinque).

Introduzione di nuove risorse hardware

Per evitare che anche le istruzioni in differenti stadi possano interferire, viene fatta una modifica importante: si introducono dei *registri di pipeline* tra stadi successivi, in modo da permettere che alla fine del ciclo di clock i risultati di una fase vengano salvati in un registro, che verrà usato come input nello stadio seguente nel prossimo ciclo di clock (Fig. 3).

Così facendo un'istruzione può entrare nello stadio che è appena divenuto libero, senza preoccuparsi della possibilità di compromettere lo stato dell'istruzione uscente che l'ha preceduta.

I registri di pipeline hanno anche il compito essenziale di trasportare risultati intermedi tra stadi che non sono direttamente adiacenti (per esempio, in un'istruzione store, il valore del registro che deve essere memorizzato viene letto nella fase ID, ma poi non viene utilizzato fino allo stadio MEM)

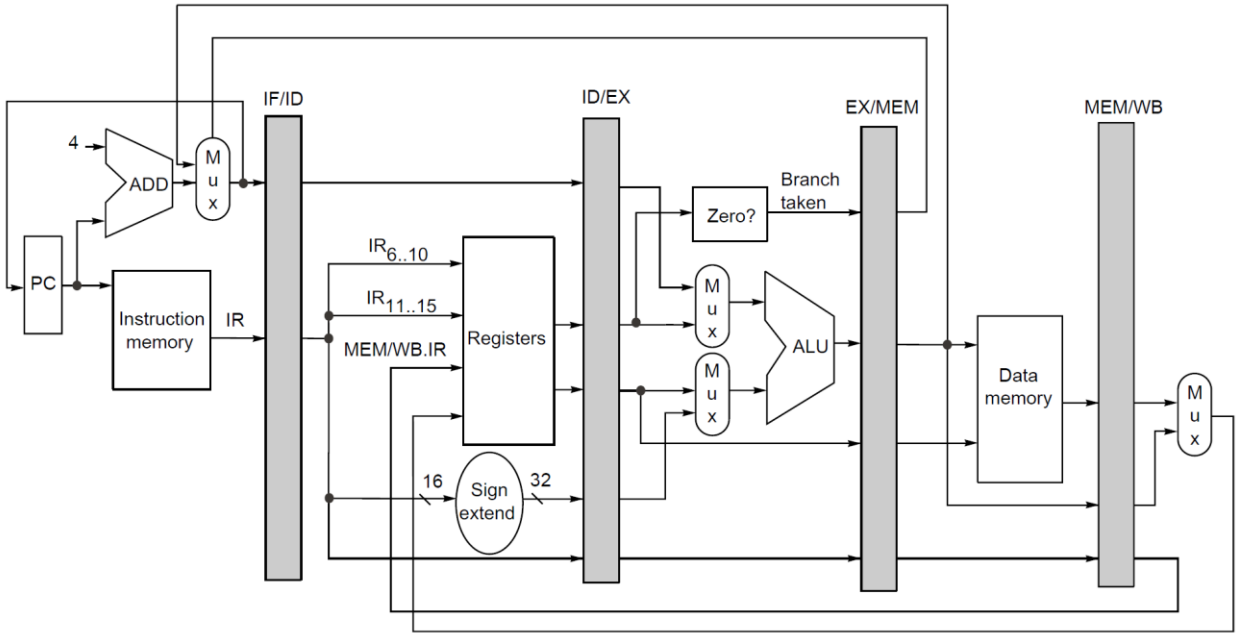

Figura 3: Per implementare il pipelining, si aggiunge nel data path una serie di registri di pipeline tra ogni coppia di stadi del pipeline. I registri trasportano valori e informazioni utili al controllo da uno stadio al successivo. Come si può vedere, alcuni segnali vengono retroazionati e questi sono causa di gran parte della complessità del pipelining, in quanto fonti di possibili conflitti.

Dipendenze e Conflitti

Per comprendere il grado di parallelismo che può essere sfruttato in un programma è necessario individuare e distinguere le istruzioni parallele, che possono quindi essere eseguite simultaneamente, da quelle che invece sono dipendenti e devono essere eseguite in ordine, eventualmente sovrapponibili in modo parziale.

Esistono tre tipi di dipendenze tra istruzioni:

Dipendenze per i Dati

Dette anche *vere dipendenze* (true data dependence).

Un'istruzione *j* è dipendente per i dati dall'istruzione *i*, se l'istruzione *i* produce un risultato che può essere utilizzato dall'istruzione *j*, o esiste una catena di tali dipendenze tra le due.

Due istruzioni legate da una dipendenza per i dati non possono essere sovrapposte completamente o venir eseguite simultaneamente.

Dipendenze per i Nomi

Chiamate in questo modo perché le istruzioni usano lo stesso "*nome*" (registro o indirizzo di memoria), ma non c'è flusso di dati tra di esse a differenza delle *vere* dipendenze per i dati.

Si considera un'istruzione *j* che segue un'istruzione *i* nell'*ordine del programma*, cioè l'ordine nel quale dovrebbero essere effettuate le istruzioni se venissero eseguite una alla volta nell'ordine sequenziale del programma originale.

Sussistono due tipi di dipendenze per i nomi:

Antidipendenze, quando un'istruzione *j* tenta di scrivere un registro o un indirizzo di memoria che l'istruzione *i* legge;

Dipendenze sull'uscita (output dependence), quando *i* e *j* scrivono lo stesso registro o locazione di memoria.

Le istruzioni coinvolte in una dipendenza per i nomi possono venire eseguite contemporaneamente o essere riordinate se il nome utilizzato viene cambiato in modo da evitare la contesa della risorsa (tecniche di *renaming*).

Dipendenze per il Controllo

Determinano l'ordine di una istruzione, rispetto a una *diramazione* (istruzione *branch*), in modo che tale istruzione venga svolta secondo l'ordine del programma corretto e solo quando deve essere effettivamente eseguita.

Sussistono due vincoli imposti da una dipendenza per il controllo:

- un'istruzione dipendente per il controllo da una diramazione non può essere spostata prima del branch, permettendo che la sua esecuzione non sia più controllata dal salto.
- un'istruzione che non è dipendente per il controllo da una diramazione non può essere mossa dopo il branch, così che la sua esecuzione sia ora controllata dal salto.

Per conservare la correttezza di un programma si devono garantire due proprietà: il *comportamento delle eccezioni* (*exception behavior*) e il *flusso dei dati* (*data flow*); mantenere la prima proprietà, significa che la riorganizzazione delle istruzioni non deve causare nuove eccezioni nel programma (una condizione più stringente è che il riordinamento non cambi il modo in cui le eccezioni vengono sollevate), mentre la seconda è il flusso dei valori dei dati tra le istruzioni che producono risultati e quelle che li consumano.

Entrambe sono di norma preservate conservando le dipendenze per i dati e per il controllo.

Non sempre l'esecuzione delle istruzioni tramite pipelining presenta una situazione ideale, in cui è possibile immettere nel pipeline una istruzione ad ogni ciclo senza mai incontrare problemi.
Quando si tenta di sovrapporre l'esecuzione di istruzioni, possono insorgere dei conflitti, perché si rischiano di violare delle dipendenze oppure perché le risorse a disposizione sono insufficienti.
Tali circostanze conflittuali vengono chiamate *hazard* (termine tradotto anche con alea, in quanto si tratta di situazioni aleatorie, ma che in questa tesina verrà denominato più semplicemente come *conflitto*) che, se ignorate, potrebbero portare il processore in uno stato illecito, diverso da quello che verrebbe raggiunto se le istruzioni venissero eseguite sequenzialmente, seguendo l'ordine del programma.
Anche le tipologie di conflitto sono tre, ma solo le ultime due sono conseguenza di dipendenze fra istruzioni.

