

Le gerarchie di memoria: memoria cache

Fino ad ora, si sono considerate le tecniche che consentono di migliorare le prestazioni di un calcolatore modificando le caratteristiche e il modo di funzionamento della CPU. Come si è già accennato, il “collo di bottiglia” in termini di prestazioni è però in genere costituito dalle memorie; i tempi di accesso alle normali memorie RAM sono nettamente più alti dei tempi di propagazione attraverso le unità di una CPU, tanto più se si considerano CPU *pipelined* e se il riferimento è quindi allo *stadio* piuttosto che all’intera CPU. La memoria principale viene realizzata normalmente utilizzando componenti *DRAM* (*Dynamic Random Access Memory*, ovvero memoria dinamica ad accesso casuale); memorie più veloci (tipicamente usate per le *Memorie Cache* di cui si parlerà in questo capitolo) sono composte da *SRAM* (*Static Random Access Memory*). Il costo per bit delle *DRAM* è più basso di quello delle *SRAM*; d’altra parte, le prime sono molto più lente delle seconde. La differenza di prezzo dipende dal fatto che le memorie *DRAM* utilizzano meno transistori per ogni bit da memorizzare: consentono quindi di raggiungere capacità superiori a parità di area di silicio.

Viste le differenze di costo e di velocità, diventa conveniente costruire una *gerarchia di livelli di memoria*, con la memoria più veloce posta più vicino al processore e quella più lenta, meno costosa, posta più distante. L’obiettivo è quello di fornire all’utente una quantità di memoria pari a quella disponibile nella tecnologia più economica, consentendo allo stesso tempo una velocità di accesso pari a quella garantita dalla tecnologia più veloce.

Le tecniche che realizzano gerarchie di memoria sono tutte basate sul *principio di località*, verificato analizzando l’esecuzione di un gran numero di programmi particolarmente significativi:

Località temporale: quando si fa riferimento a un elemento di memoria, c’è la tendenza a far riferimento allo stesso elemento entro breve (un tipico esempio è costituito dal riutilizzo di istruzioni e dati contenuti nei cicli);

Località spaziale: quando si fa riferimento a un elemento di memoria, c’è la tendenza a far riferimento entro breve tempo ad altri elementi che hanno indirizzo vicino a quello dell’elemento corrente (tipicamente, eseguita un’istruzione si tende ad eseguire quella immediatamente successiva; quando si accede a dati organizzati in vettori o matrici, l’accesso a un dato è seguito dall’accesso al dato immediatamente successivo, etc.).

Giocando appunto sul principio di località, la memoria di un calcolatore viene realizzata come una *gerarchia di memoria*, che consiste in un insieme di livelli di memoria, ciascuno di diversa velocità e dimensione. A parità di capacità, le memorie più veloci sono più costose di quelle più lente, perciò sono in genere più piccole: al livello più alto (quello più vicino alla CPU) troviamo memorie più piccole e veloci, a quello più basso memorie più lente e meno costose. L’obiettivo della gerarchia di memoria è quello di dare al programmatore l’illusione di poter usufruire di una memoria al tempo stesso veloce (idealmente, quanto la memoria al livello più alto) e grande (quanto quella al livello più basso).

Inevitabilmente, se le memorie di livello più alto hanno dimensioni più ridotte (in genere, molto più di quanto sarebbe necessario per ospitare l’intero programma o tutti i suoi dati) diventa necessario durante l’esecuzione dei programmi *trasferire* informazione fra memorie di livelli diversi. Anche se una gerarchia di memoria è in genere composta da più livelli, le informazioni vengono di volta in volta copiate solo tra *due livelli adiacenti*. D’ora in avanti, per semplicità, si parlerà sempre del *trasferimento di dati* fra livelli di memoria, fermo restando che il trasferimento può coinvolgere sia istruzioni sia dati.

Tecnicamente, si dovrebbe identificare il livello più elevato della gerarchia nel banco di registri (*register file*) interno alla CPU; la sua gestione è però molto particolare ed estremamente semplice (i problemi che la riguardano vengono gestiti essenzialmente dal compilatore), quindi non vi faremo riferimento. Si concentrerà invece l’attenzione sulle gerarchie più alte di memoria

di lavoro, cioè sulle cosiddette *memorie cache*; nei calcolatori moderni è sempre presente *almeno un livello di cache*: spesso, ne sono presenti due o anche più.

La memoria cache viene vista come organizzata in *blocchi* o *linee* (*cache lines*); la minima quantità di informazione che può essere trasferita fra due livelli è appunto un *blocco*. Se il dato richiesto dalla CPU fa parte di uno dei blocchi presenti nella cache di livello superiore, si dice che la richiesta ha avuto successo (*hit*). Se invece il dato non si trova nel livello superiore, si dice che la richiesta fallisce (*miss*): per trovare il blocco che contiene i dati richiesti, bisogna accedere al livello inferiore della gerarchia e trasferire il blocco corrispondente al livello superiore.

Si indica come *frequenza dei successi* (*hit rate*) la frazione di accessi alla memoria che hanno trovato il dato desiderato nel livello superiore; spesso questo valore viene utilizzato come indice delle prestazioni della memoria gerarchica. Dualmente, la *frequenza dei fallimenti* (*miss rate*), pari a $\text{miss rate} = (1.0 - \text{hit rate})$ è la frazione di accessi alla memoria che *non* hanno trovato il dato desiderato nel livello superiore.

Il *tempo di successo* (*hit time*) è il tempo di accesso al livello superiore della gerarchia di memoria, e comprende anche il tempo necessario per stabilire se il tentativo di accesso si risolve in un successo o in un fallimento.

La *penalità di fallimento* (*miss penalty*) è il tempo necessario per sostituire un blocco nel livello superiore con un altro blocco preso dal livello inferiore e per passare alla CPU le informazioni contenute in questo nuovo blocco.

Il *tempo di fallimento* (*miss time*) è dato da:

$$\text{miss time} = \text{hit time} + \text{miss penalty}$$

Poiché il livello superiore è più piccolo e più veloce, il tempo di successo è molto inferiore al tempo necessario per accedere al secondo livello della gerarchia, che invece rappresenta la componente principale della penalità di fallimento.

