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

3 Input output **9**
3.1 Legge di Amdahl . . . 9
3.2 Connessione bus . . . 9
3.2.1 Bus design . . . 10
3.3 Gestione dell I/O . . . 11
3.4 Memory-mapped I/O . . . 11
3.5 Programmed I/O . . . 11
3.6 Interrupt-driven I/O . . . 12
3.7 Data transfers (DMA) . . . 12

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?

3 Input output

Quando andiamo a parlare dei sistemi di input output le cose principale da considerare sono i dispositivi di I/O i device controller, i buses, la gestione dell'I/O ed i device drivers.

Solitamente le operazioni di I/O hanno un grosso impatto sul tempo di esecuzione dei programmi. Supponiamo per esempio di andare a calcolare il tempo di esecuzione con $T_{exe} = T_{cpu} + T_{I/O} = \frac{1}{10}T_{cpu}$ perciò $T_{exe} = \frac{11}{10}T_{cpu}$.

Se andiamo ad aumentare il tempo T_{cpu} di 10 volte lasciando inalterato il $T_{I/O}$ abbiamo un $T_{cpu}^{enhanced} = \frac{1}{10}T_{cpu}$ così $T_{exe} = \frac{1}{10}T_{cpu} + \frac{1}{10}T_{cpu} = \frac{1}{5}T_{cpu}$ Consideriamo copi che l'aumento di velocià $= \frac{T_{exe}\text{before enhancmente}}{T_{exe}\text{after enhancmente}} = \frac{11}{5} = 5.5$

3.1 Legge di Amdahl

Proposto da Gene Amdahl nel 1967. Si occupa della potenziale velocità massima raggiungibile da un programma parallelo quando si aumenta il numero di processori da 1 a N. Può essere applicato a qualsiasi processo di ottimizzazione. Si consideri un programma in cui solo la frazione f può essere ottimizzata (ad esempio, parallelizzata utilizzando N processori), mentre la frazione (1-f) rimane inalterata (ad esempio, è intrinsecamente sequenziale).

$$\text{Speedup} = \frac{\text{Tempo di esecuzione proma del miglioramento}}{\text{Tempo di esecuzione dopo il migliormaneto}} = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

Osservazione 3.1.1. L'accelerazione è vincolata dalla frazione sequenziale (1-f), cioè dalla parte del processo che non posso (o non sono in grado) di valorizzare!

Le prestazioni del sottosistema I/O sono molto importanti, ma non sono tutto. Altri aspetti importanti sono:

- **Affidabilità** gestite da metriche del tempo medio al guasto (MTTF), prevenzione dei guasti (componenti migliori), tolleranza ai guasti (introduzione di un certo livello di ridondanza).
- **Disponibilità** invece gestite da Tempo medio di riparazione (MTTR), e dalla formula $\frac{MTTF}{MTTF+MTTR}$

I dispositivi I/O hanno due tipologie di porte: porte di controllo, e porte data.

- Controllo: sia comandi che rapporti di stato, come diciamo al dispositivo cosa fare, come il dispositivo ci racconta le sue caratteristiche, come il dispositivo ci informa sullo stato operativo.
- Data: Alla/dalla memoria del dispositivo.

I/O funzioni di controllo abbiamo invece il control and imting, la comunicazioni fra processi (command decoding, data exchange, status reporting, address recognition), comunicazione fra i device, il data buffering (per ottimizare il trasferimento dei dati e compensare le differenze di velocità).

3.2 Connessione bus

Servono per il controllo del trasferimento da un dispositivo al processore:

1. la CPU controlla lo stato del dispositivo del modulo I/O.
2. Il controller I/O restituisce lo stato se pronta.
3. La CPU richiede il trasferimento dei dati tramite un comando al controllore.
4. Il controller I/O riceve i dati dal dispositivo periferico.
5. il controller I/O trasferisce i dati al processore.

