Integrazioni sull'architettura con cache

Queste note integrano la trattazione delle architetture con cache, rispetto al Cap. VIII sez. 3 e al Cap. X sez. 8, studiando più a fondo la *cache secondaria* e valutato il suo impatto sulle prestazioni dei programmi.

Sommario

1 Caratteristiche della cache a due livelli 1
1.1 Blocchi-C1 e blocchi-C2 1
1.2 Cache a due livelli "inclusive" *vs* "victim" 2
1.3 Architettura firmware 3
1.3.1 Sottosistema di memoria interallacciata 4
1.3.2 Parallelismo tra la gerarchia M-C2 e la gerarchia C2-C1 4
1.3.3 Latenza di strutture pipeline 5
2 Cache secondaria su domanda 7
2.1 Esempio 1: fault consecutivi appartenenti allo stesso blocco-C2 7
2.2 Esempio 2: fault consecutivi appartenenti a blocchi-C2 distinti 8
3 Cache secondaria con prefetching 9

1 Caratteristiche della cache a due livelli

La gerarchia cache secondaria – cache primaria sarà anche detta *cache a due livelli* C2-C1.

Come studiato nel Cap. VIII sez. 3.4, assumeremo la presenza di cache secondaria C2 all'interno del chip CPU. Ricordiamo come la sua utilità (rispetto a un unico livello di cache) sia legata alla possibilità di operare in parallelo con la cache primaria C1 e con l'esecuzione delle istruzioni, con l'obiettivo di procurare informazioni in anticipo rispetto a quando siano richieste da C1.

1.1 Blocchi-C1 e blocchi-C2

Indicheremo con σ_1 e σ_2 le ampiezze dei blocchi di cache primaria (cache dati) e di cache secondaria rispettivamente, con σ_2 multiplo intero di σ_1:

$$\sigma_2 = q\ \sigma_1$$

Ad esempio, σ_1 = 8 – 16 e σ_2 = 128 – 256. Indicheremo con γ_1 e γ_2 le capacità dei due livelli di cache, ad esempio γ_1 = 32K – 64K e γ_2 = 512K – 1M.

Per brevità chiameremo *blocco-C1* un blocco di ampiezza σ_1 e *blocco-C2* un blocco di ampiezza σ_2. Ovviamente C1 contiene solo blocchi-C1, mentre C2 contiene γ_2/ σ_2 blocchi-C2 e γ_2/ σ_1 blocchi-C1.

La traduzione da indirizzi di memoria principale a indirizzi di cache secondaria nell'unità C2 riguarda blocchi-C2. I blocchi trasferiti da memoria principale (o cache terziaria) a C2 sono sempre *blocchi-C2*, ognuno dei quali contiene q blocchi-C1.

Quando avviene una richiesta da C1 a C2, in seguito a fault di C1,

- il blocco-C1 può già essere presente in C2, e in tal caso ha luogo direttamente il trasferimento da C2 a C1 con latenza $2\ \sigma_1\ \tau$,
- oppure viene generato anche un fault di C2, che provoca il trasferimento in C2 del blocco-C2 contenente il blocco-C1 richiesto da C1. Come sfruttare il trasferimento del blocco-C2 per servire la richiesta corrente di C1 e quelle successive sarà esaminato nelle prossime sezioni.

Rispetto a quanto contenuto nel Cap. VIII sez. 3.4 è importante precisare che, poiché un blocco-C2 contiene q blocchi-C1, viene sempre implicitamente attuata una forma di *prefetching nei confronti dei blocchi-C1*, anche se C2 opera *su domanda*. Ad esempio, se i blocchi-C1 sono riferiti sequenzialmente dal programma, la lettura di un blocco-C2, il cui primo blocco-C1 è quello richiesto da C1, fa sì che i successivi (fino a $q-1$) blocchi-C1 che C1 richiederà in seguito siano già presenti in C2.

