



# 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 Giuntori

---

Anno Accademico 2022/2023

# Contents

<b>1</b>	<b>Memory Hierarchy</b>	<b>2</b>
1.1	Memory Technologies . . . . .	2
1.2	Cost vs Capacity vs Access Time . . . . .	2
1.3	Von Neumann architecture . . . . .	3
1.4	Terminologia . . . . .	3
1.5	The locality principle . . . . .	4
1.5.1	Locality characterization . . . . .	4
1.6	Traferimento dati . . . . .	4
1.6.1	Cache Memories . . . . .	5
1.6.2	Gestione del movimento dei dati . . . . .	5
1.7	Utilizzo della cache . . . . .	5
1.8	Cache performance . . . . .	6
1.9	Design cache system . . . . .	6
1.10	Cache associativa . . . . .	8
1.11	Cache miss . . . . .	8
1.12	Gestione delle scritture . . . . .	8
1.12.1	Write-Through . . . . .	8
1.12.2	Write-Back . . . . .	9
1.12.3	Ottimizzare le scritture . . . . .	9
1.13	Cache replacement . . . . .	9
1.13.1	Cache replacement policy . . . . .	9
1.14	Designing the Memory System . . . . .	10
1.15	Problemi cache . . . . .	10
1.15.1	Problemi di corenza . . . . .	10
1.15.2	Write invalidate protocol . . . . .	10
<b>2</b>	<b>Input output</b>	<b>11</b>
2.1	Legge di Amdahl . . . . .	11
2.2	Connessione bus . . . . .	12
2.2.1	Bus design . . . . .	13
2.3	Gestione dell I/O . . . . .	14
2.4	Memory-mapped I/O . . . . .	14
2.5	Programmed I/O . . . . .	14
2.6	Interrupt-driven I/O . . . . .	15
2.7	Data transfers (DMA) . . . . .	16
<b>3</b>	<b>Dischi rigidi</b>	<b>18</b>
3.1	Dischi magnetici . . . . .	18
3.1.1	Settore . . . . .	18
3.1.2	Disk performance . . . . .	19
3.2	Dishi SSD . . . . .	19
3.2.1	Flash translation layer . . . . .	19
3.2.2	File system flash . . . . .	20

# Architettura e Sistemi Operativi

Realizzato da: Giuntoni Matteo

A.A. 2022-2023

---

## 1 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.

### 1.1 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 volatili invece ci sono:

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

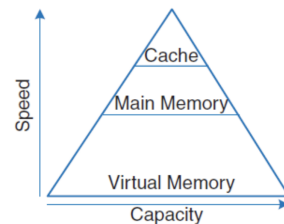


Figure 1: Gerarchia memoria

### 1.2 Cost vs Capacity vs Access Time

Un altro interessante confronto da fare fra le memorie è 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 stesso tempo sufficientemente veloci per evitare il **von Neumann Bottleneck**.

### 1.3 Von Neumann architecture

Le performance 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.

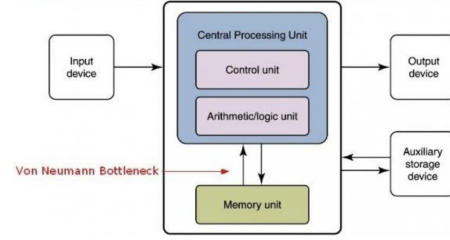


Figure 2: Von-Neumann

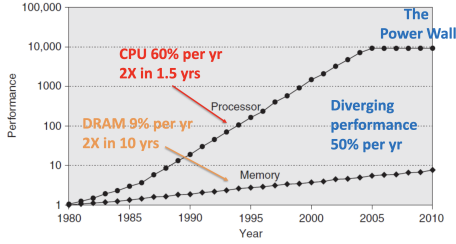


Figure 3: Von-Neumann Bottleneck

L'obiettivo è quindi di fornire l'illusione di avere una memoria grande quando la 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 andare a spostare i dati. Un primo problema è che fra le memorie diverse i dati saranno salvati in indirizzi diversi, che quindi dovranno essere 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.

### 1.4 Terminologia

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

**Definizione 1.4.1** (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 1.4.2** (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 1.4.3** (Miss rate). *Il miss rate è la frazione di accessi alla memoria non trovati nel livello superiore.*

**Definizione 1.4.4** (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 1.4.5** (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:

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

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

Definiamo anche il AMAT (Average Memory Access Time)

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

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

**Osservazione 1.4.1.** 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).

## 1.5 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 riutilizzata 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 precedentemente mossi da un livello  $n$  ad un livello  $n-1$ , questo riduce il miss rate.

### 1.5.1 Locality characterization

Andiamo a distinguere due tipologie di località.

- **La località temporale** (o riutilizzo 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 probabilmente 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.

## 1.6 Traferimento 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 o segmenti, mentre per i dischi abbiamo blocchi di dischi.

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

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.

