# Appunti di Architetture Avanzate dei Calcolatori

7 febbraio 2013

Matteo Guarnerio, Francesco Gusmeroli

Revisione 1

# Indice

# Parte I
# Introduzione

## 1 Definizioni

### 1.1 Compatibilità binaria tra processori

Tra un processore P1 (tipicamente più “vecchio”) e un processore P2.
P2 è in grado di eseguire qualsiasi programma oggetto compilato per P1.
L’insieme delle istruzioni di macchina di P1 è un sottoinsieme di quello di P2.

### 1.2 Macchina Virtuale

Se D è la descrizione di una macchina, il componente V che emula D è la macchina virtuale (Virtual Machine, VM).

### 1.3 Architettura astratta

Specifica funzionale di un calcolatore, può essere vista:

- **Dal punto di vista del programmatore:** si ha a che fare con un modello di programmazione, equivalente alla descrizione del linguaggio macchina (ISA – Instruction Set Architecture);
- **Dal punto di vista del progettista:** si ha a che fare con un modello hardware (descrizione a scatola nera fornita al progettista).

### 1.4 Architettura concreta (micro-architettura)

Implementazione di una architettura astratta;

### 1.5 Instruction Set Architecture (ISA)

Architettura astratta definita dall’insieme delle istruzioni che il calcolatore è capace di decodificare ed eseguire.

### 1.6 Paradigma architetturale

Composizione della struttura hardware e del modo di esecuzione del programma.

### 1.7 Embedded computing

Presenti in numerose applicazioni (cellulari, auto, elettrodomestici, videogiochi, etc.), di varie capacità di elaborazione e costi.

In molti casi è necessario prestare attenzione alla quantità di memoria (vincolo sul software applicativo) e ridurre il consumo di potenza (impatto sul progetto della CPU, HW e SW).

# 2 Prestazioni

Con prestazioni si indica il reciproco di tempo di risposta, il tempo di esecuzione.
Tempo di risposta minore indica prestazioni migliori.

- **Tempo di CPU**: tempo durante il quale la CPU sta eseguendo istruzioni che si divide in:
  - **Tempo di CPU Utente**: tempo speso eseguendo il programma utente.
  - **Tempo di CPU di Sistema:** tempo speso eseguendo attività del Sistema Operativo richieste dal programma.
- **Ciclo di Clock**: dipende dalla tecnologia hardware e dalla organizzazione della CPU.
- **Instruction Count (IC):** dipende dall'ISA e dalla tecnologia del compilatore.
- **CPI**: dipende dall'organizzazione della CPU e dall'ISA: $CPI = CPU_{clock\ cycles\ for\ a\ program}/IC$

## 2.1 Colli di bottiglia nel miglioramento delle prestazioni

- **Logic-Memory gap**: la memoria è più lenta della logica, memorie più veloci sono più costose.
- **Esecuzione totalmente seriale**: nella macchina di Von Neumann un'istruzione deve essere completata prima che si possa leggere l'istruzione successiva.
- **Esecuzione totalmente sequenziale**: le istruzioni vengono eseguite rigorosamente nell'ordine in cui sono state scritte dal programmatore, una per volta.

## 2.2 Come migliorare le prestazioni

- **Modificare la struttura della memoria**: il programmatore deve vedere una memoria indirizzabile molto ampia. La CPU deve vedere una "memoria equivalente" molto veloce. Il cliente finale deve trovare il costo accettabile (gerarchie di memoria).
- **Ottenere migliore efficienza nell'esecuzione del programma**: modificare il paradigma di esecuzione della sequenza di istruzioni, superando il paradigma di esecuzione seriale e sequenziale, introdurre qualche tipo di parallelismo nella CPU.
- **Superare il paradigma imperativo di esecuzione:** esecuzione dominata non più dal controllo ma dalla disponibilità dei dati.
- **Superare il vincolo di esecuzione di un singolo flusso di istruzioni**: estendere il ricorso a varie forme di parallelismo così da eseguire simultaneamente più flussi differenti.

# Parte II
# Cache

La memoria cache è una memoria temporanea, che memorizza un insieme di dati che possano successivamente essere velocemente recuperati.

Il suo collocamento funzionale: CPU $\leftrightarrow$ Cache $\leftrightarrow$ RAM

## 3 Località

Fortemente dipendente dall'applicazione.

E' **alta**, sia temporale che spaziale, per cicli interni che operano su dati organizzati in vettori. **Bassa** nel caso di ripetuto chiamate a procedura.

Principio di località: vale a molti livelli.

**Località temporale**: quando si fa riferimento a una posizione di memoria, con alta probabilità si farà di nuovo riferimento alla stessa posizione entro breve tempo.

**Località spaziale**: quando si fa riferimento a una posizione di memoria, con alta probabilità si farà entro breve tempo riferimento a posizioni vicine.

## 4 Problemi cache e soluzioni per ottimizzare

Problemi legati al livello più alto della gerarchia:

I primi tre dipendono fortemente dall'architettura. L'ultimo è ortogonale ai primi tre.

### 4.1 Block placement

Dove si può portare un blocco proveniente da un livello gerarchico inferiore.

### 4.2 Block identification

Come si trova un blocco in un livello gerarchico superiore.

### 4.3 Block replacement

Quale blocco sostituire in caso di Miss per fare posto ad un blocco del livello gerarchico sottostante (FIFO, LRU, Random).

#### 4.3.1 Cache Miss

**Regola del 2:1**: il $Miss_{rate}$ di una Cache Direct Mapped di $N$ byte è circa pari a quello di una cache set-associative a $2$ vie di $N/2$ byte.

- **Compulsory misses**: al primo accesso un blocco non è mai presente in cache. Non dipendono dalle dimensioni ne dall'architettura della cache.
- **Capacity misses**: quando si esegue un programma, alcuni blocchi devono essere sostituiti di tanto in tanto. Diminuiscono all'aumentare delle dimensioni della cache.

- **Conflict misses**: può rendersi necessario ricaricare più tardi nello stesso set i blocchi che si sono sostituiti da poco, miss da collisione. Diminuiscono all'aumentare delle dimensioni della cache e dell'associatività. Una cache completamente associativa li evita al meglio.

### 4.3.2 Cache Thrashing

Gli stessi blocchi vengono ripetutamente trasferiti alla e dalla cache.

Riduce di molto le prestazioni.

Nei sistemi application-specific è possibile identificare blocchi di dati o istruzioni che potrebbero essere soggetti a thrashing.

Per limitare il problema e garantire prestazioni deterministiche su particolari sezioni dell'applicazione:

- **Cache locking**: vincolo particolari blocchi (locked) dal programmatore (quindi la cache diventa visibile al programmatore). Utilizzata soprattutto nei sistemi embedded.

## 4.4 Write policy o Problema della Scrittura

Come gestire le scritture e modifiche dei blocchi (Write Back, Write Through). Indipendente dalla particolare architettura della cache. Si presenta sempre come problema ed è necessario che sia veloce e consistente, ovvero che l'informazione in RAM deve sempre essere consistente con quella in cache.

Due strategie fondamentali di scrittura:

- **Write-Through**: dati scritti simultaneamente nel blocco di cache e nel blocco di RAM. Consistenza sempre rispettata. Tempo di accesso per le scritture è quello della RAM, quindi le prestazioni diminuiscono. Read Miss meno costose che con Write Back, non richiedono mai di scrivere in RAM prima di effettuare una lettura. Soluzione più facile da implementare. Per essere efficiente, deve essere dotata di un **Write Buffer**, una piccola memoria FIFO, in modo che la RAM non rallenti la cache, così trasferimenti dalla cache al buffer alla velocità della cache, e dal buffer alla RAM alla velocità della RAM. Un elemento del Write Buffer include dato e indirizzo del dato. Si possono accodare tante scritture quante sono le posizioni del buffer. Quando il buffer è pieno si verificano dei **write stalls**, e la CPU deve attendere che le scritture giungano al completamento. In genere utilizzano l'opzione **No Write Allocate**.
- **Write-Back** o **Copy-Back**: i dati scritti solo nel blocco di cache. Il blocco di RAM viene modificato solo quando il blocco di cache deve essere sostituito. Subito dopo una scrittura cache e RAM sono inconsistenti. In realtà la riscrittura in RAM è necessaria solo se il blocco in cache è stato modificato. Il **blocco di cache si dice clean** se non è stato modificato, **dirty** se modificato. Ad ogni blocco di cache aggiungiamo un **dirty bit** che ne segnala lo stato e che viene posto a 1 in caso di modifiche. **La scrittura in RAM viene effettuata solo se dirty bit = 1**. **Si preferisce questa politica**, poiché c'è una buona probabilità di scritture multiple prima di dover fare una sostituzione. Le singole parole vengono scritte dalla CPU alla velocità imposta dalla cache, non dalla RAM. **Più scritture in uno stesso blocco comportano solo una scrittura in RAM**. Quando si fa una scrittura in RAM, si sfrutta la maggior larghezza del bus, poiché si trasferisce l'intero blocco. Una Miss in lettura implica che se il blocco è dirty, prima di leggere il nuovo blocco devo scrivere in RAM. In genere utilizzano l'opzione **Write Allocate**, sperando che le scritture successive coinvolgano lo stesso blocco.

### 4.4.1 Write Miss

Possono essere indotti da scritture quando vi sono tentativi di scrivere in un indirizzo non presente in cache.

Soluzioni possibili:

- **Write Allocate** chiamata anche **Fetch on Write**: carico il blocco in cache e poi effettuo la scrittura in Write-Through o Write-Back.
- **No Write Allocate** chiamate anche **Write Around**: modifico il blocco direttamente in RAM senza essere trasferito in cache.

# 5 Ottimizzazione prestazioni cache

A livello architetturale si può cercare di:

## 5.1 Ridurre $Miss_{penalty}$

Introduzione di cache multilivello.

Cache L1 piccola per essere molto veloce.

Cache L2 grande da catturare molti degli accessi che altrimenti andrebbero alla RAM.

$Miss_{penalty\,L1} = Hit_{time\,L2} + Miss_{rate\,L2} * Miss_{penalty\,L2}$

$T_A = Hit_{time\,L1} + Miss_{rate\,L1} * Miss_{penalty\,L1} = Hit_{time\,L1} + Miss_{rate\,L1} * (Hit_{time\,L2} + Miss_{rate\,L2} * Miss_{penalty\,L2})$

dove $Miss_{rate\,L2}$ si valuta sugli accessi alla cache L1 che non hanno avuto successo.

L'analisi delle prestazioni dimostra che:

- L2 è utile se molto più grande di L1
- La velocità di L1 influenza $Clock_{CPU}$. La velocità di L2 influenza $Miss_{penalty\,L1}$
- L'architettura di L2 può essere basata su meccanismi semplici (Direct Mapped)

### 5.1.1 Replicazione dei dati L1 in L2

1. **Multilevel inclusion**: dati in L1 sempre presenti in L2. Se i blocchi di L1 sono più piccoli di quelli di L2, la sostituzione diventa difficile.
2. **Multilevel exclusion**: dati in L1 mai presenti in L2. Ragionevole per L2 piccole.

### 5.1.2 Critical Word First / Early Restart

Soluzioni utili se i blocchi sono grandi.

1. **Critical Word First** o **Wrappped Fetch** o **Requested Word First**: quando Read Miss la parola mancante viene richiesta alla memoria e inviata alla CPU non appena arriva. La CPU inizia l'esecuzione mentre si riempie anche il resto del blocco.
2. **Early Restart**: le parole vengono lette nell'ordine normale, non appena la parola richiesta arriva la si invia alla CPU.

### 5.1.3 Priorità ai Read Miss rispetto alle scritture

1. **Cache di tipo Write-Through dotate di write buffer**: in caso di Read Miss, controlla il contenuto del Write Buffer, non c'è conflitto si serve il Read Miss, che riceve priorità sulla scrittura.
2. **Cache di tipo Write-Back**: si copia il blocco dirty in un buffer, poi si compie la lettura dalla memoria, infine la scrittura del blocco dirty dal buffer di memoria.

### 5.1.4 Fusione del Write Buffer

Si usa il Write Buffer anche per cache Write-Back al momento di una scrittura in RAM.

- Se WB vuoto: vi si scrivono il dato e il relativo indirizzo, la CPU ha finito la scrittura.
- Se WB contiene blocchi modificati: si controlla l'indirizzo dei dati da memorizzare. Se il nuovo indirizzo è uguale a quello di un elemento valido nel WB, i nuovi dati vengono combinati con tale elemento (**write merging**).
- **Se il WB è pieno e non ci sono coincidenze di indirizzo**: la CPU deve aspettare finché nel WB si svuota una posizione.

### 5.1.5 Victim Cache

Ciò che si è scartato potrebbe presto tornare nuovamente utile.

Inserisco una piccola cache associativa, Victim Cache, tra la L1 e il suo percorso di riempimento.

La Victim contiene solo blocchi scartati dalla cache in caso di Miss, chiamati **victims**.

In caso di nuovo Miss cerco prima nei blocchi vittima. Se i dati sono nella Victim, vengono posti in cache.

## 5.2 Ridurre $Miss_{rate}$

### 5.2.1 Blocchi più grandi

Sfruttano meglio la località spaziale.

Diminuiscono i Compulsory Miss.

Blocchi più grandi $\rightarrow$ numero minore di blocchi in cache $\rightarrow$ possibile aumento numero di Conflict Miss e anche di Capacity Miss.

$Miss_{penality}$ aumenta $\rightarrow$ cresce il tempo medio di accesso.

Le statistiche mostrano che la dimensione ottima di blocco è fra 32 bytes, per cache piccole, e 64 bytes per cache più grandi.

Il $Miss_{rate}$ aumenta se il blocco è troppo grande in rapporto alle dimensioni della cache.

### 5.2.2 Associatività più grande

Le statistiche mostrano che una cache set-associative a 8 vie porta a riduzione dei Miss quasi identica a quella di una cache totalmente associativa.

Aumenta lo $Hit_{time}$ e la circuiteria è più complessa. Si può preferire la soluzione Direct-Mapped perché consente frequenza di clock più elevata.

## 5.3 Ridurre $Miss_{penalty}$ e $Miss_{rate}$ con il parallelismo

Sovrapporre l'esecuzione del programma all'attività nella gerarchia di memorie.
La CPU continua a leggere istruzioni dalla I-cache durante un D-cache Miss.
La D-cache continua a consentire accessi anche durante un Miss (**Hit Under Miss**).
Ha senso per CPU dotate di parallelismo intrinseco.
La complessità del controllore della cache è molto maggiore.

### 5.3.1 Prefetching hardware di istruzioni e dati

Lettura anticipata di elementi da una memoria di livello inferiore a una di livello superiore prima che siano richiesti dalla CPU.

In caso di Miss la CPU legge il blocco richiesto e anche quello immediatamente consecutivo. Il blocco richiesto viene portato in cache, quello prefetched viene posto in uno **Instruction Stream Buffer** (ISB).

Se un blocco richiesto è presente nell'ISB, lo si legge dall'ISB, si cancella l'originaria richiesta alla cache, si invia la prossima richiesta di prefetching.

Si può adottare un approccio similare per la D-cache.

Ampiamente usato in architetture ad alte prestazioni.

### 5.3.2 Prefetching software - controllato dal programmatore o dal compilatore

Da parte del programmatore permette notevoli ottimizzazioni, richiede conoscenza di dettaglio dell'architettura.

Il compilatore inserisce istruzioni di prefetch per richiedere i dati prima che si debbano usare:

**Register prefetch**: carica il valore in un registro

**Cache prefetch**: carica i dati in cache (e non in un registro).

Le soluzioni possono essere **faulting** o **non faulting**, si genera o no eccezione nel caso di fault sull'indirizzo virtuale o di violazione delle protezioni.

**Nonfaulting prefetch**: si comporta come una no-op altrimenti provocherebbe un'eccezione. La maggior parte delle moderne CPU lo adottano. Approccio utile se la CPU può procedere mentre si leggono i dati prefetched, la memoria non va in stallo, continua a fornire istruzioni e dati mentre aspetta i dati prefetched.

**Prefetching più efficace**: semanticamente invisibile al programma, non modifica il contenuto di registri o memoria, ne provoca fault sulla memoria virtuale.

## 5.4 Ridurre $Cache_{miss}$

### 5.4.1 Way Prediction

Aggiungo altri bit in cache per predire il blocco dentro il set del prossimo accesso alla cache.

Un MUX nello schema di selezione dei dati viene pre-posizionato all'istante della lettura, si effettua un solo confronto sui tag, invece di confrontare tutti i tag nel set.

Se tale confronto da Miss, in cicli di clock successivi si controllano gli altri blocchi.

Nel caso di sistemi embedded, riduce i consumi.

### 5.4.2 Ottimizzazione da compilatore in compilazione

Orientate a ridurre separatamente Instruction Miss e Data Miss.

Tecniche:

**Riorganizzazione del codice**: per ridurre Conflict Miss, migliorando la località spaziale e temporale.

**Blocking**: migliora la località temporale. Per un algoritmo in cui ad alcune matrici si accede per righe e ad altre per colonne, e ad ogni iterazione si accede sia a righe sia a colonne, memorizzare le matrici per righe piuttosto che per colonne non migliora le prestazioni. Invece di operare su righe o colonne intere, l'algoritmo viene modificato così da operare su sottomatrici (blocchi). Lo scopo è massimizzare gli accessi ai dati presenti in cache prima che vengano sostituiti.

## 5.5 Ridurre $Hit_{time}$

Influenza $CPU_{clock}$

Il tempo di accesso alla cache vincola il ciclo di clock, anche quando la CPU richiede più di un ciclo per un accesso alla cache.

Architetture semplici di cache garantiscono tempo di accesso inferiore, hanno un percorso critico più breve, la Direct-Mapped può essere una soluzione.

### 5.5.1 Accesso pipelined alla cache

La latenza reale può essere di diversi cicli.

Il pipelining riduce il $T_{clock}$ ma introduce hit più lenti.

Aumenta il numero di stadi della pipeline della CPU $\rightarrow$ Aumenta la penalità quando si sbaglia la predizione sui salti, aumentando così il numero di cicli fra la richiesta di un dato e l'istante in cui il dato è disponibile.

# 6 Gerarchie di memorie

Dal livello più alto al più basso.

## 6.1 Registri

La memoria più veloce, intrinsecamente parte della CPU.

Dimensioni tipiche: $64 - 256$ parole

Struttura: banco di registri dotato di più porte di lettura e scrittura. Possibile un accesso parallelo (limitato) a più registri

Gestiti dal programmatore in ASM o dal compilatore.

## 6.2 Cache

Strutturata in blocchi o linee.

**Blocco**: minima quantità di informazione trasferibile tra due livelli di memoria (Cache-Cache o RAM-Cache).

**Dimesione (larghezza) del blocco**: influenza direttamente la larghezza del bus, che implica la banda.

Più i blocchi sono larghi, meno trasferimenti tra livelli, ma meno efficienza nell'uso della memoria e costo di bus più alto.

La **CPU** indirizza lo spazio generale di memoria considerando indirizzi fisici ed accede al livello più alto di cache.

Il **controllore della cache** determina se l'elemento richiesto è effettivamente presente in cache:

- Si $\rightarrow$ **Hit** $\rightarrow$ trasferimento tra cache e CPU.
- No $\rightarrow$ **Miss** $\rightarrow$ trasferimento da memorie di livello inferiore a livello superiore.

**Hit rate**: frazione degli accessi a memoria ricompensati da 1 hit (cifra di merito per le prestazioni della cache). $Hit_{rate}$ alto $\rightarrow$ tempo medio di accesso prossimo allo $Hit_{time}$.

**Miss rate**: frazione degli accessi a memoria cui risponde 1 miss ($Miss_{rate} = 1 - Hit_{rate}$)

**Hit time**: tempo di accesso alla cache in caso di successo (include il tempo per determinare se l'accesso si conclude con hit o miss).

**Miss penalty**: tempo necessario per sostituire un blocco in cache con un altro blocco dalla memoria di livello inferiore (si usa un valore medio). Dipende prevalentemente dalle tecnologia.

**Miss time**: $Miss_{penalty} + Hit_{time}$ tempo necessario perché la CPU ottenga l'elemento richiesto in caso di miss.

**Tempo medio di accesso**: $T_A = (Hit_{rate} * Hit_{time}) + (Miss_{rate} * Miss_{time}) = Hit_{time} + (Miss_{rate} * Miss_{penalty})$

**Memory stall cycles**: numero di cicli durante i quali la CPU è in stallo in attesa di accesso alla memoria. Il tempo di ciclo include il tempo necessario per gestire un cache hit. Durante un cache miss la CPU è in stallo.

$CPU_{exec\,time} = (CPU_{clock\,cycles} + Memory_{stall\,cycles}) * Clock_{cycle\,time}$

$Memory_{stall\,cycles} = IC * (Memory_{accesses}/Instruction) * Miss_{rate} * Miss_{penalty}$

Altra cifra di merito, indipendente dall'implementazione HW ma dipendente dall'architettura, poiché legata al numero medio di accessi a memoria per istruzione:

$Misses/Instruction = Miss_{rate} * (Memory_{accesses}/Instruction)$

**Cache di 1° livello (L1)** Tecnologia SRAM, oggi vista come parte integrante del "nucleo" che include la CPU ("Core") sul chip.