Se diciamo che C2 opera *con prefetching*, intendiamo che il prefetching è applicato esplicitamente ai *blocchi-C2*. È come se venisse attuato, nei confronti dei blocchi-C1, un "doppio prefetching" che, *nei casi più favorevoli,* può rendere trascurabile l'impatto dei livelli di memoria superiori a C2.

1.2 Cache a due livelli "inclusive" *vs* "victim"

Una cache a due livelli C2-C1 si dice *inclusive* se vale la proprietà che, se un blocco-C1 è presente in C1 allora è presente anche in C2; in altri termini, i blocchi presenti in C1 sono un sottoinsieme dei blocchi-C1 presenti in C2.

Una cache a due livelli non inclusive è chiamata *victim*: con questa modalità è possibile che un blocco-C1 sia presente in C1 ma non in C2.

Le due modalità, inclusive e victim, danno luogo a modi diversi di gestire i blocchi nei due livelli di cache. Sostanzialmente, la modalità victim tende a rendere maggiormente indipendenti la gestione dei blocchi di C2 da quella di C1.

Nella modalità inclusive, un blocco-C1 può essere deallocato da C1 senza che questo evento abbia ripercussioni sulla gestione di C2. Invece, la deallocazione di un blocco-C2 in C2 deve provocare la deallocazione in C1 di tutti i blocchi-C1 appartenenti a quel blocco-C2: ciò richiede una certa cooperazione tra le due unità cache in termini di informazioni scambiate e di informazioni di stato associate ai blocchi.

Il controllo "non deallocare", usato per sfruttare riuso, viene effettuato sia in C1 che in C2 con modalità inclusive, ma è preferibile che venga effettuato (best-effort) anche con modalità victim.

Se le scritture sono gestite con il metodo Write-Through, sappiamo che ogni singola scrittura in C1 viene propagata anche a C2 e a tutti i livelli superiori: ciò è sempre vero se C2 è inclusive, mentre con modalità victim ovviamente ciò avviene solo se il blocco-C1 è presente anche in C2 (il controllo viene effettuato dall'unità C2).

Per architetture uniprocessor, la distinzione tra le due modalità non è particolarmente rilevante, ed è stata citata soprattutto a scopo di conoscenza tecnologica. La distinzione è invece rilevante nei

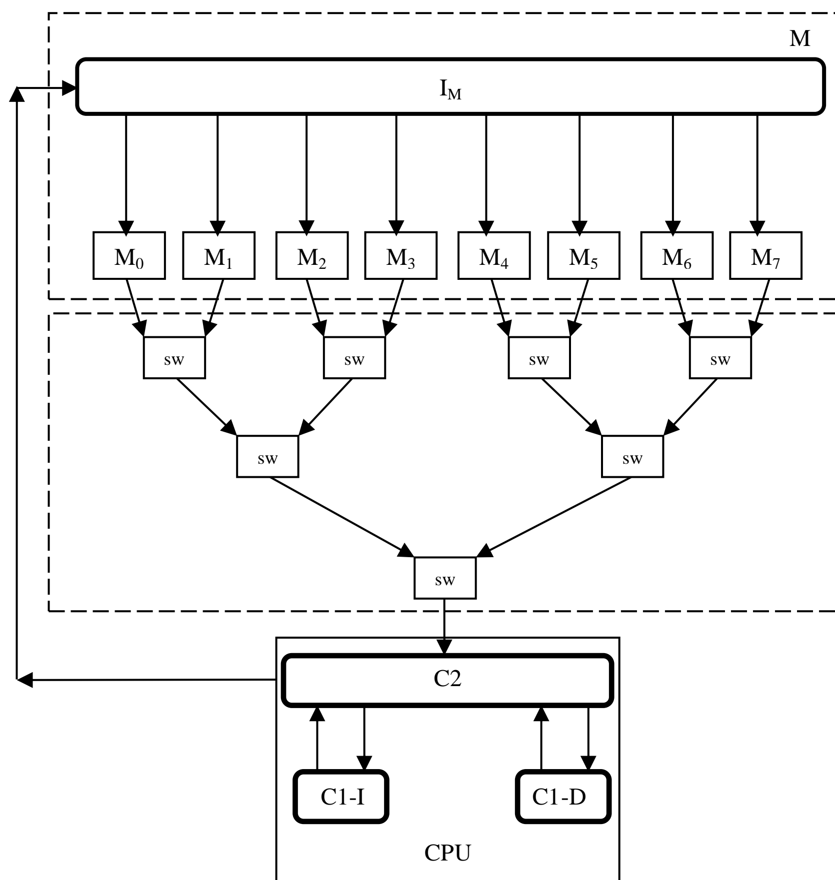
multiprocessor in relazione al problema della cache coherence di blocchi condivisi. Senza perdere in generalità, ai nostri scopi possiamo assumere una qualunque delle due modalità, ponendo il vincolo che un blocco-C2 non venga deallocato in C2 se esiste almeno un *suo* blocco-C1 attualmente presente in C1 (in C2 viene mantenuto anche lo stato di allocazione dei blocchi-C1).