Conflitti Strutturali

Avvengono quando una combinazione di istruzioni non può essere supportata dall'hardware, che si rivela insufficiente a far fronte a contese nell'uso di risorse condivise (es.: due istruzioni cercano di accedere contemporaneamente ad una stessa unità funzionale); i casi più frequenti avvengono quando in una unità funzionale non è stato pienamente implementato il pipelining oppure in presenza di una risorsa che non è stata replicata a sufficienza.

Conflitti sui Dati

Quando sussiste una dipendenza per i dati o per i nomi tra istruzioni, ed esse sono abbastanza vicine affinché la loro sovrapposizione cambi l'ordine di accesso agli operandi coinvolti nella dipendenza, si crea un conflitto sui dati.
E' necessario conservare l'*ordine del programma*, e questo è garantito rilevando ed evitando tali conflitti. A seconda dell'ordine delle operazioni di lettura e

scrittura determinate da due istruzioni *i* e *j*, con *j* successiva ad *i* nell'ordine del programma, i conflitti sui dati si suddividono in RAW, WAR e WAW:

- RAW (read after write) – quando *j* tenta di leggere un operando prima che *i* lo scriva: *j* ottiene il valore vecchio. Deriva da una vera dipendenza per i dati.
- WAR (write after read) – nei casi in cui *j* tenta di scrivere una destinazione prima che venga letta da *i*: *i* ottiene il valore aggiornato. Tale conflitto insorge da una antidipendenza e può avvenire quando le istruzioni vengono riordinate.
- WAW (write after write) – quando *j* tenta di scrivere un operando prima che sia scritto da *i*, le scritture vengono eseguite in ordine errato e l'ultimo valore rimasto è quello di *i* invece che quello di *j*. Corrisponde a una dipendenza sull'uscita e può avere luogo nei pipeline che permettono a un'istruzione di procedere anche se un'istruzione precedente è in stallo.

RAR (read after read) non è un conflitto.

Conflitti sulle Diramazioni

Le diramazioni sono fonte di problemi al normale procedimento del pipeline, poiché possono essere risolte solo in fase di decodifica dell'istruzione, quando la destinazione di un salto diventa nota: dato che non si conosce immediatamente l'esito di un branch, non si può essere certi di immettere nel pipeline l'istruzione successiva alla diramazione nell'ordine del programma; se il pipeline è stato alimentato con le istruzioni del percorso sbagliato, insorge un conflitto sulle diramazioni e il processore deve svuotare il pipeline, ricominciando l'esecuzione dall'istruzione successiva al salto corretta.

Uno dei modi più semplici di affrontare una diramazione è rieseguire la fase di fetch dell'istruzione che segue un branch, cioè dopo che la diramazione viene risolta nella fase ID.

In qualsiasi caso, che la diramazione venga *intrapresa* (*taken* branch) o meno, il processore rimane inutilizzato per un ciclo (causando un degrado delle prestazioni ad ogni istruzione di branch), ma la ripetizione della fase IF è completamente infruttuosa se la diramazione non viene seguita (*untaken* branch), poiché era stata prelevata l'istruzione corretta.

Stalli e degradazione delle prestazioni

Per risolvere i conflitti, può essere necessario mettere in *stallo* il pipeline per un numero sufficiente di cicli, inserendo istruzioni fittizie (come delle nop, no-operation), che vanno ad occupare spazio negli stadi (per questo, gli stalli vengono chiamati anche *bolle*, come fossero delle bolle d'aria nella tubatura); l'esecuzione dell'istruzione che ha causato il conflitto viene rimandata e le successive vengono bloccate (le istruzioni precedenti, che si trovano già nel pipeline, continuano la loro esecuzione altrimenti il conflitto non potrebbe mai essere risolto).

Ovviamente questo comporta una riduzione delle prestazioni del pipeline:

$$\text{Speedup} = \frac{\text{Profondità Pipeline}}{1 + \text{Cicli in stallo per istruzione nel pipeline}}$$

il numero di CPI nel processore con pipeline, che idealmente è pari ad 1, in presenza di stalli aumenta.

La formula qui sopra si raggiunge a partire dalla definizione generale di speedup

$$\text{Speedup} = \frac{\text{Tempo medio per istruzione senza pipeline}}{\text{Tempo medio per istruzione con pipeline}}$$

$$= \frac{\text{CPI senza pipeline}}{\text{CPI con pipeline}} \times \frac{\text{Periodo di Clock senza pipeline}}{\text{Periodo di Clock con pipeline}}$$

considerando gli stadi perfettamente bilanciati e ignorando l'overhead dovuto al pipelining.

Queste semplificazioni comportano la possibilità di trascurare uno dei due rapporti (a seconda che lo speedup dovuto al pipelining si veda come riduzione

dei CPI o del periodo di clock, rispettivamente si considerano i periodi di clock uguali, oppure il numero di CPI senza pipeline pari a 1) mentre l'altro equivale semplicemente al numero di stadi del pipeline.

Fondamentalmente è la presenza di dipendenze che stabilisce un limite superiore al grado di parallelismo disponibile.
E' importante notare che le dipendenze sono una proprietà dei programmi, mentre i fatti che una dipendenza porti ad un conflitto che viene rilevato, e che quest'ultimo causi uno stallo, sono proprietà dell'organizzazione del pipeline.
Per superare i limiti causati dalle dipendenze esistono due modi:
eliminare le dipendenze trasformando il codice, oppure mantenere le dipendenze e aggirare i conflitti, evitando quindi di mettere in stallo il pipeline.

Forwarding

Al fine di ridurre i conflitti sui dati, esiste una tecnica hardware (che però viene utilizzata dai pipeline con scheduling statico, cui si farà cenno nel seguito), che prende il nome di *inoltro* (*forwarding*, anche detto bypassing o short-circuiting).
Si basa su una semplice osservazione: un risultato che deve essere utilizzato come operando da un'istruzione, può essere prelevato appena viene calcolato dall'istruzione che deve produrlo, senza aspettare che venga scritto nei registri o nella memoria.
L'hardware di inoltro rileva quando l'uscita di una unità funzionale corrisponde ad un dato di ingresso per la stessa unità funzionale o per un'altra (quindi dello stesso stadio, o di uno successivo) e perciò attiva il percorso di inoltro corrispondente, permettendo il passaggio del risultato dal registro di pipeline che corrisponde all'output della prima unità verso l'input della seconda (Fig. 4).
Riassumendo, un risultato viene inoltrato direttamente all'unità cui serve.
Da notare l'esistenza di un forwarding che avviene tramite l'archivio dei registri, in cui si utilizza l'accorgimento, introdotto in precedenza (che però era servito a

risolvere un conflitto strutturale mentre qui ha lo scopo di evitare il conflitto sui dati), di leggere i registri nella seconda metà del ciclo, e scriverli nella prima metà.

