

CALCOLATORI

La gerarchia di memoria

Giovanni Iacca
giovanni.iacca@unitn.it

*Lezione basata su materiale preparato
con i Prof. Luigi Palopoli e Marco Roveri*

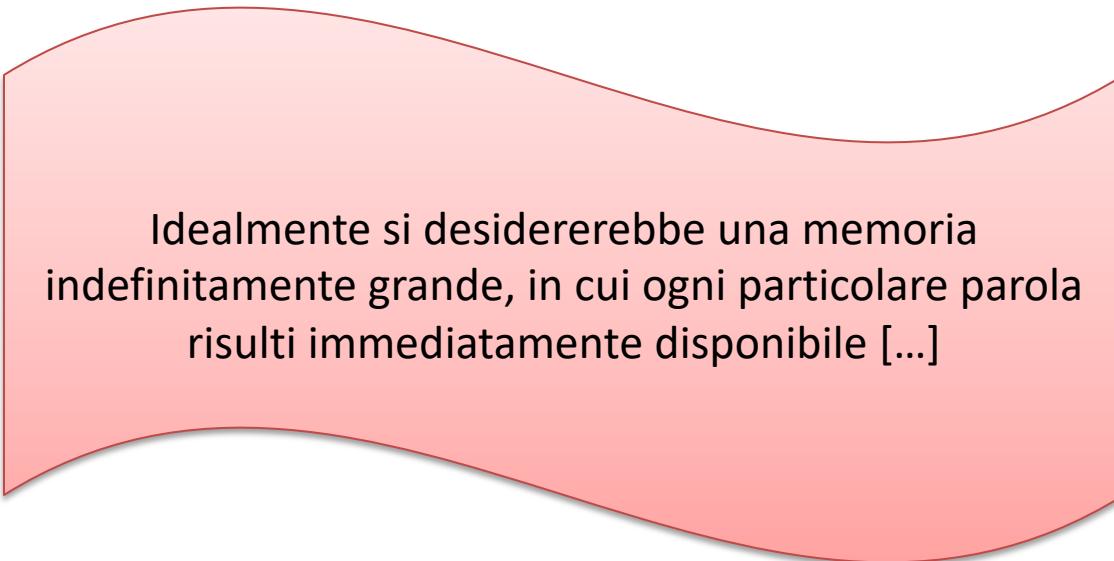


UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Gerarchia di memoria

- Abbiamo visto i vari blocchi di memoria con diverse caratteristiche di velocità e capacità
- Ma ritorniamo a come ottenere velocità e capacità insieme
- Il problema è noto da molto tempo....



Idealmente si desidererebbe una memoria
indefinitamente grande, in cui ogni particolare parola
risulti immediatamente disponibile [...]

Burks, Goldstine, Von Neumann, 1946

Gestione “piatta”

- Immaginiamo di essere un impiegato che lavora al comune
- Per effettuare il mio lavoro ho bisogno di avere accesso ad un archivio dove sono presenti le varie pratiche
- Ogni volta che mi serve una pratica vado a prenderla, ci opero su, e poi la rimetto a posto

Gestione “piatta”

- Per accedere alla pratica scrivo su un bigliettino lo scaffale dove la pratica può essere trovata, lo affido a un attendente, e aspetto che me la porti
- Osservazioni:
 - la mia capacità di memorizzazione è molto grande
 - la gran parte del mio tempo (direi il 90%) la spengo aspettando che l'attendente vada a prendere le pratiche
 - Sicuramente non è una gestione efficiente del mio tempo
- **Posso essere più veloce?**

Gestione “veloce”

- In alternativa posso tenere le pratiche sul mio tavolo e operare solo su quelle
- Osservazioni
 - Sicuramente non perdo tempo (non ho da aspettare attendenti che vadano in su e in giù)
 - Tuttavia il numero massimo di pratiche che possono gestite è molto basso
- **Posso operare su più dati?**

Capre e cavoli

- Per riuscire ad avere al tempo stesso velocità e ampiezza dell'archivio, posso fare due osservazioni:
 1. Il numero di pratiche su cui posso concretamente lavorare in ogni giornata è limitato
 2. Se uso una pratica, quasi sicuramente dovrò ritornare su di essa in tempi brevi... tanto vale tenersela sul tavolo

Un approccio gerarchico

- L'idea è che ho un certo numero di posizioni sulla mia scrivania
 - Man mano che mi serve una pratica la mando a prendere
 - Se ho la scrivania piena faccio portare a posto quelle pratiche che non mi servono più per fare spazio
- In questo modo posso contare su un'ampia capacità di memorizzazione, *ma* la maggior parte delle volte accedo ai dati molto velocemente

Torniamo ai calcolatori

- Fuori di metafora, all'interno di un calcolatore possiamo dare al processore (e ai programmi) l'illusione di avere uno spazio di memoria molto grande ma con grande velocità
- Questo è possibile grazie a due principi
 - Principio di località spaziale
 - Principio di località temporale

Località temporale

- Quando si fa uso di una locazione, la si riutilizzerà presto con elevata probabilità
- Esempio.

```
Ciclo: slli x10, x22, 3  
       add x10, x10, x25  
       ld  x9, 0(x10)  
       bne x9, x24, Esci  
       addi x22, x22, 1  
       beq x0, x0, Ciclo
```

Esci:

Queste istruzioni vengono ricaricate ogni volta che si esegue il ciclo

Località spaziale

- Quando si fa riferimento a una locazione, nei passi successivi si farà riferimento a locazioni vicine

Ciclo:

```
slli x10, x22, 3  
add x10, x10, x25  
ld x9, 0(x10)  
bne x9, x24, Esci  
addi x22, x22, 1  
beq x0, x0, Ciclo
```

Esci: ...

ISTRUZIONI: la modalità di esecuzione normale è il prelievo di istruzioni successive

DATI: Quando si scorre un array si va per word successive

Qualche dato

Facciamo riferimento a qualche cifra (relativa al 2012)

Tecnologia di Memoria	Tempo di accesso tipico	\$ per GB (2010)	\$ per GB (2012)
SRAM	0.5-2.5 ns	\$2000 - \$5000	\$500 - \$1000
DRAM	50 – 70 ns	\$20 - \$75	\$10 - \$20
Memoria flash	70 – 150 ns	\$4 - \$12	\$0.75 - \$1
Dischi magnetici	5 000 000 – 20 000 000 ns	\$0.2 - \$2	\$0.05 - \$0.1