La seguente tabella riporta le caratteristiche di alcuni processori commerciali, tutti disponibili in versione multicore:

	C1	C2	Inclusive vs Victim	C3
AMD Opteron	64K	512K	V	6M
Intel Sandy Bridge	32K	256K	I	20M
Intel Itanium	16K	256K	I	6M
IBM Power 7	32K	256K	V	4M
ARM Cortex A9	32K	2M	I	–

1.3 Architettura firmware

Una possibile struttura complessiva di un sistema con cache a due livelli è mostrata nella figura seguente:



La comunicazione tra C1 e C2, all'interno del chip, avviene con semplici collegamenti dedicati e modalità domanda-risposta: C1 richiede a C2 (usando l'indirizzo fisico di M) un blocco-C1 e, se il blocco-C1 è presente in un blocco-C2 già allocato, C2 risponde inviando a C1 uno stream di σ_1

parole con tempo di interpartenza 2τ (nelle ipotesi del Cap. VIII sez. 3.4). La latenza del trasferimento vale quindi $2\,\sigma_1\,\tau$, essendo ogni parola scritta nella memoria di C1 in un singolo ciclo di clock in parallelo alla successiva lettura in C2.

La comunicazione tra M, interallacciata con *m* moduli, e C2 avviene attraverso una struttura di interconnessione di grado limitato, allo scopo di minimizzare il numero di bit (pin-count) dell'*interfaccia di memoria esterna* della CPU.

Come visto nel Cap. X sez. 8.3, una struttura ad albero permette di ottenere, per la lettura di blocchi, la stessa *banda* di una struttura con *m* collegamenti dedicati

$$B_M = B_{M-max} = \frac{m}{\tau_M}$$

con un aumento di *latenza* di ordine soltanto logaritmico rispetto alla soluzione teorica con *m* collegamenti dedicati.

Questa soluzione rende il modello dei costi compatibile con quello delle architetture multiprocessor facenti uso di reti di interconnessione di grado limitato (come butterfly, mesh, fat tree) e con i moderni/previsti sottosistemi di I/O (Infiniband).

1.3.1 Sottosistema di memoria interallacciata

I moduli di memoria sono interfacciati alla CPU dall'unità indicata con I_M, la quale ha il compito di interpretare le richieste dalla CPU per ottimizzare l'uso della struttura interallacciata.

In particolare, ricevendo un indirizzo fisico base con richiesta di *lettura blocco-C2*, I_M provvede a comandare la lettura di σ_2 parole: per ogni blocco di *m* indirizzi consecutivi, richiede in parallelo ad ognuno degli *m* moduli la lettura di una parola allo stesso indirizzo.

Per semplicità, e senza perdere in generalità, nel seguito assumeremo che

$$m = \sigma_1$$

In tale ipotesi, per la lettura di un blocco-C2, I_M provoca la generazione di uno stream di *q* blocchi-C1, ognuno dei quali è ricevuto da C2 come uno stream di σ_1 parole, senza soluzione di continuità rispetto al successivo stream di σ_1 parole.