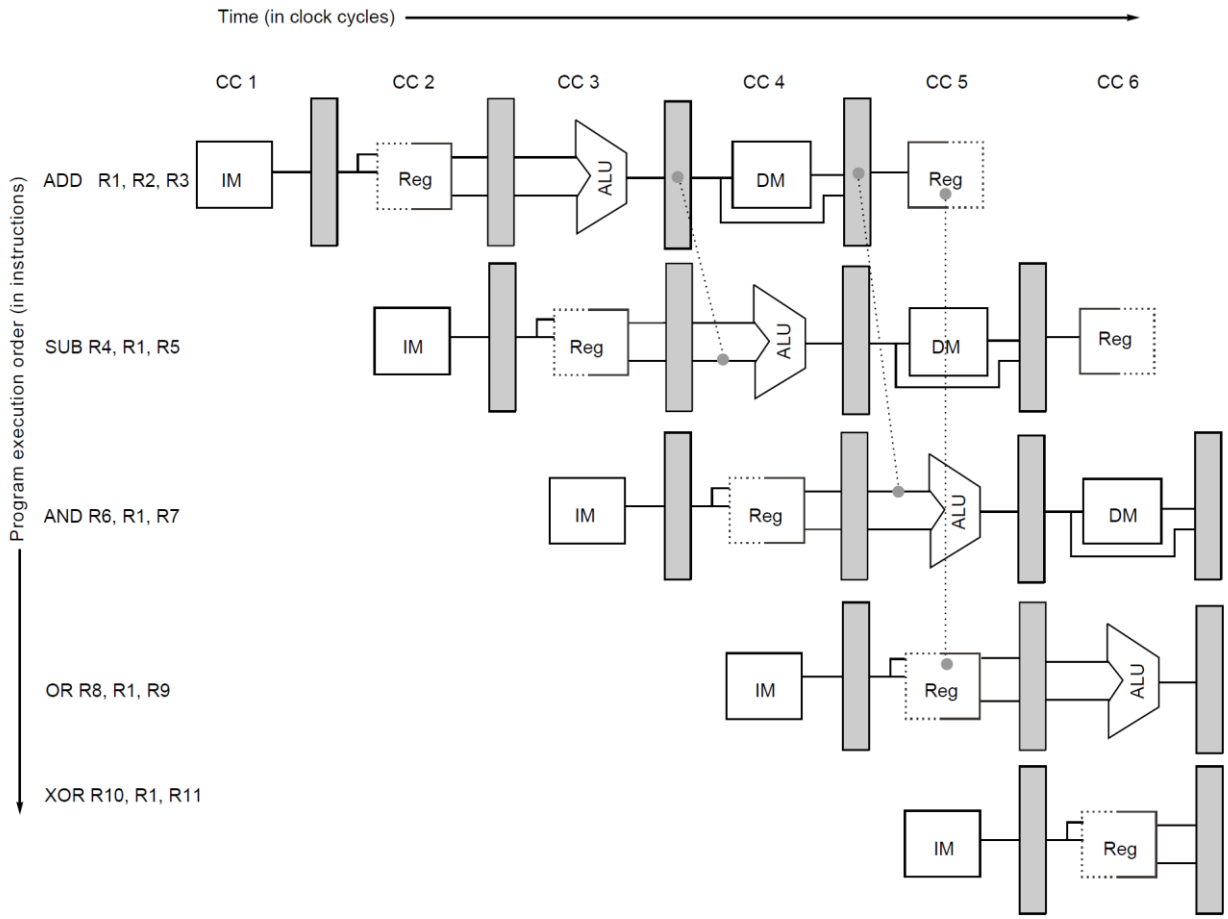

Figura 4: Un insieme di istruzioni che dipendono dal risultato della prima istruzione, usa i percorsi di inoltro per evitare i conflitti sui dati. Tra la prima e la quarta istruzione avviene un inoltro tramite l'archivio dei registri.

Per i casi in cui la strategia di inoltro non può essere applicata, il conflitto sui dati deve essere risolto tramite uno stallo; viene usata una struttura hardware che garantisce il corretto schema di esecuzione: il *pipeline interlock* ("sistema di blocco interno tra due stadi del pipeline") si occupa della rilevazione del conflitto e della conseguente messa in stallo del pipeline per il tempo necessario.

Supporto delle Eccezioni

In un processore con pipeline, la sovrapposizione delle istruzioni rende più difficoltosa la gestione delle eccezioni (indispensabile al corretto funzionamento dell'elaboratore), poiché non si sa quando un'istruzione può cambiare lo stato del processore con sicurezza.

Le situazioni eccezionali più difficili da amministrare sono quelle che vengono sollevate all'*interno* di un'istruzione (*within* instruction), cioè mentre un'istruzione è in esecuzione, e prevedono la ripresa (*resuming*) dell'esecuzione da dove era stata interrotta.

Per gestire queste situazioni particolari, il processore deve essere "*riavviabile*" (*restartable*), deve cioè far sì che un programma specifico salvi lo stato del programma correntemente in esecuzione, corregga la causa scatenante dell'eccezione e ripristini lo stato del programma, prima che l'istruzione responsabile dell'interruzione possa essere ritentata.

Se il pipeline può essere fermato in modo che le istruzioni prima della "colpevole" siano completate e quelle successive possano essere riavviate da zero, il pipeline viene detto avere "*eccezioni precise*" (*precise exceptions*).

Sussiste un altro problema dovuto alla sovrapposizione delle esecuzioni: le eccezioni possono comparire fuori ordine; ad esempio se due istruzioni consecutive sollevano entrambe un'eccezione, potrebbe accadere che la seconda sollevi la rispettiva eccezione in uno stadio precedente a quello in cui la prima solleva la propria.

Per questo motivo può rendersi necessario l'utilizzo di un *vettore di stato* (*status vector*) associato ad una particolare istruzione, che contiene tutte le eccezioni generate da essa; il vettore di stato viene controllato quando un'istruzione entra nella fase WB e se trova delle eccezioni, queste vengono risolte nell'ordine in cui verrebbero sollevate in un processore senza pipeline.

Appena un'eccezione viene lanciata, inoltre, i segnali di controllo che permettono la scrittura su registri o memoria vengono disabilitati, impedendo cambiamenti di stato da parte dell'istruzione coinvolta.

Predizione delle Diramazioni

L'obiettivo di evitare cali di prestazioni dovuti alle dipendenze sul controllo, può essere raggiunto riducendo il numero di conflitti sulle diramazioni, tramite la tecnica di svolgimento del ciclo (loop unrolling) accennata in precedenza, o anche predicendo il comportamento delle diramazioni, sia staticamente sia grazie all'hardware.

Nei metodi di predizione statica eseguiti dal compilatore, rientra l'esempio presentato nella descrizione dei conflitti sulle diramazioni, di ripulire il pipeline cancellando le istruzioni dopo il branch finché questo non viene risolto.

Tra le altre tecniche software si citano rapidamente gli schemi di "predizione non intrapresa" (*predicted-untaken*) e di "predizione intrapresa" (*predicted-taken*), che continuano l'esecuzione trattando nell'immediato un salto come untaken o taken finché non si conosce l'esito definitivo, e la strategia denominata "diramazione ritardata" (*delayed branch*), che dopo il branch esegue un "successore sequenziale" (possono essere anche molteplici) : un'istruzione che viene eseguita qualsiasi sia l'esito della diramazione e che il compilatore ha l'incarico di trovare.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figura 5: Il comportamento di una diramazione ritardata è lo stesso nel caso che il salto sia preso o meno.

La predizione dinamica delle diramazioni (dynamic branch prediction), nella sua forma più semplice, permette di ridurre il ritardo della diramazione quando è più lungo del tempo per calcolare la destinazione del salto; si realizza utilizzando una cache chiamata *branch history table* o *branch-prediction buffer*, indicizzata dalla porzione inferiore dell'indirizzo di una istruzione di salto, e contenente uno o più bit che indicano se la diramazione è stata recentemente intrapresa o meno: in generale, disponendo di n bit, il salto verrà predetto come intrapreso quando il valore dei bit è maggiore o uguale alla metà del valore massimo ($2^n - 1$).

Ogni volta che si ha una diramazione, si inverte un bit in modo che lo stato venga cambiato verso la direzione del salto appena imboccato.

I "*predittori*" a singolo bit sono poco performanti: un salto quasi sempre intrapreso, viene scorrettamente predetto per due volte quando di tanto in tanto non viene intrapreso, poiché ad ogni errore di predizione si inverte l'unico bit (una volta quando risulta untaken e la seconda quando ritorna ad essere taken).

Sebbene i predittori ad n bit dovrebbero consentire una maggiore precisione, gli studi effettuati su questi ultimi hanno dimostrato che i predittori a 2 bit hanno un'efficacia praticamente identica.