Queste cifre suggeriscono l'idea della gerarchia di memoria

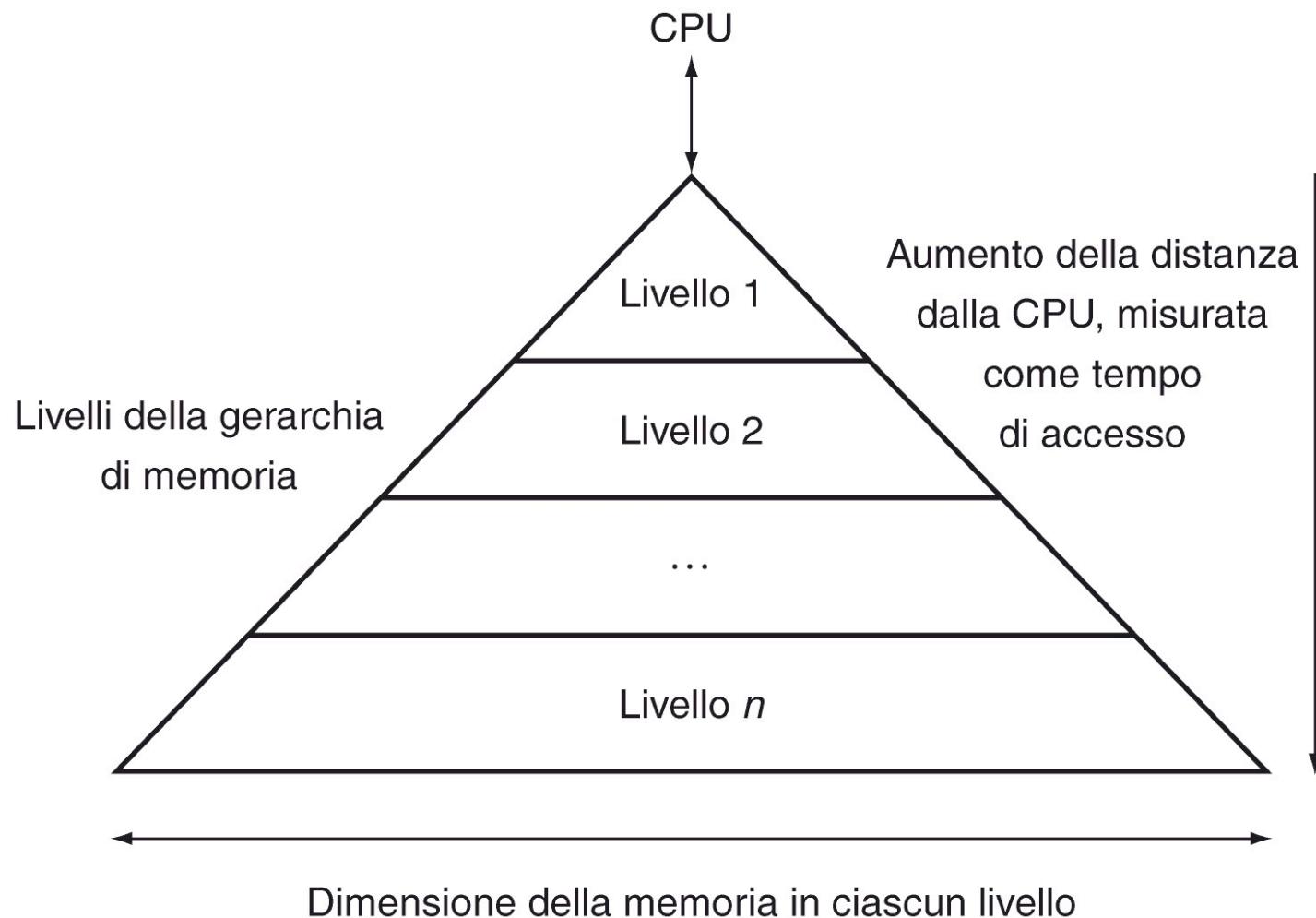
- Memorie piccole e veloci vicino al processore
- Memorie grandi e lente lontane dal processore

Gerarchia di memoria

- Struttura base

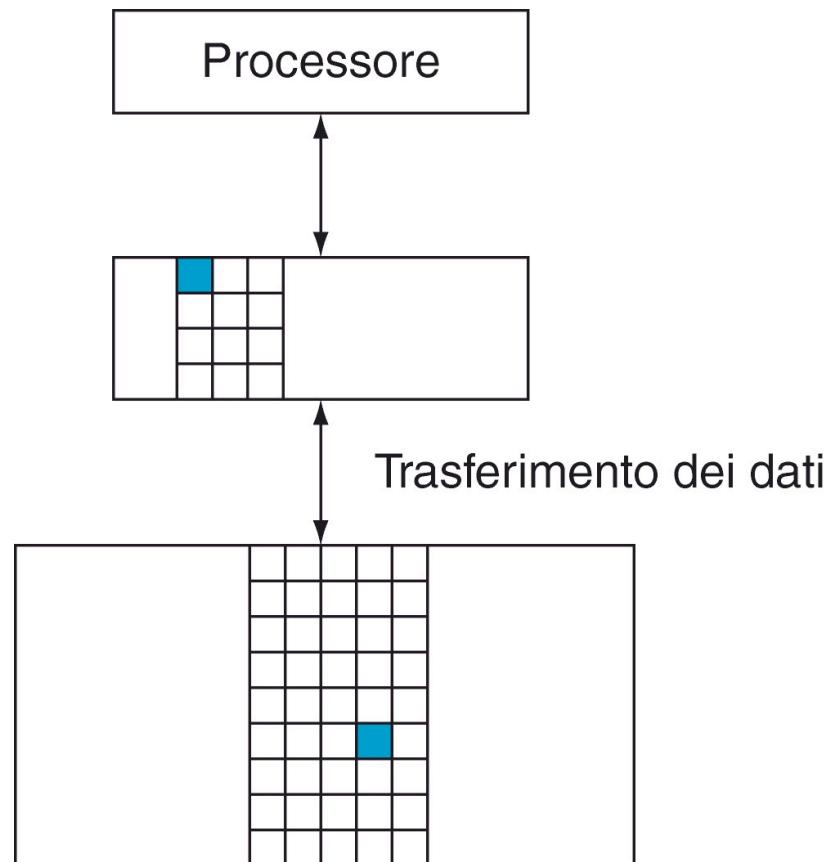
Velocità	Processore	Dimensione	Costo (\$/bit)	Tecnologia corrente
Più veloce	Memoria	Più piccola	Più elevato	SRAM
	Memoria			DRAM
Più lenta	Memoria	Più grande	Più basso	Disco magnetico

Struttura della gerarchia



Esempio

- Due livelli



Terminologia

- **Blocco:** unità minima di informazione che può essere presente o assente in ciascun livello
 - Un faldone nell'esempio di un archivio
- **Hit rate:** Frequenza di successo = frazione degli accessi in cui trovo il dato nel livello superiore
 - Quante volte trovo il faldone che mi serve nella scrivania
- **Miss Rate:** $1.0 - \text{Hit rate}$: frazione degli accessi in cui non trovo il dato nel livello superiore
 - Quante volte devo andare a cercare un faldone in archivio
- **Tempo di hit:** Tempo che occorre per accedere al dato quando lo trovo nel livello superiore
 - Quanto mi ci vuole a leggere un documento nel faldone sulla scrivania
- **Penalità di miss:** quanto tempo mi ci vuole per accedere al dato se non lo trovo nel livello superiore
 - Tempo per spostare il faldone dall'archivio alla scrivania + tempo di accesso al documento nel faldone (dopo che arriva sulla scrivania)