Inoltre, la richiesta dalla CPU a I_M può essere di

- *scrittura singola*: la richiesta contiene anche indirizzo fisico e parola di dato, che vengono instradati da I_M al modulo il cui identificatore è contenuto nell'indirizzo;
- *scrittura di blocco-C1* o *di blocco-C2*: il primo messaggio contiene indirizzo fisico base del blocco e prima parola di dato, ed è seguito da uno stream di $\sigma_1 - 1$ o $\sigma_2 - 1$ parole di dato che completano il blocco-C1 o il blocco-C2 rispettivamente. Le parole sono inviate da I_M ai moduli in round-robin con tempo di interpartenza uguale al tempo di interarrivo T_A a I_M stesso (essendo certamente il tempo di servizio ideale di I_M minore di T_A), quindi il tempo di interarrivo a ogni modulo è uguale a $m\,T_A$.

1.3.2 Parallelismo tra la gerarchia M-C2 e la gerarchia C2-C1

Torniamo alla lettura di un blocco-C1 in seguito a fault di C1. Studiamo il comportamento dell'unità C2 nel caso in cui C1 richieda un blocco-C1 e questo non sia allocato in C2 (fault anche di C2). L'unità C2 determina facilmente la posizione (diciamo *i*, con $0 \le i \le q - 1$) del blocco-C1 all'interno del blocco-C2 che C2 richiede a M. Quando le parole del blocco-C1 *i*-esimo sono

ricevute dall'unità C2, questa le scrive nella propria memoria e, in parallelo, ognuna di esse è inviata immediatamente all'unità C1 senza attendere la conclusione del trasferimento dell'intero blocco-C2.

È frequente (anche se non l'unico) il caso in cui $i = 0$, che corrisponde all'utilizzo di una struttura dati a partire da un indirizzo base che sia uguale alla base di un blocco-C2. In molti casi il compilatore può forzare tale allineamento ("padding" di strutture dati).

Ancora più importante, successive richieste (fino a $q-1$ richieste) di blocchi-C1 appartenenti al blocco-C2 e, nel frattempo, già copiate in C2 sono servite semplicemente con un trasferimento da C2 a C1 (di latenza $2\ \sigma_1\ \tau$)

Si ha quindi un certo (in molti casi un sostanziale) parallelismo tra C2 e le altre unità della CPU (in particolare C1), che rappresenta una delle principali motivazioni dell'esistenza della cache secondaria separata dalla primaria.

La conseguenza favorevole sulle prestazioni è la possibilità di minimizzare/ridurre la penalità dovuta ai fault di C2.

Nel seguito assumeremo sempre che *C1 operi su domanda*. Distingueremo invece il caso di *C2 operante su domanda* da quello di *C2 con prefetching*, con la precisazione alla fine della sez. 1.1.

Nel valutare il modello dei costi, ci concentreremo sulle penalità dovute alle latenze di trasferimento blocchi. Per questa ragione i dati degli esempi di programmi saranno caratterizzati solo da località.

1.3.3 Latenza di strutture pipeline

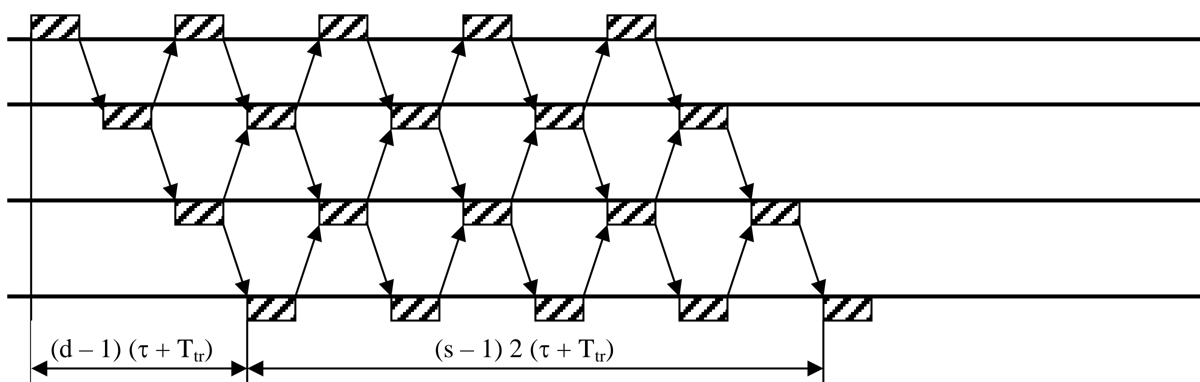
Invece di usare le formule della latenza di trasferimento blocco del Cap. X sez. 8.3, in presenza di cache a due livelli è più comodo utilizzare l'espressione generale che esprime la latenza di una struttura di unità di elaborazione operanti in pipeline. La formula si ricava come segue.