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

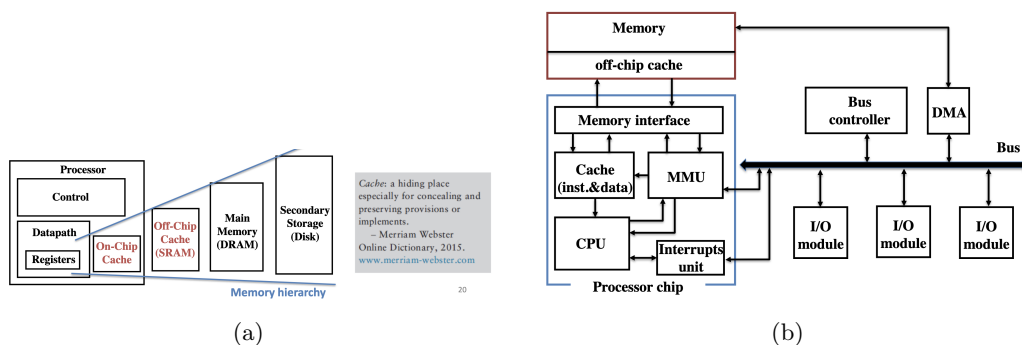
```
loop: cmp r3, r3
      beq end
      ldr r12, [r0, r3, lsl #2]
      ldr r4, [r1]
      add r4, r4, r12
      str r4, [r1]
      add r3, r3, #1
      b loop
end: ...
```

### 1.6.1 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.

### 1.6.2 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 storage viene fatta dal sistema operativo.



## 1.7 Utilizzo della cache

L'organizzazione della cache avviene non a blocchi ma a linee, ogni linea contiene blocchi di memoria (8-16 memory words). la prima volta che il processore richiede la memoria 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 1.7.1.** 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  $AMAT = t_{L1} + MR_{L1} * t_M = 1 + 0.05 * 50 = 3.5ns$

## 1.8 Cache performance

$CPU_{time} = ClockCycles * ClockCycleTime = IC * CPI * ClockCycleTime$  in questa 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 considerazione questo fattore, e per farlo si usa il seguente calcolo

$$CPI_{stall} = \frac{MemoryInstructions}{ProgramInstructions} * Missrate * Misspenalty$$

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

**Esempio 1.8.1.** Assumiamo che abbiamo un miss rate del 2% per la cache delle istruzioni mentre un 4% per la cache dei dati, ed una miss penalty 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  $CPI = 2 + 3.44 = 5.44$ .

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

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 penalty
- Ridurre l'hit time

## 1.9 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 numero di parole per blocco. Da qui possiamo distinguere vari metodi di organizzazione:

1. Direct mapped
2. N-way set associative, 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. Andiamo ora a visualizzare 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 indicizzare 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  $b > 1$ , prendiamo  $C = 8$  e  $b = 4$ , Abbiamo quindi  $B = C/b = 2$  quindi  $S = 2$ , dobbiamo quindi andare a "affettare l'indirizzo" nel seguente modo:

Vediamo un ultimo esempio. Questa è una 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

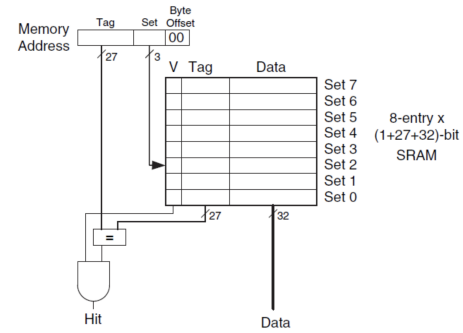


Figure 5: Direct mapped

**Esempio 1.9.1.** Supponiamo di avere indirizzi a 32 bit, una cache ad accesso diretto con  $S = B = 128$  ed ognuna contiene  $b = 8$ . Dobbiamo trovare la struttura con cui organizzare gli indirizzi. Ci servono innanzitutto 2 bit per l'offset, poi ci servono  $\log_2 8 = 3$  bits, dobbiamo capire quanti bit ci servono per l'indice, che si calcolano con  $\log_2 128 = 7$  bits, la parte rimanente di bits sono per TAG ed è di 20 bits.

**Esempio 1.9.2.** Supponiamo sempre di avere indirizzi a 32 bit, una cache ad accesso diretto con  $S = B = 128$  ed ognuna contiene  $b = 8$ . Dobbiamo calcolare la linea di cache e l'offset all'interno della linea di cache che conterrà l'indirizzo  $0xFFAC$ .  $0xFFAC = 0...01111\ 1110\ 1010\ 1100$  dove si può raggruppare in 3 parti fatte nel seguente modo (TAG, IDX, OFF) =  $(0...01111, 1110101, 01100)$ . Quindi la linea di cache ha indice 117 quindi la 118th entry. L'offset nel blocco di cache è 3 quindi la 4th parola di memoria.

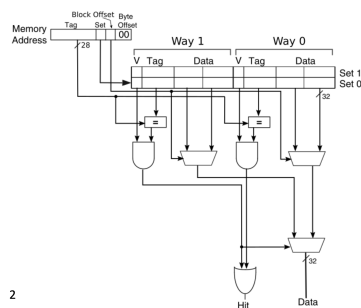
**Esempio 1.9.3.** Consideriamo il seguente codice in C: `for(i=0; i<16; i++) C[i] = A[i] + B[i]`. Ed andiamo a considerare solamente le operazioni di ldr. Poi prendiamo un processore con 2Ghz con un mapping diretto L1 ai dati dalla cache con  $C = 128$ ,  $b=8$ ,  $t_m = 100$  cicli e  $t_{L1} = 4$  cicli. 'A' inizia all'indirizzo  $0x00000000$ , 'B' inizia all'indirizzo  $0x00000040$  e 'C' inizia all'indirizzo  $0x00000080$ .