I dati in cache ricevono un nome attraverso lo spazio di indirizzamento della memoria.

Di norma la cache è "trasparente" al programmatore e al compilatore.

Può essere o unificata, un'unica cache ospita sia istruzioni sia dati, o come più abitualmente nei microprocessori moderni organizzata in Instruction-cache (**I-cache**) e Data-cache (**D-cache**) separate, ovviando così ai conflitti di accesso avendo due cache distinte.

**Cache separate possono essere ottimizzate individualmente**, le statistiche dimostrano che la **I-cache** ha miss rate più basso della **D-cache**.

**I-cache** è di tipo "read-mostly" località spaziale molto buona, salvo nel caso di chiamate a procedura molto frequenti.

**D-cache** località fortemente dipendente dall'applicazione.

L'**Accesso** può essere o mediante **indirizzo fisico** o mediante **indirizzo virtuale** (**cache virtually addressed**).

Problemi di **indirizzo virtuale**:

- Richiede meccanismi di traduzione degli indirizzi associati alla cache.
- Più processi possono usare lo stesso indirizzo virtuale per accedere a spazi di indirizzamento diversi. Occorre svuotare la cache, **cache flushing**, quando si cambia il contesto. Oppure associare ai blocchi di cache **ASID - Address Space Identifier** come tag unitamente al tag dell'indirizzo virtuale.
- Esistenza di "sinonimi": più processi possono puntare alla stessa pagina fisica che viene "condivisa".

E' $\sim 5$ volte più veloce della L2 e $\sim 100$ volte più veloce della RAM, dato da uno hit time molto minore del tempo per accedere alle memorie di livello inferiore.

**Cache di 2° e 3° livello (L2 e L3)** Se presente L2 è sullo stesso chip della CPU. Tecnologia SRAM.

L3 posizionamento e tecnologia dipendono dal progetto e produttore (es: IBM Power 7: L3 sul chip, tecnologia eDRAM).

Numero dei livelli di cache e dimensioni: determinati dai requisiti di prestazioni.

Di norma L2 e L3 sono "trasparenti" al programmatore e compilatore.

Trasferimenti gestiti da hardware.

### 6.2.1 Cache Direct-Mapped

Ogni blocco nello spazio degli indirizzi trova il suo corrispondente in uno e un solo blocco in cache.

$N_B$ numero di blocchi nella cache.

$N_R$ numero di blocchi nella RAM.

$N_W$ numero di parole nel blocco.

$B_{AC} = |B_{AR}|_{\mod N_B}$ indirizzo del blocco in cache (indice).

$B_{AR}$ indirizzo del blocco in RAM.

**Selezione della parola nel blocco**: $log_2(N_W)$ bit meno significativi dell'indirizzo della parola. Se la memoria è indirizzata al byte occorre tener conto anche di bit riservati allo spiazzamento del byte nella parola.

**Indirizzo del blocco in cache (indice della cache)**: si estrae usando i $log_2(N_B)$ bit dell'indirizzo immediatamente più significativi.

Ogni linea della cache include:

- Bit di validità
- Tag costituito dai $log_2(N_R) - log_2(N_B) - log_2(N_W)$ bit più significativi dell'indirizzo in RAM.
- Blocco di parole (i dati).

Pro e Contro:

- Non tiene conto della località temporale, il blocco sostituito potrebbe essere stato usato molto di recente.
- Implementazione facile
- Area circuitale ristretta
- E' veloce, percorso critico relativamente breve
- Non molto efficiente per la sostituzione. Facile fenomeno di "thrashing".

### 6.2.2 Cache Totalmente Associativa

Ogni blocco di RAM può essere mappato su un qualsiasi blocco di cache.

Ogni linea di cache include:

- Indirizzo completo in memoria, il Tag è costituito dall'indirizzo completo della parola.

La ricerca viene effettuata mediante confronto parallelo dell'indirizzo ricercato con tutti i Tag.

La parte di indirizzamento della cache è costituita da una CAM (Content Addressable Memory).

Pro e Contro:

- Molto complessa la CAM e consumo elevato.
- Logica di confronto lunga poiché il tag è lungo quanto tutto l'indirizzo.

### 6.2.3 Cache Set-Associativa a N vie

E' il compromesso tra soluzioni direct-mapped e associative.

La cache è organizzata come insieme di set, ognuno dei quali include N blocchi.

La RAM viene vista come organizzata in set. Ogni set della RAM viene correlato a uno e un solo set della cache, con una filosofia direct-mapped.

Placement: ogni blocco in un set della RAM può mapparsi su uno qualsiasi dei blocchi di cache nel set della cache correlato, mediante tecniche associative. Ogni blocco di memoria si può mappare su una delle N posizioni all'interno del set.

L'indice del Set in cache corrispondente a un dato set in RAM è identificato come:
$Set_{cache} = (RAM_{block\,address}) \mod (\#Cache_{sets})$
$\#Cache_{sets}$ è una potenza di 2.

L'indice della cache viene estratto usando i $log_2(\#Cache_{sets})$ bit meno significativi dell'indirizzo (a parte lo spiazzamento).

A dimensioni di cache costanti:

- Aumentando l'associatività di un fattore $2$ raddoppia il numero di blocchi in un set, dimezzando il numero di set. L'indice diventa più corto di un bit, il tag più lungo di un bit (il numero dei confrontatori raddoppia, poiché tutti i confronti sui tag sono fatti in parallelo).
- Cache set-associativa a N vie: N confrontatori in un set.

**Block Replacement policy**: Quale blocco sostituire in caso di cache miss?

1. **Random**: scelta casuale. La soluzione più semplice.
2. **LRU (Least Recently Used)**: tiene conto della località temporale. Il blocco sostituito è quello che da più tempo non viene utilizzato. Ad ogni blocco è associato un contatore all'indietro che viene portato al massimo valore in caso di accesso e decrementato di uno ogni volta che si accede a un blocco diverso. E' la soluzione più costosa in area e consumo al crescere del numero di blocchi, dato dal numero maggiore di contatori che contano tutti a ogni accesso.
3. **FIFO (First In First Out)**: si approssima la soluzione LRU determinando il blocco più vecchio, quello caricato da più tempo, invece di quello non usato da più tempo.

### 6.2.4 Cache Multi-Livello

**Local misses**: $\#Miss_{accesso\,cache}/\#Accessi_{stessa\,cache}$
**Global misses**: $\#Miss_{accesso\,cache}/\#Accessi_{totali\,generati\,da\,CPU}$
**Global miss rate L2**: $Miss_{rate\,L1} * Miss_{rate\,L2}$

## 6.3 RAM - Random Access Memory

RAM volatile: di norma in tecnologia (S-)DRAM.

Applicazioni embedded: spesso parte almeno dello spazio di indirizzamento primario è implementato con tecnologie (semi)permanenti (ROM, Flash, etc.).

## 6.4 Memoria di massa ("storage")

Tecnologie **statiche** (flash) possono sostituire i dischi (SSD), in particolare per i dispositivi mobili.

I trasferimenti sono gestiti dal Sistema Operativo (supporto HW necessario, ma in genere lo si mantiene il più possibile limitato).

Non sempre presente nei sistemi embedded.

## 6.5 Memoria Flash

Memoria statica, mantiene il contenuto anche in assenza di alimentazione.
Richiede molta meno potenza per essere alimentata in standby.
Architettura e proprietà sono molto diverse da quelle delle normali DRAM.
Cancellazione per blocchi (invece di parole o bytes) e prima di scriverci.
Per scrivere bisogna comporre un intero blocco.
Numero limitato di cicli di scrittura per blocco: ordine di $100000$.
Costa meno della DRAM ma più dei dischi.
E' $4$ volte più lenta della DRAM nel migliore dei casi, ma molto più veloce del disco.

# Parte III
# CPU

## 7 Instruction Set Architecture (ISA)

Architettura astratta definita dall'insieme delle istruzioni che il calcolatore è capace di decodificare ed eseguire.

E' la parte di architettura che è visibile al programmatore e al compilatore.

Le ISA per le tre classi di calcolatori (PC, server, embedded) sono in buona misura simili.

Per i mainframe deve mantenere la compatibilità con molte generazioni precedenti, e rispettare vincoli dovuti agli aspetti "legali" delle applicazioni (es. aritmetica decimale).

## 8 Le istruzioni macchina

Sono un punto critico per le prestazioni del calcolatore e per la definizione delle architetture concrete che implementeranno l'ISA; comprendono la scelta delle istruzioni e dei modi di indirizzamento.

**Insieme delle istruzioni:** determina il programma oggetto che viene prodotto da un programma sorgente di alto livello, influenza le prestazioni. Ci sono due diverse tendenze:

### 8.1 CISC (Complex Instruction Set Computer)

Singole istruzioni di macchina realizzano operazioni anche di notevole complessità, si consentono modalità complesse di indirizzamento, il programma oggetto risulta relativamente breve e una singola operazione complessa viene eseguita in un tempo molto minore rispetto ad una sequenza di istruzioni semplici. Questa scelta però necessita di un'unità di controllo microprogrammata ed è quindi più lenta della FSM con un periodo di clock più lungo, andando così ad incidere su tutte le operazione di sincronizzazione della CPU.

### 8.2 RISC (Reduced Instruction Set Computer)

Questa filosofia punta a semplificare l'unità di controllo e quindi ridurre il ciclo di clock oltre a ridurre il numero di modalità di indirizzamento per ottenere una maggiore regolarità nella lettura e nell'esecuzione delle istruzioni.

## 9 Modalità di indirizzamento

### 9.1 Numero degli indirizzi

La **macchina a tre indirizzi** permette una programmazione semplice, nella quale gli operandi sono espliciti.

Sorge un problema, il numero di bit necessario a codificare gli indirizzi degli operandi e dei risultati.

## 9.2 Tipo di indirizzo MIPS

Nelle **macchine con riferimento a memoria** tutte le istruzioni possono fare direttamente riferimento alla memoria con **accesso diretto** ($M[xx]$ indica la parola di memoria il cui indirizzo è precisato nella parte indirizzo dell'istruzione), con **puntatore a un indirizzo di memoria** ($M[R_i]$ indica la parola di memoria il cui indirizzo è precisato nel registro $R_i$), costruendo l'indirizzo con uno **spiazzamento** rispetto alla base contenuta in un registro ($100, (R_i)$ l'indirizzo si calcola sommando $100$ al contenuto del registro $R_i$), indicando una **parola di memoria che contiene l'indirizzo della parola che contiene l'operando** ($M(M[xx])$).

**Istruzioni di tipo** $R$

Nelle **macchine registro-registro (Load-Store)** solo le istruzioni lettura da memoria in un registro (Load) e scrittura da un registro in memoria (Store) possono accedere alla memoria.

Sono adatte a CPU con un elevato numero di registri interni di tipo generale, in cui tutte le istruzioni hanno la stessa lunghezza.

Un problema riscontrato è la generazione di un programma oggetto più lungo rispetto alle macchine precedenti.

# 10 Parallelismo

SI possono sfruttare simultaneamente più tipi di parallelismo.

Parallelismo funzionale che può essere visto a diversi livelli di astrazione.

Parallelismo intrinseco, adottato nelle moderne CPU, il quale prevede di eseguire simultaneamente più istruzioni in linguaggio macchina dello stesso programma.

## 10.1 Pipelining

Si applica a operazioni complesse che vengono ripetute identicamente su flussi continui di dati, e che possono essere frazionate in sotto operazioni simili.

All'unità funzionale unica si sostituisce una cascata di unità più semplici corrispondenti a ciascuna sotto operazione, dette stadi della pipeline.

Ogni elemento del flusso di dati passa attraverso tutti gli stadi generando una sequenza di risultati intermedi.

Non appena uno stadio ha prodotto un risultato intermedio e lo ha inviato allo stadio successivo, risulta pronto a ricevere operandi dallo stadio precedente.

Il sistema è rigorosamente sincrono.

Se $T$ è la latenza dell'intera operazione e $N$ sono gli stadi, idealmente ogni stadio ha latenza $T/N$.

Si può riassumere il tutto affermando che più unità funzionali vengono usate in sequenza per compiere una singola computazione, formando una pipeline.

Il pipelining è essenziale per rendere competitivo l'approccio RISC ma può essere presente anche nei processori CISC.

## 10.2 Replica di unità funzionali

Più unità eseguono le loro operazioni (eventualmente identiche) in parallelo su diversi dati.

Le due tecniche, pipelining e parallelismo come replica di unità funzionali, possono coesistere nella stessa architettura (i due concetti sono ortogonali).

# 11 Miglioramento delle prestazioni

**Speed-up riferito all'esecuzione di codice benchmark** è il rapporto tra il tempo di esecuzione del codice sulla macchina originale in assenza di miglioramento e il tempo di esecuzione dopo il miglioramento introdotto.

$Speedup_{tot}$: per l'esecuzione completa del codice si deve tener conto sia della parte influenzata dalla modifica sia della parte non influenzata.

## 11.1 Legge di Amdhal

$T_{exec\,new} = T_{exec\,old}((1 - fraction_{enhanced}) + fraction_{enhanced}/Speedup_{enhanced})$

$Speedup_{overall} = T_{ecec\,old}/T_{exec\,new} = 1/((1 - fraction_{enhanced}) + fraction_{enhanced}/Speedup_{enhanced})$

# 12 La macchina MIPS

E' un'architettura a $32$ bit di tipo registro-registro.

Nel Register File sono contenuti $32$ registri di tipo generale indifferenziati.

Tutte le istruzioni sono lunghe $4$ Byte.

Le istruzioni ALU hanno $2$ operandi che si trovano in registri sorgente, mentre il risultato viene scritto nel registro destinazione.

Solo le funzioni Load e Store accedono alla memoria Dati.

L'indirizzamento in memoria fa riferimento al Byte e le istruzioni di controllo, **jump** e **branch**, indicano la destinazione mediante **indirizzamento relativo rispetto al Program Counter (PC) o ad un registro base**.

## 12.1 Formati e tipi delle istruzioni

**Istruzioni di tipo L:** Load e Store nelle istruzioni macchina, si precisano codice operativo, numero d'ordine di un primo registro (sorgente per le istruzioni ALU, registro base per Load e Store o per i salti condizionati), numero d'ordine di un secondo registro (destinazione per le istruzioni ALU e per la Load, sorgente per la Store, registro su cui si verifica la condizione per i salti condizionati), un valore immediato (operando per le istruzioni ALU, spiazzamento per le altre).

**Istruzioni di tipo R**: istruzioni ALU registro-registro. Gli operandi sono in due registri sorgente, il risultato va in un registro destinazione, tutti appartenenti al Register File.

**Istruzioni di tipo J:** salti incondizionati, chiamate a subroutine (indirizzamento relativo al PC).

## 12.2 Pipelining

Fasi possibili nella vita di un'istruzione:

1. **(IF) Fetch (lettura):** l'istruzione viene letta da memoria e inserita nell'Instruction Register: $IR \Leftarrow M[PC]$. Nella stessa fase, il PC viene incrementato di 4.
2. **(ID) Decodifica dell'istruzione / Lettura dei Registri:** l'istruzione viene decodificata. Si leggono i registri in cui gli indirizzi vengono prelevati dai bit $21-25$ e $16-20$ dell'istruzione. Si porta il loro contenuto in due latch A and B. I bit $0-15$ dell'istruzione vengono interpretati come se fossero un valore immediato che viene letto ed esteso a $32$ bit.

3. **(EXE) Esecuzione o calcolo di un indirizzo di memoria:** fase che coinvolge l'ALU per qualunque tipo di istruzione. Nessuna istruzione MIPS richiede l'esecuzione di un'operazione ALU e anche l'accesso alla memoria dati (non verranno mai richieste simultaneamente). Funzioni eseguite: **Istruzioni Load/Store:** il valore immediato precedentemente esteso a $32$ bit (che qui rappresenta uno spiazzamento) viene sommato al registro base. Il risultato viene scritto nel registro d'indirizzamento della D-cache. Per una una store-store. Il valore in $R_d$ viene trasferito al registro di scrittura della D-cache in vista della scrittura in memoria. **Istruzioni ALU**: i contenuti di $R_{s1}$ and $R_{s2}$ vengono inviati all'ALU insieme ai comandi di funzione.
4. **(MEM) Accesso alla memoria / completamento dei salti condizionati: Load/Store:** si invia il segnale di Read/Write alla D-Cache. **Load**: i dati vengono trasferiti dalla cache al registro di lettura. **Store**: i dati presenti nel registro di scrittura vengono scritti in memoria. **Salti Condizionati:** si verifica la condizione, se è soddisfatta l'indirizzo destinazione (calcolato nella fase 3) viene trasferito al PC, altrimenti si mantiene valido l'indirizzo sequenziale.
5. **(WB) Write Back (Scrittura) nel Register File**: **Load**: i dati presenti nel registro di lettura/scrittura vengono scritti nel Register File. **Istruzioni ALU**: il risultato dell'operazione viene scritto nel registro destinazione.

## 12.3 Data Path

Il percorso di elaborazione della CPU, chiamato Data Path, è di tipo **pipelined**.

Ogni stadio della pipeline corrisponde ad una fase.

**Tutte le istruzioni attraversano tutti gli stadi**, anche se non attive nella fase corrispondente.

**Tutti gli stadi sono sincroni**, l'attraversamento di ogni stadio richiede lo stesso tempo, ovvero tutte le istruzioni hanno identica latenza.

Non appena l'istruzione $i$ ha lasciato uno stadio, l'istruzione $i+1$ può occupare lo stesso stadio.

Idealmente **ad ogni ciclo si può leggere/completare una nuova istruzione** $\rightarrow$ il CPI diventa asintoticamente uguale a $1$.

## 12.4 Problemi

La presenza nella pipeline di più istruzioni simultaneamente attive può portare a vari tipi di conflitti che si possono trasformare in **alee** (cause di errore).

Un esempio di errore si ha in presenza di accesso alla medesima risorsa nello stesso ciclo di clock. In questo caso si possono introdurre dei cicli vuoti, riducendo quindi il throughput. Una possibile soluzione la si ha con la replicazione delle risorse, evitando così i conflitti.

Per il funzionamento ideale: $CPI = 1$

Nel funzionamento reale, con la comparsa di conflitti e alee, occorre modificare sia il compilatore che lo HW, quindi $CPI > 1$ con conseguente peggioramento delle prestazioni.

### 12.4.1 Alee e dipendenze

Le alee sono dovute a dipendenze fra istruzioni.

- **Dipendenze sulle risorse:** due diverse istruzioni richiedono la stessa risorsa per essere eseguite. Per eliminare questo problema si può fare attenzione in fase di compilazione (**soluzione SW**) ponendo le istruzioni che richiedono la stessa risorsa a debita distanza le une dalle altre, altrimenti si fraziona o si replica la risorsa (**soluzione HW**).

- **Dipendenze di dati:** due diverse istruzioni hanno in comune un operando in un registro o in memoria.
  - **Dipendenza RAW (Read After Write)**: l'istruzione $i_{k+1}$ richiede come operando sorgente l'operando di destinazione $i_k$.
  - **Dipendenza WAR (Write After Read)**: l'operando di destinazione dell'istruzione $i_{k+1}$ è un operando sorgente di $i_k$.
  - **Dipendenza WAW (Write After Write)**: le istruzioni $i_{k+1}$ e $i_k$ hanno lo stesso operando di destinazione.
  - Per ovviare a queste dipendenze è necessario modificare il programma o lo HW. RAW è **dipendenza "vera"**, WAR e WAW sono dovute al riuso di registri o indirizzi di memoria da parte del compilatore e si dicono **dipendenze "false"**.
- **Dipendenze di dati tra iterazioni dei cicli:** un'istruzione eseguita in un'iterazione dipende da istruzioni di iterazioni precedenti di norma si risolve via SW.
- **Dipendenze di controllo:** dovute alle istruzioni che modificano il flusso di controllo, possono ridurre le prestazioni delle CPU con parallelismo intrinseco. Possono generare errori se si avviano all'esecuzione le istruzioni successive in sequenza a quella di controllo. **Salti incondizionati**: si riconoscono nella fase ID, si annulla l'istruzione appena letta e si legge la nuova istruzione (generando della latenza)**. Salti condizionati**: occorre aspettare che la condizione sia stata valutata per stabilire qual'è la prossima istruzione.

### 12.4.2 Conflitti define-use sui dati

Una volta scoperto un problema di dipendenza dati lo si può risolvere via SW o via HW.

- **Soluzione SW:** il compilatore inserisce tante istruzioni non dipendenti quante ne sono necessarie tra la define e la prima use (spostamento di codice, non influenza le prestazioni), altrimenti si inseriscono delle NOP (No OPeration, aumentano il PC e basta) tra la define e la prima use, peggiorando le prestazioni.
- **Soluzione HW:** l'equivalente HW della NOP via SW.
  - Si bloccano (**stall**) le istruzioni che seguono la define fino a che non è risolto il conflitto. Il blocco della CPU per un ciclo è detto **inserimento di una bolla**. La perdita di throughput è equivalente all'inserimento di NOP, ma non aumenta l'occupazione di memoria.
  - Un'altra soluzione HW è **modificare il circuito del Data Path**, si introduce la tecnica del **data forwarding**, se le dipendenze sono rispetto a dati che si trovano "a sinistra" non occorrono stalli. Non c'è perdita di prestazioni.