Il *tempo medio di accesso alla memoria* T_M è dato da:

$$T_M = \text{hit rate} * \text{hit time} + \text{miss rate} * \text{miss time} = \text{hit rate} * \text{hit time} + \text{miss rate} * (\text{hit time} + \text{miss penalty}) = \text{hit time} * (\text{hit rate} + \text{miss rate}) + \text{miss rate} * \text{miss penalty}.$$

Dato che ovviamente è $\text{hit rate} + \text{miss rate} = 1$, si ottiene infine

$$T_M = \text{hit time} + \text{miss rate} * \text{miss penalty}$$

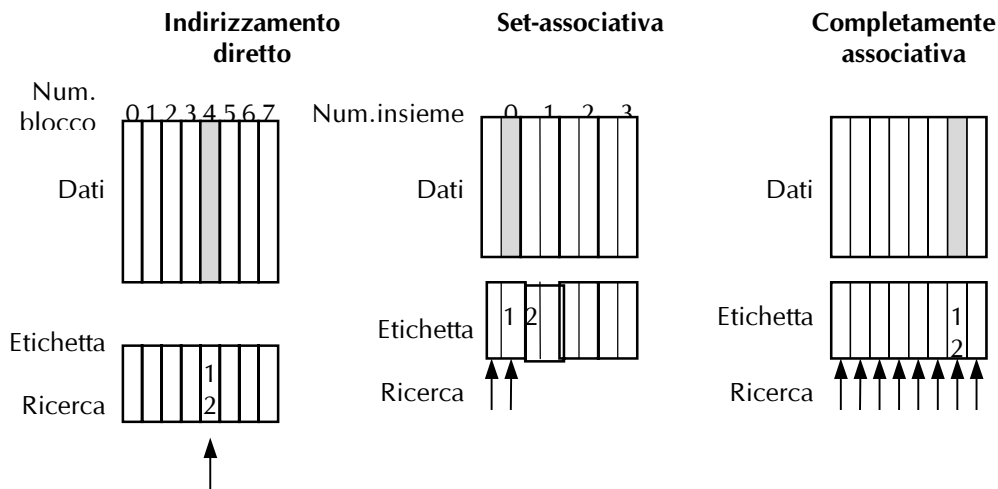
Durante il progetto di una gerarchia di memoria, quattro sono le domande a cui bisogna rispondere:

1. Dove si può mettere un blocco che viene portato dal livello inferiore al livello superiore? (*Problema del piazzamento di un blocco*).
2. Dove si trova il blocco che contiene il dato richiesto? (*Problema della ricerca di un blocco*).
3. Quale blocco presente al livello superiore deve essere sostituito da uno del livello inferiore, in caso di fallimento? (*Problema della sostituzione di un blocco*).
4. Cosa succede in caso di scrittura? (*Problema della strategia di scrittura*).

Consideriamo innanzitutto il **problema del piazzamento di un blocco**. In fase di esecuzione, la CPU può a priori tentare di accedere a *una qualunque parola nello spazio totale di indirizzamento*: spazio che può essere qui visto come corrispondente all'intera memoria RAM (livello più basso nelle gerarchie della memoria di lavoro), ma che certamente eccede di gran lunga quello disponibile nella cache di livello superiore. Occorre quindi definire le possibili modalità per consentire ad ogni parola della memoria indirizzabile di trovare (ove necessario) una posizione della cache in cui possa essere trasferita – in altre parole, occorre definire una corrispondenza tra l'indirizzo in memoria della parola e la locazione nella cache. A questo scopo, sono state definite essenzialmente tre soluzioni diverse:

1. Cache a indirizzamento diretto

2. Cache set-associativa a n vie
3. Cache completamente associativa



1. Cache a indirizzamento diretto

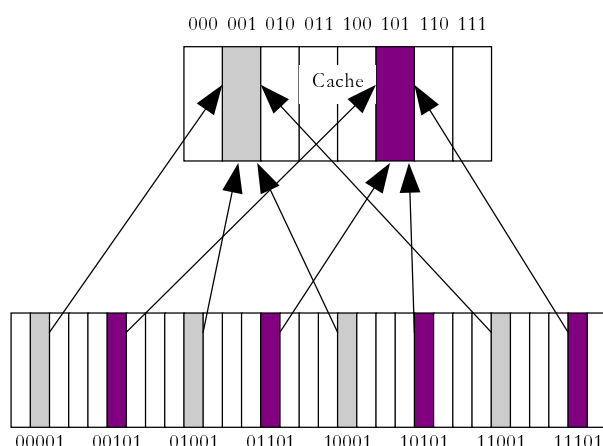
In una cache a *indirizzamento diretto* (*direct mapped*), ogni locazione di memoria corrisponde esattamente a una locazione della cache.

La corrispondenza tra indirizzo di memoria e locazione nella cache è data da:

$$(Ind. \text{ blocco})_{cache} = (Ind. \text{ blocco})_{mem} \bmod (\# \text{ blocchi nella cache})$$

Essendo il numero degli elementi nella cache una potenza di 2, l'operazione di *modulo* può essere effettuata considerando i $\log_2(\# \text{ blocchi nella cache})$ bit meno significativi, che sono utilizzati come *indice* della cache.

Si consideri ad esempio una cache a indirizzamento diretto di 8 parole e si supponga che lo spazio totale di indirizzamento della memoria sia compreso tra 0 e 29 (gli indirizzi di memoria sono quindi di *cinque* bit); per quanto detto prima, un indirizzo di memoria X corrisponde all'elemento $X \bmod 8$ della cache. I $\log_2(8)=3$ bit meno significativi dell'indirizzo sono quindi utilizzati come *indice* della cache.



Tutti gli indirizzi che *terminano* con la configurazione *001* (cioè *00001*, *01001*, *10001* e *11001*) corrispondono all'elemento *001* della cache; allo stesso modo, le parole il cui indirizzo termina con *101* (cioè *00101*, *01101*, *10101* e *11101*) possono essere caricate nell'elemento *101* della

cache, e così via. Il problema del piazzamento di un blocco è risolto in modo elementare, esaminando semplicemente la configurazione degli ultimi tre bit dell'indirizzo.

2. Cache completamente associativa

In una cache *completamente associativa* (*fully associative*) un blocco di memoria può essere messo in una qualsiasi posizione della memoria cache; si vedrà poi come viene creata la corrispondenza fra locazione della cache e indirizzo in RAM. Poiché il blocco può essere messo in un posto qualsiasi della cache, al momento della ricerca tutti i blocchi della cache dovranno essere esaminati.

3. Cache set-associativa a n vie

Una cache set-associativa a n vie è costituita da numerosi insiemi (*set*), ognuno dei quali comprende n blocchi. Anche la memoria RAM è vista come organizzata in blocchi: ogni blocco della memoria corrisponde ad un unico *insieme* della cache ed il blocco può essere messo in uno *qualsiasi* degli n elementi dell'insieme. In altre parole, in una cache set-associativa a n vie, ogni blocco della memoria può essere trasferito in un numero prefissato n di posizioni all'interno di un insieme.