Quindi per risumere possiamo dire che i pro sono:

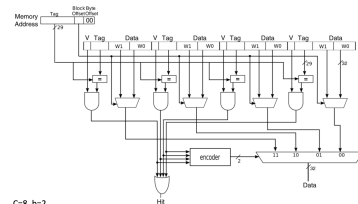
- La realizzazione è semplice
- E' molto veloce in caso di hits

Mentre nel caso dei contro abbiamo:

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



(a) Fully associative



(b) N-way associative



## 1.10 Cache associativa

La cache 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 esserci più indirizzi sulla stessa linea di cache. Che si divide a sua volta in cache Fully associative e N-way set-associative cache.

## 1.11 Cache miss

Le tipologie di cache miss sono le seguenti:

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

A questo punto, visti i vari strumenti della cache e le tipologie di miss possiamo andare a vedere come ridurre i cache miss tramite l'utilizzo di alcune tecniche:

- **Aumentare la dimensione dei blocchi.** Andando così ad aumentare la località spaziale ma aumentando anche la miss penalty.
- **Aumentare l'associatività.** Meno conflitti ma un maggiore tempo di hit.
- **Aumentare dim. dei cache.** Porta a meno capacity miss e conflitti ma aumenta l'hit time.
- **Usare algoritmi di cache-oblivious.**

In caso comunque avvengano dobbiamo andarli a gestire. Un errore nella cache blocca l'intero processore, congelando il contenuto di tutti i registri durante l'attesa della memoria: in effetti, un processore più avanzato consente l'esecuzione fuori ordine di altre istruzioni durante l'attesa della memoria. Gli steps che vengono eseguiti per la gestione di un cache miss (**fault management**):

1. Indica al livello di memoria successivo di leggere il valore mancante.
2. Attendi che la memoria risponda (questo può richiedere più cicli).
3. Aggiorna la riga della cache corrispondente con i dati ricevuti.
4. Riavvia l'esecuzione dell'istruzione (ora è un hit della cache).

## 1.12 Gestione delle scritture

Dobbiamo definire quelle che sono le **write hits**, se l'istruzione di store scrive il dato soltanto dentro la cache, dopo la cache è la memoria potrebbero avere differenti valori (quindi abbiamo un'inconsistenza). Abbiamo poi **write miss**.

- Chiediamoci se possiamo recuperare il blocco dalla memoria alla cache e poi andiamo a sovrascrivere la parola mancante? La risposta è sì e questa policy si chiama **write-allocate**. Il blocco viene caricato nella cache seguito da un hit di scrittura (utilizzato principalmente nella politica Write-Back).
- Poi chiediamoci anche se dovremmo scrivere la parola direttamente al livello di memoria successivo? Questa politica è chiamata **no-write-allocate**, il blocco non viene caricato nella cache (utilizzato principalmente nella politica Write-Through)

### 1.12.1 Write-Through

Gli hit di scrittura aggiornano sempre sia la cache che il successivo livello di memoria.

- **Pro:** Soluzione semplice, facile da implementare. I dati sono sempre coerenti tra i due livelli di memoria.
- **Contro:** La velocità di scrittura dipende dalla velocità di scrittura del livello di memoria inferiore. Maggiore traffico di memoria, per ogni singola scrittura potrebbero esserci più scritture in ogni livello di memoria.

### 1.12.2 Write-Back

I colpi di scrittura aggiornano solo la cache, quindi il blocco modificato viene scritto nel livello di memoria inferiore quando viene sostituito.

- **Pro:** la velocità delle scritture è quella della cache: minore traffico di memoria rispetto a Write-Through, le successive scritture sullo stesso blocco di cache non producono traffico con il livello di memoria inferiore più costoso.
- **Contro:** Abbiamo bisogno di tenere traccia dei blocchi modificati (bit sporchi). Più complesso da implementare rispetto al Write-Through. La sostituzione della riga della cache è più costosa.

Il write-back può migliorare le prestazioni specialmente quando la CPU genera istruzioni di memorizzazione più velocemente di quanto la memoria principale può gestire, tuttavia, il costo delle scritture nella cache è maggiore se si verifica un errore di scrittura, dobbiamo prima riscrivere il blocco in memoria (se il dirty bit è 1). Ciò richiede almeno due cicli anche per un hit di scrittura: un ciclo per verificare un hit seguito da un ciclo per eseguire effettivamente la scrittura. In alternativa, possiamo usare un buffer di scrittura per trattenere temporaneamente i dati da scrivere mentre il blocco della cache è controllato per un colpo. Il processore esegue la ricerca nella cache e inserisce i dati nel buffer di scrittura durante il normale ciclo di accesso alla cache. Supponendo un riscontro nella cache, i nuovi dati vengono scritti dal buffer di scrittura nella cache al successivo ciclo di accesso alla cache inutilizzato (pipelining degli accessi).

### 1.12.3 Ottimizzare le scritture

Poiché la scrittura nella memoria off-chip è costosa, gli archivi di memoria sono bufferizzati, un **buffer di scrittura** viene utilizzato per conservare i dati in attesa di essere scritti nella memoria. L'esecuzione continua immediatamente dopo la scrittura dei dati nella cache e nel buffer di scrittura, la memoria memorizza nella memoria principale viene eseguita in parallelo con il calcolo della CPU, la CPU va in stallo solo se il buffer di scrittura è pieno, quindi la larghezza di banda richiesta dalla memoria principale è un fattore critico, in particolare per il modello di cache Write-Through!