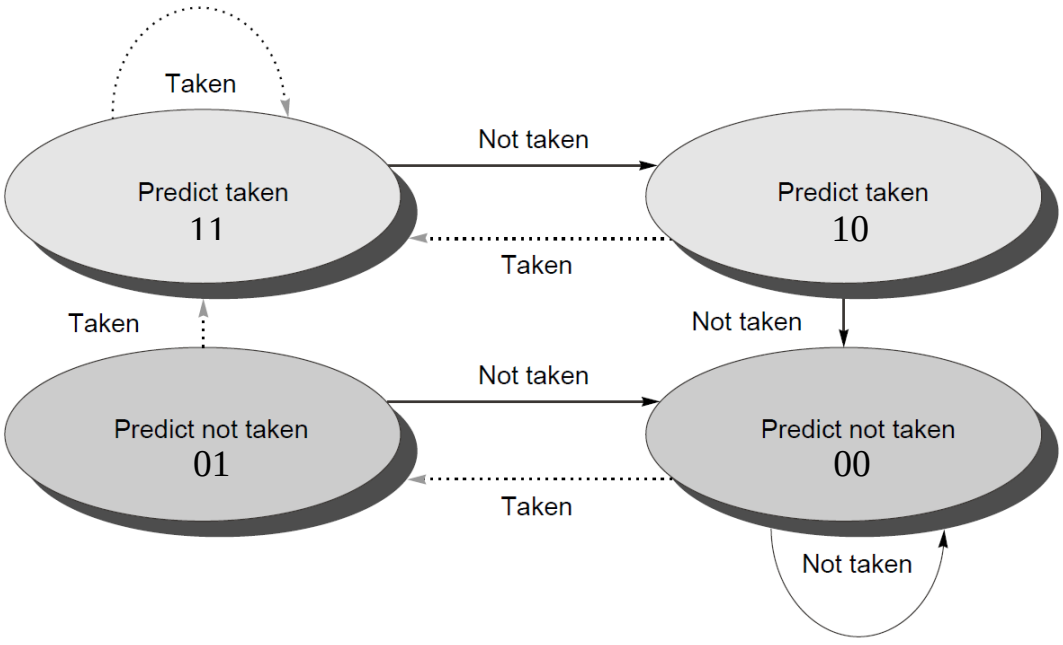

Figura 6: Gli stati in uno schema di predizione a 2 bit.

Oltre ai predittori che usano il comportamento recente di un singolo salto, esistono anche i predittori a due livelli, o "predittori correlativi" (*correlating predictors*), che sono individuati da una coppia di numeri (m,n), e usano il comportamento degli ultimi m salti per scegliere fra 2^m predittori, ognuno dei quali è un predittore a n bit per una singola diramazione.
Una terza tipologia di predittore è il "predittore a torneo" (*tournament predictor*) che usa entrambi i precedenti due tipi di predittore, locale e globale (solitamente uno per tipo), combinandoli con un selettore che sceglie quale dei due usare sulla base di chi è stato il più efficace nelle predizioni recenti.

Scheduling Dinamico

Quando un conflitto sui dati non può essere evitato tramite i meccanismi di inoltro, il pipeline viene posto in stallo, partendo dall'istruzione che l'ha causato, cioè quella che utilizza il risultato della dipendenza sui dati in questione, e nessuna istruzione successiva a questa può essere prelevata finché la dipendenza non viene risolta; come accennato in precedenza, la tecnica dell'inoltro è associata allo scheduling statico, cioè la riorganizzazione dell'ordine delle istruzioni effettuata dal compilatore; un esempio di scheduling statico è la tecnica di srotolamento (unroll) del ciclo effettuata in compilazione.
Esiste anche una tecnica hardware, con la quale l'ordine di esecuzione delle istruzioni può essere cambiato al fine di ridurre gli stalli, che prende il nome di scheduling dinamico (riorganizzazione dinamica).
Il punto chiave del metodo, consiste nella suddivisione dello stadio che prima è stato indicato come "decodifica dell'istruzione" (Instruction Decode), in due stadi separati:

Issue (*Emissione*) – decodifica l'istruzione e controlla la presenza di conflitti strutturali.

Read Operands (Lettura degli operandi) – aspetta fino a che non sussistono più conflitti sui dati (cioè attende che i dati siano pronti), poi legge gli operandi.

Viene garantito che le istruzioni siano *emesse* in ordine (in-order issue) e allo stesso tempo permesso che possano entrare la fase di esecuzione con un ordine cambiato (out-of-order execution); quest'ultimo fatto implica che anche il completamento sia "fuori ordine".

Di conseguenza le istruzioni vengono eseguite appena possibile, ma per poter ottenere un vantaggio dall'esecuzione fuori ordine, più istruzioni devono trovarsi nello stadio di esecuzione (EX) simultaneamente; si può consentire ciò, con unità funzionali multiple, con unità funzionali che implementano un pipeline, oppure con entrambe le strategie.

Una tecnica di scheduling dinamico è l'algoritmo di Tomasulo che rileva quando gli operandi per una istruzione sono disponibili, minimizzando così i conflitti RAW, e impiega la "*rinominazione dei registri*" (*register renaming*), al fine di ridurre i conflitti WAR e WAW, che possono insorgere come conseguenza dell'esecuzione fuori ordine: rinominando i registri destinazione, nessuna istruzione write fuori ordine può disturbare un'istruzione che dipende da un valore precedente di un operando.

Queste funzioni vengono effettuate grazie alla presenza di "*stazioni di prenotazione*" (*reservation stations*), che memorizzano gli operandi delle istruzioni in attesa di essere emesse appena questi diventano disponibili, e al fatto che le istruzioni pendenti utilizzino come riferimento per i dati in ingresso, le stazioni di prenotazione in cui gli operandi devono essere memorizzati, invece dei registri dove essi dovevano essere scritti.