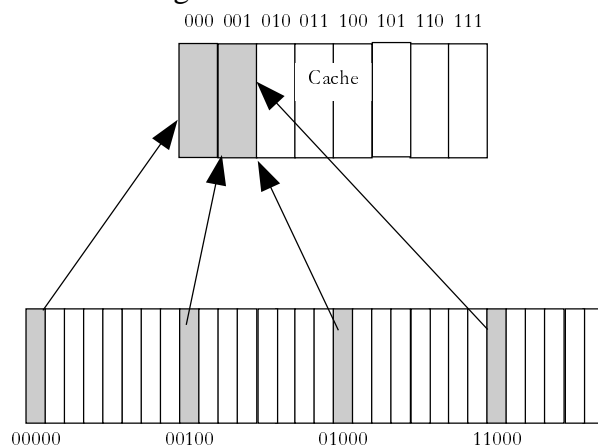
Si tratta di uno schema intermedio tra quello a indirizzamento diretto e quello completamente associativo: la corrispondenza blocco di RAM – insieme avviene con un indirizzamento diretto, mentre tutti i blocchi nell'insieme vengono poi esaminati per verificare la corrispondenza.

In una cache set-associativa, l'*insieme* che contiene il blocco viene individuato da:

$$(\text{Insieme})_{\text{cache}} = (\text{Ind. blocco})_{\text{mem}} \text{ modulo } (\# \text{ insiemi nella cache})$$

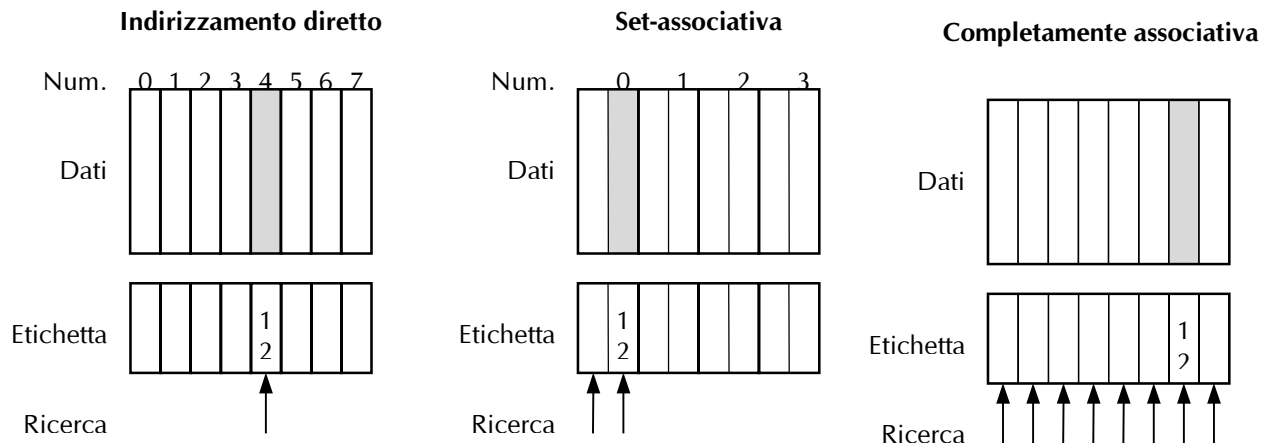
Essendo il numero degli insiemi nella cache una potenza di 2, l'operazione di *modulo* può essere effettuata considerando i $\log_2(\# \text{ insiemi nella cache})$ bit meno significativi, che sono utilizzati come *indice* della cache. Il blocco può poi essere messo in un qualsiasi elemento dell'insieme: la ricerca deve quindi essere effettuata su tutti gli elementi dell'insieme.

Si consideri ad esempio una cache set-associativa a 2 vie composta da 8 parole, e si supponga ancora che lo spazio di indirizzamento della memoria sia compreso tra 0 e 29: un indirizzo di memoria X corrisponde ad un elemento dell'insieme dato da $X \text{ modulo } 4$ della cache. I $\log_2(4) = 2$ bit meno significativi dell'indirizzo di memoria sono utilizzati come *indice* della cache.



Tutti gli indirizzi che terminano con *00* – cioè *00000*, *00100*, ecc. – corrispondono all'insieme *00* della cache, e così via.

Per approfondire il problema del piazzamento, si supponga ora di volere operare una variazione del piazzamento del blocco di indirizzo *12*, rispettivamente, in cache da 8 blocchi a indirizzamento diretto, set-associativa a 2 vie e completamente associativa.



Nella cache da 8 blocchi a indirizzamento diretto il blocco 12 si può trovare in un solo blocco della cache definito da: $(12 \text{ modulo } 8) = 4$.

Nella cache da 8 blocchi set-associativa a 2 vie ci sono 4 insiemi: il blocco 12 si può trovare in un qualsiasi blocco dell'insieme $(12 \text{ mod } 4) = 0$.

Nel piazzamento completamente associativo il blocco 12 può trovarsi in uno qualsiasi degli 8 blocchi.

Ogni politica di piazzamento può in realtà essere considerata una variazione di quella set-associativa: una cache a indirizzamento diretto è una cache set-associativa a 1 vie, in cui ogni elemento della cache contiene un blocco e forma un insieme di un solo elemento.

Una cache di m elementi completamente associativa è una cache set-associativa a m vie: c'è un solo insieme di m blocchi e un elemento può trovarsi in uno qualsiasi dei blocchi dell'insieme.

La dimensione complessiva della cache in termini di blocchi è uguale al numero degli insiemi moltiplicato per l'associatività (ovvero la dimensione degli insiemi).

Definita la dimensione della cache, al crescere dell'associatività diminuisce il numero degli insiemi, mentre cresce il numero di elementi compresi in un insieme.

A titolo di esempio, si considerino le possibili configurazioni di una cache composta da 8 blocchi

Set-associativa a 1 via

(A indirizzamento diretto)

Blocco	Etichetta	Dati
0		
1		
2		
3		
4		
5		
6		
7		

Set-associativa a 2 vie

Insieme	Etichetta	Dati	Etichetta	Dati
0				
1				
2				
3				

Set-associativa a 4 vie

Insieme	Etichetta	Dati	Etichetta	Dati	Etichetta	Dati	Etichetta	Dati
0								
1								

Set-associativa a 8 vie (Completamente associativa)

Etich. Dati Etich. Dati Etich. Dati Etich. Dati Etich. Dati Etich. Dati Etich. Dati Etich. Dati

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Si passi ora ad affrontare il **problema della ricerca di un blocco**.