**Esempio:**
LW $17, 200($19)
ADD $12, $17, $18
**Soluzione software:** come per i conflitti define-use.
**Soluzione hardware:** anche ricorrendo al data forwarding rimane necessario l'inserimento di una bolla. I dati sono disponibili solo al termine dello stadio MEM, nuovi percorsi di bypass.

### 12.4.3 Conflitti di controllo

Meno frequenti di quelli define-use ma con soluzioni meno efficienti.

- **Stallo automatico:** ogni volta che si decodifica un salto-salto , la CPU va in stallo finché non si è risolta la condizione del salto (l'istruzione avviata nella pipeline dopo quella di controllo viene annullata). Penalizzazione inutile se non si fa il salto, CPI aumenta più del necessario. L'unità di controllo si comporta in modo "irregolare" per un'istruzione di controllo, generando stalli invece di continuare il flusso nella pipeline.
- **Branch hosting:** su CPU "vecchie". L'unità di controllo non "tratta" la branch in modo diverso dalle altre istruzioni, la fa procedere e intanto avvia nella pipeline le istruzioni successive senza "porsi problemi". Il compilatore deve spostare la branch tre istruzioni "prima" di quanto richiesto e le tre istruzioni che seguono vengono sempre eseguite. Quando queste sono completate, il risultato del salto condizionato è noto, il valore corretto del prossimo indirizzo è nel PC, non occorrono stalli. (La soluzione ovviamente può portare a inserire delle NOP).
- **Branch prediction:** di gran lunga la tecnica più usata, ha molte soluzioni alternative:
  - **Branch not taken:** si predice che il salto non sia mai effettuato. Le istruzioni che seguono il salto sono lette e inviate alla pipeline. Se il salto deve essere fatto, si svuota la pipeline dalle istruzioni lette dopo quella di salto (**flushing**), impedendo che vengano completate e modifichino lo stato della macchina. Si riprende la lettura dall'istruzione destinazione del salto.
  - **Branch taken:** Si predice che il salto sia sempre effettuato. L'istruzione che segue immediatamente il salto viene annullata e si resta in stallo per un ciclo, in attesa che l'ALU abbia calcolato l'indirizzo destinazione del salto. La lettura riprende poi da tale indirizzo perdendo così due cicli. Se il salto non deve essere fatto, si svuota la pipeline dalle istruzioni lette e avviate nella pipeline, impedendo che vengano completate e modifichino lo stato della macchina. Si riprende la lettura dall'istruzione successiva a quella di salto.

### 12.4.4 Gestione delle eccezioni

Le eccezioni possono essere dovute a:

- **Condizioni in fase di esecuzione** (es: overflow, divisione per 0): il programma di gestione dell'eccezione deve essere in grado di identificare le condizioni che hanno condotto all'eccezione. L'utente può voler conoscere il risultato dell'istruzione che ha provocato eccezione. L'eccezione viene sollevata nel corso dello stadio EXE. PC salvato = PC dell'istruzione successiva.
- **Interrupt esterni**: non esiste a priori una specifica fase di un'istruzione associata a un interrupt. Convenzionalmente l'interrupt può essere associato all'istruzione nella fase di WB. Lo stesso vale per l'eccezione dovuta all'identificazione di un guasto hardware.
- **Chiamate a SO**.
- **Page Fault**.
- **Uso di codici operativi non definiti**: può essere dovuto a un errore del compilatore, ad un guasto di memoria, a istruzioni che invocano unità funzionali non presenti nel particolare dispositivo usato o a istruzioni che invocano l'uso di un coprocessore. Si solleva l'eccezione nello stadio ID. L'indirizzo di rientro è nel buffer IF/ID.
- **Malfunzionamenti HW**: vedi interrupt esterni.

# 13 Miglioramento delle prestazioni

## 13.1 Superpipelining

**Macchine pipelined:** CPI reale della pipeline derivato come:

$CPI_{pipeline} = CPI_{ideale} + Stalls_{structura} + Stalls_{Data\,hazard} + Stalls_{Control} \geq CPI_{ideale}$

Riducendo un qualsiasi termine a destra significa ridurre $CPI_{pipeline}$, altre tecniche per migliorare $CPI_{pipeline}$ possono creare problemi con le alee.

Per ottenere prestazioni più elevate quindi si cerca di aumentare il numero di istruzioni simultaneamente attive all'interno della CPU, quindi si riduce il tempo medio di esecuzione per istruzione:

$TPI = T_{clock} * CPI$

Si deve quindi estrarre **maggior parallelismo a livello di istruzione (ILP - Instruction Level Parallelism)**.

In teoria $T_{clock}$ **è inversamente proporzionale al numero degli stadi della pipeline** $\rightarrow$ aumentando il numero degli stadi diminuisce il TPI.

La riduzione della profondità degli stadi (aumento della profondità della pipeline) è limitata per i seguenti motivi:

- **Buffer inter-stadio:** il numero di livelli logici nel buffer deve essere minore del numero di livelli logici nello stadio: $\#Livelli\,logici_{Buffer} < \#Livelli\,logici_{Stadio}$
- **Problemi di progetto elettronico**: ad esempio la **clock skew** (sfasamento nell'arrivo dello stesso segnale su più registri, può essere positivo o negativo. Se completamente sincronizzato è zero, come nel caso ideale).
- **Ritardi Define/Use e Load/Use:** aumentando la profondità della pipeline, diventano maggiori i ritardi in termini di numero di cicli (in caso di conflitto, si deve inserire un maggior numero di bolle), ovvero **aumenta la perdita di prestazioni dovuta ad alee sui dati**.
- Per lo stesso motivo, le **alee di controllo** implicano salti condizionati più lenti (una speculazione errata porta al "flushing" di un maggior numero di stadi).

In altre parole il **superpipelining** consente un tempo di ciclo più breve, ma spesso introduce anche un maggior numero di cicli richiesti, e al crescere della frequenza di clock aumenta il consumo di potenza.

Si sfruttano altre forme di **ILP**.

## 13.2 Pipelining parallelo

### 13.2.1 Speed-up potenziale

Supponendo di avere un oracolo perfetto, si potrebbe astrarre parallelismo per alcune centinaia di istruzioni.

Prendendo in esame dei sistemi dedicati ad una specifica applicazione, si hanno degli speed-up maggiori in quanto si ha a che fare con un programma applicativo perfettamente noto, sono disponibili dati reali e gli effettivi risultati di profiling sono usati come base per le politiche di scheduling.

### 13.2.2 Macchina scalare

Un vincolo pesante della pipeline è il fatto che tutte le istruzioni attraversano tutti gli stadi anche se non attivano le unità funzionali.

Un modello evoluto di pipeline prevede una **diramazione** dopo lo stadio di decodifica, le istruzioni sono decodificate una alla volta, ma le tre pipeline possono essere attive contemporaneamente dopo che l'unità di controllo ha verificato che non esistono dipendenze tra le istruzioni attive.

Se esiste una dipendenza, la pipeline interessata viene posta in attesa.

In ogni caso, decodificando un'istruzione alla volta, non si potrà mai raggiungere $CPI < 1$. Per raggiungere questo risultato occorre avere più pipeline parallele.

Quando sono presenti più pipeline, si presentano errori anche per i conflitti WAW e WAR dovuti al completamento fuori ordine. Per ovviare a questo problema si possono verificare le dipendenze tramite l'unità di controllo, altrimenti si modifica l'architettura HW per forzare le istruzioni ed attraversare lo stadio WB in ordine.

### 13.2.3 Intel Atom

Capace di attivare due istruzioni per volta, l'ordine dell'esecuzione è garantita dal bloccaggio delle istruzioni a latenza breve avviate successivamente.

## 13.3 Pipelining parallelo - Problemi e soluzioni

Più pipeline in parallelo possono generare problemi di dipendenza dati tra diverse pipeline, alla terminazione fuori programma di diverse istruzioni e devono mantenere la consistenza sequenziale, ovvero i risultati devono essere identici a quelli che si avrebbero con la macchina di Von Neumann, occorre quindi prevedere un **riordinamento.**

Occorre identificare e risolvere le dipendenze per predisporre per l'esecuzione in modo da raggiungere il massimo parallelismo possibile.

### 13.3.1 Scheduling dinamico e calcolo Speed-Up

Le istruzioni vengono lette e lanciate per l'esecuzione in ordine di programma.

Lo HW riordina l'esecuzione delle istruzioni a runtime in modo da ridurre gli stalli.

Consente di gestire le dipendenze che non si possono identificare al momento della compilazione.

Cresce notevolmente la complessità dello HW di controllo.

Il codice oggetto può essere indistinguibile da quello per un processore scalare.

L'esecuzione fuori ordine introduce alee di dati WAR e WAW oltre alle RAW.

**Calcolo speed-up**: $SpeedUp_{sc} = T_s/T_{sc}$ con $T_s$ tempo di esecuzione di CPU scalare. $T_{sc}$ tempo di esecuzione CPU superscalare.

Se $SpeedUp_{sc} > 1 \rightarrow$ CPU superscalare più veloce.

### 13.3.2 Branch prediction dinamica: rilevamento e risoluzione dinamica delle dipendenze

In ogni ciclo le istruzioni della finestra di issue vengono controllate riguardo alle dipendenze di tutte le istruzioni della finestra di esecuzione e di issue.

Dopodiché si lanciano le istruzioni che possono essere lanciate.

Date infinite risorse, il parallelismo è limitato solo dalle caratteristiche del programma applicativo, ovvero dalle dipendenze all'interno di ogni blocco basico (insieme di $4-7$ istruzioni).

### 13.3.3 Issue multipla: esecuzione parallela con più unità di esecuzione, architetture superscalari

Sono necessarie più pipeline di esecuzione e più capacità di lettura per poter lanciare e decodificare più istruzioni per volta.

**Limite teorico**: avendo $n$ pipeline di esecuzione, se sono state lette e decodificate $n$ istruzioni prive di dipendenze di dati o di controllo e compatibili con le pipeline libere, tutte le $n$ istruzioni vengono lanciate in esecuzione simultaneamente.

Esistono meccanismi che **garantiscono l'aggiornamento dello stato** in modo da supportare la consistenza sequenziale.

Come conseguenza $CPI < 1$.

Idealmente con $n$ numero di istruzioni lette ed eseguite contemporaneamente: $CPI = 1/n$

chiaramente il numero di istruzioni trasferite da cache a CPU deve essere superiore ad una parola.

**Singole attività**:

- **Fetch parallelo**: lettura parallela di più istruzioni nello stesso ciclo.
- **Decodifica parallela**: la complessità aumenta con il grado di parallelismo.
- **Instruction Issue superscalare**: le istruzioni vengono analizzate per identificare possibili dipendenze e lanciate su diverse pipeline di esecuzione. Più istruzioni da lanciare aumentano i problemi di dipendenza. Si verificano più **dipendenze di controllo** rispetto ai processori scalari, è quindi necessario introdurre delle politiche di lancio per evitare le **dipendenze di dati**.
- **Esecuzione parallela delle istruzioni**: prevede il coinvolgimento di più unità funzionali, le istruzioni sono **lanciate in base alla disponibilità dei dati**, introducono delle **esecuzioni fuori ordine** e nasce il problema di conservare la **consistenza sequenziale**.
- **A valle dell'esecuzione fuori ordine**: lo stato della macchina viene aggiornato seguendo il corretto ordine di programma, anche in caso di eccezioni.

La complessità della **decodifica superscalare** rischia di dilatare il tempo di ciclo, si ricorre quindi alla **pre-decodifica**.

In fase di caricamento dalla RAM vengono decodificate in parte le istruzioni o delle classi di esse.

Vengono aggiunti dei bit che indicano la classe dell'istruzione, il tipo di risorse richieste e viene effettuato il pre-calcolo dell'indirizzo obiettivo del salto.

Con **Issue Policy** si intendono le specifiche di gestione delle dipendenze in fase di lancio.

**Issue Rate** indica il numero massimo di istruzioni che la CPU è in grado di inviare in un ciclo.

Le Issue Policy superscalari includono:

- Gestione delle **false dipendenze di dati**: le dipendenze WAR/WAW vengono superate ridenominando i registri, il risultato viene scritto in un opportuno spazio allocato dinamicamente.
- Gestione delle **dipendenze di controllo non risolte**: gestione dei salti condizionati quando la condizione non è ancora stata risolta. O si attende fino a quando la condizione referenziata non è diventata disponibile, oppure si adotta una esecuzione speculativa.
- Uso di **tecniche di shelving** per garantire frequenza di issue elevata.
- Gestione del blocco dell'issue.

### 13.3.4 Speculazione: vedi Esecuzione Parallela più avanti

## 13.4 Tecniche di Shelving

Tecniche per evitare il blocco della issue provocato da dipendenze di dati non risolte.

O si consente che le istruzioni dipendenti provochino il blocco di istruzioni successive (**blocking issue**) oppure si evita il blocco mediante shelving.

Il **non-blocking issue mediante shelving** prevede di disaccoppiare issue e controllo delle dipendenze, è necessario introdurre dei buffer (**reservation station**) in testa alle unità funzionali.

La gestione delle dipendenze viene effettuata da una logica distribuita nelle reservation station, le istruzioni liberate dalle dipendenze vengono poi inoltrate alle unità funzionali, a patto che ci siano risorse disponibili per l'esecuzione dell'istruzione.

Ci sono molte tecniche di shelving, per esempio con le **reservation stations individuali** si ha una station per ogni unità funzionale, con le **stazioni di gruppo** una stazione contiene istruzioni per un gruppo di unità funzionali, la tecnica di **reservation station centrale** prevede di avere una station più grande delle stazioni di gruppo, ovvero capace di accettare/fornire più istruzioni per ciclo.

Un ultimo approccio prevede un buffer combinato, **ReOrder Buffer (ROB)**, per sheving, ridenominazione e riordino.

In ogni caso le politiche di shelving prevedono scelte per:

- **Politica di dispatch** (come scegliere le istruzioni per l'esecuzione e come gestire i blocchi del dispatch): sono eseguibili le istruzioni i cui operandi sono disponibili, quindi non sono utilizzati da nessun altro. Se il numero di istruzioni eseguibili è maggiore del numero di risorse disponibili, in genere si eseguono le istruzioni in attesa da più tempo.
- Frequenza di dispatch: numero massimo di istruzioni inviate in un ciclo dalle reservation stations.
- Controllo della disponibilità degli operandi.
- Gestione di una reservation station vuota (la stazione può essere bypassata o no).

### 13.4.1 Le false dipendenze: ridenominazione

La ridenominazione procede allo scheduling per l'esecuzione delle istruzioni di una reservation station e le invia alle unità funzionali allocate.

Utilizza tecniche per rimuovere le false dipendenze (WAR, WAW) e può essere **statica** o **dinamica**.

Nel primo caso avviene durante la compilazione, nel secondo durante l'esecuzione prevedendo una ridenominazione parziale per certi tipi di istruzioni, o completa per tuttte le istruzioni eleggibili. Può essere effettuata attraverso un buffer di ridenominazione.

La ridenominazione avviene attraverso le seguenti tecniche:

- **Fusione del Register File architetturale con Register File di ridenominazione:** i Register File architetturale e di ridenominazione vengono allocati dinamicamente a particolari registri dello stesso RF fisico.
- **Register File architetturale e di ridenominazione separati:** l'istruzione specifica il registro destinazione $R_d$ che provocherebbe false dipendenze. Si alloca a $R_d$ un nuovo registro di ridenominazione, che resta valido finché o un'istruzione successiva fa riferimento allo stesso $R_d$, e il registro architetturale è riallocato a un diverso registro di ridenominazione, o l'istruzione giunge a completamento e l'allocazione perde validità.

- **Uso del ReOrder Buffer (ROB)** per implementare i registri di ridenominazione: ad ogni istruzione al momento dell'issue, viene allocato un diverso elemento del ROB. I risultati intermedi vengono registrati in quello stesso elemento.

Per gestire la ridenominazione mediante reservation station in presenza di salti è stato proposto l'**algoritmo di Tomasulo**.

### 13.4.2 Algoritmo di Tomasulo

Garantisce l'esecuzione di un'istruzione solo quando gli operandi sono disponibili, evitando alee RAW.

Sono presenti più Reservation Station (RS) che registri fissi, perciò si eliminano anche le alee dovute a false dipendenze.

- All'**issue** di un'istruzione i cui operandi non sono pronti, i nomi dei corrispondenti **registri sorgente vengono sostituiti** con quelli delle RS dove sono registrate le istruzioni che li forniranno, eliminando così la necessità di ridenominare i registri. In questo modo vengono eliminate le alee WAR e WAW.
- **Esecuzione**: dopo aver controllato se gli operandi sono disponibili si passa all'esecuzione. Se il controllo è negativo, si ritarda l'esecuzione evitando alee RAW. Per effettuare Load e Store si calcola preventivamente l'indirizzo di memoria, le Load vengono eseguite non appena l'unità di memoria è disponibile, le Store aspettano che sia disponibile il valore da memorizzare. Nessuna istruzione può iniziare l'esecuzione fino a che i salti condizionati che la precedono in ordine di programma non siano stati risolti. In alternativa si usa branch prediction o speculazione.
- **Scrittura dei risultati**: quando il risultato è disponibile, lo si scrive sul **Common Data Bus**. Dal CDB passa ai registri del Register File e a qualsiasi Reservation Station che lo attende.

Una Load e una Store possono essere eseguite in ordine diverso da quello di programma, purché accedano a indirizzi di memoria diversi, altrimenti si potrebbe avere un conflitto WAR (scambiando la sequenza load-store) o RAW (scambiando la sequenza store-load) oppure WAW (scambiando due store).

Si possono riordinare liberamente le Load.

#### Vantaggi

- Grazie all'uso delle Reservation Stations e del CDB, **la logica per l'identificazione delle alee è distribuita**. Se più istruzioni attendono uno stesso risultato, e ognuna ha già disponibile l'altro operando, tutte le istruzioni possono essere avviate simultaneamente in esecuzione non appena il risultato viene diffuso sul CDB, eliminando il conflitto per l'accesso al Register File.
- Si eliminano alee WAW e WAR grazie alla implicita ridenominazione dei registri mediante le RS e alla memorizzazione degli operandi nelle RS non appena diventano disponibili.

#### Svantaggi

- **Complessità HW:** ogni RS necessità di un buffer e di una logica di controllo complessa.
- **Limite di prestazioni dovuto al Common Data Bus:** costituisce un punto di serializzazione, aggiungendo più CDB ognuno deve interagire con ogni RS e bisogna duplicare lo HW per la verifica associativa dei tag.

### 13.4.3 Reservation Station: componenti

- Tag: identifica la RS.
- OpCode (Operation Code).
- $V_j\, V_k$ valori degli operandi sorgente.
- $Q_j\, Q_k$ puntatori alle RS che produrranno $V_j V_k$.
- Busy indica che la RS è occupata.

All'interno del Register File ogni registro ha un campo $Q_j$ che ospita il numero della RS che contiene l'istruzione il cui risultato verrà scritto nel registro.

$Q_j$ vuoto indica che nessuna istruzione attiva sta calcolando un risultato destinato a quel registro, quindi il valore presente è utilizzabile.

## 13.5 Esecuzione Parallela

Istruzioni eseguite in parallelo, in genere finiscono fuori ordine.

Si distinguono le seguenti possibilità:

- L'istruzione è **finita**: l'operazione richiesta dal codice operativo è completata.
- L'istruzione è **completata**: il risultato è stato scritto nel registro destinazione o in memoria.
- L'istruzione è **committed** (retired): così si definisce il completamento quando l'architettura include il ReOrder Buffer (ROB).

### 13.5.1 Consistenza Sequenziale

Si riferisce all'ordine di completamento delle istruzioni per quanto riguarda la **consistenza del processore**:

- **Debole** (**Weak**): le istruzioni **possono essere completate fuori ordine**, purché non si sacrifichi alcuna dipendenza di dati.
- **Forte** (**Strong**): le istruzioni **vengono forzate al completamento strettamente in ordine di programma**, usando il ReOrder Buffer (ROB).

Si riferisce all'ordine di accesso alla memoria per la **consistenza di memoria:**

- **Debole** (**Weak**): **gli accessi alla memoria possono essere fuori ordine**, purché non si violi alcuna dipendenza di dati. E' possibile il riordinamento Load/Store, consentendo migliori prestazioni.
- **Forte** (**Strong**): **gli accessi alla memoria si verificano strettamente in ordine di programma**.

Un modello di consistenza sequenziale di un processore **integra tutti e due gli aspetti**, le alternative sono WW, WS, SS, SW (tutte le combinazioni possibili), i processori più recenti tendono ad usare l'ultimo dei quattro.

### 13.5.2 Riordinamento Load/Store

Una volta calcolato l'indirizzo di memoria, una **Load** accede alla D-cache per leggere il dato che viene reso disponibile per il registro destinazione (Load finita).

La Load è completata quando il dato viene effettivamente scritto nel registro architetturale, una **Store** deve attendere che l'operando sia disponibile.

Quando si è in presenza di una consistenza di memoria debole, è possibile **riordinare gli accessi di memoria**:

- Load/Store bypassing.
- Rende fattibili Load/Store speculative.
- Consente di mascherare i cache miss.