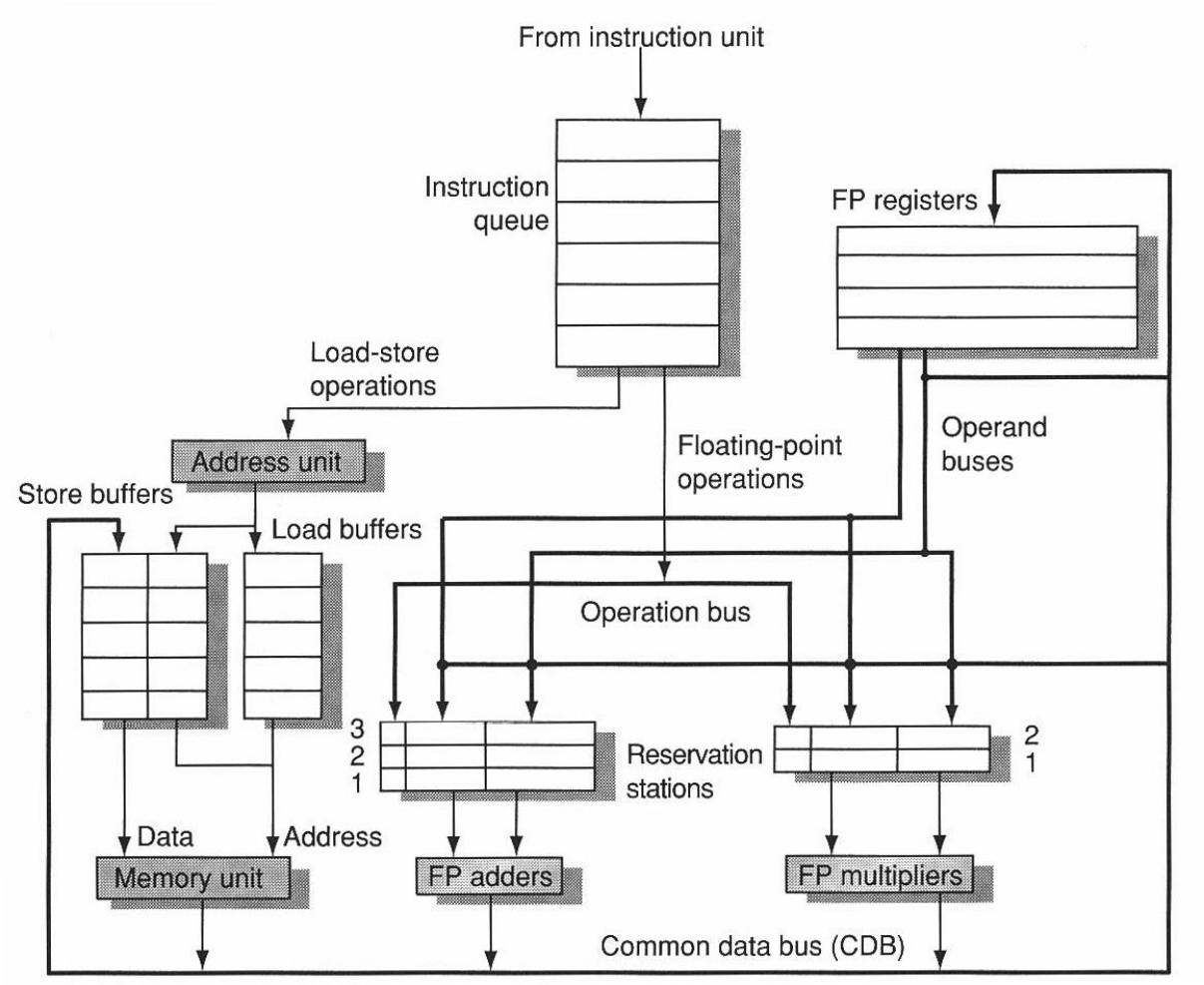

Figura 7 : La struttura fondamentale di una unità floating point che utilizza l'algoritmo di Tomasulo.

Ogni unità funzionale ha la propria stazione di prenotazione, che possiede tutte le informazioni per stabilire quando un'istruzione può iniziare l'esecuzione in quella unità (Fig. 7): di conseguenza, la rilevazione dei conflitti e il controllo sull'esecuzione sono distribuiti.

Una stazione di prenotazione infatti, trattiene le istruzioni emesse in attesa di essere eseguite in un'unità funzionale, e memorizza i valori degli operandi necessari o i nomi delle stazioni di prenotazione da cui verranno forniti; appena gli operandi diventano tutti disponibili, un'istruzione può iniziare l'esecuzione.

I risultati vengono passati direttamente alle unità funzionali, dalle stazioni dove sono memorizzati, piuttosto che passare tramite i registri, grazie all'impiego di un bus dei dati comune (CDB: *common data bus* oppure *common result bus*),

che sostituisce la funzione del meccanismo di inoltro dei processori con scheduling statico.
L'algoritmo di Tomasulo prevede inoltre due buffer per le istruzioni store e load, che mantengono gli indirizzi e gestiscono gli accessi alla memoria.
Per conservare il comportamento delle eccezioni, una istruzione non può iniziare l'esecuzione, se prima tutte le diramazioni che la precedono nell'ordine del programma non sono state risolte. I processori con scheduling dinamico, usano la predizione dinamica delle diramazioni e per questo motivo devono attendere che la predizione venga confermata, prima di procedere all'esecuzione delle istruzioni che seguono la diramazione.

La tecnica della Speculazione

Come si è visto, lo scheduling dinamico migliora le prestazioni, ma le dipendenze per il controllo pongono dei limiti al parallelismo fruibile.
La speculazione basata sull'hardware è una tecnica che permette di predire la direzione di un salto, proseguendo l'esecuzione delle istruzioni prima che esso venga risolto e ha la capacità di annullare gli effetti di una predizione scorretta.
Questo metodo consente dunque di superare i limiti dovuti alla presenza di diramazioni e integra lo scheduling dinamico, necessario per la riorganizzazione delle istruzioni provenienti da vari blocchi, con la predizione dinamica delle diramazioni, che decide la direzione del salto da intraprendere, e la speculazione, la quale permette di continuare l'esecuzione prima che vengano risolte le dipendenze per il controllo e ripristinare lo stato nel caso di un salto errato.
L'algoritmo di Tomasulo può essere esteso alla speculazione, separando il passaggio dei risultati tra le istruzioni, che serve per l'esecuzione speculativa delle istruzioni, dall'effettivo completamento di un'istruzione; tramite questa distinzione, un'istruzione può compiere la sua fase di esecuzione e passare il

risultato ad altre istruzioni, senza che le sia permesso effettuare aggiornamenti irreversibili, finché non è sicuro che l'istruzione non sia più speculativa.

Si definisce un ulteriore passo nella sequenza di esecuzione dell'istruzione chiamato *consolidamento* (*commit*), che avviene quando una istruzione non è più speculativa e ha il permesso di aggiornare l'archivio dei registri o la memoria.

Le istruzioni possono dunque essere eseguite fuori ordine, ma sono costrette a effettuare il consolidamento in ordine, e si impedisce ogni azione irrevocabile finché non si arriva al consolidamento.

In questo modo viene distinto il completamento dell'esecuzione, dal consolidamento di un'istruzione.

I risultati di un'istruzione che ha completato l'esecuzione ma non è stata consolidata, vengono salvati in un *buffer di riordino* (*reorder buffer*, *ROB*), che viene impiegato anche per il passaggio degli operandi tra le istruzioni (sempre nell'intervallo di tempo tra completamento dell'esecuzione e il consolidamento di un'istruzione); le stazioni di prenotazione, continuano ad essere il luogo dove le operazioni e gli operatori, vengono memorizzati tra l'emissione e l'inizio dell'esecuzione.

Il ROB provvede alla funzione di rinominazione dei registri che veniva effettuata dalle stazioni di prenotazione, e integra la funzione del buffer per le istruzioni store prendendone il posto (Fig. 8).

Una volta che un'istruzione raggiunge la testa del ROB e il risultato è memorizzato, diventa pronta per essere consolidata, poiché non è più speculativa; a quel punto il risultato viene scritto nei registri o nella memoria e l'istruzione viene rimossa dal ROB.

Dato che ogni istruzione ottiene una collocazione nel ROB prima di consolidarsi, i risultati vengono individuati utilizzando come riferimento il numero della posizione nel ROB, invece del nome della stazione di prenotazione.