Poiché ogni elemento della cache può contenere parole memorizzate in diverse locazioni di memoria, per sapere se una parola di memoria si trova nella cache bisogna aggiungere nella cache un insieme di *etichette* (*tag*), che contengono le informazioni necessarie per verificare se una delle parole presenti nella cache corrisponde o meno alla parola cercata.

È necessario disporre di un metodo per riconoscere se un blocco della cache contiene informazioni *valide*: ad esempio, quando un processore viene acceso la cache è vuota e le informazioni nelle etichette non hanno nessun significato. Si aggiunge quindi a ogni elemento della cache un *bit di validità* (*validity bit*) che indicare se l'elemento stesso contiene dei dati validi. Il *validity bit* è ovviamente indipendente dalla particolare filosofia scelta per il piazzamento, ed è quindi presente in tutte le soluzioni.

Si consideri dapprima la ricerca di un blocco in una cache di 8 blocchi a *indirizzamento diretto*, nella quale ogni "blocco" corrisponda a una singola parola. A parte il bit di validità, dato che per ogni parola di memoria c'è un'unica posizione della cache dove il trasferimento è possibile, l'etichetta associata alla posizione della cache dovrà contenere la parte più significativa dell'indirizzo della parola che vi è stata trasferita. Per ogni parola presente nella cache, l'indirizzo completo *con riferimento allo spazio totale indirizzabile* sarà la concatenazione *etichetta-indirizzo in cache*.

Per chiarire il meccanismo della ricerca, si esamina un esempio costituito da una sequenza di richieste fatte dalla CPU al sistema di memoria, e si vede come evolve di conseguenza una cache a indirizzamento diretto da 8 parole, nella quale i 3 bit meno significativi indicano il blocco, mentre *il campo etichetta contiene la parte più significativa dell'indirizzo di memoria*.

Indirizzo del dato in memoria	Indirizzo binario dato in memoria	Blocco della cache assegnato (dove trovare o mettere i dati)	Etichetta attuale associata al blocco della cache	Risultato dell'opera zione
22	10110	$(10\textcolor{red}{110}_{\text{due}} \bmod 8) = \textcolor{red}{110}_{\text{due}}$	00	Fallimento
26	11010	$(11\textcolor{red}{010}_{\text{due}} \bmod 8) = \textcolor{red}{010}_{\text{due}}$	01	Fallimento
22	10110	$(10\textcolor{red}{110}_{\text{due}} \bmod 8) = \textcolor{red}{110}_{\text{due}}$	10	Successo
26	11010	$(11\textcolor{red}{010}_{\text{due}} \bmod 8) = \textcolor{red}{010}_{\text{due}}$	11	Successo
16	10000	$(10\textcolor{red}{000}_{\text{due}} \bmod 8) = \textcolor{red}{000}_{\text{due}}$	11	Fallimento
4	00100	$(00\textcolor{red}{100}_{\text{due}} \bmod 8) = \textcolor{red}{100}_{\text{due}}$	01	Fallimento
16	10000	$(10\textcolor{red}{000}_{\text{due}} \bmod 8) = \textcolor{red}{000}_{\text{due}}$	10	Successo
18	10010	$(10\textcolor{red}{010}_{\text{due}} \bmod 8) = \textcolor{red}{010}_{\text{due}}$	10	Successo

Come si è ampiamente detto, nella cache a indirizzamento diretto c'è un solo posto dove poter inserire i dati appena richiesti, quindi non c'è nessuna possibilità di scegliere quale elemento sostituire. I bit meno significativi dell'indirizzo costituiscono l'indice per individuare l'unico elemento della cache al quale l'indirizzo può corrispondere. Si consideri ora in maggior dettaglio come si svolgono alcune operazioni di accesso, supponendo di partire dall'accensione della macchina, quando tutti i bit di validità sono negativi (N) e i contenuti delle etichette privi di significato: la ricerca della parola con indirizzo di memoria 10110 viene fatta esaminando il blocco 110, la cui etichetta è negativa. L'accesso viene gestito trasferendo il dato dalla memoria e portando a valore positivo (S) l'etichetta del blocco 110; l'etichetta del blocco diventa ora 10.

Indice	V	Etichetta	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Stato iniziale della cache dopo l'accensione.

Indice	V	Etichetta	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	S	10	Memoria(10110)
111	N		

Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (10110).

Si supponga ora di avere già “popolato” la cache con vari trasferimenti dalla memoria, e di avere la situazione riportata di seguito: si cerchi di leggere la parola di memoria 10101. Il bit di validità del blocco 101 è positivo, ma l'etichetta è diversa da quella voluta (01 invece di 10): si ha quindi un *fallimento*. La parola cercata viene trasferita dalla RAM alla cache, e l'etichetta del blocco 101 opportunamente aggiornata.

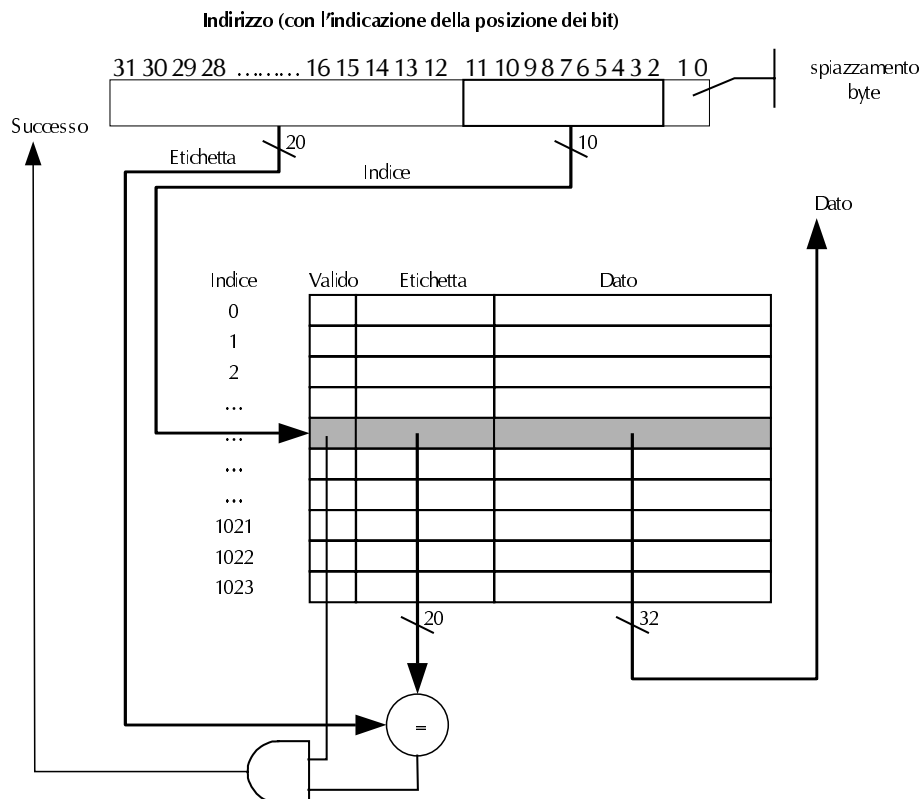
Indice	V	Etichetta	Dato
000	S	00	Memoria (00000)
001	S	01	Memoria (01001)
010	S	10	Memoria (10010)
011	S	11	Memoria (11011)
100	S	11	Memoria (11100)
101	S	01	Memoria (01101)
110	S	10	Memoria(10110)
111	S	10	Memoria (10111)

