UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 2° anno - 15 CFU

Architettura e sistemi operativi

Professore:
Prof.

Autore:
Matteo Giuntoni

Anno Accademico 2022/2023

Contents

1 L'evoluzione dei processori **2**
1.1 Rivoluzione dei transistor . . . 2
1.2 Circuiti integrati . . . 2
1.3 Legge di Moore . . . 2
1.4 Livelli di astrazione . . . 3

2 Memory Hierarchy **3**
2.1 Memory Technologies . . . 3
2.2 Cost vs Capacity vs Access Time . . . 4
2.3 Von Neumann architecture . . . 4
2.4 Terminologia . . . 4
2.5 The locality principle . . . 5
2.5.1 Locality characterization . . . 5
2.6 Traferimiento dati . . . 6
2.6.1 Cache Memories . . . 6
2.6.2 Gestione del movimento dei dati . . . 6
2.7 Utilizzo della cache . . . 6
2.8 Cache performance . . . 7
2.9 Design cache system . . . 7
2.10 Cache associativa . . . 8
2.11 Cache miss . . . 8
2.12 Gestione delle scritture . . . 8

Architettura e Sistemi Operativi

Realizzato da: Giuntoni Matteo

A.A. 2022-2023

---

1 L'evoluzione dei processori

I primi calcolatori nati durante la seconda guerra mondiale erano basati sulle valvole, strumenti che servivano per simulare le porte logiche. Questa tecnologia aveva due principali problemi, il primo era che si rompevano frequentemente, quindi era necessaria una figura che in maniera ricorrente le sostituisse, il secondo invece era che occupavano molto spazio date le loro dimensioni.

Definizione 1.0.1 (Stored program)**.** *Lo stored program è un sistema di funzionamento dei pc inventato da John Von Neuman nel 1945, esso consisteva nel suddivisione i pc in una parte operativa, la CPU, ed una parte dove salvare i programmi.*

Con questo sistema il programma può essere rappresentato in una forma idonea alla memorizzazione in memoria insieme ai dati, invece di dover riprogettare tutta la macchina per ogni nuovo programma. La prima implementazione fu nel 1952 con una memoria di solamente 40kb.

1.1 Rivoluzione dei transistor

La tecnologia dei transistor è stata inventata nel 1947. E' una tecnologia simile a quella dei valvole, infatti anche loro servono per creare le varie porte logie, però hanno il vantaggio di essere molto più piccoli. Il loro funzionamento è dato dal fatto che sono realizzati in silicio.

Definizione 1.1.1 (Cicli di clock)**.** *Scandisce il tempo in cui un determinato bit all'interno del PC passa da avere il valore 0 ad il valore 1. Questo è un tempo standard.*

1.2 Circuiti integrati

In seguito all'invenzione dei transistor un ulteriore evoluzione all'interno degli strumenti di calcolo è stata quella dei circuiti integrati, inizialmente infatti ogni transistor era appoggiato su un pezzo di silicio dedicato, grazie ai circuiti integrati abbiamo un grande numero di piccoli transistor su un unico pezzo di semiconduttore. Questo porta ad avere una serie di vantaggio:

- Inanzi tutto a livello di prestazioni i processori sono più veloci.
- SI riesce a ridurre in maniera considerevole le dimensioni.
- Si abbattano anche i costi visto che il pezzo di silicone viene diviso in piccole aree dove in ciascuno viene replicato il circuito, in questo modo si può utilizzare una produzione i serie.

1.3 Legge di Moore

Definizione 1.3.1 (Legge di Moore)**.** *Questa legge dice che i processori o gli strumenti calcolo raddoppiano la loro potenza circa ogni anno, e questo data la crescente capacità di ridurre di dimensione i circuiti integrati.*

Questa legge era validità fino a pochi decenni fa, quello che accade ora è che raddoppia il numero di transistor, questo però non sempre conduce ad un aumento di prestazioni. Oggi giorno infatti la frequenza a cui può arrivare un processore è bloccata dal fatto che non si può ridurre i cicli di clock, essi infatti portano ad un aumento del consumo di corrente e di conseguenza l'aumento di temperatura,

essendo che attualmente i sistemi di raffreddamento sono limitati, anche le prestazioni si limato.

Per ovviare a questo problema nei processori moderni lo spazio che si guadagna riducendo di dimensioni i circuiti integrati si sfrutta affiancare processori uguali, questo però porta ad un altro problema, la necessità di scrivere codi e concorrente, che è una tipologia di programmazione che richiede maggiori accortezze.