Consideriamo un pipeline costituito da d unità comunicanti mediante collegamenti dedicati aventi latenza di trasmissione T_{tr}.

Supponiamo che tutte le unità abbiano tempo di servizio interno uguale a τ. Come nel Cap. X sez. 8.3, indichiamo con $T_{hop} = \tau + T_{tr}$ la latenza di un passo di "hop" attraverso la struttura (dall'interfaccia di uscita di uno stadio all'interfaccia d'ingresso dello stadio successivo).

Sia s la lunghezza dello stream.Supponiamo che *lo stream sia generato dal primo stadio*.

Il funzionamento temporale della struttura pipeline, con comunicazioni a *singola bufferizzazione*, è mostrato nella figura seguente (per d = 4, s = 5) che mette in evidenza le singole comunicazioni RDY-ACK:



Come si vede la latenza è data da:

$$L\ (s) = (2s + d - 3)\ T_{hop}$$

Se le comunicazioni sono a *doppia bufferizzazione* (o se possono essere realizzate "elettronicamente" in maniera dipendente dal tempo senza ACK), con un ragionamento analogo si ricava:

$$L\ (s) = (s + d - 2)\ T_{hop}$$

ottenendo un sensibile miglioramento in quando la lunghezza dello stream viene pagata una sola volta.

Se lo stream è generato da un sottosistema a monte con tempo di interarrivo T_A, allora le formule si generalizzano in:

$$singola\ bufferizzazione\colon L\ (s) = (2s + d - 3)\max(T_{hop}, T_A)$$

$$doppia\ bufferizzazione\colon L\ (s) = (s + d - 2)\ \max(T_{hop}, T_A)$$

in quanto se gli stadi non sono collo di bottiglia il tempo di servizio vale T_A.

Come primo esempio, applichiamo la formula con doppia bufferizzazione al trasferimento di un blocco-C1 in un sistema con *sola cache primaria*, cioè nelle stesse condizioni del Cap. X sez. 8.3 e inserendo anche l'unità I_M come nella figura precedente.

La richiesta dalla CPU a I_M ha latenza T_{hop}, così come la latenza della richiesta da I_M a M. La lettura in parallelo di un blocco di $m = \sigma_1$ parole dalla memoria interallacciata ha latenza τ_M. A questo punto viene innescato un pipeline dall'interfaccia di uscita dei moduli di memoria fino a C1. La lunghezza dello stream è

$$s = \sigma_1$$

Gli stadi del pipeline sono: *1)* l'insieme delle interfacce di uscita dei moduli di memoria, *2)* i *livelli* della struttura ad albero, *3)* l'unità C1 (eventualmente potremmo considerare ulteriori unità, come l'interfaccia di memoria esterna se costituita da un'unità distinta dalla cache), quindi la *distanza* (lunghezza del pipeline) *in questo caso* vale:

$$d = 2 + \log_2 \sigma_1$$

da cui la latenza del trasferimento da M a CPU:

$$L\ (\sigma_1) = (\sigma_1 + \log_2 \sigma_1)\ \max(\ T_{hop}, \frac{\tau_M}{\sigma_1}\ )$$

Ad esempio, con $T_{hop} = 10\tau$, $\tau_M = 30\tau$ e $\sigma_1 = 8$, T_{hop} prevale sul tempo di servizio τ_M/σ_1 della memoria interallacciata.