Considerazioni

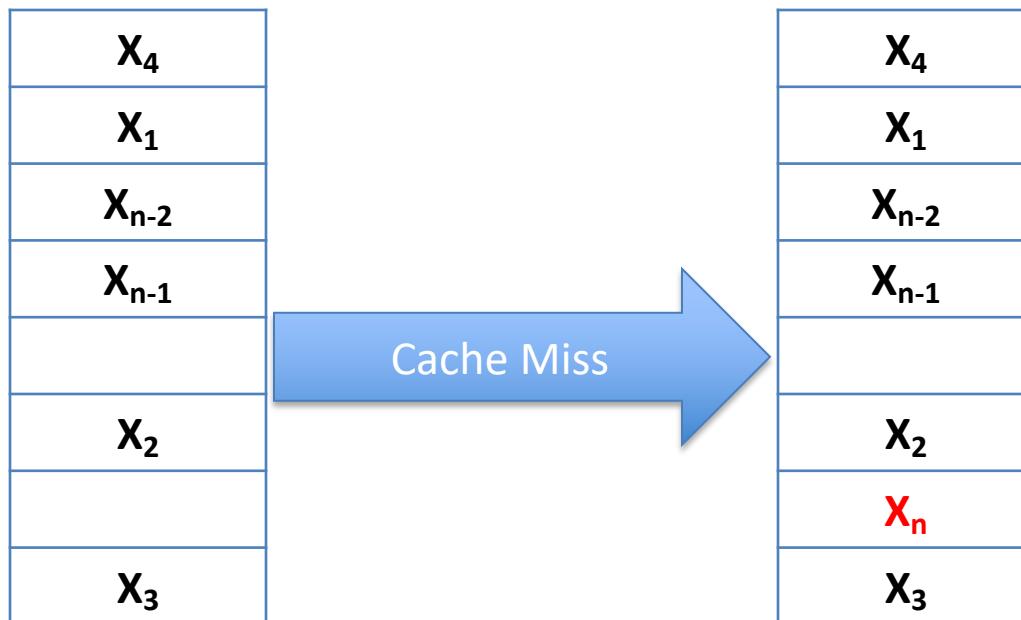
- La penalità di miss è molto maggiore del tempo di hit (e anche del trasferimento in memoria di un singolo dato)
 - Da cui il vantaggio
- Quindi è importante ridurre frequenza di miss
 - In questo ci aiuta il principio di località che il programmatore deve sfruttare al meglio

Cache

- **Cache:** posto sicuro [nascosto] dove riporre le cose
- Nascosto perché il programmatore non vede la cache direttamente (non vi accede)
 - L'uso della cache è interamente trasparente
- L'uso della cache fu sperimentato per la prima volta negli anni '70 e da allora è stato largamente adottato in tutti i calcolatori

Un esempio semplice

- Partiamo da un semplice esempio in cui i blocchi di cache siano costituiti da una sola word
- Supponiamo che a un certo punto il processore richieda la parola X_n che non è in cache

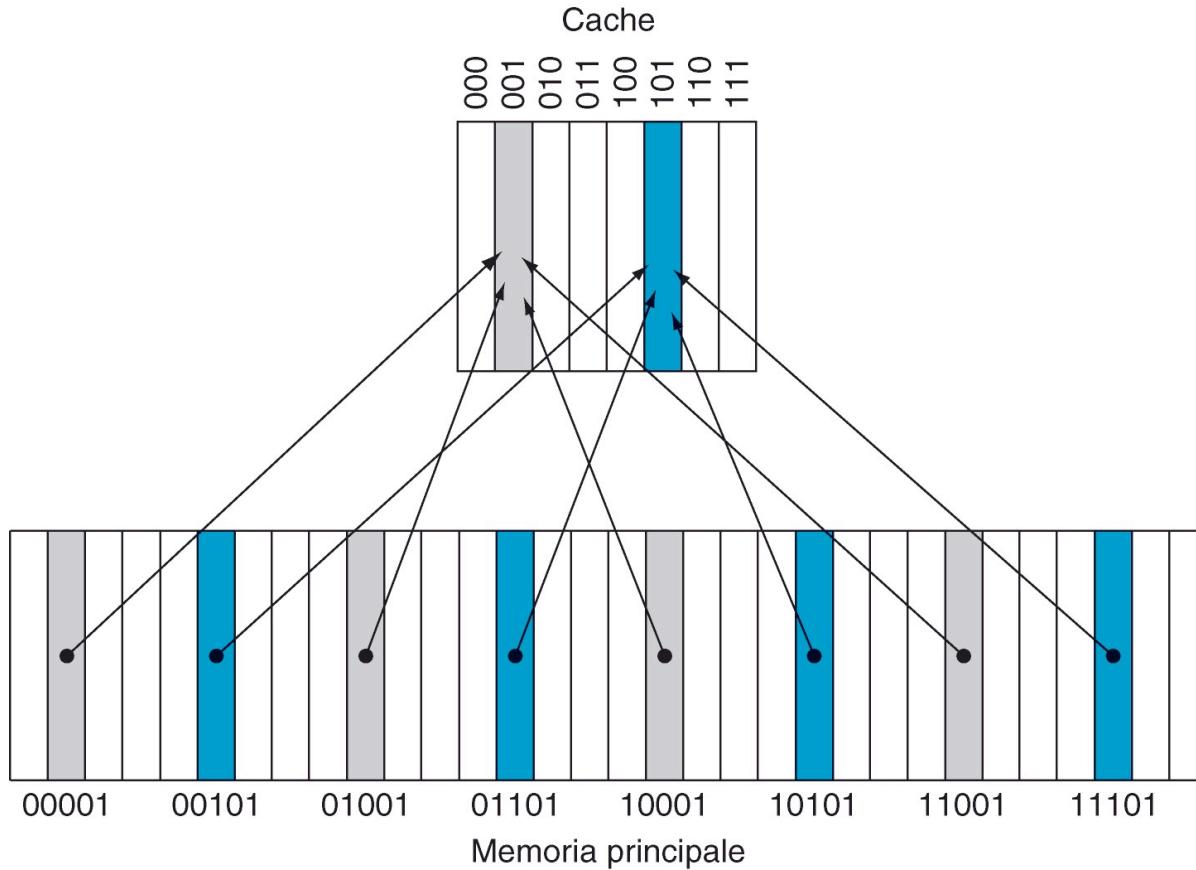


Domande

- Come facciamo a capire se un dato richiesto è nella cache?
- Dove andiamo a cercare per sapere se c'è?
- *Cache a mappatura diretta*: a ogni indirizzo della memoria corrisponde una precisa locazione della cache
- Possibilità: indirizzo locazione dove un indirizzo è mappato = (indirizzo blocco) *modulo* (numero di blocchi in cache)
 - Se la dimensione (=numero di blocchi) della cache è potenza di due, è sufficiente prendere i bit meno significativi dell'indirizzo in numero pari al logaritmo in base due della dimensione della cache