1.4 Livelli di astrazione

Un calcolatore può essere visto sotto vari livelli di astrazione, possiamo immaginarli come una lente di ingrandimento sopra un processore. Questi livelli sono:

1. Applicazione
2. Sistema operativo
3. Architettura. Per esempio le architetture dei processori M86.
4. Micro-architettura. E' l'implementazione dell'architettura.
5. Componenti logici. Sono per esempio le porte logiche.
6. Dispositivi fisici. Sono quello con cui vengono construtie le porte logiche.

Ogni livello fornisce al livello superiore un interfaccia per il livello superiore, quindi se cambia qualcosa nel livello inferiore ma che va a rispettare l'interfaccia, per il livello superiore non è necessaria nessuna modifica.

2 Memory Hierarchy

I principi fondamentali che andremo a vedere sono legati alle tecnologie con cui vengono costruite le memorie, la gerarchia delle varie tipologie di memorie, le memorie caches ed in fine come andare a misurare e migliorare le performance delle caches.

2.1 Memory Technologies

Partiamo dicendo che esistono varie tipologie di memorie, che possono essere distinte in primo luogo in **memorie volatili**, e **memorie non volatili**. Fra memorie volatili abbiamo:

- Latches, flip-flops, register files (o semplici registri).
- SRAM (Static Random-Access Memory).
- DRAM (Dynamic Random-Access Memory).

Fra le memorie non volatile invece ci sono:

- ROM
- NVRAM
- Flash memory
- Magnetic disks
- And others ...

2.2 Cost vs Capacity vs Access Time

Un altro interessanto confronto da fare fra le memorie e relativo ai costi, le capacità ed il tempo di accesso per ciascuna.

- **SRAM**. Access Time (ns): 0.5 - 1, Bandwidth (GB/s): 25+. Price ($/GB): 5000. Used for registers and caches.
- **DRAM**. Access Time (ns): 10 - 50, Bandwidth (GB/s): 10. Price ($/GB): 7. Used for RAM.
- **Flash memory**. Access Time (ns): 20.000 (20us), Bandwidth (GB/s): 0.5. Price ($/GB): 0.40. Used for: SSD disk (secondary and virtual memory - non volatile).
- **Magnetic disks**: Access Time (ns): 5.000.000 (5ms), Bandwidth (GB/s): 0.75. Price ($/GB): 0.05. Used for: HDD disk (secondary/tertiary storage - non volatile).

Da questa classificazione possiamo trarre alcune regole generali:

- Le memorie di grandi dimensioni sono solitamente lente e economiche.
- Le memorie di piccole dimensioni sono più veloci ma anche più costose.

Da qui possiamo capire che nella selezione della memorie va trovato un compromesso fra i parametri visti precedentemente per andare ad avere memorie sufficientemente grandi per contenere i dati richiesti ma allo steso tempo sufficientemente veloci per evitare il **von Neumann Bottleneck**.

2.3 Von Neumann architecture

Le perfomance dei computer sono limitate nella velocità della CPU dal trasferimento di dati fra le memorie esterne dall'unità di calcolo.
Per andare a mitigare questo problema andiamo a inserire memorie più piccole vicino al processore, mentre più ci si allontana si avrà memorie di grosse ma anche più lente. Da qui vorremmo che il processore lavori alla velocità della memoria più vicina.

L'obbiettivo è quindi di fornire l'illusione di avere una memoria grande quando al memoria più lontana e veloce quando la memoria più vicina. Per creare questa illusione andiamo a far risiedere i dati inizialmente nel livello più lontano e più capiente. Per far accedere il processore bisognerà a andare a spostare i dati. Un primo problema è che fra le memorie diverse i dati saranno salvati in indirizzi diversi, che quindi dovranno esser mappati.

Un altro aspetto è che i trasferimenti di memoria possono non essere di una singola parola, un altro problema è che se un dato viene modificato in una memoria bisogna decidere come propagare le modifiche. Capiamo dunque che si sono un insieme di problemi per andare a implementare questa soluzione.

2.4 Terminologia

Introduciamo innanzitutto un po' di terminologia che ci servirà successivamente.