Una Load può scavalcare una Store non ancora completata o viceversa, purché non si violi alcuna dipendenza di dati in memoria.

O si ritarda l'esecuzione del bypassing (**esecuzione non speculativa del bypass**), o si accede alla memoria nonostante le verifiche di indirizzo non siano ancora risolte.

Infatti con molta probabilità l'indirizzo nella Load sarà diverso da quello delle Store precedenti non ancora completate.

Una volta che gli indirizzi delle store sono noti, si procede alla **verifica della correttezza speculativa**. In caso di coincidenza degli indirizzi, la Load viene annullata e rieseguita.

### 13.5.3 ReOrder Buffer (ROB)

Il ROB è un buffer circolare con un puntatore di testa che indica il prossimo elemento libero ed un puntatore di coda che indica l'istruzione che per prima giungerà a commit.

Le istruzioni vengono inserite nel ROB in ordine di programma, quando si lancia un'istruzione la si alloca nel ROB in sequenza.

Un'istruzione giunge a commit solo se è finita e tutte le istruzioni precedenti sono già giunte a commit.

Il ROB contiene i seguenti campi:

- **Tipo di istruzione**: indica se l'istruzione è un salto condizionato, una Store una Load o un'istruzione ALU.
- **Indirizzo**: dell'istruzione stessa.
- **Destinazione**: è il numero del registro (per Load e istruzioni ALU) o l'indirizzo in memoria (per le Store) in cui scrivere il risultato.
- **Valore:** usato per contenere il valore del risultato in attesa che l'istruzione giunga a commit.
- **Stato dell'istruzione**: lanciato (i), in esecuzione (x), finito (f).

Il ROB può supportare sia l'**esecuzione speculativa** che la **gestione delle eccezioni**.

Nel primo caso ogni elemento del ROB viene esteso per includere un campo di stato speculativo, che indica se l'istruzione è stata eseguita in modo speculativo. L'istruzione non può avere stato commit finchè è in stato speculativo, occorre quindi controllare se la speculazione risulta corretta, e quindi si annulla lo stato speculativo dell'istruzione, oppure se risulta errata, svuotando il ROB delle istruzioni eseguite in modo speculativo.

Le **eccezioni** vengono gestite in ordine, accettando la richiesta di eccezione solo quando l'istruzione diventa prossima al commit. Le **interruzioni esterne** si associano all'istruzione in coda al ROB che viene svuotato dal contenuto successivo all'istruzione associata all'eccezione.

### 13.5.4 Predizione dei salti: Branch Prediction

La soluzione più semplice si basa sul fatto che quando si esegue nuovamente una branch, con alta probabilità la valutazione della condizione è la stessa fatta in precedenza.Questo comporta la creazione Speculazione: vedi Esecuzione Parallela

## 13.6 Tecniche di Shelving

Sono basate su **Branch Prediction Buffer (BPB)**.

E' una memoria indirizzata mediante i bit meno significativi dell'indirizzo di istruzione di salto.

Dopo la fase di ID si verifica se il salto è già stato eseguito e, in caso affermativo, si predice che l'esecuzione sia quella precedente.

Se si scopre che la predizione è errata si inverte il bit di predizione e si registra nel buffer il nuovo valore.

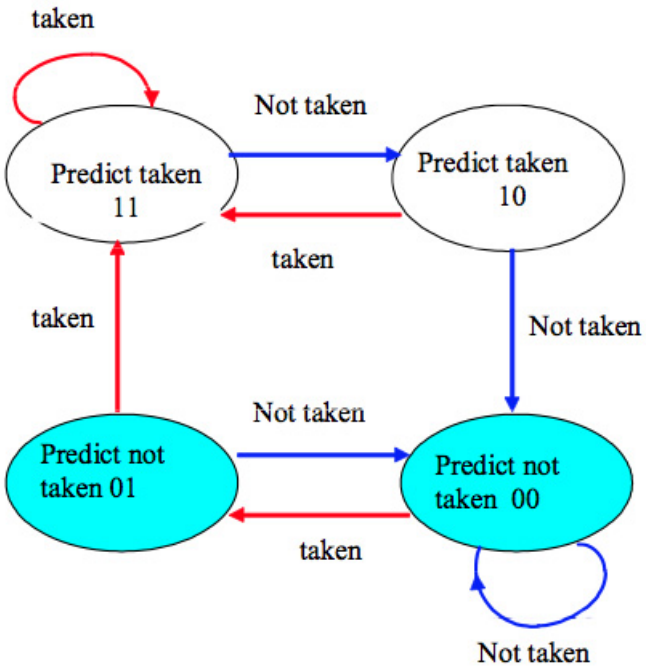
In un ciclo, almeno $2$ volte la predizione risulterà errata: all'inizio, se la predizione era "non fatto" e alla fine, quando la predizione è inevitabilmente errata, in quanto precedentemente il ciclo era stato eseguito.

Con l'introduzione di più bit per la predizione, si incrementa il valore per ogni salto fatto, si decrementa per ogni salto non fatto.

La **predizione è taken** se il bit più significativo è $1$ (quindi quando il contatore ha valore maggiore o uguale alla metà del massimo valore esprimibile con $n$ bit ($Contatore \geq 2^n/2$)).

Altrimenti la **predizione è not taken**.

Gli schemi a $2$ bit funzionano quasi altrettanto bene rispetto a quelli con $n > 2$.




Lo schema base del BPB si basa sul comportamento dell'ultimo salto.

Per migliorare l'accuratezza occorre considerare il comportamento anche di altri salti.

Un ulteriore passo in avanti si ha controllando il flusso di istruzioni coinvolgendo la predizione dell'indirizzo destinazione del salto, introducendo il **Branch Target Buffer (BTB)**, il quale memorizza l'indirizzo di arrivo del salto predetto.

Al BTB si accede nello stadio IF, usando come informazione di accesso l'indirizzo di una possibile istruzione di salto.

Se l'accesso è uno Hit, ovvero l'indirizzo dell'istruzione letta coincide con un indirizzo presente nel buffer, significa che l'istruzione è un salto che è già stato eseguito in precedenza.

Se è un Miss, l'istruzione non è un salto e si prosegue in sequenza, oppure è un salto mai eseguito prima, l'elemento nel BTB verrà creato al termine dell'esecuzione del salto.

### 13.6.1 Speculazione

Una volta predetto il prossimo indirizzo, si scommette (specula) sul risultato dei salti condizionati, il programma viene eseguito come se la speculazione fosse corretta.

La **speculazione hardware-based** si occupa di **predire** dinamicamente i salti, **speculare** per poter eseguire le istruzioni prima di aver risolto le dipendenze (e disfarle in caso di speculazione errata) e gestire lo **scheduling dinamico** per la combinazione dei blocchi di istruzioni.

Si **estende l'algoritmo di Tomasulo** utilizzando il **Re-Order Buffer** per separare la fase di commit dalla fase di esecuzione.

Le Reservation Station registrano le operazioni tra l'istante in cui vengono lanciate e quello in cui inizia l'esecuzione, i risultati vengono etichettati col numero dell'elemento nel ROB invece che col numero della Reservation Station. Il ROB si sostituisce inoltre alla ridenominazione.

Tutte le istruzioni tranne i salti errati giungono al commit quando arrivano al ROB.

In caso di predizione errata, il ROB viene svuotato e l'esecuzione riprende dal successore corretto del salto.

I passi nell'esecuzione delle istruzioni sono così riassunti:

1. **Issue**: l'istruzione viene estratta dalla coda delle istruzioni. Se ci sono una Reservation Station libera e una posizione vuota nel ROB, l'istruzione viene lanciata. Se gli operandi sono disponibili nel RF o nel ROB, vengono inviati alla RS, alla quale si invia anche la posizione del ROB allocata per il risultato. Se RS o/e ROB sono pieni, l'istruzione va in stallo.

2. **Execute**: se uno o più operandi non sono ancora disponibili, il CDB viene monitorato nell'attesa che il valore venga calcolato. Si controlla la presenza di alee RAW. Quando ambedue gli operandi sono disponibili nella RS si esegue l'operazione. Per una Store: basta che sia disponibile il registro base.

3. **Write result**: risultato disponibile, scritto sul CDB, dal CDB al ROB e alle eventuali RS che lo aspettano. La RS che ospitava l'istruzione viene liberata. Per una **Store**: se il risultato è disponibile, lo si scrive nel campo Valore dell'elemento corrispondente nel ROB, altrimenti il CDB viene monitorato fino a quando non si trasmette il valore.

4. **Commit**: tre diverse possibili sequenze:

   (a) **Commit normale**: l'istruzione raggiunge la coda del ROB, il risultato è presente nel buffer. Si scrive il risultato nel registro, si cancella l'istruzione dal ROB.

   (b) **Commit di una Store**: come sopra, salvo il fatto che si aggiorna la memoria invece di un registro.

   (c) **L'istruzione è un salto**: se la predizione era **errata**, si svuota il ROB, l'esecuzione riprende dal successore corretto dell'istruzione di salto. Se la predizione del salto era **corretta**, si porta a compimento l'istruzione di salto e si validano le istruzioni successive, in cui si annulla il campo speculazione.

### 13.6.2 Speculazione e scheduling dinamico

Non si scrivono valori ne nei registri ne nella memoria fino al commit di un'istruzione.

La CPU può facilmente annullare le azioni speculative quando si scopre che la predizione di un salto era errata, svuotando il ROB da tutti gli elementi inseriti dopo un salto predetto in modo errato, la lettura riparte dall'indirizzo corretto.

Le **eccezioni** non vengono riconosciute fino a quando l'istruzione non è pronta al commit.

Se un'istruzione speculata solleva eccezione, si registra nel ROB, se la predizione di salto errata, è tale per cui l'istruzione non avrebbe dovuto essere eseguita, si cancellano sia l'istruzione che l'eccezione svuotando il ROB. Se l'eccezione raggiunge la testa del ROB non è più speculativa e va servita.

Le alee WAW e WAR vengono eliminate con la speculazione, aggiornando in modo ordinato la memoria, le alee RAW si risolvono introducendo delle restrizioni: una Load non può passare al secondo passo dell'esecuzione (lettura) se un elemento attivo nel ROB corrispondente a una Store ha il campo Destinazione uguale al campo sorgente della Load e **si mantiene l'ordine di programma** per il calcolo degli indirizzi della Load rispetto a tutte le Store precedenti.

### 13.6.3 Ridenominazione dei registri e ROB

In presenza di speculazione i valori possono trovarsi temporaneamente nel ROB.

Se per un certo tempo non si lanciano nuove istruzioni, tutte le istruzioni nel ROB giungono a commit, i valori dei registri vengono scritti nel Register File, che corrisponde direttamente ai registri architetturalmente visibili, in alternativa si attua un uso esplicito di un insieme di registri più grande con la **ridenominazione dei registri**, contenente sia i valori architetturalmente visibili sia i valori temporanei.

Durante il lancio il processo di ridenominazione mappa i nomi dei registri architetturali sui numeri dei registri fisici nell'insieme esteso, allocando nuovi registri inutilizzati per la destinazione dei risultati.

Le **alee** WAW e WAR vengono evitate grazie alla ridenominazione, il **recupero delle speculazioni errate** si attua in base al fatto che il registro fisico che riceve il risultato di un'istruzione speculata diventa il registro architetturale solo al momento del commit.

In conclusione il commit è più semplice che con il ROB ma la deallocazione dei registri è più complessa, inoltre il mapping dinamico dei registri architetturali su quelli fisici complica il progetto e la messa a punto del software.

# 14 Limiti dell'ILP

Sono dovuti all'effettivo flusso dei dati attraverso i registri o la memoria, in un **processore ideale**:

1. **Ridenominazione dei registri**: sono disponibili infiniti registri, si evitano alee WAW, WAR, il numero di istruzioni che possono iniziare simultaneamente non ha un limite.
2. **Predizione dei salti condizionati e incondizionati perfetta**: si immagina una CPU con speculazione perfetta e buffer istruzioni illimitato, si superano le barriere dovute a istruzioni di controllo.
3. **Analisi degli indirizzi in memoria perfetta**: tutti gli indirizzi in memoria sono noti esattamente, se gli indirizzi sono diversi una Load può sempre scavalcare una Store.
4. **Caches perfette**: ogni accesso a cache richiede esattamente un ciclo di clock.

Con la seconda ipotesi non esistono dipendenze di controllo nel programma dinamico, con le ipotesi 1 e 3, si eliminano le dipendenze di dati false, inoltre, qualsiasi istruzione può essere schedulata nel ciclo immediatamente successivo a quella da cui dipende (speculazione perfetta).

## 14.1 Il processore perfetto

Il processore perfetto dovrebbe:

- **Guardare in avanti** nel programma a distanza arbitrariamente grande per trovare istruzioni da lanciare, predicendo i salti in modo perfetto.
- Determinare se ci sono **dipendenze di dati** fra le istruzioni nel pacchetto di lancio e, nel caso, ridenominare.
- Determinare se ci sono **dipendenze sulla memoria** fra le istruzioni in fase di lancio e gestirle.
- Fornire abbastanza unità funzionali da **permettere il lancio di tutte le istruzioni pronte**.

### 14.1.1 Limiti della dimensione della finestra

La **dimensione della finestra** influisce sul numero di confronti da operare per determinare le dipendenze RAW ($2\sum_{i=1}^{n-1} i = 2\frac{(n-1)n}{2} = n^2 - n$), attualmente $n = \{32 - 126\}$ istruzioni.

Altri limiti sono il **numero di registri**, la ricerca delle coppie di **istruzioni dipendenti** e le **issue in ordine**, il **numero di unità funzionali**, il **numero di bus**, il **numero di porte del Register File**.

Di conseguenza esiste un limite al numero di istruzioni iniziate nello stesso ciclo, infatti il massimo numero di istruzioni che si possono lanciare, eseguire, o portare a commit nello stesso ciclo di clock è molto più piccolo della dimensione della finestra.

### 14.1.2 Limite di predizione

Un'analisi perfetta è impossibile, non può essere perfetta in compilazione, e, se eseguita a run-time, richiederebbe un numero di confronti potenzialmente infinito.

### 14.1.3 Pentium IV

**Caratteristiche**:

- Architettura superscalare con scheduling dinamico.
- Uso di ridenominazione dei registri: possibili fino a $128$ risultati in attesa di finalizzazione.
- $7$ unità di esecuzione su interi.
- Disegno aggressivo di ALU e D-cache.
- Soluzione estremamente aggressiva per la gestione dinamica della speculazione.

La microarchitettura "**NetBurst**" raggiunge frequenza di clock più alta ($2.4 - 3$ GHz), mantiene throughput di esecuzione prossimo al massimo, con una Hyper-pipeline a $20$ stadi.

Sono presenti tre meccanismi di prefetching:

1. Prefetch hardware delle istruzioni (basato sul BTB).
2. Prefetching controllato dal software sulla D-cache.
3. Meccanismo hardware di prefetching di istruzioni e dati da cache L3 a cache L2.

Inserimento di **Execution Trace Cache** che genera il flusso di micro operazioni in alternativa alle normali istruzioni.

Le Load possono essere riordinate rispetto alle altre Load e alle Store. Sono possibili le Load Speculative.

Ammessi $4$ cache miss su Load.

Le misprediction vanno dall'$1,4\%$ allo $0,5\%$.

In conclusione l'IPC va da $1,24$ a $5,58$ con $100$ Watt di consumo.

### 14.1.4 Nuove architetture Intel

Eliminata la Trace Cache perché consumava troppo.

Front end deve fornire alle unità di esecuzione istruzioni decodificate per un motore di esecuzione di issue pari a $6$.

- **Branch Prediction Unit (BPU)**.
- **Instruction Fetch Unit (IFU)**: effettua il pre-fetching delle istruzioni che probabilmente verranno eseguite, pre-codificate e mantiene pieno l'instruction buffer.
- **Instruction Queue (IQ)**.
- **Unità di decodifica**.

L'**Instruction Queue** include anche una loop cache: nella BPU c'è un **Loop Stream Detector (LSD)**, che identifica i cicli corti che vengono bloccati e inviati dalla IQ in streaming fino a quando una misprediction non lo termina.

In questo modo non ci sono penalizzazioni dovute ai salti fatti e buona parte del front end resta fermo, diminuendo il costo energetico.

L'ISA viene esteso per introdurre istruzioni vettoriali e specifiche per l'esecuzione di algoritmi crittografici (AES).

### 14.1.5 VLIW - Very Long Instruction Word

Si tratta di un'alternativa alla soluzione superscalare, le istruzioni macchina eseguibili simultaneamente sono identificate dal compilatore e assemblate in istruzioni lunghe (**bundle**) eseguite in parallelo da diverse unità funzionali.

Le istruzioni sono **eseguite in ordine di programma** e avviate per l'esecuzione su diverse unità di esecuzione pipelined.

Con la gestione dello scheduling delle istruzioni gestita dal compilatore, si riduce notevolmente la complessità dall'unità di controllo.

Schema di funzionamento con numero di percorsi pari a quattro:

- Il compilatore identifica $4$ operazioni mutuamente indipendenti, che possono essere impaccate in un bundle ed eseguite simultaneamente.
- Il compilatore risolve anche i **conflitti di risorse**, le $4$ operazioni si devono servire di risorse distinte.
- **Conflitti di controllo**: in un bundle può essere presente al più una operazione che modifica il flusso del controllo (salto condizionato o no, chiamata, o ritorno da sottoprogramma).

- Applicazioni con parallelismo superiore: CPU organizzata in “cluster”, ognuno corrispondente ad un parallelismo limitato che fa riferimento a un proprio banco di registri, se appartengono a un unico flusso istruzioni, in tutto può essere presente una sola operazione di controllo.
- A ogni sillaba viene associato un percorso (**lane**) nel Data Path della CPU, lungo il quale viene eseguita.
- Sono disponibili $\mathbf{n}$ **ALU**, ognuna collegabile a un lane. La cache dati ha una sola porta di accesso ed è raggiunta da un solo lane.
- Register File: $\mathbf{2n}$ porte di lettura, $\mathbf{n}$ di scrittura.

Volendo mantenere l’esecuzione in ordine, non è possibile ammettere che sillabe diverse di uno stesso bundle giungano al Write Back in istanti diversi, ma se e il parallelismo non è elevato, le istruzioni contengono un numero elevato di sillabe NOP, portando così ad un’**esplosione della memoria**.

Per evitarla, i bundle contengono solamente sillabe significative, nessuna NOP. Dei **codici separatori** distinguono un bundle dall’altro, l’unità di controllo scompone il bundle nella sequenza di bundle eseguibili.

**Pro e contro:**

- **Architettura**: più semplice di una CPU superscalare con uguale parallelismo e quindi più veloce, ciclo di clock più breve, determinato dagli stadi della pipeline e non dall’unità di controllo.
- **Compilatore**: deve conoscere non solo l’architettura della CPU, ma anche parametri tecnologici (latenza e throughput delle unità funzionali, Load-Use delay, etc.), si riduce la portabilità del codice oggetto e quindi anche tra successive generazioni della stessa famiglia può mancare la compatibilità binaria. Per ovviare a questo problema si può ricorrere alla traduzione del codice oggetto mediante code **morphing software** che traduce il codice x86 in istruzioni VLIW provocando però un degrado nelle prestazioni.
- **Data cache miss** generato da una sillaba: occorre porre in stallo l’intera CPU, superabile con l’uso di una Load speculativa, facendo però crescere la complessità dell’unità di controllo.

### 14.1.6 Esecuzione Predicata

Come Esecuzione Predicata, si intendono una serie di tecniche che convertono le dipendenze di controllo in dipendenze di dati, dando al compilatore un’alternativa ai salti condizionati, chiamata anche **guarded execution**.

Implica una forma di esecuzione condizionale delle istruzioni basata su una **guardia booleana**.

In pratica l’istruzione fa riferimento a una condizione, che viene valutata come parte dell’esecuzione dell’istruzione:

- **Condizione vera**: l’istruzione viene eseguita normalmente.
- **Condizione falsa**: l’istruzione continua come se fosse una NOP.

In questo modo si eliminano le penalizzazioni per predizioni errate, pagando con “uso inutile” di risorse: si inseriscono nella schedule una computazione inutile e una utile.

Questo tipo di esecuzione aumenta in modo rilevante la dimensione del codice e il consumo di potenza, non adatto a sistemi embedded.

# 15 Le CPU per sistemi Embedded - Processori DSP

Un esempio tipico sono i **processori per elaborazione numerica del segnale (DSP - Digital Signal Processor)** utilizzati in diversi ambiti, dall'elettronica di consumo alle telecomunicazioni ai controlli di automazione, etc.

## 15.1 Requisiti:

- Computazioni numeriche iterative su grandi insiemi di dati e, spesso, di flussi di dati continui.
- Computazioni numeriche ad alta fedeltà.
- Larga banda di memoria:
  - Memoria Istruzioni e Memoria Dati Separate.
  - Cache Istruzioni.
  - HW Dedicato per calcolare gli indirizzi di memoria.
  - Indirizzamento Circolare.
- Elaborazione in tempo reale.
- Elevato throughput.
- I/O veloce: Direct Memory Access (DMA) per trasferire dati praticamente senza l'intervento del processore.

### 15.1.1 Formato dei Dati

Molto spesso in virgola fissa piuttosto che in virgola mobile, in questo modo risulta meno costoso, consuma minor potenza.