Complessivamente, tenendo conto anche della comunicazione da CPU a I_M e dell'accesso ai moduli di memoria:

$$T_{trasf}(\sigma_1) = 2\ T_{hop} + \tau_M + L\ (\sigma_1)$$

Per $T_{hop} \geq \tau_M/\sigma_1$:

$$T_{trasf}(\sigma_1) = \tau_M + (\sigma_1 + \log_2 \sigma_1 + 2)\ T_{hop}$$

che fornisce valori del tutto confrontabili con la formula del Cap. X sez. 8.3 (che non considerava la presenza di I_M e le comunicazioni CPU-I_M), ma risulta di uso più generale nei casi di interesse per le architetture con cache a più livelli.

2 Cache secondaria su domanda

In presenza di cache a due livelli C2-C1 la penalità dovuta ai fault di cache primaria si può valutare come:

$$T_{fault} = N_{fault-C2}\, T_{trasfM-C2} + N_{fault-C1}\, T_{trasfC2-C1}$$

Grazie al fatto che un blocco-C2 contiene q blocchi-C1, una certa percentuale di (al limite tutti i) successivi fault di C1 potranno essere serviti da C2 senza ulteriori trasferimenti da M oppure senza pagare l'intera latenza di un trasferimento da M: questo finché non si verifichi un nuovo fault di C2, in corrispondenza del quale occorre trasferire da M in C2 un blocco-C1 senza parallelismo con la CPU.

A partire dalla richiesta successiva di blocco-C1 verrà di nuovo esplicitato, in tutto o in parte, parallelismo, e così via.

La caratterizzazione del parallelismo tra M, C2 e C1, e quindi la valutazione di $T_{trasfM-C2}$, $T_{trasfC2-C1}$, varia da programma a programma.

In questa sede non tenteremo di ricavare delle formule generali, ma mostreremo come utilizzare i concetti e le tecniche della sezione precedente per ricavare il modello dei costi programma per programma.

Vediamo alcune classi di esempi con caratteristiche diverse.

2.1 Esempio 1: fault consecutivi appartenenti allo stesso blocco-C2

Consideriamo il seguente semplice programma:

int A[M], B[M];

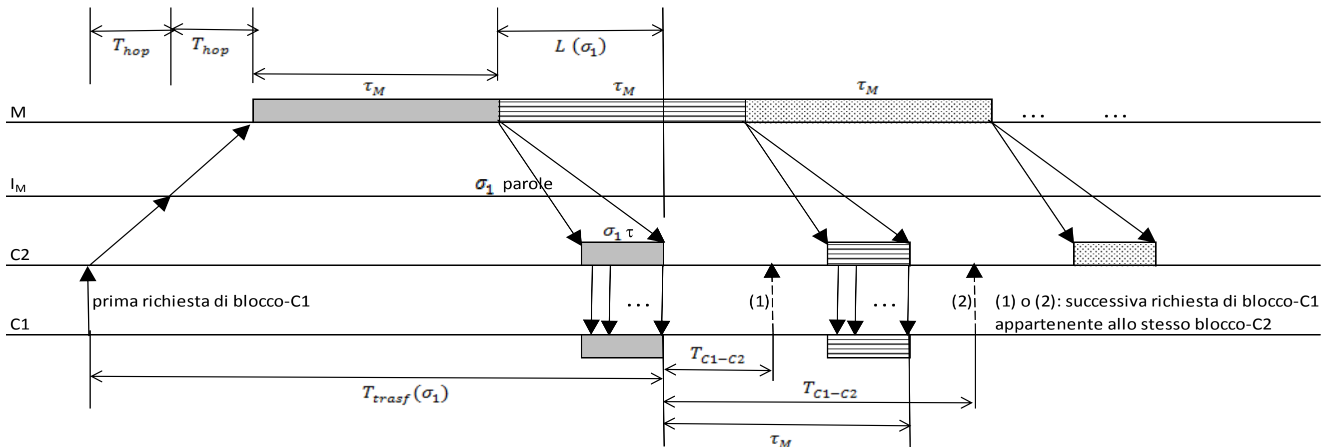
$\forall\ i = 0 .. M-1$:

B[i] = F(A[i])

In questo caso:

$$N_{fault-C2} = \frac{M}{\sigma_2} \quad , \quad N_{fault-C1} = \frac{M}{\sigma_1}$$

I fault su A danno luogo al funzionamento temporale schematizzato nella figura seguente:



In base a quanto spiegato nella sez. 1.3.2, la copia del primo blocco-C1 in C2 *e* in C1 richiede:

$$T_{trasfM-C2} = 2\, T_{hop} + \tau_M + L\ (\sigma_1)$$

grazie al parallelismo delle scritture, parola per parola, in C2 e in C1. Nell'esempio specifico, questa è tutta la penalità pagata *per ogni fault di C2*.

Consideriamo l'intervallo di tempo T_{C1-C2} tra la fine della memorizzazione del blocco-C1 in C1 e la successiva richiesta da C1 a C2. Nell'esempio si tratta del tempo di servizio per iterazione del loop in assenza di fault di cache. Come si vede dalla figura, se:

$$T_{C1-C2} \geq \tau_M$$

(caso 2) allora il nuovo blocco-C1 richiesto è già presente in C2, per cui viene pagata solo la latenza del trasferimento da C2 a C1:

$$T_{trasfC2-C1} = 2\,\sigma_1\,\tau$$

Se invece (caso 1 – calcolo *F* di "grana" relativamente fine):

$$T_{C1-C2} < \tau_M$$

il blocco-C1 richiesto non è ancora presente in C2: esso deve essere atteso e le sue parole saranno copiate in parallelo in C2 *e* in C1 (come è accaduto con il primo blocco-C1). In questo caso la latenza vale:

$$T_{trasfC2-C1} = \tau_M - T_{C1-C2}$$

(A rigore $N_{fault\text{-}C1}$ andrebbe decrementato di $N_{fault\text{-}C2}$).

In conclusione, abbiamo visto come, anche per una cache secondaria su domanda, una opportuna progettazione dell'unità C2 (invio a C1 delle parole del blocco-C1 richiesto in parallelo alle loro scritture nella memoria di C2) permetta di anticipare il trasferimento da M di blocchi-C1 rispetto alle richieste di C1, pagando in più solo una singola latenza $T_{trasfM-C2}$ (σ_1) per ogni fault di C2.

2.2 Esempio 2: fault consecutivi appartenenti a blocchi-C2 distinti

Consideriamo la seguente variante dell'esempio 1:

int A[M], B[M], C[M];

$\forall$ *i = 0 .. M – 1 :*

C[i] = G(A[i], B[i])

In questo caso:

$$N_{fault-C2} = 2\,\frac{M}{\sigma_2} \quad , \quad N_{fault-C1} = 2\,\frac{M}{\sigma_1}$$

Supponiamo che il riferimento ad A[i] sia seguito dal riferimento a B[i], quindi ogni fault su A è immediatamente seguito da un fault su B. L'unità C2, dopo aver trasferito da M il primo blocco-C1 di A (in parallelo viene trasferito anche in C1), non riceve richieste per A, bensì riceve la prima richiesta di blocco-C1 di B. Questa rimarrà in attesa che venga completato il trasferimento di tutti i blocchi-C1 del blocco-C2 di A. Quindi avrà luogo il trasferimento di tutti i blocchi-C1 di B in C2 (il primo anche in C1). Solo da questo punto in poi C2 potrà prendere in considerazione le richieste dei successivi blocchi-C1 di A e di B, che saranno tutti già presenti in C2, finché non si verificherà un nuovo fault di C2; dopo di che il funzionamento descritto si ripeterà.