Stato iniziale della cache

Indice	V	Etichetta	Dato
000	S	00	Memoria (00000)
001	S	01	Memoria (01001)
010	S	10	Memoria (10010)
011	S	11	Memoria (11011)
100	S	11	Memoria (11100)
101	S	10	Memoria (10101)
110	S	10	Memoria(10110)
111	S	10	Memoria (10111)

Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (10101).

Una cache a indirizzamento diretto da 4Kbyte e blocco corrispondente a una sola parola di 32 bit può essere schematizzata come segue:



L'*indice* è utilizzato per selezionare (mediante normale indirizzamento) un elemento della cache che consiste di una parola di dati, un'*etichetta* (che viene confrontata con la parte alta dell'indirizzo per verificare se l'elemento della cache corrisponde all'indirizzo richiesto) e un bit di validità. La cache contiene 2^{10} (cioè 1024) parole e la dimensione di un blocco è di 1 parola: 10 bit sono utilizzati per l'*indice* della cache.

I 2 bit meno significativi dell'indirizzo (*spiazzamento* o *offset*) specificano un byte all'interno di una parola o "blocco" da 32 bit: rimangono quindi $32 - 10 - 2 = 20$ bit per l'*etichetta*.

Se l'*etichetta* della cache e i 20 bit più alti dell'indirizzo sono uguali e se il bit di validità ha valore positivo, allora la ricerca nella cache ha successo (*hit*), e la parola viene fornita al processore. Altrimenti si verifica un fallimento (*miss*).

La memorie cache descritta finora non sfrutta il principio di *località spaziale degli accessi* in quanto ogni parola corrisponde ad un blocco. Per trarre vantaggio dalla località spaziale è necessario che *la dimensione del blocco della cache sia maggiore della dimensione della parola di memoria*, in modo che il blocco contenga più di una sola parola. È necessario quindi un campo aggiuntivo dell'indirizzo che rappresenti lo *spiazzamento (offset) della parola nel blocco*. Quando si verifica un fallimento, dalla memoria centrale vengono prelevate più parole adiacenti che hanno una elevata probabilità di essere richieste a breve. Il numero totale di etichette è inferiore nella cache con blocchi di più parole, perché ogni etichetta è utilizzata per più parole. Questo migliora il grado di efficienza di utilizzo della memoria cache.

Un indirizzo di memoria è ora diviso in 3 campi:

Etichetta	Indice	Spiazzamento
-----------	--------	--------------

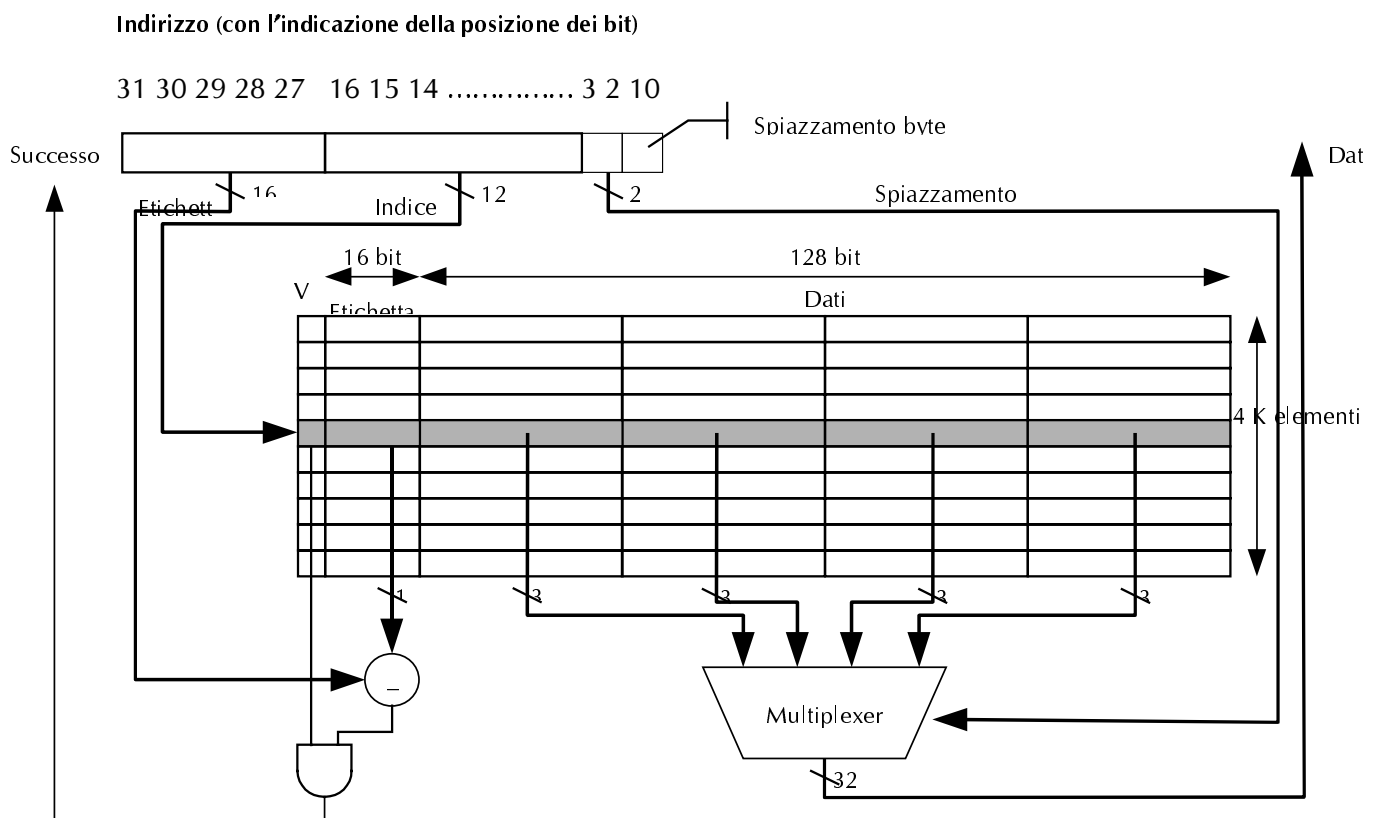
Questa struttura viene utilizzata sia per cache a indirizzamento diretto, sia per quelle associative e quella set-associative: diverso è l'uso che se ne fa nei diversi casi. L'*etichetta*, da confrontare con il contenuto del campo etichetta della cache, viene infatti utilizzata per controllare tutti i