Esempio

- Se la nostra cache dispone di 8 parole devo prendere i tre bit meno significativi



Problema

- Siccome molte parole posso essere mappate sullo stesso blocco di cache, come facciamo a capire se, in un dato momento, vi si trova l'indirizzo che serve a noi?
- *Si ricorre a un campo, detto tag, che contiene un'informazione sufficiente a risalire al blocco correntemente mappato in memoria*
- *Ad esempio possiamo utilizzare i bit più significativi di una parola, che non sono usati per la mappatura sulla cache, per trovare la locazione di memoria corrispondente all'indirizzo mappato da quel blocco di cache.*

Validità

- Usiamo i bit più significativi (due nell'esempio fatto) per capire se nel blocco di cache memorizziamo l'indirizzo richiesto
- Inoltre abbiamo un bit di validità che ci dice se quello che memorizziamo in un blocco di cache in un certo momento sia o meno valido

Esempio

Indice	V	Tag	Dati
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

- a. Lo stato iniziale della cache dopo l'accensione del calcolatore

Indice	V	Tag	Dati
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	S	10 _{due}	Memoria (10110 _{due})
111	N		

- b. Dopo avere gestito una miss all'indirizzo (10110_{due})

Indice	V	Tag	Dati
000	N		
001	N		
010	S	11 _{due}	Memoria (11010 _{due})
011	N		
100	N		
101	N		
110	S	10 _{due}	Memoria (10110 _{due})
111	N		

- c. Dopo avere gestito una miss all'indirizzo 11010_{due}

Indice	V	Tag	Dati
000	S	10 _{due}	Memoria (10000 _{due})
001	N		
010	S	11 _{due}	Memoria (11010 _{due})
011	N		
100	N		
101	N		
110	S	10 _{due}	Memoria (10110 _{due})
111	N		

- d. Dopo avere gestito una miss all'indirizzo 10000_{due}

Indice	V	Tag	Dati
000	S	10 _{due}	Memoria (10000 _{due})
001	N		
010	S	11 _{due}	Memoria (11010 _{due})
011	S	00 _{due}	Memoria (00011 _{due})
100	N		
101	N		
110	S	10 _{due}	Memoria (10110 _{due})
111	N		

- e. Dopo avere gestito una miss all'indirizzo 00011_{due}

Indice	V	Tag	Dati
000	S	10 _{due}	Memoria (10000 _{due})
001	N		
010	S	10 _{due}	Memoria (10010 _{due})
011	S	00 _{due}	Memoria (00011 _{due})
100	N		
101	N		
110	S	10 _{due}	Memoria (10110 _{due})
111	N		

- f. Dopo avere gestito una miss all'indirizzo 10010_{due}

Accesso a
10110:
Miss

Accesso a
11010:
Miss

Accesso a
11010: hit

Accesso a
10010:
miss

Esempio RISC-V a 64 bit

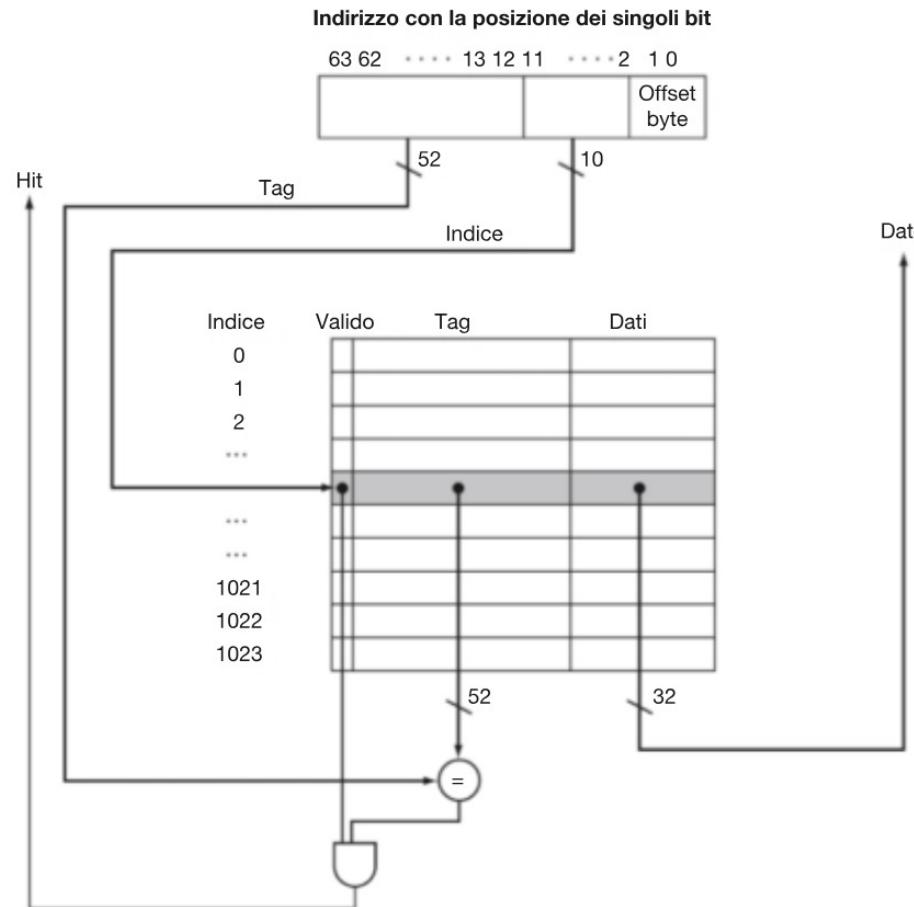
- Esempio
 - Indirizzo su 64 bit
 - Cache a mappatura diretta
 - Dimensioni della cache: 2^n blocchi, di cui n bit usati per l'indice
 - Dimensione del blocco di cache: 2^m parole, ossia 2^{m+2} byte (m bit usati per individuare una parola nel blocco), due bit per individuare un byte in una parola

In questo caso la dimensione del tag è data da:

$$64-(n+m+2)$$

Schema di risoluzione

- Un indirizzo viene risolto in cache con il seguente schema ($n=10$, $m=0$, dimensione tag = 52):



Se il campo tag è uguale ai 52 bit più significative dell'indirizzo e se il bit di validità è 1, allora si ha una hit e si da il dato al processore, altrimenti scatta una miss.