## 1.13 Cache replacement

In una direct mapped cache, il blocco richiesto può andare esattamente in una posizione, quindi non abbiamo scelta: se il blocco da sostituire è stato modificato e la politica di scrittura è Write-Back, dobbiamo aggiornare la memoria di livello inferiore. Cache associativa, possiamo scegliere dove posizionare il blocco richiesto:

- Fully associative cache, tutti i blocchi sono candidati per la sostituzione.
- N-way set-associative cache, dobbiamo scegliere tra gli N blocchi nel set selezionato.

### 1.13.1 Cache replacement policy

Per decidere quali blocchi andare a rimpiazzare si possono applicare delle policy di rimpiazzamento, alcune di queste sono:

- Lo schema più utilizzato è **Least Recent Used (LRU)**. Considera la località temporale, il blocco sostituito è quello che è rimasto inutilizzato per il tempo più lungo. Per una cache set-associativa a 2 vie, può essere implementato con 1 bit (usare bit -U), per un 4 vie è ancora fattibile con 2 bit, per più di 4 vie diventa abbastanza complicato (pseudo-LRU).
- Per le cache altamente associative, una politica **Random** offre all'incirca le stesse prestazioni di LRU.

**Esempio 1.13.1.** Considera una piccola cache con 4 blocchi,  $b=1$ . Trova il numero di errori per una cache a mappatura diretta per la seguente sequenza ordinata di indirizzi di blocco 0, 8, 0, 6, 8.

## 1.14 Designing the Memory System

La penalità di errore può essere ridotta aumentando la larghezza di banda del bus tra DRAM e cache. Ci sono tre possibili organizzazioni:

- **Semplice:** una parola alla volta viene letta dalla memoria.
- **Ampia memoria:** N parole alla volta vengono lette dalla memoria.
- **Interleaved:** K banchi di memoria indipendenti in grado di servire K richieste contemporaneamente.

Consideriamo il tempo di trasferimento del blocco della cache per l'organizzazione della memoria Simplevs Interleaved.

## 1.15 Problemi cache

La memorizzazione nella cache è essenziale per ridurre il collo di bottiglia di von Neuman e ottenere prestazioni ragionevoli sui sistemi moderni. Tuttavia, la memorizzazione nella cache introduce alcuni problemi nei sistemi multiprocessore/multicore. Problema di coerenza della cache, senza condivisione degrado delle prestazioni nei multiprocessori coerenti con la cache, due variabili non correlate sono collocate nello stesso blocco della cache e accesso in modalità lettura/scrittura da thread diversi!

### 1.15.1 Problemi di coerenza

Supponiamo un'architettura SMP: SMP: Symmetric Multiprocessors, caching di dati privati e condivisi, i dati core privati vengono memorizzati nella cache in L1, riducendo così l'AMAT e le comunicazioni di memoria off-chip. Quando i dati condivisi vengono memorizzati nella cache, i valori condivisi può essere replicato in più core cache private. Questo riduce anche la contesa di memoria! Cosa succede se i dati condivisi vengono scritti? La memorizzazione nella cache dei dati condivisi introduce un nuovo problema: **la coerenza della cache**.

### 1.15.2 Write invalidate protocol

Il protocollo di invalidazione della scrittura invalida le copie dei dati presenti in altre cache durante un'operazione di scrittura. Una semplice implementazione utilizza un protocollo bus di snooping per garantire che il processore acquisisca l'accesso esclusivo a un blocco di cache prima che scriva in esso.

## 2 Input output

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

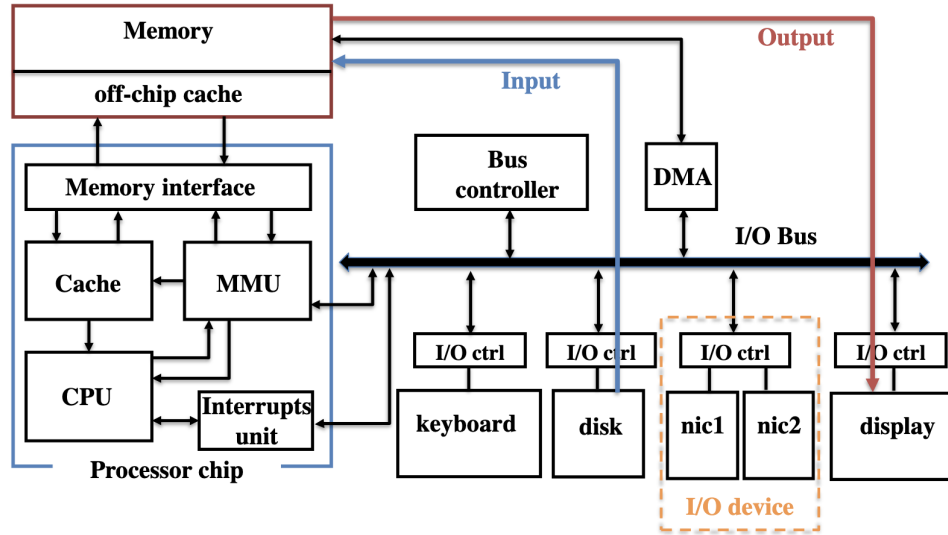


Figure 7: Input/output e I/O devices

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 copri che l'aumento di velocità uguale a:

$$\frac{T_{exe} \text{ before enhancement}}{T_{exe} \text{ after enhancement}} = \frac{11}{5} = 5.5$$