Questi passaggi richiedono una o più azioni di arbitrato del bus per implementare il protocollo di comunicazione. Andiamo ora a definire l'interconnesione fra bus, essa si definisce come una raccolta di linee di dati trattate insieme come un singolo segnale logico utilizzato anche per indicare una raccolta condivisa di linee con più dispositivi connessi (chiamati rubinetti), si definiscono su essi alcuni fattori di prestazione: lunghezza fisica, numero di prese collegate.

Un bus è un mezzo di trasmissione condiviso, solo un dispositivo alla volta può trasmettere con successo. Il bus di sistema collega i principali componenti (processore, memoria, bus I/O), centinaia di linee separate, linee classificate in tre gruppi: dati, indirizzo, controllo.
Alcuni tipi di bus sono:

- **System bus** Collega processore e memoria, I/O interfacciato con adattatori: breve, pochi tocchi e quindi più veloce e larghezza di banda elevata, inoltre non è standard (ovvero specifico del sistema).
- **I/O bus** Connette dispositivi I/O, nessuna connessione diretta con processore e memoria: più tempo, più tocchi è uguale a più lentezza e larghezza di banda inferiore. E' uno standard di settore (ad es. ATA/SCSI).
- **Backplane bus** Connette CPU, memoria, dispositivi I/O. Lunghe, molti tocchi allora lento e larghezza di banda medio/bassa. Gestisce diversi dispositivi con diverse larghezze di banda ed è standard di settore.

3.2.1 Bus design

Il design di un bus si basa su soddisfare gli obbiettivi di avere un alta performance, standardizzazione e bassi costi. Per esempio il bus di sistema enfatizza le prestazioni, l'I/O e i bus backplain enfatizzano i costi e la standardizzazione.
Uno dei problemi principali di progettazioni da affrontare è il seguente:

- I cavi sono condivisi o separati? La risposta è che i bus più ampi (ovvero, una maggiore larghezza di banda) sono più costosi e più suscettibili allo skew.
- Come viene acquisito e rilasciato il controllo del bus? La riposta è tramite due metodologie **Atomic transactions** che permette una bassa utilizzazione ed una bassa complessita, e le **Split-transactions** dove le richieste/risposte possono essere intervallate; ciò significa un utilizzo più elevato ma un design più complesso (ad es. ID per identificare richieste diverse).
- Il bus ha è cloccato (ha un tempo di clock)? Per rispondere al problema del clock bisogna introdurre due opzioni possibili.
  - **Sincrono**. Tutti i dispositivi collegati al bus condividono lo stesso bus clock. Gli eventi si verificano all'estremità del segnale di clock. Un protocollo possibile è che al ciclo X l'unità di I/O scrive una richiesta READ sul bus; al ciclo $X + \Delta$ l'unità può leggere i dati dal bus. Δ è il tempo massimo per scrivere i dati sul bus da parte di un'unità collegata.
    Potenziale abbiamo il problema di skew dell'orologio e questa soluzione è limitata a bus brevi (ad esempio, bus di sistema).
  - **Asincrono**. Il bus non ha il clock. Nessun disallineamento dell'orologio, si occupa di dispositivi con velocità diverse: può essere più lungo, quindi più lento.
    Richiede protocolli di handshaking. Inoltre il protocollo possibilo funzione nel seguente modo (3 linee di controllo, 1 linea dati in cui i dati e l'indirizzo sono multiplexati): UIO1 scrive una richiesta READ (WRITE) ReadReq (WriteReq) nella linea di controllo e l'indirizzo nella linea dati, UIO2 legge il indirizzo e scrive un ACK nella linea di controllo a UIO13. UIO2 scrive i dati sul bus e scrive la riga DataReadycontrol per notificare UIO14. UIO1 legge i dati e invia un ACK nella linea di controllo a UIO2.
- Come si devide chi prende un bus per trasferire dati? Le comunicazioni bus devono essere gestite da un protocollo di comunicazione. Ne possiamo vedere due:

- **Bus master**. Il concetto dietro questo protocolle è che, un unità che può avviare una richiesta di bus. Esso ha una configurazione più semplice: un solo master mentre di solito i bus hanno più master.
- **Arbitration**. Il concetto invece di questo protocolle è che si va ha scegliere un master tra più richieste cercando di implementare la priorità e l'equità (prevenendo la fame). Solo un master alla volta (tutti gli altri ascoltano il bus). Il ruolo dell'arbitro è gestire le richieste del bus e assegnare le sovvenzioni considerando le priorità e l'equità.

  Per andare ad implementale questo sistema possiamo usare un metodo **centralizzato**, dove un dispositivo dedicato (Arbiter) raccoglie le richieste e poi decide (potenziale problema di collo di bottiglia). Si può usare un metodo ri raccolta Daisy-chain, non equo, oppure con linee di richiesta/risposta indipendenti (ampiamente utilizzate).

  Oppure possiamo usare un metodo **distribuito** dove ogni dispositivo vede le richieste contemporaneamente e partecipa alla selezione del master successivo (più complesso da realizzare e necessita di molte linee di controllo).

3.3 Gestione dell I/O

Quando andiamo a parlare di gestione dei dispositivi di input output ci chiedimao come la CPU da i comandi hai dispositivi di I/O? Come la CPU sa quando i dispositivi di I/O completando le operazioni? Come fanno i disponsitivi di I/O ad eseguire il data transfers?

Partiamo dal rispondere alla prima domanda. Inanzitutto soltato il sistema operativo può mandare comandi hai dispositiv di I/O ed i programmatori per mandare comandi all'I/O deve utilizzare delle chiamate di sistema. Da qui possiamo avere due opzioni per mandare comandi hia device di I/O:

- I/O istruzioni: istruzioni ISA che indirizzano i registri dei dispositivi I/O. Sono delle istruzioni di privilegio disponibile soltato in kernel mode.
- Memory-mapped I/O: una porzione degli indirizzi fisici rispervata all'I/O.

3.4 Memory-mapped I/O

I registri interni dei dispositivi sono mappati su locazioni di memoria principali (a indirizzi fisici riservati). I comandi di I/O sono lettura/scrittura di memoria standard. Le operazioni in quelle posizioni vengono reindirizzate ai controller del dispositivo dalla MMU. Gli accessi in modalità utente alle aree mappate in memoria generano eccezioni di violazione della memoria.

3.5 Programmed I/O

Per andare invece ora a rispondere alla seconda domanda che era stata posta. Come la CPU sa quando i dispositivi di I/O completando le operazioni? Introduciamo due concetti che possano risolvere questo probelma:

- Programmed I/O: la cpu gestisce direttamente le operazioni e gli status di I/O.
- Interrupt-driven I/O: Un interrupt è un evento asincrono proveniente da un dispositivo (ad es. modulo I/O, timer, controller DMA). A volte le eccezioni sono anche c hiamate interruzioni (o comprendono interruzioni). In generale, le eccezioni sono eventi sincroni che interrompono l'esecuzione del programma (ad esempio, overflow aritmetico, istruzione non definita).

Nel caso specifico del programmed I/O come scritto sopra la CPU ha un diretto controllo sopra l'I/O, quindi gestisce lo stato di rilevamento, i comandi di lettura e scrittura ed il trasferimento dei dati. Con questo sistema i ll dispositivo I/O ha un ruolo passivo e la CPU attende che il modulo I/O completi le operazioni (polling).

Nel pulling il processore interroga il registro di stato del dispositivo I/O. Esso ha come pro una facile implmentazione mentre come contro una perdita di cicli del processore perchè la CPU è molto più veloce che i dispositivi di I/O. La cpu deve leggere lo sato dei registri molte volte solo per scoprire che il dispositivo non ha completato l'operazione.