Si ha quindi (modificando opportunamente il diagramma temporale precedente):

$$T_{trasfM-C2} = 2\,T_{hop} + 2\,q\,\tau_M \sim 2\,q\,\tau_M$$

in quanto $L(\sigma_1) < \tau_M$ (le scritture nella memoria di C2 sono sovrapposte ai trasferimenti) e la latenza da CPU a M per la richiesta di B è mascherata, e

$$T_{trasfC2-C1} = 2\,\sigma_1\,\tau$$

in quanto tutti i blocchi-C1 richiesti da C1 sono già presenti in C2, eccetto i primi due la cui valutazione rientra in $T_{trasfM-C2}$ (a rigore $N_{fault-C1}$ andrebbe decrementato di 2 $N_{fault-C2}$).

Gli esempi 1 e 2 mostrano il modo di ragionare su casi più articolati:

- in casi come quello dell'esempio 1, per ogni fault di C2 più blocchi-C1 del blocco-C2 trasferito sono richiesti da C1 senza essere inframmezzati da richieste per altri blocchi-C2;
- in casi come quello dell'esempio 2, si ha un "burst" di richieste di blocchi-C1 che appartengono a blocchi-C2 distinti, ciò che obbliga l'unità C2 a leggere tutti questi blocchi-C2 prima di esaudire le successive richieste di blocchi-C1 (a meno di non applicare politiche sofisticate, e generalmente poco efficienti, per mischiare i trasferimenti di blocchi-C2 distinti).

3 Cache secondaria con prefetching

Come sappiamo dal Cap. VIII, la tecnica del prefetching, che a prima vista appare come la panacea per la località, in realtà è di delicata applicazione. In generale, occorre che il programma indichi esplicitamente la relazione tra il blocco corrente e il blocco successivo. Di regola, quando sia implementato, il prefetching viene applicato nel caso più semplice in cui i due blocchi sono consecutivi. Proprio per distinguere chiaramente questo caso da altre situazioni, in cui l'applicazione del prefetching al blocco successivo risulterebbe solo dannosa, il set di istruzioni dovrebbe includere istruzioni speciali o, come in D-RISC, annotazioni opportune.

Nel nostro caso supporremo che la cache primaria operi solo su domanda e che la tecnica del prefetching sia *applicata alla cache secondaria*, quindi le annotazioni riguardano C2.

È anche importante notare come tutte le valutazioni di prestazioni, in presenza di prefetching, considerino solo le strutture dati visibili al programma e non quelle del supporto a tempo di esecuzione del processo, a diverse delle quali può essere utile applicare prefetching. Queste ultime comportano, rispetto alla valutazione ottimistica, un certo overhead di tempo e di spazio, che trascureremo per semplicità.

Consideriamo l'***esempio 1*** della sez. 2.1. Una volta richiesto il primo blocco-C1, C2 legge uno stream di M/σ_2 blocchi-C2, cioè uno stream di M/σ_1 blocchi-C1 senza soluzione di continuità, nel contempo servendo in parallelo le richieste di C1. Quindi, trascurando il trasferimento del primo blocco-C1 da M a C2, nel modello dei costi possiamo porre:

$$N_{fault-C2} \sim 0$$

e valutare la sola penalità dei trasferimenti da C2 a C1, *ognuno* dei quali ha latenza

$$T_{trasfC2-C1} = 2\,\sigma_1\,\tau \qquad se\ T_{C1-C2} \geq \tau_M$$

$$T_{trasfC2-C1} = \tau_M - T_{C1-C2} \quad se\ T_{C1-C2} < \tau_M$$

Consideriamo ora l'***esempio 2*** della sez. 2.2. L'unità C2 legge, senza soluzione di continuità, blocchi-C2 di A alternati a blocchi-C2 di B. La penalità è ancora prevalentemente quella dei soli trasferimenti da C2 a C1, *ognuno* dei quali ha latenza:

$$T_{trasfC2-C1} = 2\,\sigma_1\,\tau$$

eventualmente aggiungendo, *una volte per tutte*, la latenza del trasferimento in C2 del primo blocco-C2 di A e del primo blocco-C2 di B:

$$T_{trasfM-C2} \sim 2\,q\,\tau_M$$