### 2.1 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).

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

**Osservazione 2.1.1.** 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.

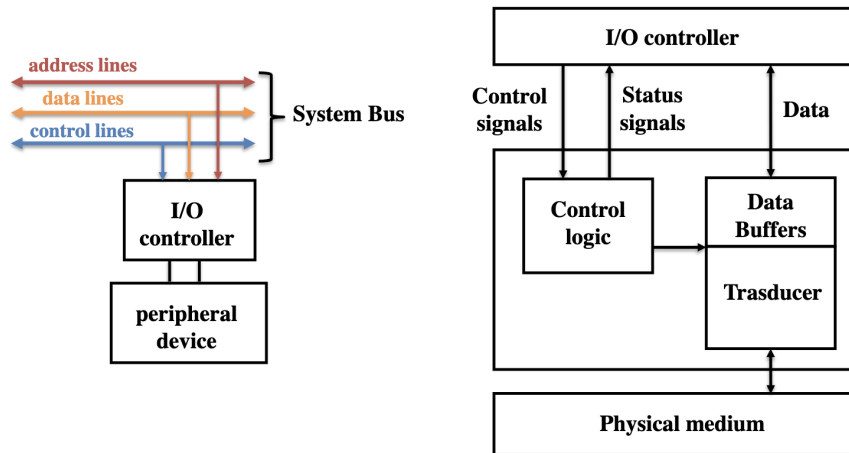


Figure 8: I/O device generico

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 ottimizzare il trasferimento dei dati e compensare le differenze di velocità).

## 2.2 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'interconnessione 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.

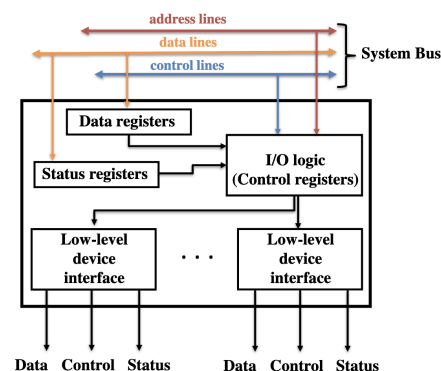


Figure 9: I/O device generico

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.

### 2.2.1 Bus design

Il design di un bus si basa su soddisfare gli obiettivi di avere un alta performance, standardizzazione e bassi costi. Per esempio il bus di sistema enfatizza le prestazioni, l'I/O e i bus backplane 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 risposta è tramite due metodologie **Atomic transactions** che permette una bassa utilizzazione ed una bassa complessità, 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 un clock (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.  $\Delta$  è il tempo massimo per scrivere i dati sul bus da parte di un'unità collegata. Potenzialmente 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 possibile funziona 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 l'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 decide 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 protocollo 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 protocollo è che si va a 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 implementare 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 di 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).

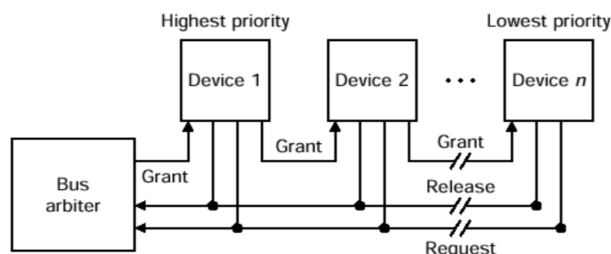


Figure 10: Daisy chain arbitrain

## 2.3 Gestione dell I/O

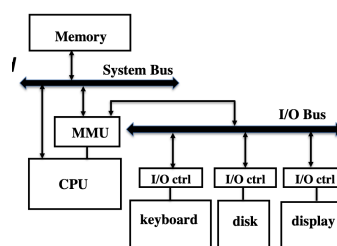
Quando andiamo a parlare di gestione dei dispositivi di input output ci chiediamo come la CPU dà i comandi ai dispositivi di I/O? Come la CPU sa quando i dispositivi di I/O completano le operazioni? Come fanno i dispositivi di I/O ad eseguire i data transfers?

Partiamo dal rispondere alla prima domanda. Inanzitutto soltanto il sistema operativo può mandare comandi ai dispositivi 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 ai dispositivi di I/O:

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

## 2.4 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.



## 2.5 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 completano le operazioni? Introduciamo due concetti che possano risolvere questo problema.

- 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 chiamate 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 il dispositivo I/O ha un ruolo passivo e la CPU attende che il modulo I/O completi le operazioni (polling).