3.6 Interrupt-driven I/O

Il problema principale visto nel programmed I/O è la perdita di tempo per l'alta velocità della CPU, come alternativa al polling si introduce quello che viene chiamato interrupts.

- La CPU invia comandi a un dispositivo e continua a svolgere altre attività.
- Il dispositivo I/O genera un interrupt quando lo stato cambia, cioè i dati sono pronti.
- Gli interrupt di I/O sono asincroni.
- I/O interrupts sono prioritari, dove in particolare i dispositivi I/O con larghezza di banda elevata hanno una priorità maggiore rispetto a quelli con larghezza di banda ridotta.

Dal punto di vista della CPU viene inviato un comando I/O (ad esempio, un carico mappato in memoria), nel fra tempo viene fatto altro lavoro, viene poi controllato l'interrupt alla fine del ciclo fetch-execute e se l'iterrupt viene ricevuto salva il contesto corrente (ovvero i registri, non necessariamente tutti) e passa al livello privilegiato (ovvero PL1 su processore ARM), sulla base dell'ID dell'interrupt e della priorità dell'interrupt, il gestore dell'interrupt associato è pronto per essere eseguito, [1] esegui poi il gestore (ovvero, il sistema operativo gestisce l'interrupt) e ripristina il contesto per l'esecuzione dell'istruzione successiva: [2]

Esempio 3.6.1. Facciamo ora un esempio, assumiamo di avere una CPU ad una velocità di 500MHz ed un costo di polling di 400 cicli. Abbiamo poi un mouse come dispositivo a banda lenta. Vogliamo aver un poll di 30 volte al secondo.

Per calcolare i cicli al secondo per polling scriviamo (30 poll/s) * 400 = 120000 cycles/s, e la percentuale di cicli spesi per polling si calocla fancendo 12K/500M = 0.002%. Abbiamo dunque un overhead accettabile.

Prendiamo ora un disco (che è un dispositivo ad alta lunghezza si banda) che ha 4MB/s di transfer rate e 16B interface.
Per non mancare i dati dobbiamo eseguire il poll abbastanza spesso, quindi (4MB/s)/16B = 250 K/s, vista come percentuale di cicli abbiamo 100M / 500 M = 20%, questo risultato non è accetabile perchè Abbiamo speso il 20% dei cicli solo per controllare i registri I/O, non per il trasferimento dei dati

3.7 Data transfers (DMA)

Andiamo ora a rispondere all'ultima domanda che ci eravamo posti, come i device di I/O eseguno il trasferimento dei dati? Sappiamo ora che l'I/O guidato da interrupt elimina i problemi di polling, tuttavia, il sistema operativo deve trasferire i dati una parola alla volta. Questo va bene solo per dispositivi I/O a bassa larghezza di banda (ad es. mouse), per ovviare a questo problma introduciamo la direct memory access (DMA).

Esso è meccanismi che forniscono a un dispositivo di controllo I/O la capacità di trasferire i dati direttamente da e verso la memoria principale senza coinvolgere la CPU. DMA disaccoppia il protocollo CPU-memoria dal protocollo memoria-dispositivo I/O. Interrupt utilizzati dal dispositivo I/O per comunicare con la CPU solo al completamento del trasferimento I/O o quando si verifica un errore.

DMA è implementato con un controller specializzato. Per eseguire DMA, un dispositivo I/O è collegato a un controller DMA: è possibile collegare più dispositivi allo stesso controller. Il controller stesso è visto come un dispositivo I/O mappato in memoria, inoltre il controller DMA si occupa dell'arbitrato

[1] I passaggi precedenti sono atomici dal punto di vista del sistema operativo!
[2] il ripristino è un'operazione atomica per il sistema operativo!

del bus e del trasferimento dei dati: diventa il master del bus. Il controller DMA e la CPU si contendono il bus di memoria. Ci sono tre passi nel transfer DMA:

1. La CPU imposta DMA fornendo identità, op, indirizzo di memoria e numero di byte da trasferire.
2. DMA avvia l'operazione sul dispositivo e arbitra la connessione. Trasferisce i dati dal dispositivo o dalla memoria.
3. Una volta completato il trasferimento DMA, il controller invia un interrupt alla CPU.