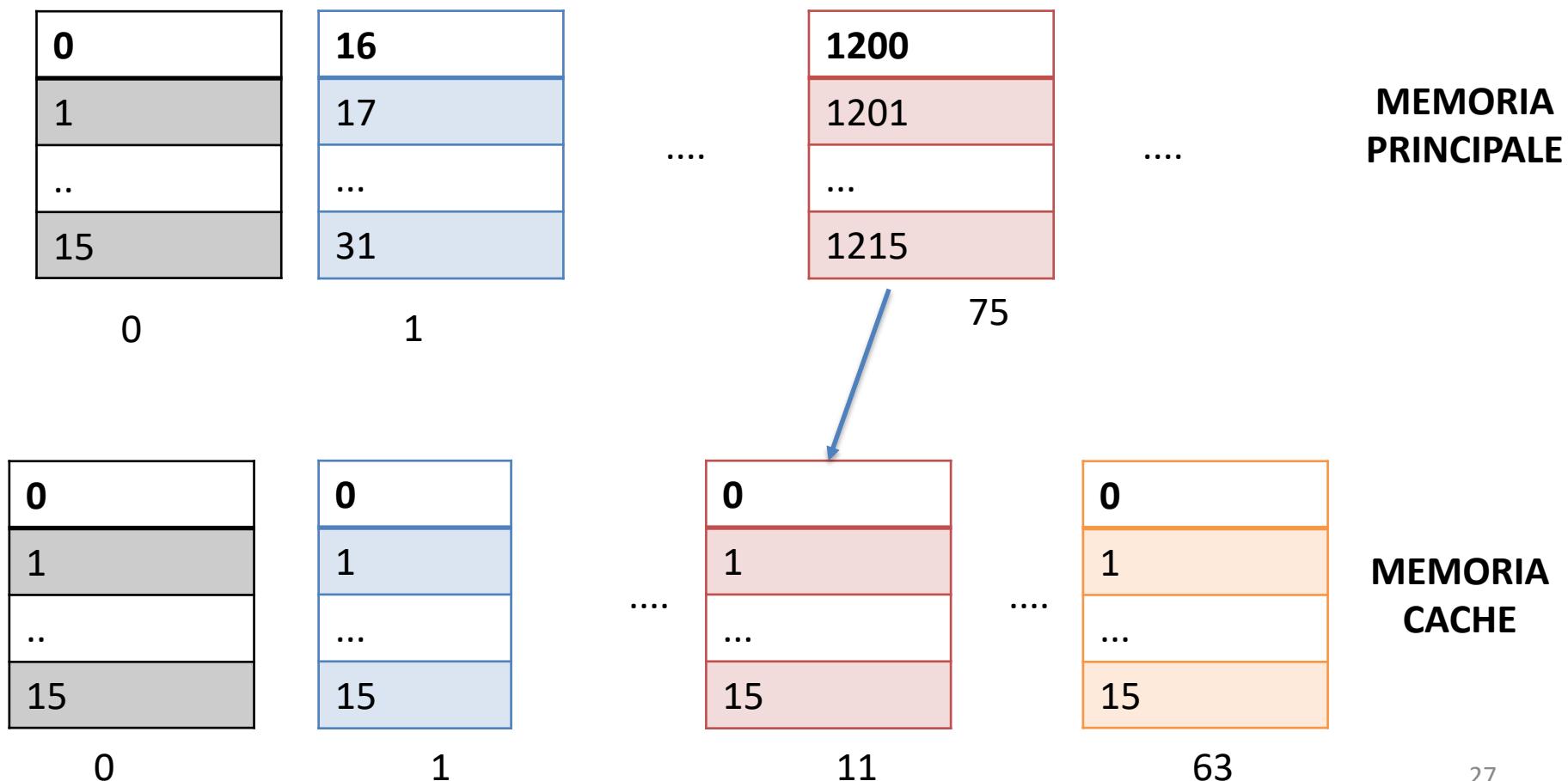
Esempio

- Si consideri una cache con 64 blocchi di 16 byte ciascuno. A quale numero di blocco corrisponde l'indirizzo 1200 espresso in byte?
- Blocco identificato da:
(indirizzo blocco) *modulo* (numero blocchi in cache)
- Dove:

$$\text{Indirizzo Blocco} = \frac{\text{Indirizzo del Dato in byte}}{\text{Byte per blocco}}$$

Esempio

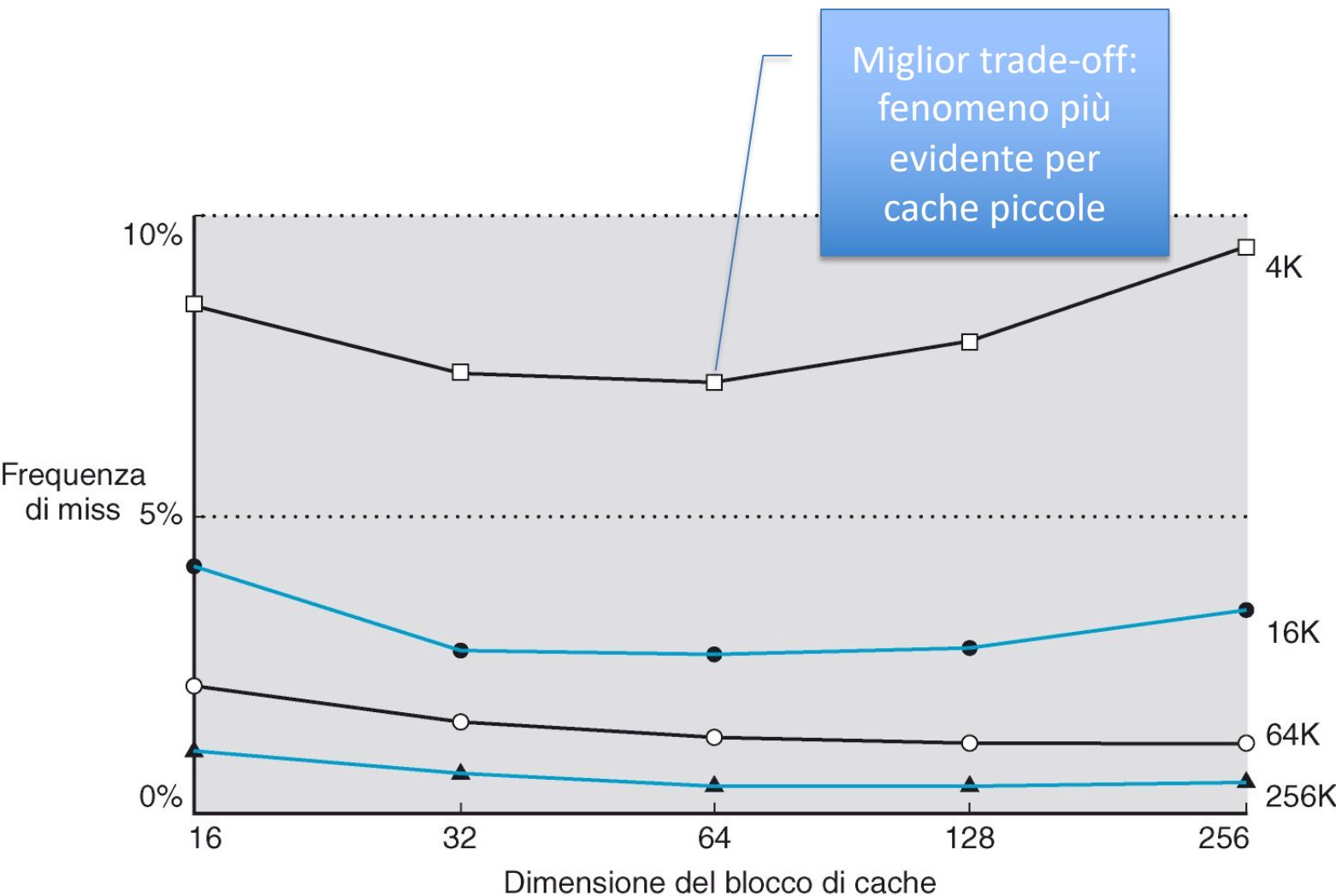
- Quindi l'indirizzo del blocco è $1200/16=75$
- Blocco contenente il dato è: $75 \text{ modulo } 64 = 11$