blocchi nell'insieme selezionato dall'indice (cache set-associativa), il blocco selezionato dall'indice (cache a indirizzamento diretto) oppure tutti i blocchi (cache completamente associativa).

L'*indice* serve a identificare l'insieme (cache set-associativa) oppure il blocco (cache a indirizzamento diretto). In una cache completamente associativa, il campo indice non serve poiché c'è un solo insieme.

Lo *spiazzamento* (offset) nel blocco indica l'indirizzo della parola o del byte desiderati all'interno del blocco.

Si consideri una cache a indirizzamento diretto da 64Kbyte e blocco da 128 bit (quattro parole):



La cache contiene 4K (2^{12}) blocchi: 12 bit sono utilizzati per l'*indice* della cache.

Ogni blocco è composto da 4 parole ($4 \times 32 \text{ bit} = 16 \text{ byte}$): è necessario un campo aggiuntivo da 2 bit (i bit 3-2) per lo *spiazzamento della parola nel blocco*. Tali bit controllano il multiplexer in modo da selezionare la parola richiesta tra le 4 parole che si trovano nel blocco individuato.

I 2 bit meno significativi dell'indirizzo (*spiazzamento o offset*) specificano un byte all'interno di una parola da 32 bit: rimangono $32 - 12 - 2 - 2 = 16$ bit per l'*etichetta*.

Si passi ora al *problema della ricerca di un blocco in una cache set-associativa a n vie*.

Ogni blocco della cache comprende ancora un'etichetta che permette di individuare l'indirizzo del blocco. Il valore dell'indice serve a selezionare l'insieme che contiene l'indirizzo desiderato; per ogni blocco dell'insieme che potrebbe contenere l'informazione cercata viene controllata l'etichetta per verificare se corrisponde all'indirizzo richiesto dalla CPU. Le etichette di tutti i blocchi compresi in questo insieme debbono quindi essere controllate. Per ottimizzare le prestazioni, tutte le etichette dell'insieme selezionato vengono esaminate *in parallelo*, in modo associativo (cercando cioè la corrispondenza fra le etichette e il segmento di indirizzo).

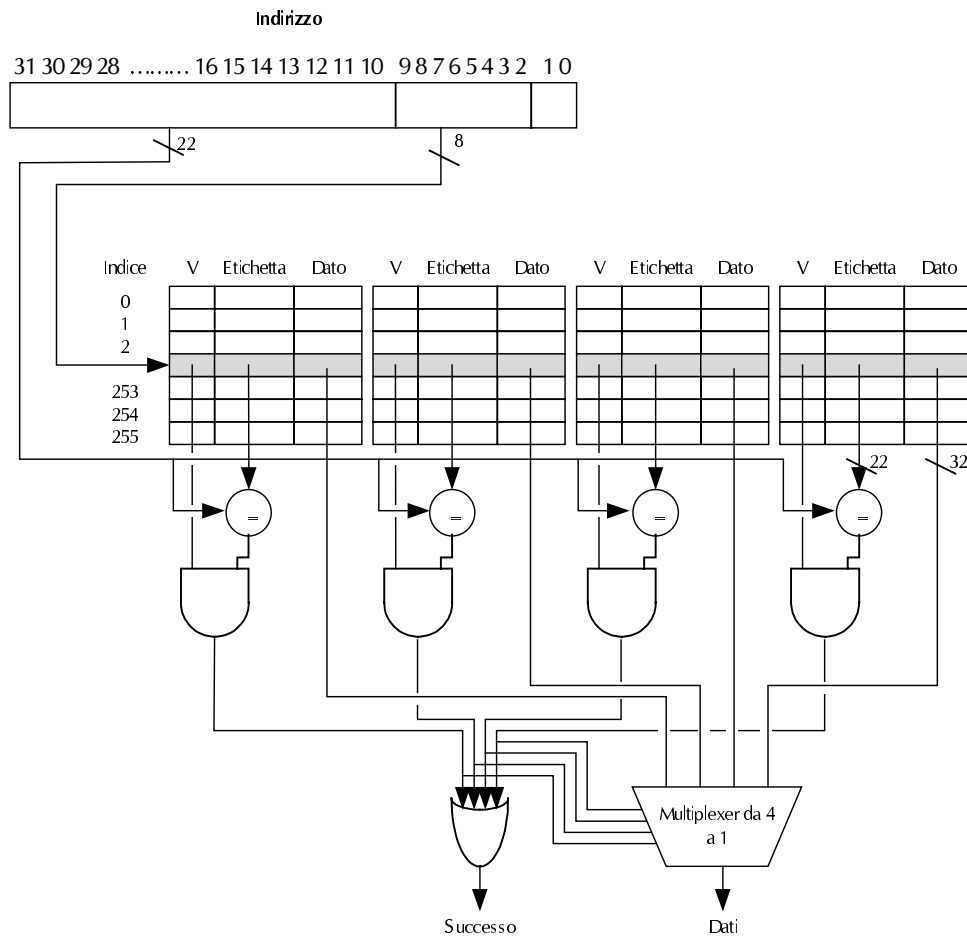
Se si mantiene costante la dimensione totale, al crescere dell'associatività aumenta anche il numero dei blocchi compresi nell'insieme, e questo corrisponde al numero dei confronti associativi che debbono essere effettuati per realizzare una ricerca in parallelo. Ogni volta che si raddoppia il grado di associatività si raddoppia il numero di blocchi compresi in un insieme e si dimezza il numero degli insiemi. Al tempo stesso, ogni incremento dell'associatività di un fattore due fa diminuire di un bit la dimensione dell'indice e aumentare di un bit la dimensione dell'etichetta. (vale la pena di osservare che aumentando l'associatività aumenta il numero dei confrontatori e aumenta anche la dimensione di ogni singolo confrontatore – si ha cioè una maggiore complessità circuitale). In una cache completamente associativa c'è un unico insieme e tutti i blocchi debbono essere esaminati in parallelo: di conseguenza, non c'è indice e l'intero indirizzo, a parte lo spiazzamento nel blocco, viene confrontato con l'etichetta di ogni blocco: occorrono quindi tanti comparatori quanti sono i blocchi. (Dualmente, in una cache a indirizzamento diretto è necessario un solo comparatore, dato che l'elemento può essere in una sola posizione).