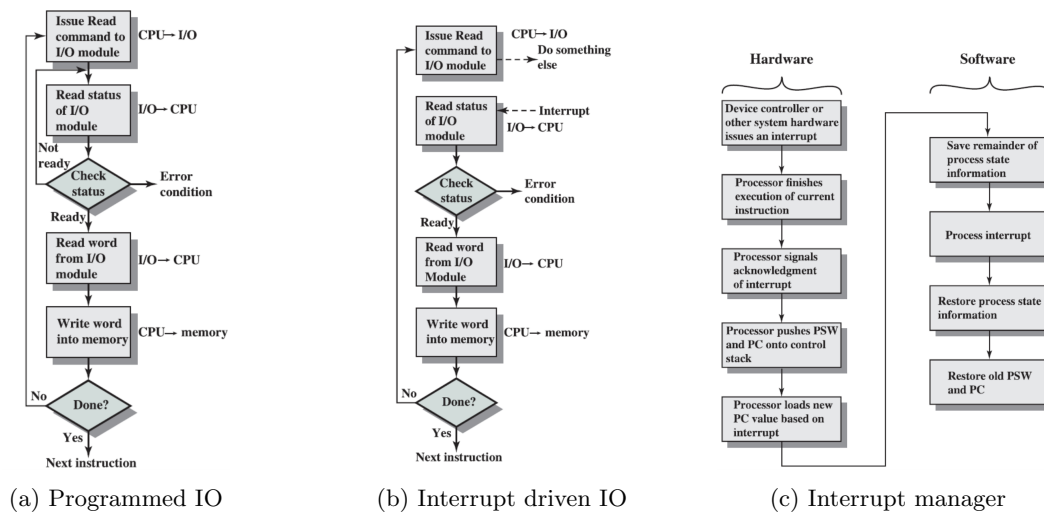
Nel pulling il processore interroga il registro di stato del dispositivo I/O. Esso ha come pro una facile implementazione mentre come contro una perdita di cicli del processore perchè la CPU è molto più veloce che i dispositivi di I/O. La cpu deve leggere lo stato dei registri molte volte solo per scoprire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

## 2.6 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'interrupt 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,<sup>1</sup> esegui poi il gestore (ovvero, il sistema operativo gestisce l'interrupt) e ripristina il contesto per l'esecuzione dell'istruzione successiva:<sup>2</sup>



**Esempio 2.6.1.** 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  $(30 \text{ poll/s}) * 400 = 120000 \text{ cycles/s}$ , e la percentuale di cicli spesi per polling si calcola faccenda  $12K/500M = 0.002\%$ . Abbiamo dunque un overhead accettabile.

<sup>1</sup>I passaggi precedenti sono atomici dal punto di vista del sistema operativo!

<sup>2</sup>il ripristino è un'operazione atomica per il sistema operativo!



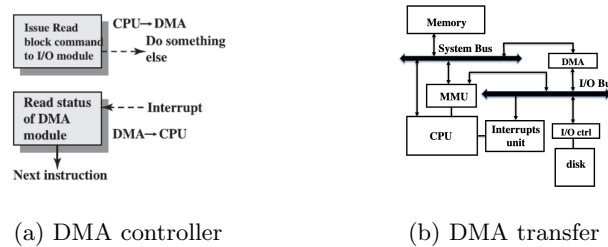
Prendiamo ora un disco (che è un dispositivo ad alta lunghezza di banda) che ha 4MB/s di transfer rate e 16B interface.

Per non mancare i dati dobbiamo eseguire il poll abbastanza spesso, quindi  $(4\text{MB/s})/16\text{B} = 250\text{ K/s}$ , vista come percentuale di cicli abbiamo  $100\text{M} / 500\text{ M} = 20\%$ , questo risultato non è accettabile perché Abbiamo speso il 20% dei cicli solo per controllare i registri I/O, non per il trasferimento dei dati

## 2.7 Data transfers (DMA)

Andiamo ora a rispondere all'ultima domanda che ci eravamo posti, come i device di I/O eseguono 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 problema introduciamo la direct memory access (DMA).

Esso è meccanismo 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 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 2.7.1.** Facciamo un esempio, assumiamo i seguenti dati: 500Mhz CPU, il device di un disco ha un tempo di trasferimento 4MB/s, 16B di interfaccia 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  $0.5 * (4\text{MB/s}) / (16\text{B}) * [(400 + 100) \text{ cicli} / 500\text{M cicli/s}] = 12.5\%$ .

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  $0.5 * (4\text{M B/s}) / (16\text{KB}) * [(1600 + 400) / (500\text{M cicli/s})] = 0.05\%$

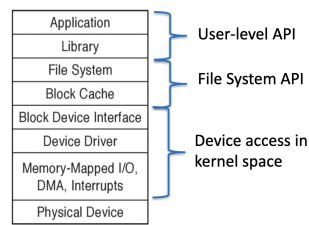
Notiamo da questo esempio che Senza DMA: il processore avvia tutti i trasferimenti di dati, tutti i trasferimenti passano attraverso la traduzione degli indirizzi (MMU), i trasferimenti possono essere di qualsiasi dimensione e attraversare i limiti della pagina virtuale, nessun impatto sulla gerarchia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

cache, le cache non contengono mai dati obsoleti.

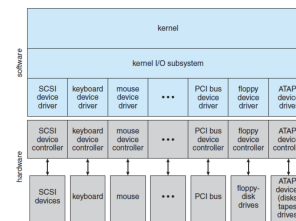
Mentre con DMA: il controller DMA avvia i trasferimenti di dati, esiste un altro percorso per il sistema di memoria, potenziali problemi: i controller DMA utilizzano indirizzi virtuali o fisici? Cosa succede se scrivono i dati in una posizione di memoria cache?

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.