Trade-off

- Blocchi di cache molto grandi esaltano la località spaziale e da questo punto di vista diminuiscono le probabilità di miss
- Tuttavia, a parità di dimensioni della cache avere pochi blocchi diminuisce l'efficacia nello sfruttamento della località temporale
- Quindi abbiamo un trade-off
- Inoltre avere dei *miss* con blocchi grandi porta a un costo di gestione alto (bisogna spostare molti byte)

Frequenza delle miss



Gestione delle miss

- La presenza di una cache non modifica molto il funzionamento del processore (con pipeline) fino a che abbiamo delle hit
 - Il processore non si «accorge» neanche della presenza della cache
 - In caso di miss, bisogna generare uno stallo nella pipeline e gestire il trasferimento da memoria principale alla cache (ad opera della circuiteria di controllo)

Gestione delle miss

- Ad esempio, per una miss sulla memoria istruzioni, bisognerà:
 1. Inviare il valore PC – 4 alla memoria (PC viene incrementato all'inizio, quindi la miss è su PC-4)
 2. Comandare alla memoria di eseguire una lettura e attenderne il completamento
 3. Scrivere il blocco che proviene dalla memoria della cache aggiornando il tag
 4. Far ripartire l'istruzione dal fetch, che stavolta troverà l'istruzione in cache

Scritture

- Gli accessi in lettura alla memoria dati avvengono con la stessa logica
- Gli accessi in scrittura sono un po' più delicati perché possono generare problemi di consistenza
- Una politica è la cosiddetta **write-through**
 - Ogni scrittura viene direttamente effettuata in memoria principale (sia che si abbia una hit che una miss)
 - In questo modo non ho problemi di consistenza, ma le scritture sono molto costose
 - Posso impiegare un buffer di scrittura (una coda in cui metto tutte le scritture che sono in attesa di essere completate)

Scritture

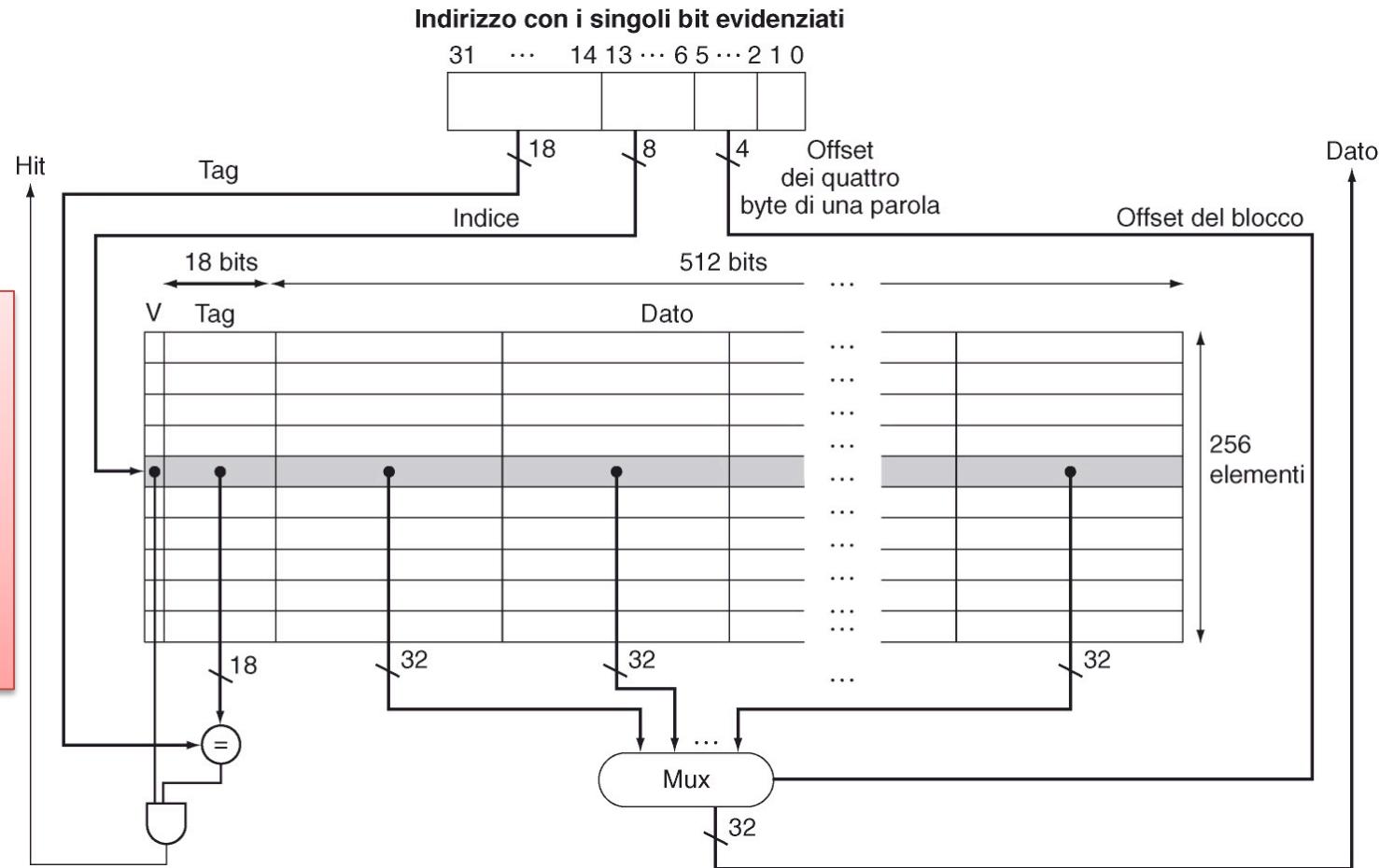
- Un'altra possibile politica è la **write-back**
 - Se il blocco è in cache le scritture avvengono localmente in cache e l'update viene fatto solo quando il blocco viene rimpiazzato (o quando una locazione nel blocco viene acceduta da un altro processore su architetture multi-core)
 - Questo schema è conveniente quando il processore genera molte scritture e la memoria non ce la fa a «stargli dietro»

Esempio

FastMath Intrinsity

(basato su architettura MIPS)

- Cache di 16K
- 16 parole per blocco
- Possibilità di operare in write-through o in write-back