In una cache set-associativa a n vie, sono necessari n comparatori, oltre a un multiplexer da n a 1 per scegliere tra gli n possibili blocchi dell'insieme selezionato. I comparatori individuano quale elemento dell'insieme corrisponde all'etichetta e forniscono quindi gli ingressi di selezione del multiplexer, in modo da avviare all'uscita uno solo degli n blocchi dell'insieme selezionato.

L'accesso alla cache si effettua utilizzando l'indice per individuare l'insieme e poi esaminando in parallelo tutti gli n blocchi dell'insieme.

Oltre al costo – correlato, come si è detto, ai comparatori aggiunti – occorre tenere conto dei ritardi imposti dalla necessità di confrontare e selezionare l'elemento desiderato tra quelli dell'insieme. D'altra parte, è chiaro che la soluzione completamente associativa permette uno sfruttamento migliore dello spazio disponibile in cache, dato che – ad esempio, in fase di scrittura – è possibile trasferire un blocco dalla RAM a un qualsiasi blocco della cache. In ogni gerarchia di memoria, la scelta tra lo schema a indirizzamento diretto, quello set-associativo e quello completamente associativo dipende dal confronto tra il costo di un fallimento e quello di realizzazione dell'associatività, sia dal punto di vista del tempo sia da quello della circuiteria aggiuntiva.

Lo schema generale di una cache set-associativa a 4 vie da 4Kbyte e blocco da 32 bit è il seguente:



Si consideri ora il **problema della sostituzione di un blocco**. Quando si verifica un fallimento nell'accesso alla cache, nel caso di cache a indirizzamento diretto c'è un solo candidato alla sostituzione: il problema si risolve quindi immediatamente. In una cache completamente associativa, invece, bisogna decidere quale blocco sostituire: ogni blocco è un potenziale candidato per la sostituzione. Occorre stabilire una *politica di sostituzione*. Infine, se la cache è set-associativa, l'insieme interessato è identificato immediatamente ma occorre stabilire una politica di sostituzione limitatamente ai blocchi compresi nell'insieme.

Le principali strategie utilizzate per la scelta del blocco da sostituire sono sostanzialmente tre: sostituzione casuale, sostituzione del blocco usato meno di recente, sostituzione del tipo "first in-first out".

Nel caso di sostituzione *casuale (random)*, la scelta tra i blocchi candidati viene effettuata a caso, eventualmente utilizzando dei componenti hardware di supporto per l'identificazione del blocco. Se la politica è quella del *blocco utilizzato meno di recente (Least Recently Used - LRU)*, il blocco sostituito è quello che è rimasto inutilizzato da più lungo tempo. A questo scopo, nei termini più semplici ad ogni blocco si associano dei contatori verso il basso che al momento della scrittura nel blocco vengono posti al valore massimo e che vengono poi decrementati di un'unità ogni volta che si effettua una lettura *in un blocco diverso*. La sostituzione toccherà quindi il blocco associato al contatore col valore più basso.

Infine, se la politica scelta è quella *First In First Out (FIFO)*, il blocco sostituito è quello utilizzato N accessi precedenti, indipendentemente dal fatto che sia stato utilizzato o meno negli ultimi $(N-1)$ accessi.

La selezione casuale ha evidentemente il vantaggio di essere semplice da realizzare; peraltro, non tiene alcun conto del principio di località temporale – si rischia di sostituire un blocco che è stato scritto da poco e cui si tenterà ben presto di accedere nuovamente.

Al crescere del numero di blocchi di cui bisogna tenere traccia, peraltro, la politica LRU (che è basata proprio sul principio di località temporale) diventa sempre più costosa; in pratica, si ricorre a qualche semplice approssimazione. Val la pena di notare che in una cache set-associativa a 2 vie, la sostituzione a caso ha una frequenza dei fallimenti pari a circa 1.1 volte quella ottenuta utilizzando la politica LRU. Al crescere delle dimensioni della cache, la frequenza dei fallimenti diminuisce per entrambe le strategie e la differenza in termini assoluti diventa minima. Il vantaggio della sostituzione LRU aumenta al crescere del grado di associatività, ma in tal caso è anche più difficile da realizzare.

Infine, affrontiamo il **problema della strategia di scrittura**. Il problema nasce dalla necessità che – quando si deve scrivere il risultato di un'operazione – si vuole certamente che anche l'istruzione di scrittura sia eseguita velocemente (e quindi accedendo alla cache), ma si vuole anche che l'informazione contenuta – in qualsiasi istante – nella cache sia *consistente* con quella contenuta nella RAM. Le possibili strategie per la gestione delle scritture sono *write-through* e *write-back*.

Nell'approccio *write-through*, quando si esegue un'istruzione di scrittura l'informazione viene scritta simultaneamente nel blocco della cache e nel blocco della memoria principale. La coerenza è quindi sempre rispettata, a prezzo però di un maggior tempo richiesto da ogni operazione di scrittura. Nella soluzione *write-back* (o *copy back*), invece, al momento dell'esecuzione dell'istruzione l'informazione viene scritta *solo* nel blocco della cache. Il blocco modificato viene scritto nel livello inferiore della gerarchia solo quando se ne decide la sostituzione. Al termine della istruzione di scrittura nella cache, quindi, la memoria RAM conterrà un valore *diverso* da quello presente nella cache; in questo caso si dice che la memoria e la cache sono *inconsistenti* (cioè *non sono coerenti*): il blocco della cache può essere *clean* (*unmodified*) se non ci sono state scritture, oppure *dirty* (*modified*) se sono state effettuate scritture.