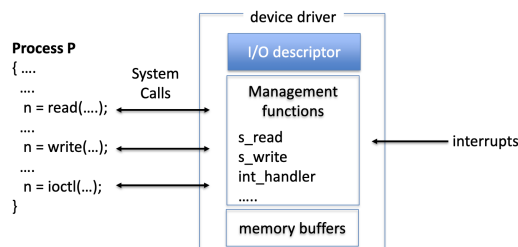
(a) Device access



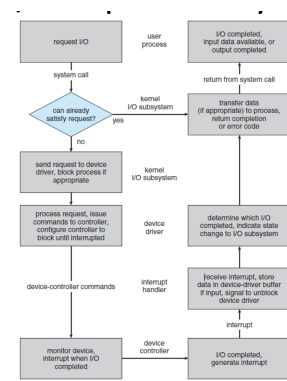
(b) Kernel IO structure

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 ai 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.



(a) Device driver



(b) IO request life cycle

### 3 Dischi rigidi

Alcuni tipi di storage device sono per esempio i dischi magnetici e le flash memory.

**Definizione 3.0.1** (Dischi magnetici). *I dischi magnetici è un tipo di archiviazione che raramente viene danneggiata. Grande capacità a basso costo. Accesso casuale a livello di blocco. Prestazioni lente per l'accesso casuale. Migliori prestazioni per l'accesso in streaming.*

**Definizione 3.0.2** (Flash memory). *Archiviazione che raramente viene danneggiata. Buona capacità a costo intermedio ( $2 \times$  disco). Accesso casuale a livello di blocco. Buone prestazioni per le letture; peggio per le scritture casuali.*

#### 3.1 Dischi magnetici

I dischi magnetici hanno circa 1 micron di larghezza, lunghezza d'onda della luce è di circa 0.5 micron, risoluzione dell'occhio umano: 50 micron, 100K su un tipico disco da 2,5 pollici. Separato da regioni di guardia inutilizzate, riduce la probabilità che le tracce vicine vengano danneggiate durante le scritture (ancora una piccola possibilità diversa da zero).

La lunghezza della traccia varia a seconda del disco, all'esterno: più settori per traccia, maggiore larghezza di banda, il disco è organizzato in regioni di tracce con lo stesso numero di settori/traccia, viene utilizzata solo la metà esterna del raggio. La maggior parte dell'area del disco nelle regioni esterne del disco

##### 3.1.1 Settore

I settori contengono sofisticati codici di correzione degli errori, il magnete della testina del disco ha un campo più ampio della traccia, nasconde le corruzioni dovute alle scritture di tracce vicine

- **Sector sparing.** Rimappa i settori danneggiati in modo trasparente ai settori di riserva sulla stessa superficie.
- **Slip sparing.** Rimappa tutti i settori (quando c'è un settore danneggiato) per preservare il comportamento sequenziale.
- **Track skewing.** Numeri di settore spostati da una traccia all'altra, per consentire il movimento della testina del disco per operazioni sequenziali.

A basso livello il controller accede ai singoli settori. La dimensione tipica del settore è di 256/512 byte. Identificato da una tripla:  $\langle \#cilindro, \#faccia, \#setto$

Al livello superiore, il driver del disco raggruppa un insieme di settori contigui in un blocco. La dimensione tipica del blocco è di  $2/4/8$  Kbyte, identificata da un puntatore su uno spazio di indirizzi contiguo (da 0 a max-blocks).

Dato un numero di sector  $b$ , ed una tripla  $\langle c, f, s \rangle$  abbiamo che.

$$b = c(\#faces \cdot \#sectors) + f(\#sectors) + s$$

$\#faces$  sono il numero di facce in un disco mentre  $\#sectors$  sono il numero di settori per track. Di conseguenza a questa formula abbiamo che:

$$c = b \operatorname{div} (\#faces \cdot \#sectors) \quad f = (b \operatorname{div} (\#faces \cdot \#sectors)) \operatorname{div} \#sectors$$

$$s = (b \operatorname{div} (\#faces \cdot \#sectors)) \operatorname{mod} \#sectors$$

**Esempio 3.1.1.** Disco con 100 cilindri, 4 facce, 2000 settori per traccia. Un file utilizza i blocchi 94421, 94422, 94423. Qual è il  $\langle c, f, s \rangle$  per ogni blocco? Possiamo calcolare facendo:

$c = 94421 \operatorname{div} (4 \cdot 2000) = 1$  (rimangono 6421).  $f = 6421 \operatorname{div} 2000 = 3$  (rimangono 421) quindi  $s = 421$ . Pertanto le triple sono:  $\langle 1, 2, 421 \rangle, \langle 1, 3, 422 \rangle, \langle 1, 3, 423 \rangle$ .

### 3.1.2 Disk performance

Fra i punti principali nel calcolo delle performance di un disco è la sua latenza. Per calcolare la disk latency:

$$\text{Disk latency} = \text{Seek time} + \text{Rotation time} + \text{Transfer Time}$$

In questa formula possiamo notare 3 componenti:

- **Seek time.** Tempo per spostare il braccio del disco sulla traccia (1-20ms). Regolazione fine della posizione necessaria affinché la testina si "assesti". Tempo di cambio testina tempo di cambio traccia (su dischi moderni).
- **Rotation time.** Tempo di attesa per la rotazione del disco sotto la testina del disco Rotazione del disco: 4, 15 ms (a seconda del prezzo del disco).
- **Transfer time.** Tempo di trasferimento dei dati su/fuori dal disco Velocità di trasferimento testina disco: 50-100 MB/s (5-10 usec/settore). Velocità di trasferimento host dipendente dal connettore I/O (USB, SATA, ...).