Esempio FastMath Intrinsity (basato su architettura MIPS)

Tipiche performance misurate su benchmark SPEC CPU2000

Frequenza di miss per le istruzioni	Frequenza di miss per i dati	Frequenza di miss totale
0,4%	11,4%	3,2%

Cache associative

- Le cache a mappatura diretta sono piuttosto semplici da realizzare
- Tuttavia hanno un problema: se ho spesso bisogno di locazioni di memoria che si mappano sullo stesso blocco, ho cache miss in continuazione
- All'estremo opposto ho una cache completamente associativa
 - Posso mappare qualsiasi blocco in qualsiasi blocco di cache

Cache completamente associativa

- Il problema per le cache completamente associative è che devo cercare ovunque il dato (il tag è tutto l'indirizzo del blocco)
- Per effettuare la ricerca in maniera efficiente, devo farla su tutti i blocchi in parallelo
- Per questo motivo ho bisogno di n comparatori (uno per ogni blocco di cache) che operino in parallelo
- Il costo HW è così alto che si può fare solo per piccole cache

Cache set-associativa

- Le cache set-associative sono un via di mezzo tra le due che abbiamo visto
- In sostanza ogni blocco di memoria può essere mappato su una linea di n blocchi diversi di cache (n «vie»)
- Quindi combiniamo due idee
 - Associamo ciascun blocco di memoria a una certa linea (e quindi a uno degli n blocchi di quella linea su cui possiamo mappare il blocco di memoria)
 - All'interno della linea, effettuiamo una **ricerca parallela** come se avessimo una cache completamente associativa

Mappatura del blocco

- In una cache a mappatura diretta un blocco di memoria viene mappato in un blocco di cache dato da:
(indirizzo blocco) *modulo* (numero **blocchi** della cache)
- In una cache set-associativa un blocco di memoria viene mappato nella linea data da:
(indirizzo blocco) *modulo* (numero **linee** della cache)
- Quindi, per trovare il blocco all'interno della linea dobbiamo confrontare (in parallelo) il tag del blocco con tutti i tag dei blocchi di quella linea

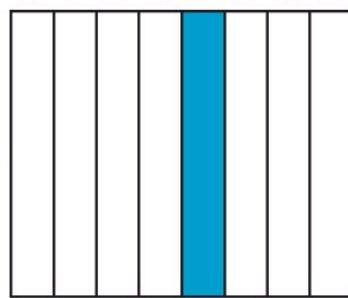
Posizione del blocco

Mappatura diretta

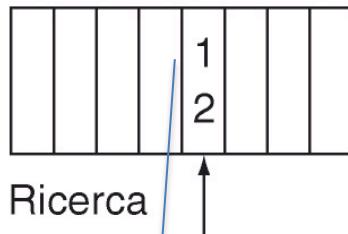
Blocco No.

0 1 2 3 4 5 6 7

Dato



Tag



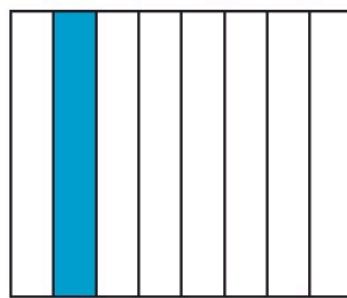
Blocco #12 solo in
posizione 4 ($12 \bmod 8$)

Set-associativa

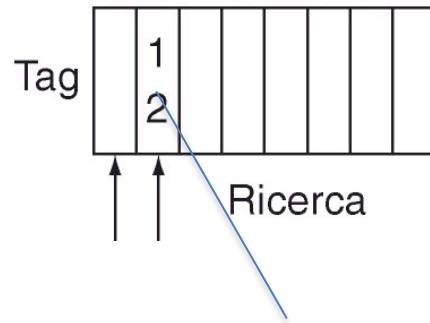
Linea No.

0 1 2 3

Dato



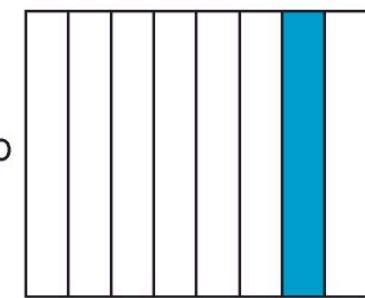
Tag



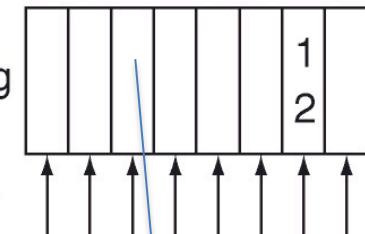
Blocco #12 solo nella linea
 $0 = (12 \bmod 4)$, blocco 0 o 1
(ipotizzando cache a 2 vie,
ovvero 4 linee da 2 blocchi)

Completamente associativa

Dato



Tag
Ricerca



Blocco #12 può
essere ovunque

Varie configurazioni

Cache set-associativa a una via

(a mappatura diretta)