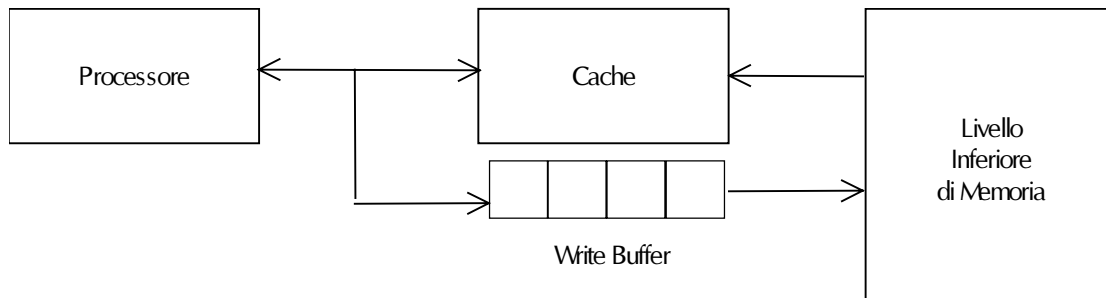
Poiché le prestazioni delle CPU continuano ad aumentare con un tasso di crescita superiore a quello delle memorie principali basate su *DRAM*, la frequenza con cui vengono generate le scritture da parte del processore sarà ben presto superiore a quella gestibile dal sistema di memoria, anche in presenza di memoria più grandi sia in termini fisici che in termini logici; è quindi ragionevole prevedere che in futuro verrà utilizzata sempre più la strategia *write-back*. Fra l'altro, ancora il principio di località porta a prevedere che se l'operazione di scrittura ha interessato un particolare blocco di cache, la probabilità che nuove scritture interessino lo stesso blocco è alta – è quindi probabile che ci siano state numerose modifiche del blocco di cache prima che questo venga sostituito e che si ricorra al *write-back*.

Le due soluzioni hanno ognuna vantaggi e svantaggi. Per quanto riguarda l'opzione *write-back*, i vantaggi sono che:

- Le singole parole possono essere scritte dalla CPU alla frequenza a cui la *cache*, e non la memoria principale, è in grado di accettarle;
 - Scritture multiple all'interno dello stesso blocco di cache richiedono poi *una sola scrittura* al livello inferiore della gerarchia.
 - Quando i blocchi vengono scritti, il sistema può trarre vantaggio dall'utilizzo di un'interfaccia (bus) più larga con il livello inferiore, visto che si trasferisce un blocco intero.
- Un'interfaccia più larga consente anche di migliorare la gestione dei fallimenti in lettura.

I vantaggi dello schema *write-through* sono riassumibili come segue:

- I fallimenti in lettura sono meno costosi, infatti non richiedono *mai* la scrittura nel livello inferiore (un fallimento in lettura con la strategia *write-back* implica la scrittura del blocco sostituito, se questo risulta “dirty”).
- È più facile realizzare uno schema *write-through* che uno *write-back*, anche se, per essere efficace in un sistema veloce, una cache *write-through* deve essere dotata anche di un buffer di scrittura (*write buffer*). in modo da non dover “attendere” il livello inferiore di memoria (si veda la figura):



Il buffer di scrittura è posto tra la cache e il livello inferiore di memoria:

- il processore scrive il dato nella cache e nel buffer di scrittura;
- il controllore del sottosistema di memoria scrive il contenuto del buffer di scrittura in memoria.

In genere, il buffer di scrittura viene realizzato mediante una semplice memoria *FIFO* di 4 posizioni; quando il processore effettua una scrittura, scrive nella cache e nel buffer, e da questo l'informazione viene trasferita (alla velocità propria della RAM) nel livello inferiore di memoria. La velocità della RAM condiziona quindi i trasferimenti dal buffer ma non quelli dalla CPU. Si possono accodare fino a quattro scritture: stalli in scrittura (*write stall*) possono evidentemente avvenire anche in presenza di buffer di scrittura, quando il buffer raggiunge la saturazione.

Anche un'operazione di scrittura può generare un “fallimento” (*write miss*) – si tenta cioè di scrivere in una parola che non è presente in cache. Le possibili opzioni nel caso di fallimenti in scrittura sono:

- *Alloca e scrivi (write allocate o fetch on write)*: il blocco viene caricato nella cache e successivamente si effettua la scrittura (secondo una delle due modalità *write-through* o *write-back*) analogamente al caso di fallimento di un'operazione di lettura.
- *Scrivi senza allocare (no write allocate o write around)*: il blocco viene modificato direttamente nel livello inferiore di memoria e non viene caricato nella cache.

Entrambe le opzioni possono essere utilizzate sia con la politica *write-through* sia con quella *write-back*, ma in genere le cache di tipo *write-back* usano l'opzione *write allocate* (sperando che le successive scritture di quel blocco siano fatte direttamente in cache, basandosi sul principio di località), mentre le cache di tipo *write-through* utilizzano spesso l'opzione *no write allocate* (poiché le successive scritture indirizzate verso quel blocco saranno effettuate ancora in memoria).

Le tipologie di utilizzo delle memorie cache sono come

- Cache Dati
- Cache Istruzioni
- Cache Unificate o miste che possono contenere sia dati sia istruzioni

Le cache separate (istruzioni + dati) possono essere ottimizzate autonomamente differenziando capacità, dimensioni dei blocchi, grado di associatività, ecc. per ottenere prestazioni migliori. Le statistiche indicano che le cache istruzioni hanno un *miss rate* più basso delle cache dati.

Si possono distinguere tre cause di fallimento di accesso alle memorie cache:

1. Obbligatorietà (*Compulsory*): durante il primo accesso un blocco non è presente nella cache e deve esservi trasferito: si tratta dei cosiddetti “fallimenti di partenza a freddo” (*cold start misses*) o “fallimenti di primo accesso” (*first reference misses*), chiaramente indipendenti dalla dimensione della cache.
2. Capacità (*Capacity*): se la cache non può contenere tutti i blocchi necessari all'esecuzione di un programma, alcuni blocchi devono essere scartati e ricaricati successivamente. Fallimenti di questo tipo diminuiscono al crescere della dimensione della cache.
3. Conflitto (*Conflict*): se la cache è di tipo set-associativo o ad indirizzamento diretto ci sono fallimenti causati dai blocchi che bisogna scartare e recuperare più tardi in conseguenza del fatto che più blocchi devono essere caricati nello stesso insieme: nascono così fallimenti per collisione (*collision misses*). Anche questi diminuiscono all'aumentare dell'associatività.

Una regola empirica (detta *regola 2:1*) definita su un grande campione di esperimenti sulle memorie cache stabilisce che il *miss rate* di una cache a indirizzamento diretto di dimensione N è circa uguale al miss rate di una cache set-associativa a 2 vie di dimensione $N/2$.