**Esempio 3.1.2.** Quanto ci vuole per completare 500 letture casuali del disco, in ordine FIFO? Ricerca: media 10,5 msec. Rotazione: media 4,15 msec. Trasferimento: 5-10 usec.  $500 * (10,5 + 4,15 + 0,01)/1000 = 7,3$  secondi.

**Esempio 3.1.3.** Quanto tempo occorre per completare 500 letture sequenziali del disco? Tempo di ricerca: 10,5 ms (per raggiungere il primo settore). Tempo di rotazione: 4,15 ms (per raggiungere il primo settore). Tempo di trasferimento: (traccia esterna)  $500 \text{ settori} * 512 \text{ byte} / 128 \text{ MB/sec} = 2 \text{ ms}$  Totale:  $10,5 + 4,15 + 2 = 16,7 \text{ ms}$ .

Potrebbe essere necessaria una testina aggiuntiva o un interruttore di traccia (+1ms). Il buffer di traccia può consentire la lettura fuori servizio di alcuni settori dal disco (-2ms)

**Esempio 3.1.4.** Disco con 100 cilindri, 4 facce, 2000 settori per traccia. Un file usa i blocchi 94421, 94422, 94423 ( $< 11, 3, 421 > < 11, 3, 422 > < 11, 3, 423 >$ ).

Un settore viene letto in 0,01 ms, il tempo di ricerca tra due tracce consecutive è di 0,01 ms e il tempo medio per raggiungere il settore desiderato è la metà del tempo di rotazione.

Considerando che il braccio è al cilindro 22 e che il controller DMA ha abbastanza buffer, calcola il tempo per leggere i blocchi di file.  $(22-11)*0.01 + 20/2 + 3*0.01 = 10.14\text{ms}$

## 3.2 Dishi SSD

Un Solid StateDrive (SSD) è un dispositivo di archiviazione non volatile basato sulla tecnologia di memoria flash. Gli SSD sono più affidabili e più veloci dei dischi rigidi (HD) perché non hanno parti mobili e nessun tempo di ricerca, gli SSD usano comunemente una semplice pianificazione FCFS policy. Consumano meno energia e sono più costosi per MB di dati. La capacità dell'HD è in genere maggiore. In alcuni sistemi, gli SSD vengono utilizzati come ulteriore livello di cache nella gerarchia della memoria.

All'interno delle flash memory abbiamo che le scritture devono essere per "pulire" le celle; nessun aggiornamento in atto. Cancellazione di blocchi di grandi dimensioni richiesta prima della scrittura. Blocco di cancellazione fra 128 ed i 512 KB. Tempo di cancellazione: diversi millisecondi, scrittura/lettura pagina (2-4 KB) in circa 50-100 usec.

### 3.2.1 Flash translation layer

Per evitare il costo di una cancellazione per ogni scrittura, le pagine vengono cancellate in anticipo. Pagine pulite sempre disponibili per una nuova scrittura, questo significa che una scrittura non può essere indirizzata ad una pagina arbitraria in memoria, ma solo ad una pagina precedentemente cancellata, cosa succede se si riscrive un blocco di un file? La pagina che contiene il blocco non può da riscrivere subito... Deve essere prima cancellato ma con le pagine circostanti! Viene utilizzata una pagina pulita per applicare la scrittura, ma questa pagina è da qualche parte nel disco... La vecchia pagina va nella spazzatura per il riciclaggio. Come sapere dove sono le pagine del mio file?

Il firmware del dispositivo flash associa la pagina logica # a una posizione fisica: sposta le pagine

live secondo necessità per la cancellazione. Garbage collect blocco di cancellazione vuoto copiando le pagine live in una nuova posizione, abbiamo dunque livellamento dell'usura. Puoi scrivere ogni pagina fisica solo un numero limitato di volte: evita le pagine che non funzionano più. Trasparente per l'utente del dispositivo.

### 3.2.2 File system flash

I file system sui dischi magnetici non hanno bisogno di dire al disco quali blocchi sono in uso: quando un blocco non è più utilizzato viene contrassegnato come libero nella bitmap, il file system lo riutilizza quando vuole. Quando questi FS sono stati utilizzati per la prima volta su unità flash, le prestazioni sono diminuite drasticamente nel tempo.

Il Flash Translation Layer si è dato da fare con la garbage collection: i blocchi live devono essere rimappati in una nuova posizione, per compattare le pagine libere per poter procedere con la cancellazione dei blocchi. Questo anche con una grande quantità di spazio libero, ad esempio, se FS sposta un file di grandi dimensioni da un intervallo di blocchi a un altro, lo storage non sa che i vecchi blocchi possono andare nella spazzatura, a meno che FS non lo dica!

Comando TRIM: il file system indica al dispositivo quando le pagine non sono più in uso. Aiuta il livello di traduzione dei file a ottimizzare la raccolta dei rifiuti. Introdotto tra il 2009 e il 2011 nella maggior parte dei sistemi operativi.