Non essendo possibile una precisione infinita, toccherà al programmatore adattare il risultato ai propri requisiti di accuratezza.

Alcuni valori sopra o sotto una certa soglia perdono di significato per l'applicazione.

### 15.1.2 Cicli Zero Overhead

Non occorrono cicli di clock in più per aggiornare e verificare il contatore di ciclo e tornare alla testa del corpo del ciclo.

Gli algoritmi di DSP spesso hanno cicli interni compatti (tight), ripetuti un numero fisso di volte: si introduce il costrutto **Repeat**: il numero di iterazioni viene caricato in un contatore HW, le istruzioni del ciclo vengono trasferite in un **Loop Buffer** all'interno della CPU, ad ogni iterazione viene decrementato il contatore, niente istruzione finale di branch ne conflitti di controllo.

### 15.1.3 Insieme di Istruzioni Specializzato:

- Uso Intensivo dello HW per sfruttare specifiche unità di calcolo, si possono consentire due letture dati in parallelo.
- Minimizzazione dello spazio di memoria: limitazione dei registri che possono essere usati dai vari tipi di istruzioni.
- Insieme di istruzioni specializzato, complicato e irregolare.

#### 15.1.4 Architetture DSP di fascia alta

- Più unità di esecuzione indipendenti.
- Più operazioni codificate in una singola istruzione.
- Pipelining.
- Più memorie dati con più bus.

### 15.2 Vantaggi

- **Tempo di sviluppo breve**: linguaggi di programmazione ad alto livello, strumenti di debug.
- **Flessibilità**: dispositivi riprogrammabili: consentono di applicare aggiornamenti e di correggere gli errori di programmazione.
- **Più economici di HW dedicato**: in particolare per applicazioni con basso volume di produzione.
- Vantaggi in termini di **velocità**, **costo** ed **efficienza energetica**.

### 15.3 Gestione della Potenza

Il consumo di potenza è un problema sempre più importante per tutte le CPU, in particolar modo per i sistemi portatili e mobili, nonché per la maggior parte dei sistemi embedded.

Lo scopo principale è quello di raggiungere alte prestazioni limitando il consumo di potenza, agendo sul livello tecnologico, architetturale, di programma e di sistema.

$P = P_{SW} + P_{SC} + P_{Lk}$

dove:

$P_{SW}$: potenza di commutazione.

$P_{SC}$: potenza da cortocircuito.

$P_{Lk}$: potenza di Leakage (stand-by).

La tensione e la frequenza hanno un forte impatto sulla potenza di commutazione e quindi sul consumo, non è quindi possibile pensare di continuare ad aumentare la frequenza.

È possibile ridurre l'attività di parti preselezionate della CPU sulla base del codice che si sta eseguendo, utilizzando la tecnica di **Clock Gating** che prevede di bloccare l'invio del ciclo di clock alle parti che si vogliono fermare, altrimenti si fa ricorso al **Power Gating** bloccando l'alimentazione.

A livello tecnologico si può ridurre la tensione di alimentazione, riducendo il consumo di commutazione e di leakage, in questo modo però si diminuisce anche la frequenza. Si introducono quindi diversi modi di funzionamento della CPU in base alla tensione di alimentazione e alle varie frequenze.

E' necessario scegliere bene le frequenze e le tensioni, in quanto passare da una modalità di funzionamento all'altra comporta un consumo energetico non banale.

## 16 Miglioramenti CPU singola

Ad ogni transizione la percentuale dei transistori essenziali per l'esecuzione di un'istruzione diventa una frazione sempre più piccola del totale, mentre cresce quella dei transistori usati per funzioni di controllo.

Il miglioramento delle prestazioni non è uniforme per tutte le applicazioni, alcune sfruttano lo HW aggiunto meglio di altre.

## 16.1 Parallelismo a livello di thread (TLP - Thread Level Parallelism)

Oltre all'Instruction Level Parallelism si possono sfruttare altre forme di parallelismo, per esempio il **Multithreading** permette che più thread condividano le unità funzionali di una stessa CPU, sovrapponendosi.

Il sistema operativo vede il processore fisico come se si trattasse di un multiprocessore simmetrico costituito da due processori.

Occorre mantenere lo **stato indipendente di ogni thread**: devono essere previsti un PC, un Register File, uno Stack Pointer, un Registro di Stato, una tabella delle pagine.

La **memoria** può essere condivisa grazie ai meccanismi di memoria virtuale.

### 16.1.1 Multithreading a grana fine

La CPU commuta da un thread all'altro ad ogni istruzione, l'esecuzione di più thread è intrallacciata, spesso si commuta seguendo un turno, saltando un thread se questo incontra uno stallo a latenza lunga.

La CPU deve riuscire a cambiare thread a ogni ciclo di clock: la commutazione è necessariamente gestita dallo HW, occorre replicare le risorse elementari quali PC, SP, Registro di Stato, etc.

Quando un thread incontra un evento a latenza elevata, i suoi cicli restano in parte inutilizzati.

**Vantaggio**: si mascherano le perdite di throughput dovute a stalli brevi e solo parzialmente quelli dovuti a stalli lunghi.

**Svantaggio**: il singolo thread viene molto spesso rallentato.

Il funzionamento dell'intera CPU deve riconoscere il ricorso al multithreading, il data forwarding e il flushing in presenza di eccezioni devono tener conto della presenza simultanea di più thread.

### 16.1.2 Multithreading a grana grezza

Il sistema commuta da un thread a un'altro solo in corrispondenza di stalli lunghi.

I due thread condividono molte risorse del sistema, la commutazione da un thread all'altro richiede diversi cicli per il salvataggio del contesto e viene gestita dal sistema operativo.

**Vantaggio**: in condizioni normali il singolo thread non viene rallentato.

**Svantaggio**: per stalli brevi non si riducono le perdite di throughput, la CPU lancia istruzioni che appartenevano a un solo thread, quando c'è uno stallo si deve svuotare la pipeline prima di lanciare il nuovo thread.

Se applicato a un'architettura superscalare con esecuzione fuori ordine e scheduling dinamico, richiede di giungere allo svuotamento di ReOrder Buffer, Reservation Stations, buffer di speculazione prima di effettuare il cambiamento di thread.

### 16.1.3 Simultaneus Multithreading (SMT)

Derivato da Multithreading a grana fine, usa le risorse disponibili in un'architettura superscalare per gestire simultaneamente ILP e TLP.

Di base una CPU superscalare ha **più risorse funzionali** di quanto un thread possa effettivamente usare, perciò più istruzioni indipendenti appartenenti a thread diversi possono essere lanciate **senza preoccuparsi delle dipendenze**, risolte dallo HW per lo scheduling dinamico.

Il sistema si adatta dinamicamente all'ambiente, permettendo l'**esecuzione di istruzioni da ogni thread**, e permettendo che le istruzioni di un singolo thread usino tutte le unità funzionali se l'altro thread trova un evento a lunga latenza.

Ad una CPU con scheduling dinamico occorre aggiungere:

- Una tabella di ridenominazione per ogni thread.
- Un PC per ogni thread.
- La possibilità che istruzioni di più thread giungano a commit indipendentemente.

Le CPU attuali hanno pipeline profonde, quindi SMT ha senso solo se abbinato a multithreading a grana fine, valutando l'impatto di latenza sul singolo thread, introducendo così una priorità.

Il rischio è quello di ridurre le prestazioni quando il thread ad alta priorità incontra uno stallo lungo.

# Parte IV
# Multiprocessore

Si è capito che apportare ulteriori aumenti di prestazioni alla singola CPU, ricorrendo a ILP e TLP, richiede costi sempre più elevati in termini di area di silicio, complessità dell'unità di controllo e consumo di potenza.

Inoltre il limite di ILP estraibile da un'applicazione, anche ricorrendo a compilatori ottimizzanti, non consente grandi miglioramenti.

## 17 Architetture parallele a livello di processo

Sistemi paralleli dotati di più unità di elaborazione.

Vengono realizzati di norma partendo da microprocessori standard.

Per sfruttare il parallelismo nei multiprocessori occorre software strutturato per essere eseguito in modo parallelo, ma il problema è stato superato in parte, in particolare, per i server e per i sistemi dedicati, dove le applicazioni mostrano un parallelismo naturale senza che sia necessario riscrivere grandi masse di software.

Può essere vista come un'estensione delle architetture convenzionali che affronta aspetti di comunicazione e collaborazione tra processori.

**Granularità del parallelismo** Quantità di lavoro compiuto da un thread in parallelo ad altri thread.

Fissata una dimensione del problema, la granularità diminuisce all'aumentare del numero di processori.

### 17.1 Tassonomia tradizionale (Flynn, 1966)

#### 17.1.1 SISD - Single Instruction Stream, Single Data Stream

Il tradizionale processore singolo.

#### 17.1.2 SIMD - Single Instruction Stream, Multiple Data Stream

Unico flusso di istruzioni eseguito simultaneamente da più CPU che operano su diversi flussi di dati.

Ogni CPU ha la propria memoria dati.

C'è una sola memoria istruzioni e un solo processore di controllo.

Dopo soluzioni di supercalcolo scientifico, il concetto SIMD ha trovato applicazione nei calcolatori vettoriali e recentemente, con opportune evoluzioni, nelle GPU.

#### 17.1.3 MISD - Multiple Instruction Streams, Single Data Stream

Più CPU operano su un unico flusso di dati.

Non si è mai costruito nessun multiprocessore commerciale di questo tipo.

Approssimato da qualche "stream processor" molto particolare.

### 17.1.4 MIMD - Multiple Instruction Streams, Multiple Data Streams

Ogni CPU legge ed esegue le istruzioni di un proprio flusso di esecuzione e opera su propri dati, eventualmente condivisi con altri.

I processori spesso sono microprocessori commerciali standard.

E' la soluzione di gran lunga più diffusa.

## 17.2 Multiprocessori SIMD

Dagli anni $'60$ con ILLIAC IV fino agli anni $'80$ con Connection Machine.

Dedicati a elaborazione con parallelismo di dati, **data-parallel processing**.

Il costo dell'unità di controllo viene ammortizzato su molte unità di esecuzione.

Ridotta dimensione della memoria di programma, poiché vi è una sola copia del codice in corso di esecuzione.

I calcolatori SIMD reali possiedono un misto di istruzioni SISD e SIMD. Di norma: un calcolatore host di tipo SISD esegue le operazioni sequenziali (salti, calcolo degli indirizzi). Le istruzioni SIMD sono inviate a tutte le unità di esecuzione, che sfruttano una rete di interconnessione per scambiarsi i dati.

Sono caratterizzati da:

- Identiche operazioni sono compiute in parallelo su ogni elemento di grandi strutture dati regolari, quali vettori o matrici.
- Il modello di programmazione viene tradotto direttamente dallo HW. Un processore di controllo invia identicamente in broadcast ogni istruzione a una matrice di elementi di elaborazione (PE) collegati in una struttura regolare e dotati ognuno di una propria memoria dati o della capacità di accedere indipendentemente a una memoria dati comune.
- Non esistono dati condivisi tra processori diversi, quindi non si pongono problemi di conflitto o di coerenza.
- Tutte le unità di elaborazione poste in parallelo sono sincronizzate ed eseguono una stessa istruzione, indirizzata da un unico contatore di programma. Il programmatore vede un modello simile a quello dei sistemi SISD.
- Ogni unità di elaborazione possiede i propri registri di indirizzamento della memoria.

### 17.2.1 Calcolatori vettoriali

Architettura accompagnata dal relativo modello di compilazione.

- Include un insieme di registri vettoriali, ognuno dei quali è un banco di registri di un certo numero di elementi (es. $64$).
- Gli operandi vettoriali devono essere caricati nei registri vettoriali prima di essere usati. I risultati di operazioni aritmetico/logiche su registri vettoriali vengono caricati in un registro vettoriale.

**Esecuzione dell'operazione: ADD.V** Somma di due vettori.

Dopo la decodifica la pipeline viene configurata per eseguire la ADD.V in modo streaming.

1. I due registri vettoriali d'ingresso vengono collegati ai due ingressi della pipeline di somma.
2. L'uscita della pipeline di somma viene collegata al registro vettoriale di uscita.

3. Le coppie di elementi dei vettori d'ingresso vengono inviate alla pipeline di somma una dopo l'altra, una per ciclo di clock, in modo streaming. Dopo un numero di cicli corrispondente alla profondità della pipeline di somma il primo risultato viene scritto nel primo elemento del registro vettoriale di uscita, e così via.

**Tempo di esecuzione**: $T_{exec} = T_{startup} + N$
dove $T_{startup}$ dipende dalla profondità della pipeline aritmetica e non dalla dimensione del vettore.

#### Vantaggi dei calcolatori vettoriali rispetto ai processori tradizionali

1. Ogni risultato è indipendente da quelli precedenti $\rightarrow$ possibilità di pipeline profonde, alta frequenza di clock.
2. Una singola istruzione vettoriale fa molto lavoro $\rightarrow$ minore numero di istruzioni lette e di istruzioni di salto, quindi anche di predizioni di salto errate.
3. Le istruzioni vettoriali accedono ogni volta a un blocco di memoria, trasferendo le parole in un registro vettoriale $\rightarrow$ si ammortizza la latenza di memoria.
4. Le istruzioni vettoriali accedono alla memoria secondo schemi noti $\rightarrow$ più banchi di memoria forniscono operandi simultaneamente. Di norma non esiste cache dati.

## 17.3 Multiprocessori MIMD

Sono soluzione flessibili, possono funzionare su un'unica applicazione operando come una macchina ad alte prestazioni, su altre applicazioni come multiprocessori multiprogrammati che eseguono simultaneamente molti compiti, o come una combinazione delle due soluzioni.

Possono essere costruite utilizzando CPU standard.

Per sfruttare un MIMD con $n$ processori occorre disporre di almeno $n$ thread o processi da eseguire.

I thread o i processi indipendenti sono tipicamente identificati dal programmatore o creati dal compilatore.

Il parallelismo viene identificato dal software.

Sono caratterizzati da:

- Ogni processore esegue il proprio flusso di istruzioni. In molti casi, ognuno esegue un processo/thread differente.
- Ogni processo può essere indipendente da quello eseguito dagli altri processori.
- Più processori possono eseguire un unico programma condividendone il codice e buona parte dello spazio di indirizzamento, programma strutturato in thread.

# 18 Modelli di programmazione parallela

## 18.1 Problemi del calcolo parallelo

### 18.1.1 Limitato parallelismo disponibile nei programmi

Si applica la legge di Amdahl.

Si supponga che il numero di processori sia $100$, e si voglia raggiungere uno speedup pari a $80$. La legge di Amdahl si esprime come:

$80 = \frac{1}{frazione_{parallel}/100+(1-frazione_{parallel})} \rightarrow 0,8 * frazione_{parallel} + 80 * (1 - frazione_{parallel}) = 1$
$\rightarrow frazione_{parallel} = 0,9975 = 99,75\%$ del codice parallelo
Quindi per ottenere lo speedup richiesto occorrerebbe che solo lo $0,25\%$ del codice sia sequenziale.

### 18.1.2 Elevata latenza degli accessi remoti

La comunicazione fra processori può costare dai $100$ ai $1000$ cicli di clock, a seconda del meccanismo di comunicazione, della rete di interconnessione e della dimensione del multiprocessore.
Si supponga:
$\#nodi = 32$, $costo_{accesso\,remoto} = 400ns$, $CPU_{clock} = 1GHz$, $IPC = 2$
tutti i riferimenti danno Hit nella cache locale.
Determinare quanto più veloce è il processore nei due casi:

1. Non ci sono comunicazioni.
2. Il $2\%$ delle istruzioni eseguite coinvolgono un accesso remoto.

Per prima cosa si calcola il CPI. Nel caso degli accessi remoti si ha:
$CPI = CPI_{base} + f_{rich.\,remote} * costo_{rich.\,remote} = 1/IPC_{base} + 0,2\% * costo_{rich.\,remote} =$
$= 0,5 + 0,2\% * costo_{rich.\,remote} = 0,5 + 0,8 = 1,3$
$costo_{rich.\,remote} = costo_{accesso\,remoto}/T_{ciclo} = 400ns/1ns = 400$ cicli.
Il sistema nel primo caso è $1,3/0,5 = 2,6$ volte più veloce.
Il costo della comunicazione ha impatto elevato.

### 18.1.3 Problemi di contention dell'accesso alle risorse condivise

Se $n$ processori accedono agli stessi moduli di memoria, $n - 1$ devono aspettare.
La **probabilità di contention aumenta con $n$.**
Questi problemi portano a mancanza di scalabilità.
Si introducono cache locali per ridurre le contention, infatti hit in cache locali non richiedono accesso alla memoria condivisa, ma nascono **problemi di cache coherence**.

## 18.2 Programmazione con spazio di indirizzamento condiviso

L'architettura di comunicazione usa le normali operazioni sulla memoria per fornire comunicazione mediante gli indirizzi condivisi. Fornisce inoltre speciali operazioni atomiche per la sincronizzazione degli accessi alle variabili condivise.

I processi possono essere configurati così da condividere una parte del loro spazio di indirizzamento, quindi mappati su una sola area fisica. Formalmente un processo è uno spazio di indirizzi virtuali con uno o più thread di controllo.

Cooperazione e coordinamento tra i processi vengono ottenuti leggendo e scrivendo variabili condivise e puntatori che si riferiscono a indirizzi condivisi.

Le scritture da parte di un thread in un indirizzo di memoria logicamente condiviso sono visibili alle letture da parte di altri thread.

Questo modello **deve specificare come i processi vedono l'ordine delle operazioni** eseguite da altri processi.

### 18.2.1 Architetture a memoria centralizzata condivisa

Ogni processore può accedere equamente a tutta la RAM.

L'interconnessione può essere costituita da un semplice bus.

L'unità di memoria primaria centralizzata ha una relazione simmetrica con tutti i processori, il tempo d'accesso è uniforme per qualunque processore $\rightarrow$ multiprocessori simmetrici a memoria condivisa (**SMP**). Detti anche architetture con Accesso a Memoria Uniforme (UMA).

#### Aumentare la scalabilità

- **Mantenere l'uniformità degli accessi ma sostituire l'interconnessione a bus con una rete di interconnessione scalabile**. Ogni accesso alla memoria si traduce in una transazione di messaggio sulla rete. Sostituisce la transazione di bus nell'architettura basata su bus. **Svantaggio principale:** la latenza degli accessi alla memoria aumenta per tutti gli accessi. Inoltre, la memoria unica resta un punto di serializzazione.
- **Memoria fisicamente distribuita** (es: MIMD con memoria fisicamente distribuita).

## 18.3 Programmazione a scambio di messaggi

La comunicazione è integrata a livello di I/O invece che nel sistema di memoria.

Ogni processo viene allocato con i suoi dati nelle memorie locali.

Le operazioni di comunicazione più usate sono varianti delle **SEND** e **RECEIVE**.

La coppia SEND e RECEIVE provoca un trasferimento di dati da un processo a un altro, realizzando un evento di sincronizzazione fra due unità e una copia da memoria a memoria, in cui ogni nodo partecipante specifica il proprio indirizzo locale dei dati.

Questo modello **non ha bisogno di presupporre un ordinamento fra le operazioni** dei diversi processi al di fuori di quello associato alle operazioni SEND e RECEIVE.

**SEND** Specifica un buffer dati locale il cui contenuto deve essere trasmesso e un processo ricevente, tipicamente è su un processore remoto.

Solitamente specifica anche un **tag** (identificatore) da associare al messaggio.

**RECEIVE** Specifica un processo che trasmette e un buffer dati locali in cui i dati trasmessi devono essere trasferiti.

Specifica anche una regola di abbinamento, **matching rule**.

#### Varianti dell'evento di sincronizzazione

1. Supporto alla comunicazione fra arbitrarie coppie di nodi, o mediante tecnica **store-and-forward** attraverso nodi intermedi. Oppure mediante l'uso di reti di interconnessione di tipo generale dotate di routers che si fanno carico dell'instradamento. Svantaggio: il tempo di comunicazione dipende dal numero di nodi intermedi.
2. La comunicazione viene effettuata inviando messaggi che richiedono un'azione o forniscono dei dati. La comunicazione può essere effettuata o dal punto di vista del **lettore dei dati** (es. chiamata a procedura remota) oppure da quello dello **scrittore dei dati**, se il processore che produce i dati sa quando altri nodi ne avranno bisogno, e quali saranno questi nodi. Se un processo ha bisogno di accedere a dei dati, dal punto di vista del **lettore dei dati**:

   (a) Invia un messaggio chiedendo i dati.

(b) Mentre attende la risposta può essere sospeso (e il processore allocato per un diverso processo), oppure continuare finché non ha bisogno della risposta andando in uno stato di **busy-wait**.

i. Nel caso il processo viene sospeso: richiede un meccanismo efficiente di **context switching**.

ii. Nel caso si pone in **busy-wait** richiede un supporto HW/SW efficiente per monitorare le comunicazioni attesa, **polling**.

### 18.3.1 Primitive di scambio messaggi sincrone

Nello scambio di messaggi sincrono, il trasmittente deve bloccarsi fino a quando il messaggio non è stato ricevuto dal ricevente, e il ricevente deve bloccarsi fino a quando il messaggio non è stato ricevuto e copiato al punto designato.