Definizione 2.4.1 (Hit e Miss)**.** *Se i dati richiesti dal processore compaiono in qualche blocco nel livello di memoria più vicino, si parla di hit. In caso contrario, la richiesta viene definita miss e si accede al livello di memoria successivo per recuperare il blocco contenente i dati richiesti.*

Definizione 2.4.2 (Hit rate)**.** *Il hit rate (frequenza di successi) è la frazione di accessi alla memoria rilevati nel livello superiore (ovvero, più vicino alla CPU), utilizzata come misura delle prestazioni della gerarchia.*

Definizione 2.4.3 (Miss rate)**.** *Il miss rate è la frazione di accessi alla memoria non trovati nel livello superiore.*

Definizione 2.4.4 (Miss penalty)**.** *La miss penalty è il tempo necessario per sostituire un blocco nel livello n con il blocco corrispondente dal livello n-1.*

Definizione 2.4.5 (Miss time)**.** *Il miss time è il tempo per ottenere l'elemento in caso di tempo di miss. miss time = penalità di miss + tempo di hit.*

Il miss rate ed il hit rate se li vogliamo calcolare lo si può fare con la seguenti formule:

$$MR = \frac{\text{Number of misses}}{\text{Number of total memory access}} = 1 - HR \tag{1}$$

$$MR = \frac{\text{Number of hits}}{\text{Number of total memory access}} = 1 - MR \tag{2}$$

Definiamo anche il AMAT (Average Memory Access Time)

$$AMAT = t_{M0} + MR_{M0} * (t_{M1} + MR_{M1} * (t_{M2} + MR_{M2} * (t_{M3} + \dots))) \tag{3}$$

t_{M0} = hit time, MR_{M0} = miss rate, $(t_{M1} + MR_{M1} * (t_{M2} + MR_{M2} * (t_{M3} + \dots))$ = miss penalty.

Osservazione 2.4.1. Se l'hit rate è abbastanza alto, la gerarchia della memoria ha un tempo di accesso effettivo vicino a quello del livello più alto (e più veloce) e una dimensione uguale a quella del livello più basso (e più grande).

2.5 The locality principle

Il principio di località di riferimento (o locality principle) si riferisce al fenomeno nel quale un programma tende ad accedere alla stessa locazione di memoria per un determinato periodo. Possiamo osservare che, se il programma fa riferimento ad una locazione di memoria allora la stessa locazione di memoria verrà riutilizzerà a breve con alta probabilità. Inoltre, gli elementi "vicini" alla posizione di memoria appena raggiunta saranno presto referenziati con un'alta probabilità.

Il principio di località è la forza trainante che rende la gerarchia della memoria funzionante. Esso infatti incrementa la probabilità di riutilizzare dei blocchi di dati che erano stati precedente mossi da un livello n ad un livello n-1, questo riduce il miss rate.

2.5.1 Locality characterization

Andiamo a distinguere due tipologie di località.

- **La località temporale** (o riuso di dati): i dati riferiti precedentemente probabilmente li riferirò nuovamente in un breve lasso di tempo.

  Esempio: consideriamo il seguente codice: for(int i=0; i¡10; i++) {s1 += i; s2 -= 1;}
  In questo caso le locazioni di memoria che contengono s1 ed s2 hanno località temporale.

  Dunque se all'interno della gerarchia della memoria andiamo a tenere i dati più recenti, secondo il principio di località ci riaccenderò con la CPU nuovamente a breve termine.

- **Località spaziale** dati vicini a quelli a cui sto riferendo saranno saranno probabilemete utilizzati a breve.

  Esempio: consideriamo il seguente codice: for(int i=0; i¡10; i++) {func(A[i])}
  In questo caso le locazioni di memoria dell'array hanno località spaziale, visto che sono implementate in modo contiguo.

  Dunque se all'interno della gerarchia della memoria andiamo a tenere i dati vicini a quelli in utilizzo, secondo il principio di località ci saranno grosse probabilità di accederci con la CPU.

2.6 Traferimiento dati

I dati si trasferiscono solamente attraverso due memorie adiacenti. Per ottimizzare il caricamento dei dati esso viene fatto come **blocchi** di granularità di dati. La dimensione dei blocchi può cambiare attraverso i livelli. Per la cache i blocchi vengono chiamati cache line o cache block (il loro valore tipico è 64 - 128 bytes). Per le RAM invece abbiamo pagine si segmenti, mentre per i dischi abbiamo blocchi di dischi.

Consideriamo il seguente codice di C è il suo corrispettivo in assembly.

```
// Sum and A are global variables
int i;
for(int i=0; sum=0; i<N, i++){
    sum += A[i];
}
// One possible compilation ====>
```

```
loop: cmp r3, r3
      beq end
      ldr r12, [r0, r3, lsl #2]
      ldr r4, [r1]
      add r4, r4, r12
      str r4, [r1]
      add r3, r3, #1
      b loop
end: ...
```

In questo frammento di codice viene eseguito il loop N volte, e quindi ogni struttura viene richiesta N volte in maniera sequenziale, in questo caso sia la localizzazione temporale che spaziale viene utilizzata. 'Sum' è ripetutamente letta e scritta, quindi utilizza la località temporale, 'A' è salvata come un insieme contiguo di celle di memoria, quindi utilizzerà la località spaziale.

2.6.1 Cache Memories

La cache memory è la memoria più vicina al processore, solitamente sono le SRAM, ma alcune volte sono implementate anche come DRAM. Ad oggi tutte le architetture hanno alcuni livelli di cache integrati nel chip, essa può essere più o meno grande, ne può avere più di un livello.

2.6.2 Gestione del movimento dei dati

Fra il primo livello di cache e i registri il trasferimento è gestione dal compilato. Il trasferimento fra caches e RAM viene invece gestito dalla microarchitettura. In fine la gestione dei trasferimenti fra RAM e storate viene fatta dal sistema operativo.

2.7 Utilizzo della cache

L'organizzazione della cache avviene non a blocchi ma a line, ogni linea contiene blocchi di memoria (8-16 memory words). la prima volta che il processore richiede la memory ad una cache miss succede che il blocco contenente la parola si trasferisce dentro la cache.
La richiesta successiva può essere di due tipologie:

- **Cache hit**: se il dato è presente nel blocco.
- **Cache miss**: se il dato non è presente dentro il blocco. In questo caso il blocco che contiene il dato viene trasferito dentro la cache line.

Vediamo ora l'effetto della cache sul AMAT. Innanzitutto l'utilizzo di grandi cache nella gerarchia delle memorie aiuta a ridurre il bottleneck di von Neumann.

Esempio 2.7.1. Vediamo un esempio quantitativo stabilendo dei valori:
$t_M = 50ns$ (main memory service time), $t_{L1} = 1ns$ (L1 hit time, cache hit service time).

Abbiamo un Miss rate (MR_{L1}) è del 5%, senza cache AMAT = 50ms mentre con L1 cache $AMAT = t_{L1} + MR_{L1} * t_M = 1 + 0.05 * 50 = 3.5ns$

2.8 Cache performance

$CPU_{time} = ClockCycles * ClockCycleTime = IC * CPi * ClockCyleTime$ in qeusta formula abbiamo:

- IC che è il numero di istruzioni che vengono effettivamente eseguite.
- CPI definito come $\frac{clockcycles}{IC}$

Bisogna sempre tenere conto di quando non è presente in cache. Quindi il calcolo del CPI deve tenere in conisderazione questo fattore, e per farlo si usa il seguente calcolo

$$CPI_{staff} = \frac{MemoryInstructions}{ProgramInstructions} * MIssrate * Misspenality$$

$$CPi = (CPI_{Perfect} + CPI_{Stall})$$

Esempio 2.8.1. Assumiamo che abbiamo un miss rate del 2% per la cache delle istruzioni mentre un 4% per la cache dei dati, ed una miss penality di 100 cicli per ogni mancanza, una frequenza Inoltre di 36% per le ldr, e le str. Se la CPI è 2 senza memory stalls, dobbiamo determinare quanto va più veloce un processore con una cache perfetta rispetto ad una cache reale (che ha le caratteristiche elencate sopra).

$$CPI_{stall-instr} = 1 * 0.02 * 100 = 2cycles$$

$$CPI_{Stall-data} = 0.36 * 0.4 * 100 = 1.44cycles$$

$$CPi_{stall} = 2 + 1.44 = 3.44$$

spendiamo quindi in media 3.44, e quindi in totale il nostro processore CPI = 2 + 3.44 = 5.44.

$$\frac{CPU_{\text{time with stalls}}}{CPU_{timePerfect}} = \frac{IC * (CPI_{perfect} + CPI_{stall}) * Clockcycletime}{IC * CPI_{perfect} * ClockCycleTime} = \frac{5.44}{2}$$

Tenendo quindi conto di questi aspetti negativi, possiamo andare ad agire su uno o più di questi fattori, andando a renderli il più bassi possibili. $AMAT = hit - time + miss - rate * miss - penalty$. Quindi possiamo:

- Ridurre il miss rate
- Ridurre la miss penality
- Redurre l'hit time

2.9 Design cache system

Una delle prime domande che dobbiamo porci è come i dati sono organizzati.
Data una cache di una certa capacità C è organizzata come S sets in cui ciascuno contiene B block (o linee). b è il nnumero di parole per blocco. Da qui possiamo distingure vari metodi di organizzaziazione:

1. Direct mapped
2. N-way set associatice, in cui N definisce il numero di blocchi conetnuti in un set quindi S=B/N
3. Fully associative, in cui in questo caso nell'insieme c'è uno unico blocco contenente tutta la cache.

Da ora andremo a considerare un sistema a 32-bit di indirizzi e 32-bit di parole quindi la memoria ha 2^{30} parole. Andimo ora a visualizafe la cache come una lista di indirizzi che parte da 0 ed arriva a quello finale. Avremo sempre i primi due bit a 0 per indicizare il bite. La funzione di mapping diretta della cache è una funzione rigida.

Per capire dov mettere l'indice della tabella, cioè l'indice di blocco, e per capire se la parola che stiamo inserendo è quella giusta si va a prendere l'indirizzo e si va a raggruppare.

Vediamo un caso realistico dove $b > 1$, prndiamo C = 8 e b = 4, Abbiamo quindi B = C/b = 2 quindi S = 2, dobbiamo quind andare af "affettare l'indirizzo" nel seguente modo:

Vediamo un ultimo esempio. Questa è un cache con una capacità C = 16k, S = B = 256, e b = 16. Avendo 16 parole dobbiamo prendere. V è il bit di validità, e ci dice se le parole seguenti possono essere considerati valide o meno, perchè ci sono casi in cui vogliamo rendere invalido il blocco. I primi due byte di offset, in una parola di 32 bit abbiamo 4 byte

Esempio 2.9.1. Supponiamo di avere indirizzi a 32 bit, una cache ad accesso diretto con S = B = 128 ed ognuna continen b = 8. Dobbiamo trovare la struttura con cui organizare gli indirizzi.
Ci servono inanzitutto 2 bit per l'offset, poi ci servno $log_2 8 = 3$ bits, dobiamo capire quanti bit ci servono per l'indice, che si calcolano con $\log_2 129 = 7bits$, la parte rimanete di bits sono per TAG ed è di 20 bits.

Esempio 2.9.2. Supponiamo sempre di avere indirizzi a 32 bit, una cache ad accesso diretto con S = B = 128 ed ognuna continen b = 8. Dobbiamo caloclare la linea di cache e l'offset all'interno della linea di cache che conterra l'indirizzo 0xFFAC. 0xFFAC = 0...01111 1110 1010 1100 dove si può ragruppare in 3 parti fatte nel seguente modo (TAG, IDX, OFF) = (0...01111, 1110101, 01100).
Quindi la liena di cache ha indice 117 quindi la 118th entry. L'offst nell blocco di cache è 3 quindi la 4th parola di memoria.

Esempio 2.9.3. Consideriamo il seguente codice in C: for(i=0; i¡16, i++) C[i] = A[i] + B[i]. Ed andiamo a considerare solamente le operazioni di ldr.
Poi prendiamo un processore con 2Ghz con un mapping diretto L1 ai dati dalla cache con C = 128, b=8, $t_m = 100$ cicli è $t_{L1} = 4$ cicli. 'A' inizia all'indirizzo 0x00000000, 'B' inizia all'indirizzo 0x00000040 e 'C' inizia all'inidirizzo 0x00000080.

Quindi per risasumere possiamo dire che i pro sono:

- La realizzazione è semplice
- E' molto veloce in caso di hits

Mentre nel caso dei contro abbiamo:

- La funzione di mapping a posizione fisse può generare potenzialmente molti conflitti.
- La troppa rigidità potrebbe avere un grosso impatto sul numero dei conflitti nella casch, essi dipendono dal posizionamento della memoria e dall'utilizzo delle strutture dati.

2.10 Cache associativa

La cahce ad accesso diretto ha una funzione di mapping fissa ed un dato indirizzo può andare solo ad una linea di cache. Mentre una cache ad accesso indiretto possono essserci più indirizzi sulla stessa linea di cache. Che si divide a sua olta in cache Fully associative e N-way set-associative cache.

2.11 Cache miss

Le tipologie di cache miss sono le seguenti:

- **Compulsory misses** questo tipologia di cache misses viene causato dal primo acceso al blocco che non è mai stato in cache.
- **Capacity misses** causato dalla cache che non coneinte tutti i bocchi necessari.
- **Conflict miss** solo per la mappatura diretta per i set associativi

2.12 Gestione delle scritture

Dobbiamo definire quelle che sono le **write hits**, se l'istruzione di stroe scribe il dato soltanto dentro la cache, dopo la cache è la memoria potrebbeero avere differenti valori (quindi abbiamo un inconsistenza)

Abbiamo poi **write miss** possiamo recuperare il blocco dalla memoria alla cache e poi andiamo a sovrascivere la parola mancante?