Blocco	Tag	Dato
0		
1		
2		
3		
4		
5		
6		
7		

Cache set-associativa a due vie

Linea	Tag	Dato	Tag	Dato
0				
1				
2				
3				

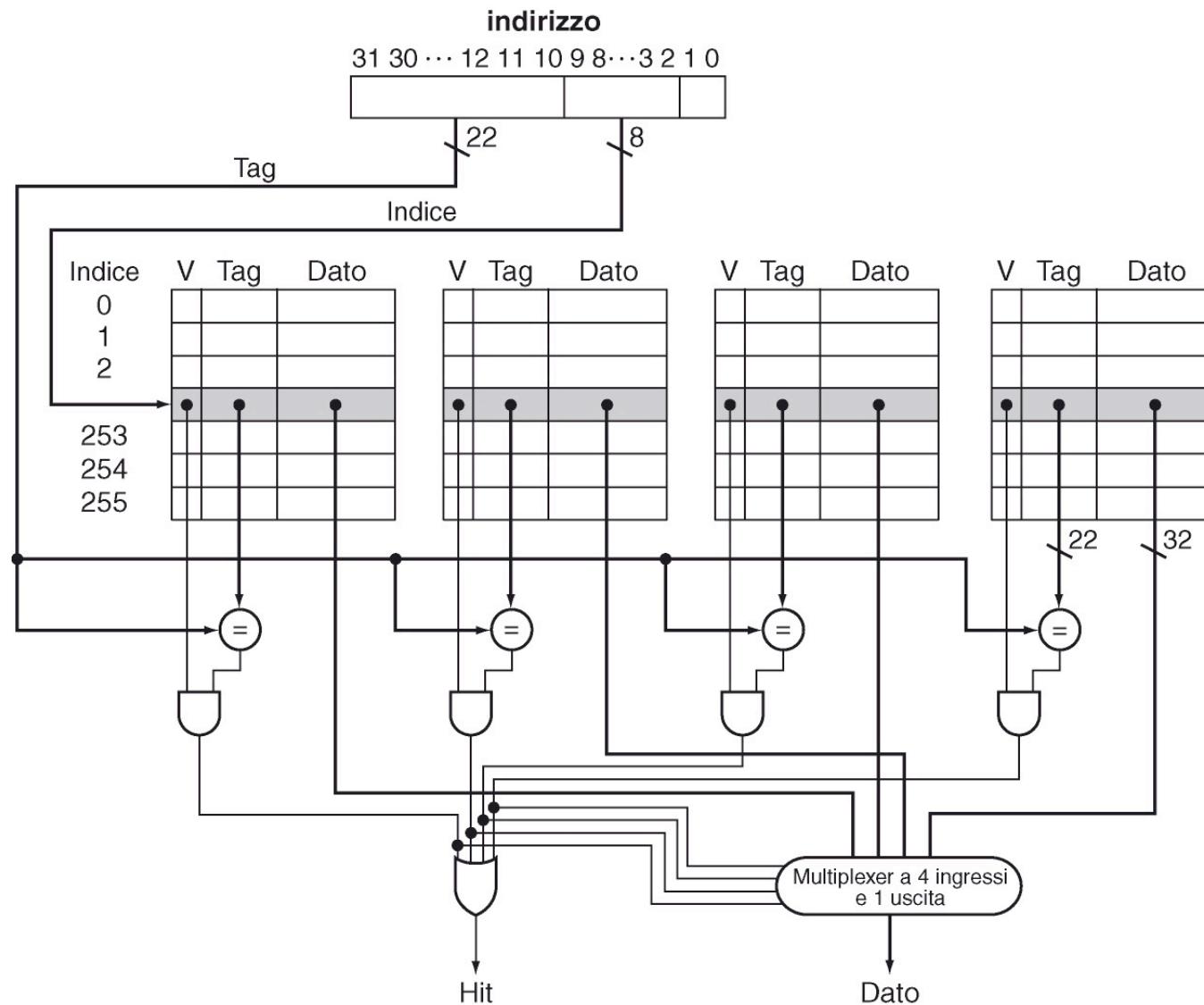
Cache set-associativa a quattro vie

Linea	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato
0								
1								

Cache set-associativa a otto vie (completamente associativa)

Tag	Dato														

Schema per cache a 4 vie



Vantaggi dell'associatività

Associatività	Frequenza di miss
1	10,3%
2	8,6%
4	8,3%
8	8,1%

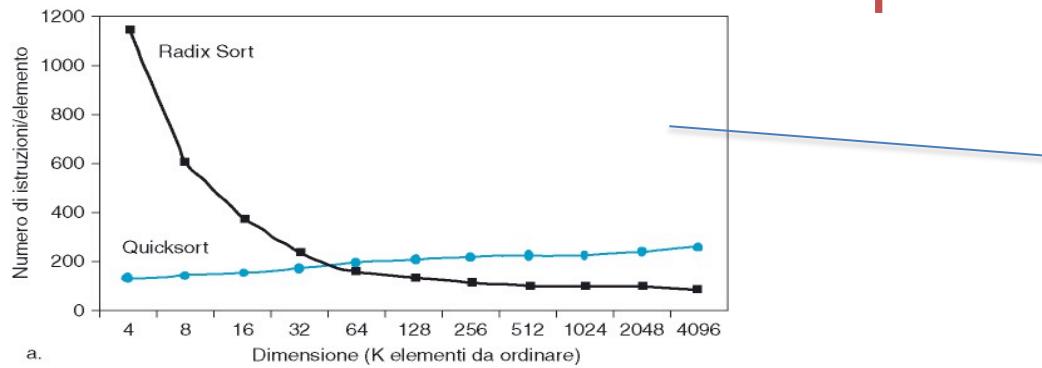
Aumentando l'associatività abbiamo vantaggi (frequenza di miss) e svantaggi (complessità). La scelta viene fatto tenendo presente questo trade-off

Un problema in più

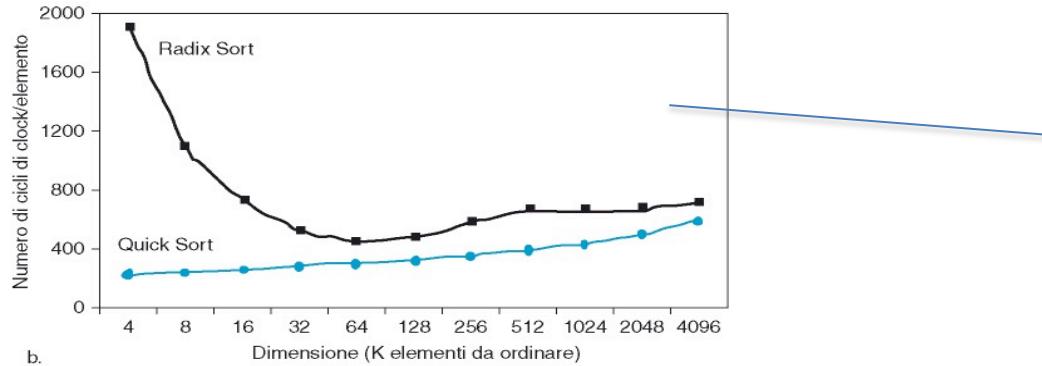
- Nelle cache a mappatura diretta quando ho una cache miss sicuramente so chi sostituire (l'unico blocco in cui posso mapparmi)
- Nelle cache associative ho più scelte: se la linea è piena chi sostituisco?
- Varie politiche
 - FIFO
 - Least Recently Used

Richiede una serie di bit
in più per contare
l'ultimo accesso

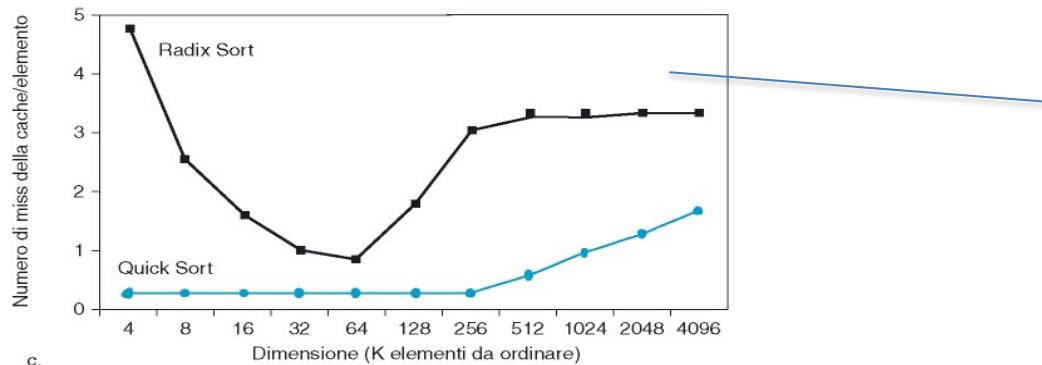
Ok, ma a che ci serve questa roba?



Numero di istruzioni
per elemento da
ordinare



Tempo di esecuzione



Cache miss