**Pro**: è semplice ragionare sul risultato di un'esecuzione, poiché forza la sincronizzazione.

**Contro**: possibilità di deadlock. Non è possibile sovrapporre computazione e comunicazione perché il trasmittente si blocca fino a quando il ricevente non ha completato la ricezione.

### 18.3.2 Primitive di scambio messaggi asincrone

Nello scambio di messaggi asincrono il trasmittente non deve attendere fino a quando i dati sono stati copiati dal suo spazio di indirizzamento a quello del ricevente.

Il trasferimento avviene in modo asincrono rispetto all'esecuzione del codice.

Ci sono $2$ possibilità:

1. **Soluzione bloccante**: SEND asincrona bloccante. Il controllo viene restituito al processo trasmittente solo dopo che il messaggio da inviare è stato copiato in una qualche area di supporto e non può essere modificato dall'esecuzione del processo trasmittente. Dualmente una RECEIVE bloccante non restituisce il controllo al processo ricevente fino a quando il messaggio non è stato copiato nello spazio locale di indirizzamento del ricevente.

2. **Soluzione non bloccante**: il controllo viene restituito al processo immediatamente. Il trasferimento viene eseguito in background.

## 18.4 Programmazione Data Parallel

Basata su programmi paralleli che consistono in uno o più thread di controllo che operano sui dati.

Specifica a quali dati i thread possono fare riferimento, quali operazioni possono essere compiute su tali dati e quale ordinamento esiste tra tali operazioni.

L'**ordinamento ha importanza fondamentale**. Un programma parallelo deve coordinare le attività dei thread in modo da garantire che le dipendenze all'interno del programma vengano rispettate. Necessità di operazioni di sincronizzazione esplicite laddove l'ordinamento implicito delle operazioni di base non sia sufficiente.

## 18.5 Il problema dell'ordinamento

I thread operano in modo indipendente e potenzialmente a velocità diverse.

Non esiste una chiara definizione di thread più recente.

Occorrono $2$ tipi di sincronizzazione:

### 18.5.1 Sincronizzazione a mutua esclusione

Garantisce che date operazioni su particolari dati vengano eseguite da un solo thread per volta.
L'ordine in cui i thread operano non è essenziale.
La mutua esclusione tende a serializzare l'esecuzione dei thread.

### 18.5.2 Sincronizzazione a eventi

Eventi vengono usati per informare altri processi che si è raggiunto un particolare punto dell'esecuzione, così che gli altri possano procedere sapendo che certe dipendenze sono state soddisfatte.

Gli eventi possono essere **punto a punto**, coinvolgendo solo due processi, o **globali**, coinvolgono un gruppo o tutti i processi.

## 18.6 OpenMP - Open MultiProcessing

La parallelizzazione del software, creando **thread** che possono essere eseguiti in parallelo, è un problema complesso.

Esiste anche software legacy che può non essere affatto parallelizzabile.

Il programmatore applicativo deve capire la semantica dell'applicazione ed estrarre i segmenti di codice che possono essere eseguiti in parallelo.

Costrutti in cui è facile trovare thread paralleli sono tipicamente **cicli** e **chiamate a funzione**.

Ogni iterazione di un ciclo può essere eseguita in parallelo insieme alle altre iterazioni se non ci sono dipendenze tra le iterazioni o se queste possono essere gestite facilmente.

Le modifiche alle strutture dati condivise devono essere protette mediante mutua esclusione, una variabile sicuramente condivisa è il contatore delle iterazioni.

Esistono **librerie software** che consentono al programmatore di lavorare a livello relativamente alto, ignorando i dettagli delle primitive hardware di sincronizzazione, fornendo opportune API.

Per esempio, in OpenMP vengono fornite le direttive al compilatore mediante **pragma**, che aiutano il programmatore a identificare i frammenti di codice che si prestano alla parallelizzazione.

I **pragma** vengono automaticamente tradotti in **thread paralleli**, mediante una combinazione di un compilatore opportuno e di un run-time support adeguati allo specifico hardware.

La direttiva **pragma** indica all'ambiente run-time che nel ciclo non ci sono dipendenze fra le iterazioni e che le iterazioni possono essere eseguite in parallelo.

Se il codice viene eseguito su un sistema che supporta $2$ contesti di **thread**, il primo contesto (CPU#1) esegue le iterazioni da $0$ a ($\frac{n}{2} - 1$), il secondo (CPU#2) esegue quelle da $n/2$ a $n$.

Se il codice viene eseguito su un sistema con Azioni di snooping della cache, sia regular cache sia transactional cache effettuano snooping sul bus.

- **Regular cache**:
  - In caso di **READ** o **T_READ**, se lo stato è VALID, la cache restituisce il valore. Se lo stato è **RESERVED** o **DiRTY**, la cache restituisce il valore e riporta lo stato a VALID.
  - In caso di **RFO** o di **T_RFO**, la cache restituisce il valore e invalida la linea.
- **Transactional cache**:
  - Se **TSTATUS** = **false**, o se il ciclo è non-transazionale, READ o RFO, si comporta **come la regular cache**, salvo il fatto che ignora tutti gli elementi con tag transazionale diverso da NORMAL.

- In caso di **T_READ**, se lo stato è VALID restituisce il valore.
- Per tutte le **operazioni transazionali**, restituisce BUSY.

Ambedue le cache possono lanciare una richiesta **WRITE** al momento di sostituire una linea.

La memoria risponde solo a richieste **READ**, **T_READ**, **RFO**, **T_RFO** a cui non risponde nessuna cache.

Risponde inoltre alle richieste **WRITE**.

$4$ contesti logici di thread, l'ambiente run-time assegna automaticamente un quarto delle iterazioni a ogni contesto.

| CPU# | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Su $2$ contesti | $i$ da $0$ a $(n/2 - 1)$ | $i$ da $n/2$ a $n$ | | |
| Su $4$ contesti | $i$ da $0$ a $(n/4 - 1)$ | $i$ da $n/4$ a $(n/2 - 1)$ | $i$ da $n/2$ a $(3n/4 - 1)$ | $i$ da $3n/4$ a $n$ |

## 18.7 Problemi lock-based

La sincronizzazione **lock-based** può facilitare errori di programmazione:.

Se il numero dei processi e dei dati condivisi cresce, aumenta la possibilità di errori di programmazione.

Tener traccia di tutti gli elementi di dati condivisi può non essere banale.

È molto difficile scalare un sistema esistente, occorre tener traccia di tutti i lock e di tutte le variabili condivise.

## 18.8 Soluzioni transaction-based

Il concetto di base è quello di **evitare il locking**.

Processi diversi lanciano indipendentemente le loro azioni sulle variabili condivise, alla fine un solo processo riuscirà a raggiungere il commit, tutti gli altri processi abortiscono e riprovano.

### 18.8.1 Concetti fondamentali

- Una struttura dati condivisa è **lock-free** se le operazioni che la riguardano non richiedono la mutua esclusione. Se un processo viene interrotto nel mezzo di un'operazione su un oggetto di dati condiviso, non si impedirà ad altri processi di operare sullo stesso oggetto.
- Le strutture dati lock-free evitano problemi comuni associati alle normali tecniche di locking in sistemi ad alta concorrenza: In particolare:
  - Si verifica **Priority inversion** quando un processo a bassa priorità subisce preemption mentre è padrone di un lock necessario a un processo con priorità più alta.
  - Si verifica **Convoying** quando un processo che è padrone di un lock subisce descheduling, ad esempio, in conseguenza di page fault, di interrupt, per avere esaurito il suo intervallo di tempo, etc. Quando si verifica una tale interruzione, altri processi che potrebbero girare vengono fermati.
  - Si può verificare **Deadlock** se più processi tentano di imporre lock allo stesso insieme di oggetti in ordine differente. Evitare Deadlock può essere difficile se i processi impongono lock a più oggetti di dati, in particolare quando l'insieme di oggetti non è noto in partenza.

# 19 Problema della Cache Coherence e Consistence per multiprocessori

Il sistema di memoria è coerente se:

1. **P legge X $\rightarrow$ P scrive X $\rightarrow$ P legge X $\Rightarrow$ Se non ci sono scritture da parte di altri processi, P legge in X il valore scritto da lui precedentemente**. Una lettura in una posizione X da parte di un processore P, dove segue una scrittura in X da parte di P, senza scritture intermedie in X da parte di altri processori, restituisce sempre il valore scritto da P. Proprietà utile a mantenere l'ordine del programma.
2. **P1 legge X $\rightarrow$ P2 scrive X $\Rightarrow$ Se non ci sono scritture da parte di altri processi, X contiene il valore scritto da P2**. Una lettura in una posizione X da parte di un processore P1, dove segue una scrittura in X da parte di un altro processore P2, restituisce il valore scritto da P2 se lettura e scrittura sono separate da un tempo sufficiente e se non si verificano altre scritture in X fra i due accessi. Proprietà che definisce la nozione di vista coerente della memoria e **implica la propagazione delle scritture**.
3. **Le scritture a una stessa posizione sono serializzate**. Due scritture nella stessa posizione da parte di una qualsiasi coppia di processori sono viste da tutti gli altri processori nello stesso ordine. Proprietà che garantisce che ogni processore in qualche istante possa vedere l'ultimo aggiornamento di una posizione.

**Modello di consistenza della memoria**: stabilisce **quando** un altro processore, lettore, vedrà il valore scritto dal processore scrittore.

Coerenza e consistenza sono complementari.

La coerenza definisce il comportamento di letture e scritture nella stessa posizione.

La consistenza definisce il comportamento di letture e scritture rispetto ad accessi ad altre posizioni di memoria.

**Scrittura completa**: solo quando tutti i processori ne vedono l'effetto.

Il processore può scambiare l'ordine delle letture ma deve completare le scritture in ordine di programma.

## 19.1 Imporre la coerenza

Bisogna tener traccia degli stati dei dati condivisi.

Solitamente la granularità è pari al blocco di cache.

In un multiprocessore, un programma che gira su più processori di solito avrà copie degli stessi dati in più cache diverse, quelle locali dei processori.

In un multiprocessore coerente, le cache consentono sia la migrazione sia la replicazione degli elementi di dati condivisi.

### 19.1.1 Migrazione

Un dato può essere spostato in una cache locale e utilizzato in modo trasparente, riducendo la latenza per l'accesso a dati condivisi allocati remotamente e la latenza della richiesta di banda per la memoria condivisa.

### 19.1.2 Replicazione

Per i dati condivisi che vengono letti simultaneamente le cache fanno copie del dato nella cache locale del processore, riducendo la latenza degli accessi e le contention per un dato condiviso in lettura.

### 19.1.3 Classi di strutture dati rispetto alla Cache Coherence

1. **Strutture a sola lettura**: non provocano problemi di cache coherence.
2. **Strutture dati scrivibili condivise**: la principale sorgente dei problemi di cache coherence.
3. **Strutture dati scrivibili private**: creano problemi solo nel caso di migrazione del processo.

### 19.1.4 Protocolli di Cache Coherence

$2$ classi di protocolli, che usano diverse tecniche per tener traccia dei dati condivisi:

1. **Snooping**: adatto in particolare alle architetture basate su bus.
2. **Directory-based**: soluzione adottata in particolare per architetture a memoria condivisa scalabili, tipicamente DSM (si veda più avanti).

## 19.2 Imporre la consistenza sequenziale

Il risultato di qualsiasi esecuzione è lo stesso che si avrebbe se le operazioni di tutti i processori venissero eseguite in un qualche ordine sequenziale, e le operazioni di ogni processore venissero eseguite in ordine di programma.

Condizioni sufficienti per garantire la consistenza sequenziale in un qualunque multiprocessore:

1. Ogni processo lancia le operazioni sulla memoria in ordine sequenziale.
2. Dopo il lancio di una write, il processo che l'ha lanciata attende il suo completamento prima di lanciare la prossima operazione.
3. Dopo il lancio di una read, il processo aspetta il completamento della read e della write i cui risultati dovranno essere restituiti dalla read prima di lanciare la prossima operazione. Implica l'atomicità delle scritture, è una condizione molto restrittiva.

# 20 Snooping per la Cache Coherence

I nodi, CPU+cache, sono collegati alla memoria condivisa tramite un bus in broadcasting.

Qualunque transazione di bus è visibile a tutti i nodi, nello stesso ordine.

Ogni nodo è dotato di un controllore di cache collegato al bus che osserva ogni transazione di bus e determina se essa è rilevante per quanto lo riguarda, ovvero se deve modificare un blocco della sua cache per preservare la coerenza.

Migliore policy di scrittura per la cache, **write-invalidate**.

## 20.1 Policy write-invalidate

Quando $P$ richiede l'accesso in scrittura ad un blocco che $P2$ ed altri processi hanno nella cache, lo invalida per tutti.

Nel momento in cui $P2$ e gli altri processi cercheranno di accedere a quel blocco, corrisponderà un Miss, e aggiorneranno la propria copia del blocco in cache locale richiedendo quella aggiornata tramite bus.

## 20.2 Proprietà della cache in base agli stati e modifica dati in cache

- **Proprietaria di un blocco**: se deve fornirne il valore quando il blocco viene richiesto.
- **Copia esclusiva del blocco**: se è l'unica che ne abbia una **copia valida** (la memoria primaria può avere o non avere una copia valida).
- **Esclusività**: la cache può modificare il blocco senza avvertire nessun altro. Se una cache non ha esclusività non può scrivere un nuovo valore nel blocco senza prima aver comunicato il fatto alle altre cache con una transazione di bus. Questa transazione viene chiamata **write miss**.

Quando un nodo $P_i$ genera una **write miss in un protocollo write-invalidate**, usa una transazione **read exclusive** per comunicare alle altre cache che sta per fare una write e per acquisire una copia del blocco con proprietà esclusiva.

Come conseguenza il blocco viene portato nella cache $C_i$ e può essere modificato.

Il bus serializza le transazione read exclusive, così che solo una cache per volta può acquisire la proprietà esclusiva.

Quando si sostituisce un blocco modificato, il valore aggiornato viene scritto in memoria.

## 20.3 Snooping asincrono

Un **segnale di handshake** deve indicare alla cache locale, quella del nodo da cui inizia la richiesta, quando tutte le cache hanno completato l'azione di snooping.

L'operazione può essere compiuta mediante un semplice circuito elettronico.

Lo svantaggio è dovuto al tempo addizionale richiesto dallo handshake.

## 20.4 Snooping sincrono

Si stabilisce inizialmente un limite superiore al tempo necessario perché tutte le cache portino a termine le azioni di snooping.

# 21 Primitive hardware fondamentali per la sincronizzazione

Si considerano varie soluzioni di mutua esclusione:

coppie lock-unlock, sincronizzazione su eventi punto-punto mediante flag, sincronizzazione globale su eventi mediante barriere.

La mutua esclusione, lock-unlock è veloce in situazioni di scarsa contention per il lock.

Tre componenti principali:

- **Metodo di acquisizione**: mediante il quale un processo tenta di acquisire il diritto alla sincronizzazione, per entrare nella sezione critica o per procedere oltre un evento di sincronizzazione.
- **Algoritmo di attesa**: il metodo con cui il processo attende che la sincronizzazione si renda disponibile. Indipendente dal tipo di sincronizzazione.
- **Metodo di rilascio**: mediante il quale un processo consente ad altri processi di procedere al di là di un evento di sincronizzazione.

## 21.1 Algoritmi di attesa

### 21.1.1 busy-waiting

Il processo cicla in attesa che una specifica variabile cambi valore.

Il rilascio dell'evento di sincronizzazione da parte di un altro processo cambia il valore della variabile e consente al processo in attesa di continuare.

Evita i costi della sospensione ma consuma tempo di processore e banda.

Risulta migliore su tempi di attesa brevi.

### 21.1.2 Blocking

Il processo in attesa si blocca rilasciando il processore se deve attendere.

Verrà risvegliato al momento del rilascio.

Ha un Overhead maggiore, perché richiama il sistema operativo, ma libera il processore per altre attività.

## 21.2 Garantire le operazioni atomiche o porzioni di codice atomiche

Serie di istruzioni indivisibili, da eseguire in maniera sequenziale.

Si può garantire con una coppia di istruzioni **Load linked** (detta anche load locked) e **Store conditional**.

La Store conditional restituisce un valore da cui si può dedurre se il codice contenuto tra la coppia di istruzioni è stata eseguito come se fosse un'operazione atomica.

Le $2$ istruzioni vengono usate in sequenza.

- Se il contenuto della posizione di memoria specificata dalla Load linked viene modificato prima di una Store conditional nella stessa posizione, la Store conditional fallisce $\rightarrow$ Le operazioni non sono state eseguite come atomiche.
- Se tra le due istruzioni il processore esegue una commutazione di contesto, la Store conditional fallisce.
- **Store conditional**: restituisce un valore che indica se ha avuto successo o no, $1$ se ha successo, $0$ altrimenti. La **Load linked** restituisce il valore iniziale.

**Spin lock**: usati quando il programmatore si aspetta che il lock venga tenuto per pochissimo tempo e desidera che quando il lock è disponibile il processo di blocco (locking) abbia bassa latenza. Sono garantiti da meccanismi di coerenza.

## 21.3 Prestazioni nei casi di lock

- **Bassa latenza**: se un lock è libero e nessun altro processore tenta di acquisirlo, chi ne fa richiesta deve acquisirlo rapidamente.
- **Basso traffico**: se molti processori tentano simultaneamente di acquisire un lock, devono riuscirvi uno dopo l'altro generando meno traffico possibile sul bus.
- **Scalabilità**.
- **Basso costo di memoria**.
- **Fairness** (imparzialità): idealmente, tutti i processori dovrebbero riuscire ad acquisire il lock nello stesso ordine in cui l'hanno richiesto.

**Se il lock è basato su scambio atomico** in caso di competizione si genera molto traffico sul bus e una scarsa scalabilità dovuta al numero di processori in competizione.

# 22 Prestazioni

Solitamente in un sistema parallelo le operazioni di maggiore interesse ed impatto sono i trasferimenti di dati, e si vuole valutarne le prestazioni.

Dal punto di vista della CPU importa sapere:

- Quanto la CPU deve attendere prima di poter usare il risultato di un evento di comunicazione.
- Quanto di questo tempo può usare per altre attività.
- Con quale frequenza può comunicare dati.

**Latenza**: Tempo richiesto da un'operazione.

**Banda**: Frequenza a cui le operazioni vengono compiute.

**Costo**: Impatto che le operazioni hanno sul tempo di esecuzione del programma, quindi il tempo speso dal programma per effettuare la comunicazione.

## 22.1 Tempo di trasferimento dati $T_{transfer}(n)$

$T_{transfer}(n) = T_0 + {}^{n}/_{B}$

$n$: numero di byte.

$B$: frequenza di spostamento dei dati (es. $B/sec$).

$T_0$: costo di start-up.

Per operazioni su memoria, $T_0$ è essenzialmente il tempo di accesso alla memoria.

Per il modello a scambio di dati $T_0$ può essere visto come il tempo necessario affinché il primo bit giunga a destinazione.

Con questa definizione, la banda dei trasferimenti dipende dalle dimensioni del trasferimento.

## 22.2 Trasferimento di dati su una rete di macchine parallele $T_{communication}(n)$

Viene iniziato dal processore tramite il **Communication Assist**.

**Communication Assist**: può essere un banale dispositivo di interfaccia, un'interfaccia di rete o un processore dedicato.

$T_{communication}(n) = T_{CPU\,Overhead} + T_{Occupancy} + T_{Network\,Delay}$

$T_{CPU\,Overhead}$: tempo speso dal processore per dare inizio al trasferimento. Può essere un costo fisso o dipendere da $n$. In questo tempo il processore non può fare altro.

$T_{Occupancy}$: tempo necessario affinché i dati passino attraverso il componente più lento lungo il percorso di comunicazione.

$T_{Network\,Delay}$: tempo di comunicazione. Include il tempo per instradare il primo bit ed altri fattori, come ad esempio il tempo per attraversare il supporto di rete.

## 22.3 Costo di comunicazione

Impatto che le operazioni hanno sul tempo di esecuzione del programma, quindi il **tempo speso dal programma per effettuare la comunicazione**.

$Costo_{communication} = f_{communication} * (T_{communication} - Overlap)$

$f_{communication}$: numero di operazioni di comunicazione per unità di lavoro nel programma. Dipende da aspetti HW e SW, in particolare HW può limitare la dimensione del trasferimento e quindi determinare il minimo numero di messaggi. HW può effettuare la replicazione automatica dei dati o la loro migrazione, etc.

$Overlap$: parte dell'operazione di comunicazione che può essere eseguita simultaneamente con altre operazioni utili. Innanzitutto computazione. Coinvolge lavoro fatto da componenti del sistema che non siano il processore, in particolare dal supporto di comunicazione dalla rete stessa, dal processore remoto o dal controllore di memoria.

## 22.4 Tempo di computazione di un algoritmo su $P$ processori

Sia $T_{alg}/P$ il tempo per compiere una computazione dell'algoritmo su $P$ processori.
$T_{init}$ il tempo necessario per iniziare un thread su un processore.
Il tempo totale per completare l'algoritmo su $P$ nodi è:
$T_{tot} = P * T_{init} + T_{alg}/P$
Lo speedup massimo si ottiene trovando il minimo di $T$ rispetto a $P$:
$T_{init} - T_{alg}/P^2 = 0$
quindi il valore ottimo di $P$ è:
$P = \sqrt{T_{alg}/T_{init}}$

# 23 MIMD con memoria fisicamente distribuita

La memoria è fisicamente distribuita, per supportare numeri più elevati di processori, altrimenti la banda della memoria non potrebbe supportare la comunicazione in modo efficiente.