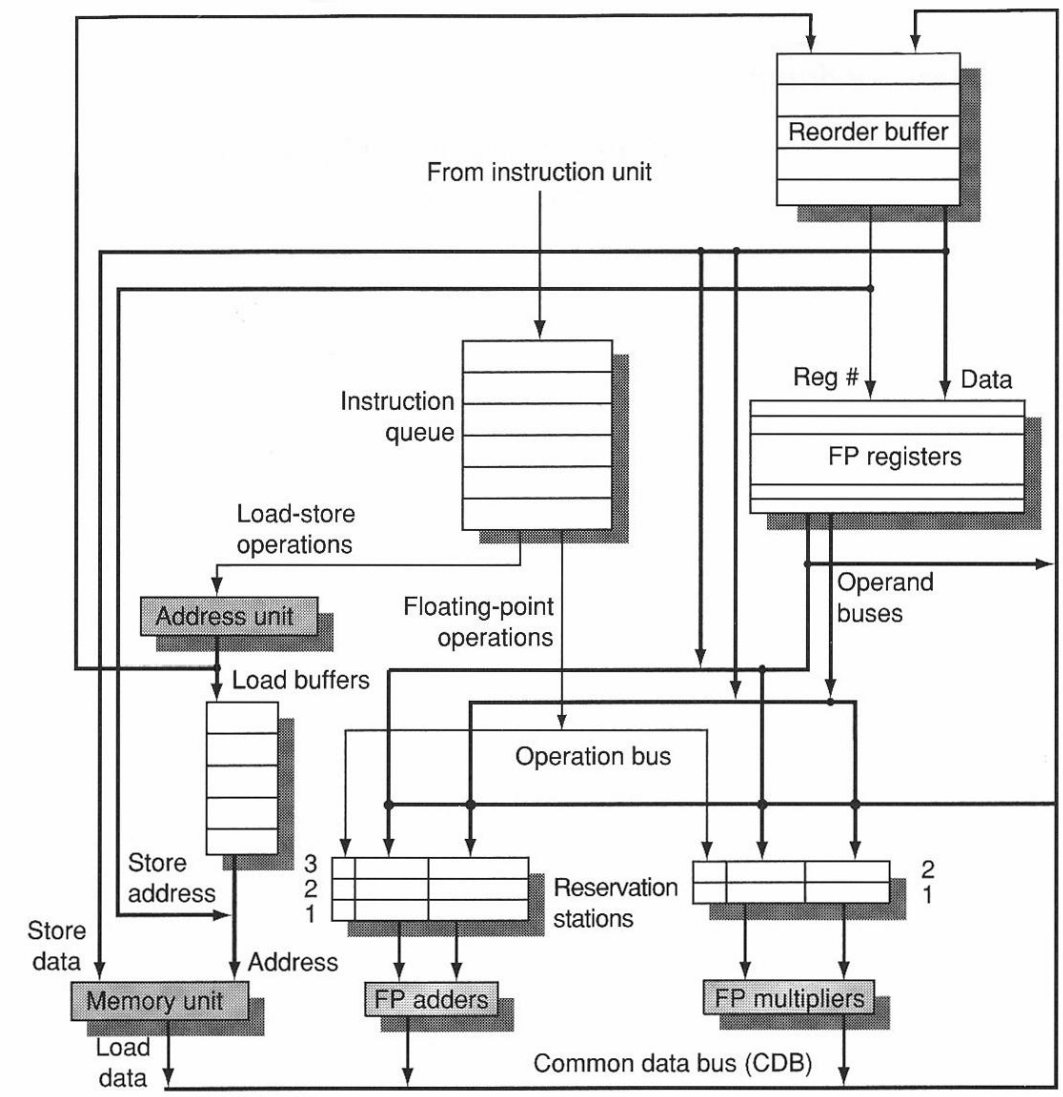

Figura 8: La struttura fondamentale di una unità FP che utilizza l'algoritmo di Tomasulo ed è stata estesa per gestire la speculazione. Il buffer delle istruzioni store è stato eliminato poiché la sua funzione viene integrata nel ROB. Questo meccanismo può essere esteso all'emissione multipla ingrandendo il bus dei dati comuni (CDB) al fine di permettere completamenti multipli per ciclo di clock.

In una implementazione alternativa la rinominazione viene effettuata da un insieme supplementare di registri, invece che dal ROB, a cui rimane la sola funzione di individuare quando un'istruzione può consolidarsi.

Se una diramazione è stata predetta in modo scorretto, quando raggiunge la testa del ROB, allora la speculazione si rivela errata, il ROB viene svuotato e l'esecuzione deve ripartire dal successore della diramazione corretta; altrimenti, se la diramazione è stata predetta giustamente, il salto può essere consolidato.

Viene preservato un modello di interruzioni precise, poiché le istruzioni si consolidano in ordine, e un'eccezione sollevata da un'istruzione viene gestita solo quando quest'ultima raggiunge la testa del ROB, cioè appena prima di consolidarsi; a quel punto tutte le istruzioni in sospeso vengono eliminate svuotando il buffer di riordino.

Poiché né i registri, né la memoria vengono aggiornati prima del consolidamento, il processore può annullare le azioni speculative in caso di predizione errata, svuotando il ROB e ripartendo col prelievo delle istruzioni dall'altro percorso.

Per questo motivo la predizione delle diramazioni è di grande importanza, visto che nella speculazione hardware influisce maggiormente sulle prestazioni.

Processori ad Emissione Multipla

I processori ad emissione multipla, come viene anticipato dal nome, permettono di emettere più istruzioni nello stesso ciclo di clock (sono chiamati anche processori superscalari)

I processori superscalari possono usare scheduling statico oppure dinamico; un caso particolare dei primi sono i processori VLIW (Very Long Instruction Word), in cui le possibilità di emissione (issue slots) sono fissate, dato che un insieme di istruzioni viene formattato come una unica lunga istruzione di estensione prestabilita; per poter eseguire una istruzione lunga, e quindi le istruzioni che la compongono in modo parallelo, si usano molteplici unità funzionali indipendenti.

In un processore superscalare che utilizza scheduling dinamico (si prende come esempio il metodo di Tomasulo), esistono due approcci per emettere più istruzioni in un ciclo di clock: uno prevede l'esecuzione delle operazioni di assegnazione di una stazione di prenotazione e aggiornamento delle tabelle di controllo del pipeline (come richiesto dall'algoritmo di Tomasulo), in una frazione del ciclo di clock, in modo che più istruzioni possano essere processate

in un ciclo; il metodo alternativo consiste nella realizzazione della logica necessaria a gestire più istruzioni allo stesso tempo e le relative dipendenze.
Entrambe le strategie possono essere utilizzate simultaneamente nel medesimo processore.
Il processore ad emissione multipla che utilizza una riorganizzazione dinamica, può anche trarre vantaggio dall'impiego della speculazione; per poterlo fare deve essere in grado di consolidare (commit) più istruzioni in un ciclo, allo stesso modo in cui prima è stata evidenziata la necessità dell'emissione multipla.

4. Una Classificazione delle Architetture Parallele

Un'architettura parallela è un insieme di elementi di elaborazione che cooperano e comunicano per risolvere velocemente problemi di dimensioni considerevoli, talvolta intrattabili su macchine sequenziali.[3]

Flynn, negli anni '60, propose una tassonomia delle architetture parallele, valida ancora tutt'oggi, che prevedeva la suddivisione dei calcolatori in quattro categorie, a seconda della quantità di flussi di istruzioni e di dati, sfruttati da una particolare architettura :

SISD (single instruction stream, single data stream) – ovvero il processore ad un singolo core, in cui un singolo flusso di istruzioni utilizza un unico flusso di dati.

SIMD (single instruction stream, multiple data streams) – in cui la stessa istruzione viene eseguita da più processori che utilizzano differenti flussi di dati. Questi computer sfruttano il *parallelismo a livello dei dati* (data-level parallelism), poiché effettuano le stesse operazioni su più oggetti in parallelo.
Le architetture SIMD negli ultimi anni sono ritornate al centro dell'attenzione per quanto riguarda il campo dell'elaborazione grafica e delle applicazioni multimediali. Un esempio di SIMD sono le architetture Vettoriali (Vector).

MISD (multiple instruction streams, single data stream) – di cui non è mai esistito alcun prodotto commerciale.

[3] Almasi & Gottlieb, 1989

MIMD (multiple instruction streams, multiple data streams) – dove ogni processore esegue le proprie istruzioni, elaborando i propri dati.
Questi elaboratori sfruttano il parallelismo a livello di thread (TLP), dato che più thread vengono eseguiti in parallelo, e hanno avuto un notevole sviluppo grazie a questa caratteristica; la flessibilità del TLP, permette infatti ai MIMD di operare in diverse modalità: concentrarsi su una singola applicazione che richiede grandi prestazioni, eseguire un insieme di compiti in maniera simultanea, oppure sfruttare una combinazione dei due.
Prima di specificare meglio questo aspetto, si fa richiamo alla distinzione tra processo e thread: per processo si intende un segmento di codice che può operare indipendentemente dagli altri processi, mentre col nome thread, spesso si individua uno dei processi che costituiscono un programma più grande, e che quindi possono condividere codice e gran parte dello spazio di indirizzamento.
Ogni processore del MIMD può quindi eseguire un processo o un thread, e di conseguenza il multiprocessore si può adattare a differenti applicazioni: lo svolgimento di differenti processi o di un programma costituito da vari thread.
Un altro vantaggio è la possibilità di effettuare investimenti prevalentemente sulla progettazione di un singolo core che poi viene replicato per costituire il multiprocessore.

Alcuni processori sono degli ibridi di queste categorie, dal momento che la classificazione di Flynn fornisce un modello grossolano.

5. Parallelismo a livello di Thread

Multithreading

Quando l'ILP è insufficiente e l'elaboratore diventa inattivo a causa di situazioni conflittuali, le unità funzionali possono rimanere inutilizzate, causando perciò uno spreco di risorse.

Per risolvere questo problema esiste la possibilità di sfruttare il TLP, adoperando un processore originariamente progettato con lo scopo di supportare l'ILP: si possono usare le istruzioni che provengono da thread distinti e risultano quindi indipendenti, al fine di mantenere impegnato il calcolatore durante gli stalli.

La tecnica del Multithreading permette ad un insieme di thread di condividere l'uso delle risorse hardware di un singolo processore, cosa che può essere realizzata eseguendo in maniera veloce lo scambio tra l'esecuzione dei diversi thread (thread-switching) e garantendo a ciascuno la duplicazione dello stato, al fine di conservare i dati necessari all'esecuzione.

Esistono due modalità principali di realizzazione del multithreading:

- a grana fine, in cui viene eseguita un'istruzione per ogni thread secondo una politica round-robin e saltando eventuali thread in stallo, i cui vantaggi sono il mantenimento costante in attività del processore, mentre i punti deboli sono il rallentamento dell'esecuzione del singolo thread e l'overhead per il thread-switching presente ad ogni ciclo di clock;
- a grana grossa, dove i thread vengono alternati solo in presenza di stalli lunghi, permettendo così di rallentare in minore misura il processore, ma creando una situazione svantaggiosa per il troughput, dato che in presenza di stalli il pipeline deve essere svuotato e successivamente essere riempito dalle istruzioni del nuovo thread.

Il multithreading simultaneo (SMT), è un metodo a grana fine che utilizza un processore superscalare dotato di scheduling dinamico, per sfruttare sia TLP sia ILP in maniera simultanea.

Nel SMT più thread usano le possibilità di emissione (issue slot) in un singolo ciclo di clock, in modo che le unità funzionali, sovrabbondanti in un processore ad emissione multipla che esegue un thread singolo, vengano sfruttate più efficacemente, grazie allo scheduling dinamico e al register renaming che permettono a diversi thread di essere eseguiti insieme.

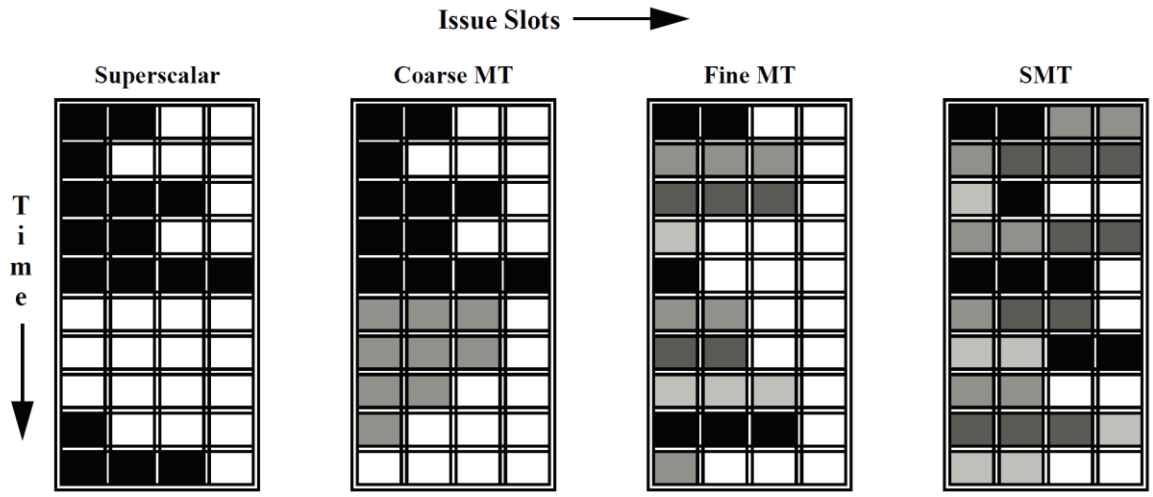

Figura 9 : Quattro approcci differenti nell'uso degli slot di emissione per un processore superscalare:

L'utilizzo di processori superscalari più ampi e di tecniche SMT più aggressive, non produce benefici tali da giustificare in modo ragionevole i costi sostenuti, perciò i progettisti sono più inclini a realizzare architetture multiprocessore, con un supporto meno aggressivo dell'emissione multipla e del multithreading.

Multiprocessori

Poiché il parallelismo a livello di thread, sfruttato dai MIMD, permette spesso una maggior flessibilità rispetto al parallelismo a livello dei dati, impiegato nei SIMD, e la stessa architettura MIMD offre una buona adattabilità alle applicazioni più svariate, i multiprocessori creati per un uso generico (general-purpose) si basano su di essa.

Grazie ai progressi tecnologici, a partire dagli anni ’90, è stato possibile disporre più processori su un singolo chip, portando ad un nuovo tipo di progettazione denominato “multi-core”, in cui i vari core (nuclei elaborativi dei processori) comunicano molto velocemente e condividono qualche risorsa come cache di secondo/terzo livello o una memoria e bus di I/O.

In un multiprocessore, la struttura della memoria dipende dal numero di processori ma, dato che tale valore è destinato a variare nel tempo, si sceglie di classificare i multiprocessori in base all’organizzazione della memoria.
L’architettura a memoria condivisa centralizzata (centralized shared-memory), è impiegata da multiprocessori con un numero basso di processori che possono condividere una singola memoria centrale (Fig. 10).
La memoria ha una relazione simmetrica con tutti i processori ed è infatti accessibile in un tempo uniforme da ciascuno, per questo tali multiprocessori sono chiamati simmetrici ovvero SMP, symmetric (shared-memory) multiprocessor, e l’architettura viene anche denominata UMA (uniform memory access).

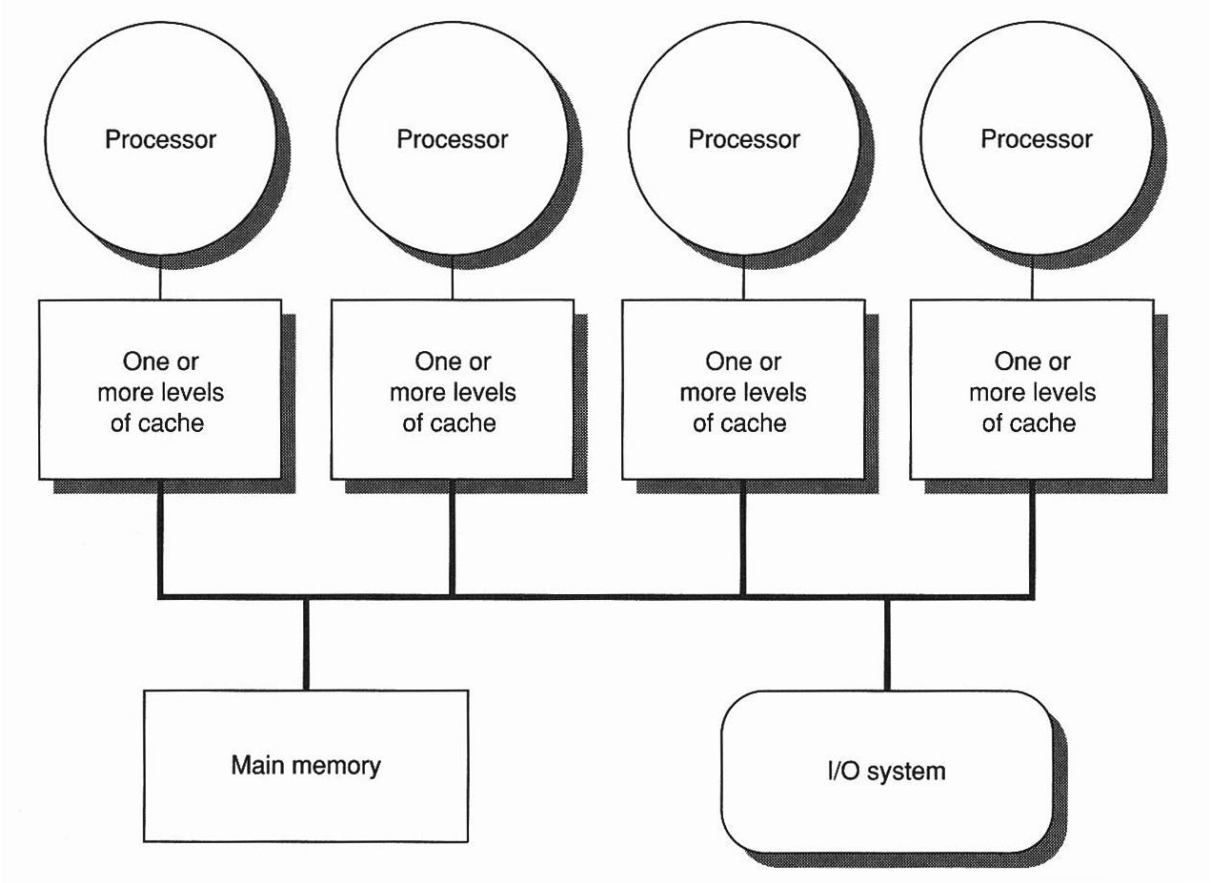

Figura 10: Struttura base di un multiprocessore con memoria condivisa centralizzata.