Esempio 3.7.1. Facciamo un esempio, assumiamo i seguenti dati: 500Mhz CPU, il device di un disco ha un tempo di trasferimento 4MB/s, 16B di interfaccie e un utilizzo di 50%, la gestione degli interrupt richiede 400 cicli, il trasferimento dei dati richiede 100 cicli, l'installazione DMA richiede 1600 cicli, trasferisce un blocco da 16 KB alla volta.

Overhead del processore per I/O guidato da interrupt senza DMA (il processore è coinvolto per ogni trasferimento di dati (16 B)) si calcola facendo 0.5 * (4MB/s) / (16B) * [(400 + 100) cicli / 500M cicli/s] = 12.5 %.

Overhead del processore per I/O guidato da interrupt con trasferimento DMA (Il processore è coinvolto una volta per trasferimento a blocchi (16 KB)) si calcola invece facendo 0.5 * (4M B/s) / (16KB) * [(1600 + 400) / (5ooM cicli/s)] = 0.05 %

Notiamo da questo esempio che Senza DMA: il processore avvia tutti i trasferimenti di dati, tutti i trasferimenti passano attraverso la traduzione degli indirizzi (MMU), i trasferimenti possono essere di qualsiasi dimensione e attraversare i limiti della pagina virtuale, nessun impatto sulla gerarchia della cache, le cache non contengono mai dati obsoleti.

Mentre con DMA: il controller DMA avvia i trasferimenti di dati, esiste un altro percorso per il sistema di memoria, potenziali problemi: i controller DMA utilizzano indirizzi virtuali o fisici? Cosa succede se scrivono i dati in una posizione di memoria cache?

Per rispondere alla domanda, quali indirizzi il processore specifica al controller DMA? Abbiamo la virtual DMA e la DMA fisica:

- Virtual DMA: Il controller DMA deve eseguire internamente la traduzione degli indirizzi utilizzando una piccola cache (TLB) inizializzata dal sistema operativo quando richiede un trasferimento I/O. Complesso ma flessibile (ad esempio, grandi trasferimenti tra pagine).
- Physical DMA: Il controller DMA funziona con indirizzi fisici. Trasferimenti alla granularità della pagina, il sistema operativo suddivide i trasferimenti di grandi dimensioni in blocchi delle dimensioni della pagina. Più semplice, ma meno flessibile.

La DMA ha anche un'integrazione con la cache essendo che la memorizzazione nella cache riduce le istruzioni della CPU e la latenza di accesso ai dati e riduce l'utilizzo della memoria da parte della CPU, lasciando così più larghezza di banda di memoria/bus per i trasferimenti DMA. Ma le cache introducono un problema di coerenza per DMA: se il DMA scrive nella memoria principale, la versione dei dati nella cache è obsoleta. Per le cache write-back, il controller DMA potrebbe leggere i vecchi valori dei dati dalla memoria mentre i dati nella cache hanno il set di bit sporchi. La soluzione è lo svuotamento/invalidamento selettivo delle linee memorizzate nella cache coinvolte nei trasferimenti DMA: richiede una logica aggiuntiva per la coerenza della cache HW!

L'accesso hai device è gestito tramite, uno stack software che fornisce modi per accedere a un'ampia gamma di dispositivi I/O. Un'interfaccia comune, in POSIX equivalente all'accesso ai file, in termini di chiamate di sistema di apertura/chiusura, lettura/scrittura. Driver di dispositivo per ogni dispositivo specifico, parte superiore HW, indipendenti (per migliorare la portabilità), parte inferiore HW-dipendente