Il sistema è costituito da nodi in ognuno dei quali oltre a CPU e cache è presente un modulo di memoria.

Si usano sia reti a interconnessione diretta, switches, sia reti indirette, maglie multidimensionali.

Di norma anche l'I/O è distribuito fra i nodi, e ogni nodo può anche essere un piccolo SMP.
Soluzione indispensabile per multiprocessori con numero di processori elevati.
Vantaggi:

- Se gli accessi alla memoria che vengono generati dalla CPU di un nodo riguardano prevalentemente il modulo di memoria locale, la banda della memoria scala in modo efficiente.
- Si riduce la latenza per gli accessi alla memoria locale.

Svantaggi:

- Comunicare dati fra i processori diventa un po' più complesso e la latenza è maggiore, almeno in assenza di conflitti.

#### Approcci architetturali alternativi

1. La comunicazione avviene ancora attraverso uno **spazio di indirizzamento condiviso**. Le memorie sono fisicamente separate ma indirizzate come un unico spazio condiviso. Qualunque CPU può accedere a qualunque posizione, se ha i corretti diritti di accesso. Architetture dette a memoria condivisa distribuita (**DSM**) o anche NUMA (non-uniform memory access). Il tempo di accesso dipende dalla posizione del dato in memoria rispetto alla CPU che richiede l'accesso.
2. Lo spazio di **indirizzamento consiste di più spazi privati logicamente disgiunti** e che non possono essere indirizzati da un processore remoto. Ogni modulo CPU-memoria è un calcolatore separato, strutture dette anche multicalcolatori. La comunicazione avviene mediante esplicito scambio di messaggi, **architetture a scambio di messaggi**. Un multicalcolatore può anche essere un insieme di calcolatori distinti connessi in rete locale, soluzione detta **cluster**.

# 24 SMP - Shared Memory Processors

Sono architetture a memoria centralizzata condivisa, quindi con un bus condiviso.

Aggiungendo CPU la banda per l'accesso alla memoria rimane la stessa e dividendola tra un maggior numero di CPU limita la scalabilità.

Più CPU creano conflitti per accedere alle risorse fisiche condivise, bus e memoria.

Sono caratterizzati da:

- Uno o più livelli di cache per ogni CPU. Ottengo un'architettura scalabile fino a qualche decina di CPU.
- Ogni accesso all'interconnessione, se questa è un bus, si traduce in una transazione di bus.
- Le cache riducono la richiesta di banda da parte di ogni CPU, ma la replicazione dei dati crea **problemi di consistenza della cache**.
- Accessi alla stessa locazione di memoria da parte di più CPU sono sincronizzati per evitare conflitti.

# 25 DSM - Distributed Shared Memory

Sono architetture a memoria condivisa distribuita, per sistemi **scalabili** che possono raggiungere centinaia o migliaia di nodi.

Si può scegliere la scalabilità della memoria, **rinunciando alla coerenza**.

La memoria è distribuita tra i nodi, che sono collegati da una rete di interconnessione.

L'accesso può essere locale o remoto, un controllore sul nodo decide, in base all'indirizzo, che tipo di accesso si deve gestire.

Una soluzione semplice prevede che solo i dati privati possano essere portati nella cache del nodo. Se il software porta in cache dati condivisi, deve poi gestire i **problemi di coerenza**. La gestione da software deve essere conservativa, ogni blocco che potrebbe essere condiviso deve essere trattato come se fosse sicuramente condiviso.

## 25.1 Cache nell'architettura DSM

In un architettura DSM lo stato di un blocco in una cache viene mantenuto in modo esplicito in una directory dove esso può essere esaminato.

Non può infatti essere determinato implicitamente portando una richiesta su un bus condiviso al quale tutti i controllori di cache accedono con azioni di snooping.

Ad ogni blocco di memoria primaria è associata l'indicazione delle cache che attualmente ne contengono una copia, insieme al relativo stato. L'insieme di queste informazioni è l'**entry della directory relativo al blocco**. Possono esistere più cache con una copia leggibile, se però esiste una cache che può effettuare una scrittura, non esistono altre copie.

Se un nodo incontra un **cache miss**:

- Comunica con l'elemento di directory relativo al blocco usando transazioni di rete punto a punto.
- L'elemento di directory si trova sullo stesso nodo che ospita il modulo di memoria del blocco, quindi la posizione viene determinata partendo dall'indirizzo.
- Dalla directory il nodo determina in **quali cache si trovano le eventuali copie** valide e decide cosa fare.

- Il nodo **comunica** eventualmente con le **copie in cache** mediante opportune transazioni di rete, ad esempio, ottiene una copia del blocco da un altra cache o nel caso di una scrittura invia invalidazioni agli altri nodi e ne riceve conferma.
- Anche i cambiamenti di stato dei blocchi vengono comunicati all'elemento di directory mediante transazioni di rete.

## 25.2 Protocolli Directory Based

Si suppone che i dati non locali siano replicati solo nelle cache: le architetture vengono definite anche **cache-coherent**.

Esistono molte varianti di protocolli basati su directory, in linea di principio si possono dividere in **memory-centric** e **cache-centric**.

In **memory-centric** la directory è associata ai moduli di memoria e per ogni modulo mantiene l'informazione su quali nodi condividano qualsiasi blocco nel modulo stesso.

In **cache-centric** l'informazione è associata agli elementi in cache per collegare i nodi che condividono il blocco.

### 25.2.1 Memory-centric

Per ogni protocollo lo stato complessivo di un blocco di memoria nell'insieme delle cache è un vettore che memorizza lo stato del blocco in ogni cache del sistema.

Uno stesso diagramma degli stati governa le copie nelle diverse cache, anche se in un generico istante lo stato presente nelle diverse cache può essere diverso.

I cambiamenti di stato sono governati mediante transazioni di rete.

Un blocco di cache può essere in uno dei seguenti stati:

- Condiviso (**shared**): uno o più nodi hanno in cache il blocco il cui valore è aggiornato in tutte le cache.
- Non in cache (**uncached**): nessun nodo ha in cache una copia del blocco.
- Esclusivo (**exclusive**): un solo nodo ha il blocco in cache e questo nodo lo ha modificato, così che la copia in memoria è scaduta. Il nodo viene detto proprietario (**owner**) del blocco.
- **Invalid**: il contenuto del blocco è stato modificato da un'altro nodo, dovrò richiedere la copia aggiornata prima di procedere.
- **Dirty**: il nodo possiede il blocco modificato più recente, da mandare in copia a tutti gli altri nodi.

Un tentativo di scrivere in una posizione che non è esclusiva genera sempre una **write miss**, e i processori si bloccano fino a quando l'accesso non è terminato.

L'interconnessione ora non è più un semplice bus ma è una rete su cui di norma non si può effettuare broadcast, la rete non può più fungere da punto unico di arbitraggio, inoltre la rete è **message-oriented** e molti messaggi devono avere risposte esplicite.

Quando il local node vuole accedere a un blocco, in caso di cache miss invia una transazione di rete di richiesta (**request**) allo home node su cui si trova l'informazione di directory per il blocco.

In caso di **read-miss** la directory indica da quale nodo si può ottenere l'informazione.

In caso di **write-miss** le invalidazioni o gli aggiornamenti vengono inviati a più copie del blocco, presenti in diverse cache.

Per determinare se una scrittura è giunta a completamento occorre che tutte le cache che contengono una copia mandino **acknowledge esplicito**.

Per ogni blocco si introduce un vettore di $p$ bit, ognuno dei quali indica se una copia è presente in cache nel corrispondente tra i $p$ nodi.

Una **read miss** determina dalla directory se esiste un dirty node con la copia aggiornata del blocco, o se il blocco è valido nella memoria dello home node.

Una **write miss** determina quali sono i nodi condivisori le cui copie del blocco devono essere invalidate.

Considerando il protocollo MSI, i nodi hanno un solo livello di cache e un solo processore. Il protocollo è controllato dall'interfaccia di rete.

In caso di read miss o write miss sul nodo $i$, il controllore locale esamina l'indirizzo per stabilire se lo home node è locale o remoto.

Se il nodo è remoto si invia allo home node una transazione di rete relativa al blocco, e sullo home node si esamina l'elemento della directory.

Possibili azioni:

- **Read miss**
  - **Dirty bit off**: l'interfaccia ottiene la copia del blocco dalla memoria e la fornisce al blocco richiedente con una transazione di rete di risposta e porta a $1$ il bit di presenza $i$.
  - **Dirty bit on**: l'interfaccia risponde al nodo richiedente con l'identità del nodo il cui bit di presenza è on. Il richiedente manda una transazione di rete di richiesta al nodo owner. Qui lo stato viene cambiato in shared e il nodo owner fornisce il blocco sia al nodo richiedente, che lo porta in stato shared nella propria cache, sia alla memoria sullo home node. Qui, il dirty bit viene portato a off e il bit $i$-esimo di presenza $i$ viene portato a on.
- **Write miss**
  - **Dirty bit off**: la memoria ha una copia valida del blocco. Si mandano messaggi di invalidazione a tutti i nodi $j$ il cui bit di presenza è on. Lo home node manda il blocco al nodo richiedente $i$, nell'elemento della directory si lascia on solo il bit $i$-esimo, tutti gli altri vengono annullati, e si porta on il dirty bit. L'interfaccia del nodo $i$ aspetta da tutti i nodi che precedentemente avevano copia del blocco un messaggio di risposta che indichi la rispettiva invalidazione.
  - **Dirty bit on**: il blocco viene prima richiamato dal dirty node usando transazioni di rete con home node e dirty node. La cache cambia il proprio stato a invalido, il blocco viene fornito al nodo richiedente che lo porta nella propria cache con stato dirty. L'elemento della directory è azzerato, lasciando on solo il bit $i$ di presenza e il relativo dirty bit.

Lo **home node** di un blocco funge da arbitro per tutte le richieste relative a quel blocco, il controllore della directory può serializzare le transazioni dirette a un blocco mediante un un ulteriore bit di stato, chiamato **busy bit** o **lock bit**, per segnalare che è in corso una transazione diretta a quel blocco.

Se il controllore della directory riceve una richiesta per un blocco col busy bit a $1$, può o accodare le richieste, oppure rifiutarle inviando al richiedente un messaggio nack.

Lo **svantaggio** dei protocolli basati sul **presence flag vector** è la quantità di memoria necessaria per registrare gli elementi della directory.

Supponendo che ogni nodo ospiti $M$ blocchi, che la dimensione di un blocco sia $B$, che il numero di nodi del multiprocessore sia $N$ e che la lunghezza di ogni elemento della directory sia $d$, l a quantità di memoria (in bit) richiesta dai presence flag vector è:

$Directory = M * N^2 * d$
Quindi la dimensione della directory cresce col quadrato del numero dei nodi.
Si può **ridurre la quantità di memoria** richiesta dal protocollo

- Sostituendo il presence flag vector con un insieme di i puntatori associato a ogni blocco, ognuno dei quali punta a un nodo.Se i nodi sono $N$, ogni puntatore identifica un nodo mediante $log_2 bit$. Se il numero dei nodi che condividono un blocco non supera $i$, è possibile tener traccia correttamente dei condivisori. Il costo in termini di memoria diventa:
  $\frac{directory}{memory+directory} * d = \frac{M*N*logN}{M*N*B+M*N*logN} * d = \frac{logN}{B+logN} * d$
- Si raggruppano i nodi in cluster o regioni. Il presence flag vector tiene traccia delle regioni invece che dei singoli nodi. Lo schema viene detto “coarse-vector”.

### 25.2.2 Cache-centric

Il numero di elementi nelle directory è proporzionale alle dimensioni delle cache invece che a quelle della memoria.

Un insieme di puntatori collega le copie dei blocchi in una lista doppia. In ogni blocco di memoria un puntatore punta a una cache con una copia del blocco.

Ogni elemento di directory in cache ha due puntatori, uno di coda che punta al prossimo nodo nella lista, uno di testa che punta al nodo precedente con una copia del blocco o, se il nodo è il primo della lista, alla memoria.

### 25.2.3 Definizioni utili

- **Home node**: il nodo nel cui modulo di memoria è allocato il blocco.
- **Dirty node**: il nodo che ha in cache una copia del blocco in stato dirty. Può essere lo home node.
- **Owner node**: è quello che al momento contiene la copia valida del blocco e che deve fornirla se necessario. È o lo home node, se il blocco non è dirty in nessuna cache, o il dirty node.
- **Exclusive node**: quello che ha nella propria cache il blocco in stato esclusivo, o dirty exclusive oppure clean exclusive. Il dirty node è anche exclusive node.
- **Local node**, o **requesting node**, contiene la CPU che ha lanciato la richiesta del blocco.
- I blocchi il cui home node è locale per la CPU che ha lanciato la richiesta vengono detti **locally allocated** o semplicemente local blocks, tutti gli altri sono **remotely allocated** o **remote blocks**.

## 25.3 Reti di interconnessione

Il compito di una rete di interconnessione è quello di **trasferire informazione** da qualunque nodo sorgente a qualunque nodo destinazione, supportando le transazioni di rete usate nel modello di programmazione.

### 25.3.1 Requisiti

- Minima latenza possibile.
- Supporto a un gran numero di trasferimenti simultanei.
- Costo relativamente basso, rispetto al resto dell'architettura.

### 25.3.2 Caratteristiche

- **Topologia**: struttura fisica dell'interconnessione (es. ad anello, a griglia, etc.). Nelle reti dirette un nodo host è collegato a tutti gli switch, nelle reti indirette gli host sono collegati solo a specifici sottoinsiemi di switch.
- **Algoritmo di instradamento** (**routing**): determina i percorsi che i messaggi possono compiere nel grafo della rete.
- **Strategia di commutazione** (**switching strategy**): determinano come i messaggi seguono il percorso. Come alternative abbiamo: **circuit switching** e **packet switching.**
- **Meccanismo di controllo del flusso**: determina quando il messaggio si muove lungo il proprio percorso. È necessario in particolare quando due o più messaggi tentano di usare la stessa risorsa di rete simultaneamente. La minima unità di informazione che può essere trasferita lungo un tratto di collegamento e accettata o rifiutata: **flit**.

### 25.3.3 Parametri

- **Diametro**: lunghezza del più lungo percorso minimo fra due nodi.
- **Distanza di instradamento** fra una coppia di nodi: numero di segmenti di collegamento attraversati. Grande almeno quanto la minima distanza fra i due nodi.
- **Distanza media** fra due nodi: media delle distanze di instradamento fra i due nodi.

### 25.3.4 Switch

Uno switch è essenzialmente costituito da porte di ingresso, porte di uscita, una matrice di commutazione interna (**crossbar**) e la logica di controllo per effettuare il collegamento ingressi-uscite in qualsiasi istante.

La **crossbar** è una struttura che consente il collegamento di qualsiasi ingresso con qualsiasi uscita.

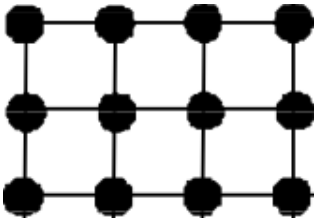
### 25.3.5 Network Interface - NI

L'interfaccia di rete (**NI – Network Interface**) contiene una o più porte di ingresso e di uscita per creare i pacchetti da inviare alla rete e ospitare i pacchetti che arrivano, secondo le indicazioni del communication assist.

La NI garantisce la formattazione dei pacchetti, crea l'informazione di instradamento e di controllo.

### 25.3.6 Topologie di rete

- **Rete totalmente connessa**: un unico switch. Il diametro è $1$. Perdita di un segmento di interconnessione implica la rimozione di un solo nodo. Il costo scala col numero $N$ di nodi. La banda è dell'ordine di $1$, dato che si può supportare una sola comunicazione per volta.
- **Vettori lineari e anelli**
- **Matrici bidimensionali**



# 26 NoC - Network on Chip

Una micro-rete on-chip flessibile e scalabile con comunicazione a pacchetti, progettata secondo una metodologia a livelli e gestita secondo un opportuno protocollo.

L'**architettura NoC** è caratterizzata da maggiore scalabilità e larghezza di banda rispetto alle precedenti connessioni on-chip.

L'approccio NoC permette:

- Modularità e scalabilità.
- Larga banda.
- Concorrenza delle comunicazioni e condivisione delle risorse.
- Efficienza in termini energetici Interconnessione affidabile.
- Supporto per Quality-of-Service.

Solitamente la rete di interconnessione ha complessità limitata, nella maggior parte dei casi una semplice architettura a bus, in soluzioni più complesse un crossbar switch che collega tutti i processori a tutti i moduli di cache.

## 26.1 Soluzione a Bus

**Vantaggi**:

- Semplice e facile da usare.
- Poco costosa.
- Standard su multiprocessori on-chip di dimensioni limitate.

**Svantaggi**:

- Un collo di bottiglia in termini di banda.
- Scarsa efficienza in termini di potenza.

Un progetto basato sulla soluzione a bus implica interconnessioni non strutturate, con un elevato consumo in potenza, area e prestazioni limitate.

E' molto difficile garantire la sincronizzazione con un unico generatore di clock.

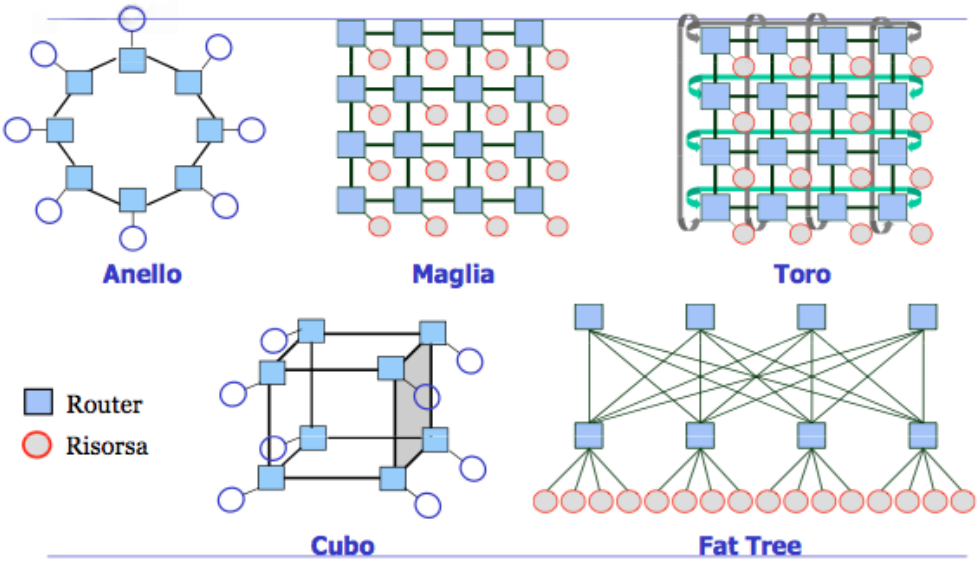
Il tempo di progetto e di verifica può essere lungo.

La presenza di guasti aumenta con la densità dei transistori e la continua riduzione delle geometrie.

Passare ad un progetto basato su NoC-design significa introdurre un sistema di comunicazione on-chip strutturato per superare i limiti di scalabilità del bus.

## 26.2 Topologie

- **Topologie regolari**: adatte a interconnettere core omogenei in un sistema multiprocessore on-chip (MultiProcessor System-on-Chip - MPSoC).




- **Topologie irregolari (Custom)**: adatte per sistemi MPSoC eterogenei caratterizzati da colli di bottiglia critici nelle comunicazioni e da dimensioni dei core non uniformi.

## 26.3 Interfaccia di rete

Si occupa di convertire il protocollo di comunicazione, nascondere i dettagli del protocollo di rete ai nuclei IP e del Data packetization; assemblaggio, trasferimento e disassemblaggio dei pacchetti.

## 26.4 Router

Supporta l'instradamento dei pacchetti da sorgente a destinazione, garantisce la corretta esecuzione del protocollo di rete e permette il parallelismo di comunicazione tra diverse coppie di nodi.

## 26.5 Instradamento Wormhole Switching

I pacchetti sono suddivisi in **flit**, la minima unità di informazione che può essere trasferita lungo un tratto di collegamento.

Si ricorre al pipelining nella comunicazione sulla base del flit.

I buffer sono piccoli.

## 26.6 Canali virtuali

Il canale fisico è a condivisione di tempo tra canali virtuali.

## 26.7 Robustezza

In caso di guasto si può modificare il percorso, ovviamente la riconfigurazione non è indolore, il nuovo percorso è più lungo e quindi la latenza aumenta.

Nasce maggior competizione per l'accesso ad alcuni segmenti di interconnessione e quindi crescono problemi di contention e di gestione delle risorse.

# 27 TM - Transactional Memory

Ci sono proposte per **implementazioni totalmente software** della concorrenza lock-free.

I risultati sperimentali suggeriscono che la mancanza di un supporto hardware possa essere molto costosa in termini di prestazioni.