L'altra tipologia di organizzazione è rappresentata da multiprocessori con memoria fisicamente distribuita, necessaria per supportare l'utilizzo di un grande numero di processori, cioè di un sistema con ampie richieste in termini di larghezza di banda per la memoria (Fig. 11).

Distribuendo la memoria tra i nodi si aumenta la larghezza di banda disponibile, se la maggior parte degli accessi in memoria avviene localmente, e si riduce la latenza per l'accesso alla memoria locale, ma allo stesso tempo la comunicazione tra i processori diventa più complessa.

Esistono due alternative per lo scambio di informazioni tra i processori.

Nella prima, le memorie fisicamente distribuite, vengono indirizzate come un singolo spazio di indirizzamento logico, che viene impiegato per la comunicazione (tramite operazioni di load e store), questi multiprocessori vengono chiamati a memoria condivisa distribuita (DSM, distribuited shared-

memory) o anche NUMA (nonuniform memory access), visto che il tempo di accesso dipende dalla posizione del dato in memoria.

La seconda alternativa, consiste in molteplici spazi di indirizzamento privati, logicamente distinti, e ogni nodo composto da processore e memoria è sostanzialmente un computer a parte (infatti vengono chiamati anche multicomputer), la comunicazione in questi sistemi deve essere effettuata con il passaggio di messaggi in maniera esplicita tra i processori.

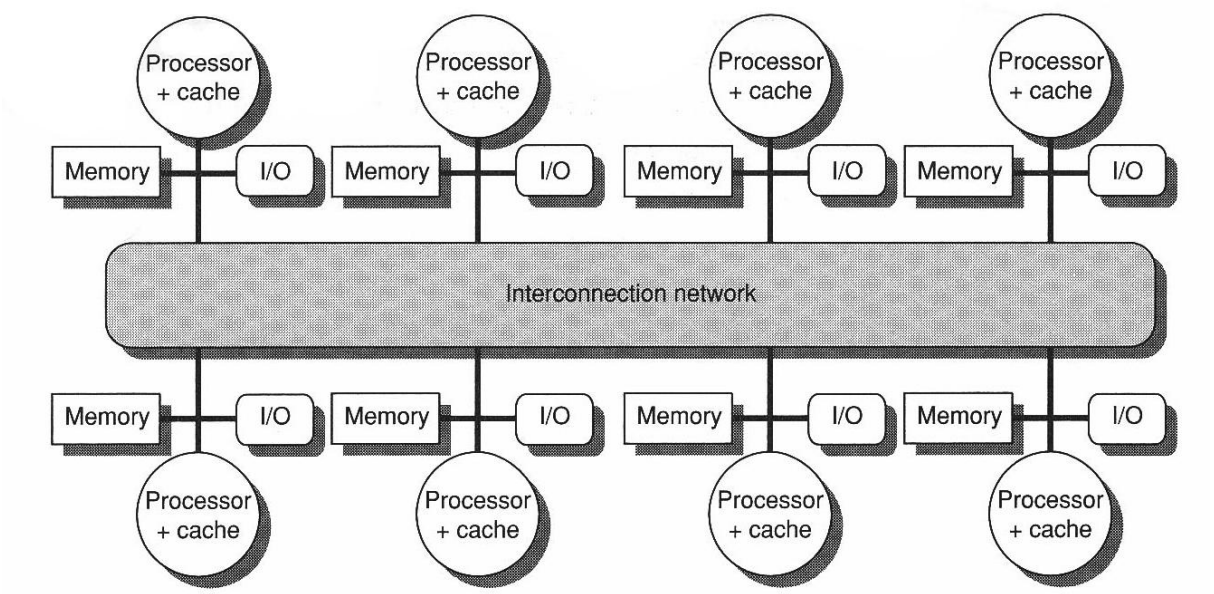

Figura 11: L'architettura di base di un multiprocessore a memoria distribuita: consiste in nodi individuali che contengono un processore, della memoria, tipicamente delle periferiche di I/O, e un'interfaccia verso una rete di interconnessione che connette tutti i nodi.

L'utilizzo di cache a più livelli può aiutare a ridurre la latenza per l'accesso locale a un'informazione e le richieste di larghezza di banda di memoria di un processore, facilitando così l'impiego di un'architettura a memoria condivisa.

Tuttavia insorgono problemi per i dati condivisi, che possono venire replicati in più cache distinte, poiché si deve garantire che le informazioni condivise siano viste nello stesso modo da tutti i processori che accedono alla propria cache locale; questi problemi prendono il nome di coerenza e consistenza della cache, e vengono risolti tramite l'impiego di hardware e protocolli appositi.

6. Conclusioni

Il parallelismo a livello di istruzione soffre di alcune importanti limitazioni, dovute al scarso grado di parallelismo intrinsecamente disponibile nei blocchi di codice, causato dalle dipendenze tra istruzioni.

Soprattutto nei processori single-core, il tentativo di sfruttare un maggior grado di ILP risulta svantaggioso, perché la quantità ristretta di benefici che si possono ottenere non riesce a giustificare i costi della progettazione di tecniche dalla maggiore complessità, anche a causa del raggiungimento di limiti fisici e tecnologici nella scalabilità dei processori singoli (consumi energetici crescenti che portano a temperature troppo elevate, divario nel tasso di miglioramento delle prestazioni tra memoria e processori, miniaturizzazione già a livelli estremi).

Viceversa, nel campo dei multiprocessori, e in particolare delle architetture multi-core, risiedono ampie possibilità di miglioramento, dato che tale ambito ha iniziato a svilupparsi solo negli ultimi anni.

L'architettura di un calcolatore, non dovrebbe più essere trasparente al programmatore, nella stessa misura in cui lo è stata in passato, poiché la conoscenza dell'hardware, permette la produzione di codice più efficace e adatto ad essere impiegato nei multiprocessori: le applicazioni parallele, poiché sono dotate di un'elevata disponibilità di parallelismo a livello di thread, permettono alle architetture parallele di raggiungere migliori performance.

Sono dunque queste le direzioni lungo le quali concentrare i futuri sforzi, al fine di ottenere un miglioramento delle prestazioni di calcolo degli elaboratori.

7. Bibliografia

John L. Hennessy, David A. Patterson, *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufman, 2006

David A. Patterson, John L. Hennessy, *Struttura e progetto dei calcolatori : l'interfaccia hardware-software*, 2. ed. condotta sulla 3. ed. americana, Zanichelli, Bologna, 2006

Materiale reperibile sul web:

Wikipedia, the free encyclopedia
(http://en.wikipedia.org/wiki/Parallel_computing)

Valeria Cardellini, Slide del corso "Sistemi Distribuiti",
Università degli Studi di Roma "TorVergata"
(http://www.ce.uniroma2.it/courses/sd0910/)

David A. Wood, Slide del corso "Advanced Topics in Computer Architecture",
University of Wisconsin, Madison
(http://pages.cs.wisc.edu/~david/courses/cs758/Fall2009/includes/lecture.html)