Lo **scopo** è quello di rendere la **sincronizzazione lock-free altrettanto efficiente quanto le tecniche lock-based**.

Ai programmatori si permette di definire operazioni **read-modify-write** su misura (customized) applicandole a più posizioni di memoria scelte in modo indipendente.

Soluzioni implementate mediante semplice estensione dei protocolli di coerenza di cache per i multiprocessori.

## 27.1 Transazioni

Sono una sequenza finita di istruzioni di macchina eseguite da un solo processore, che soddisfa le seguenti condizioni:

- **Serializzabilità**: le transazioni vengano appaiono come eseguite serialmente. In altre parole, non appare mai che i passi di una transazione si mischino (siano **interleaved**) ai passi di un'altra transazione. Non accade mai che le transazioni che sono giunte a commit vengano viste da un processore diverso come se fossero state eseguite in ordine diverso.
- **Atomicità**: ogni transazione tenta una sequenza di modifiche alla memoria condivisa. Quando la transazione giunge a completamento, o compie un **commit**, rendendo quindi le sue modifiche visibili agli altri processi in modo istantaneo, oppure **abortisce**, facendo così che le modifiche vengano scartate.

Una transazione opera in **isolamento**, cioè viene eseguita come se fosse **l'unica operazione eseguita** dal sistema e come se tutti gli altri thread fossero sospesi mentre essa viene eseguita.

Gli effetti di una transazione sulla memoria non sono visibili al di fuori della transazione fino a quando questa non giunge a commit.

Non ci sono altre scritture in conflitto nella memoria da parte di altre transazioni mentre essa viene eseguita.

La soluzione è efficace quando la probabilità di dovere ri-eseguire una transazione (**rollback**) è bassa.

Naturalmente un sistema TM non esegue veramente le transazioni in modo seriale. Vengono infatti eseguite simultaneamente **più transazioni**, a patto che si possa ancora garantire **atomicità** e **isolamento** per ogni transazione.

## 27.2 Istruzioni

Primitive per accedere alla memoria:

- **Load Transactional** (**LT**): legge il valore di una posizione di memoria condivisa e lo porta in un registro privato.
- **Load Transactional-eXclusive** (**LTX**): legge il valore di una posizione di memoria condivisa e lo porta in un registro privato suggerendo che probabilmente la posizione verrà aggiornata.
- **Store Transactional** (**ST**): tenta di scrivere un valore da un registro privato in una posizione di memoria condivisa. Il nuovo valore diventa visibile agli altri processori solo quando la transazione giunge con successo a commit.
- **Commit** (**COMMIT**): tenta di rendere permanenti le modifiche tentate dalla transazione. Ha successo (**succeeds**) solo se nessun'altra transazione ha aggiornato alcuna posizione del data set della transazione e se nessun'altra transazione ha effettuato una lettura da alcuna posizione nel write set di questa transazione. Se ha successo, le modifiche apportate dalla transazione al suo write set diventano visibili agli altri processi. Se fallisce (**fails**), tutte le modifiche apportate al write set vengono scartate scartate. COMMIT restituisce un indicatore di successo o di fallimento.
- **Abort** (**ABORT**): scarta tutti gli aggiornamenti apportati al write set.
- **Validate** (**VALIDATE**): verifica lo stato corrente della transazione. Una VALIDATE che ha successo (**successful**) restituisce **True**, indicando che la transazione non ha abortito, ma potrebbe ancora farlo più tardi. Una VALIDATE che non ha successo (**unsuccessful**) restituisce **False**, indicando che la transazione corrente ha abortito e scarta gli aggiornamenti fatti in modo tentativo dalla transazione.

In definitiva:

- **Read Set** di una transazione: insieme di posizioni di memoria lette da istruzioni **LT**.
- **Write Set** di una transazione: insieme di posizioni di memoria cui si accede con istruzioni **LTX** o **ST**.
- **Data Set** di una transazione: insieme dei suoi **read set** e **write set**.

Combinando le primitive, il programmatore può definire operazioni read-modify-write customizzate che operano su arbitrarie regioni della memoria.

Vengono supportate anche le istruzioni non-transazionali (**LOAD**, **STORE**) che non influenzano read e write set.

L'idea è che le transazioni sostituiscano delle sezioni critiche corte.

1. Si usa **LT** or **LTX** per leggere da un insieme di posizioni in memoria.

2. **VALIDATE** per verificare che i valori vengano restituiti in modo consistente.
3. Si usa **ST** per modificare un insieme di posizioni in memoria.
4. Si usa **COMMIT** per rendere permanenti le modifiche.
5. Se o la **VALIDATE** o la **COMMIT** falliscono, il processo torna al passo 1.

Un insieme di valori è detto **inconsistente** se non avrebbe potuto essere prodotto da nessuna esecuzione seriale delle transazioni.

Una transazione è detta **orfana** se la sua esecuzione continua dopo che ha abortito, vale a dire dopo che un'altra transazione che ha completato il proprio commit e non ha modificato il read set. Una transazione orfana non potrà **mai giungere a commit**, ma potrebbe essere difficile garantire che non compierà alcuna azione che generi eccezione.

### 27.2.1 Esempio di transazioni

Questa soluzione è studiata per architetture a memoria condivisa, qui la si presenta per protocolli di tipo snooping. Può essere estesa a protocolli **directory-based**.

Le operazioni non-transazionali usano le stesse cache, logica dei controllori di cache e gli stessi protocolli di coerenza che avrebbero usato in assenza della memoria transazionale.

Il supporto hardware dedicato è limitato alle cache L1 e alle istruzioni necessarie per comunicare con esse.

Compiere il **commit** o l'**abort** di una transazione è un'operazione locale alla cache.

Non richiede di comunicare con altri processori o di effettuare una write back di dati in memoria.

L'implementazione viene effettuata come modifica dei normali protocolli di coerenza di cache.

In genere, l'accesso a una posizione di memoria può essere:

- **Non-esclusivo** (**shared**): permette le letture.
- **Esclusivo**: permette le scritture.

In un qualsiasi istante, quindi, una posizione di memoria può essere:

- **Non immediatamente accessibile** da alcun processore (il dato si trova solo in memoria).
- **Accessibile in modo non-esclusivo** dAzioni di snooping della cache

Sia regular cache sia transactional cache effettuano snooping sul bus.

- **Regular cache**:
  - In caso di **READ** o **T_READ**, se lo stato è VALID, la cache restituisce il valore. Se lo stato è **RESERVED** o **DiRTY**, la cache restituisce il valore e riporta lo stato a VALID.
  - In caso di **RFO** o di **T_RFO**, la cache restituisce il valore e invalida la linea.
- **Transactional cache**:
  - Se **TSTATUS** = **false**, o se il ciclo è non-transazionale, READ o RFO, si comporta **come la regular cache**, salvo il fatto che ignora tutti gli elementi con tag transazionale diverso da NORMAL.
  - In caso di **T_READ**, se lo stato è VALID restituisce il valore.

    - Per tutte le **operazioni transazionali**, restituisce BUSY.

Ambedue le cache possono lanciare una richiesta **WRITE** al momento di sostituire una linea.

La memoria risponde solo a richieste **READ**, **T_READ**, **RFO**, **T_RFO** a cui non risponde nessuna cache. Risponde inoltre alle richieste **WRITE**.

- A uno o più processori.
- **Accessibile in modo esclusivo** esattamente da un solo processore.

L'idea di base per supportare la TM: qualsiasi protocollo capace di individuare **conflitti di accessibilità** può identificare anche **conflitti di transazioni** senza costi addizionali.

Prima che il processore $P$ possa **caricare** il contenuto di una posizione di memoria, deve acquisire **accesso non-esclusivo** a quella posizione.

Prima che un altro processore $Q$ possa scrivere in quella posizione, deve acquisire **accesso esclusivo**, quindi deve identificare e revocare il diritto di accesso di $P$.

Queste operazioni vengono sostituite dalle loro controparti transazionali.

## 27.3 Cache

Ogni processore mantiene due cache, una **regular cache** per le **operazioni non-transazionali**, e una **transactional cache** per le **operazioni transazionali**.

Le due cache sono esclusive, infatti **un elemento può risiedere nell'una o nell'altra, mai in ambedue**.

Sono ambedue cache L1, può esistere una L2, o anche una L3, fra di esse e la memoria primaria.

- **Regular cache**: normale cache direct-mapped.
- **Transactional cache**: una piccola cache totalmente associativa, dotata di logica addizionale per facilitare **commit** e **abort**. Ospita tutte le scritture fatte in modo tentativo. Queste non vengono propagate ne altri processori ne alla memoria primaria, a meno che la transazione giunga a commit. Se la transazione **abortisce**, le linee che ospitano dati scritti in modo tentativo vengono invalidate. Se giunge a **commit**, le stesse linee permettono l'accesso in snooping da parte di altri processori e possono essere scritte in memoria primaria.

Una transactional cache introduce dei transactional tags, **mantenendo** i soliti **INVALID**, **VALID**, **DIRTY**, **RESERVED**:

- **EMPTY**: l'elemento non contiene dati.
- **NORMAL**: l'elemento contiene dati giunti al commit.
- **XCOMMIT**: indica di scartare i dati al momento del commit.
- **XABORT**: indica di scartare i dati al momento di abort.

Quando una transazione completa il **commit**, posiziona gli elementi marcati **XCOMMIT** a **EMPTY**, e quelli **XABORT** a **NORMAL**.

Quando **abortisce**, posiziona gli elementi marcati **XABORT** a **EMPTY**, e quelli **XCOMMIT** a **NORMAL**.

Quando la transactional cache ha bisogno di spazio per un nuovo elemento, per prima cosa cerca un elemento EMPTY, poi cerca un elemento NORMAL, ed infine uno con etichetta XCOMMIT.

Se l'elemento XCOMMIT è DIRTY, deve essere riscritto in memoria primaria.

Gli elementi XCOMMIT sono usati per migliorare le prestazioni.

Quando una **Store Transactional** cerca di aggiornare un elemento, il valore vecchio deve essere mantenuto in caso la transazione abortisca.

Se il valore vecchio è nella transactional cache ed è DIRTY, deve o essere marcato XCOMMIT oppure scritto in memoria primaria.

Evitare i write write-back aumenta le prestazioni.

## 27.4 Tipi di cicli di bus

| Nome | Tipo | Significato | Nuovo accesso |
|---|---|---|---|
| **READ** | Regolare | Legge un valore | Condiviso |
| **RFO** | Regolare | Legge un valore | Esclusivo |
| **WRITE** | Ambedue | Write back | Esclusivo |
| **T_READ** | Trans | Legge un valore | Condiviso |
| **T_RFO** | Trans | Legge un valore | Esclusivo |
| **BUSY** | Trans | Rifiuta l'accesso | Immutato |

- **RFO** (Read-For-Ownership): acquisisce proprietà condivisa, oppure esclusiva, di una linea di cache.
- **WRITE**: aggiorna la memoria primaria nel caso che il protocollo non effettui write through. Usato anche quando si sostituiscono elementi modificati. La memoria effettua snooping sul bus, se un altro processore legge un elemento modificato, la versione in memoria primaria viene aggiornata.
- **T_READ** e **T_RFO**: agiscono come READ and RFO, ma richiedono linee di cache in modo transazionale. Le richieste transazionali possono essere rifiutate con un segnale di risposta BUSY.
- **BUSY:** quando una transazione riceve una risposta BUSY, abortisce e riprova, impedendo deadlock o aborto mutuo continuo. Si può evitare la **starvation**, ovvero l'impossibilità da parte di un processo pronto all'esecuzione di ottenere le risorse hardware di cui necessita per essere eseguito, aggiungendo un meccanismo di accodamento.

## 27.5 Azioni del processore

- Transaction Active (**TACTIVE**): indica se una transazione sta procedendo, è posizionata implicitamente quando una transazione esegue la sua prima operazione transazionale.
- Transaction Status (**TSTATUS**): indica se la transazione è attiva (**True**) o ha abortito (**False**). Le istruzioni transazionali lanciate da una transazione abortita non provocano cicli di bus e possono restituire valori arbitrari.

Se **TSTATUS** = **True**:

Azioni di snooping della cache.

Sia regular cache sia transactional cache effettuano snooping sul bus.

- **Regular cache**:

- In caso di **READ** o **T_READ**, se lo stato è VALID, la cache restituisce il valore. Se lo stato è **RESERVED** o **DiRTY**, la cache restituisce il valore e riporta lo stato a VALID.
- In caso di **RFO** o di **T_RFO**, la cache restituisce il valore e invalida la linea.

- **Transactional cache**:
  - Se **TSTATUS** = **false**, o se il ciclo è non-transazionale, READ o RFO, si comporta **come la regular cache**, salvo il fatto che ignora tutti gli elementi con tag transazionale diverso da NORMAL.
  - In caso di **T_READ**, se lo stato è VALID restituisce il valore.
  - Per tutte le **operazioni transazionali**, restituisce BUSY.

Ambedue le cache possono lanciare una richiesta **WRITE** al momento di sostituire una linea.

La memoria risponde solo a richieste **READ**, **T_READ**, **RFO**, **T_RFO** a cui non risponde nessuna cache. Risponde inoltre alle richieste **WRITE**.

- Se l'istruzione è una **LT**: si esplora la transactional cache in cerca di un elemento **XABORT** :
  - Restituisce questo valore se l'elemento esiste.
  - Se non c'è alcun elemento **XABORT**, ma ce n'è uno **NORMAL**, si modifica il NORMAL in XABORT e si alloca allo stesso dato un secondo elemento con tag XCOMMIT.
  - Se non si trova ne un elemento **XABORT** ne uno **NORMAL**, si lancia un ciclo T_READ. Se questo viene completato con successo, si predispongono due elementi nella transactional cache. Il primo con tag XCOMMIT e il secondo con tag XABORT. Lo stato è quello dato in corrispondenza del ciclo READ.
  - Se la risposta è **BUSY**, si abortisce la transazione: TSTATUS è posto a False, gli elementi XABORT vengono lasciati cadere e quelli XCOMMIT sono posti a NORMAL.
- Per una **LTX**: Si usa un ciclo T_RFO in caso di miss. Si porta la linea di cache allo stato RESERVED se la T_RFO ha successo.
- **ST**: si procede come per la LTX, aggiornando il dato nell'elemento XABORT.
- **LT** e **LTX** agiscono come LT e ST per quanto riguarda lo stato di snooping.
- **VALIDATE**: restituisce il TSTATUS flag. Se questo è false, il flag TACTIVE viene posto a false e TSTATUS a true.
- **ABORT**: scarta gli elementi della cache, pone TSTATUS a true e TACTIVE a False.
- **COMMIT**: restituisce TSTATUS, pone TSTATUS a true e TACTIVE a false. Lascia cadere tutti gli elementi di cache con tag XCOMMIT. Modifica tutti i tag XABORT in NORMAL.

## 27.6 Azioni di snooping della cache

Sia regular cache sia transactional cache effettuano snooping sul bus.

- **Regular cache**:
  - In caso di **READ** o **T_READ**, se lo stato è VALID, la cache restituisce il valore. Se lo stato è **RESERVED** o **DiRTY**, la cache restituisce il valore e riporta lo stato a VALID.
  - In caso di **RFO** o di **T_RFO**, la cache restituisce il valore e invalida la linea.

- **Transactional cache**:
  - Se **TSTATUS** = **false**, o se il cicle è non-transazionale, READ o RFO, si comporta **come la regular cache**, salvo il fatto che ignora tutti gli elementi con tag transazionale diverso da NORMAL.
  - In caso di **T_READ**, se lo stato è VALID restituisce il valore.
  - Per tutte le **operazioni transazionali**, restituisce BUSY.

Ambedue le cache possono lanciare una richiesta **WRITE** al momento di sostituire una linea.

La memoria risponde solo a richieste **READ**, **T_READ**, **RFO**, **T_RFO** a cui non risponde nessuna cache. Risponde inoltre alle richieste **WRITE**.

# Parte V
# GPU - Graphic Processing Unit

Dispositivo programmabile progettato per eseguire con altissime prestazioni programmi di sintesi grafica.

Si può utilizzare anche per applicazione caratterizzate da numeri molto alti di thread, tutti essenzialmente istanze di uno stesso thread, che possono essere eseguiti in parallelo con una predominante reciproca di indipendenza per quanto riguarda i dati.

In questo modo **non utilizzo esecuzione speculativa** e **niente predizione dei salti**.

Le cache servono per fornire banda.

Ricorro al multi-threading per mascherare la latenza della memoria.

La GPU è costituita dall'insieme dei Thread Processing Cluster, dalla rete di interconnessione e dai controllori della memoria esterna.

## 28 Architettura e prestazioni

L'architettura privilegia la presenza di un numero molto elevato di processori relativamente semplice. Centinaia di CPU su un chip.

Architettura di riferimento Nvidia Tesla e Fermi.

Le alte prestazioni si ottengono dall'esecuzione parallela di moltissimi thread, piuttosto che dall'esecuzione ottimizzata di un solo thread.

### 28.1 SIMT - Single Instruction Multiple Thread e flusso di esecuzione

La struttura multiprocessore è definita SIMT, ovvero un'estensione del concetto SIMD.

Il parallelismo a livello di thread rispecchia una gerarchia di thread, i **thread vengono raggruppati in warp**, ognuno di essi eseguito indipendentemente dall'altro.

I warp sono l'unità base del flusso di esecuzione. Sono a loro volta **strutturati in blocchi che costituiscono una griglia**.

La **sincronizzazione** tra i diversi warp all'interno di un blocco viene eseguita mediante un meccanismo di barriera.

In corrispondenza, si vede una gerarchia di strutture hardware.

I blocchi di thread vengono eseguiti da **Streaming Multiprocessors**.

**Tutti i thread di un warp** eseguono la stessa istruzione in maniera **lockstepped**, sincronizzazione forzata, ma ogni thread può seguire diramazioni diverse in corrispondenza di salti condizionati. I thread di un warp partono dallo stesso indirizzo, stesso Program Counter. Un warp che esegue un salto condizionato attende che l'indirizzo obiettivo venga calcolato per ogni thread nel warp.

#### 28.1.1 Divergenza di esecuzione causata da salto condizionato

Una divergenza si presenta solo all'interno di un warp.

Se i thread divergono in conseguenza di un salto condizionato dipendente dai dati:

- Il warp esegue in maniera seriale ogni percorso che è stato preso partendo dal punto di diramazione, disabilitando di volta in volta i thread che non sono su quel percorso. I thread entro il warp che seguono lo stesso percorso vengono eseguiti in parallelo.

- Quando l'esecuzione dei percorsi conseguenti a una divergenza termina, i thread riconvergono in uno stesso percorso di esecuzione. Essenzialmente questo aspetto caratterizza la modalità SIMT rispetto alla "classica" modalità vettoriale.

### 28.1.2 Streaming Multiprocessor (SM)

Assomiglia a un processore vettoriale di larghezza $8$ che opera su vettori di larghezza $32$.

Contiene tre diversi tipi di unità di esecuzione:

1. $8$ **Scalar Processor (SP)**: eseguono istruzioni aritmetico/logiche in virgola mobile e in virgola fissa.
2. $2$ **Special Function Unit (SFU)**: eseguono varie funzioni trascendenti e matematiche (sin, cos, etc.) oltre alla moltiplicazione in virgola mobile in singola precisione.
3. $1$ **Double Precision Unit**: esegue operazioni aritmetiche su operandi in virgola mobile di $64$ bit.

Gruppi di SM appartengono a **Thread Processing Cluster (TPC)**, che contengono anche altre risorse HW tipiche delle applicazioni grafiche (Texture Unit) e in genere non visibili al programmatore.

### 28.1.3 Load e Store

Vengono generate internamente agli SM, calcolo dell'indirizzo: registro base + spiazzamento e traduzione da indirizzo virtuale a indirizzo fisico.

Sono lanciate da un warp per volta ed eseguite in gruppi di mezzo warp, $16$ accessi a memoria per volta.

### 28.1.4 Il modello di memoria

- **Memoria globale**: accessibile da tutti i thread in esecuzione, anche se appartenenti a blocchi diversi.
- **Texture memory**: cached, a sola lettura, tipica dell'elaborazione grafica.
- **Constant memory**: costituita da due segmenti, uno accessibile all'utente, l'altro usato per le costanti generate dal compilatore.
- $3$ **livelli di cache istruzioni**: L1 di $4$ KB in ogni SM, L2 di $8$ KB allocata al TPC e L3 globale.

# 29 CPU + GPU

Una GPU non è un elaboratore di uso generale, quindi occorre prendere in considerazione un sistema CPU + GPU, nel quale la CPU scarica attività sulla GPU per incrementare le prestazioni globali del sistema.

Un problema fondamentale nel progetto di un'applicazione è stabilire quali e quante parti dell'applicazione si prestano meglio ad essere eseguite dalla GPU.

## 29.1 Modello di programmazione Nvidia CUDA

E' un'estensione proprietaria di Nvidia del C che definisce due tipologie di codice:

- Codice seriale: verrà eseguito dalla CPU.
- Codice parallelo: verrà eseguito dalla GPU.

## 29.2 Standard OpenCL

Standard industriale orientato al calcolo task-parallel e data-parallel su sistemi eterogenei per CPU, GPU e DSP (Digital Signal Processor - processore dedicato e ottimizzato per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti).