Memoria Cache

G. Lettieri

12 Marzo 2023

1 Introduzione

La memoria centrale è molto più lenta del processore. Possiamo rendercene conto scrivendo un programma che accede ripetutamente agli elementi di un array, misurando quanti cicli di clock sono in media necessari per ogni accesso. La Figura 1 mostra i risultati ottenuti su un processore Intel Core i7-12700 con clock massimo a 4.9 GHz e memoria DDR4 a 3200 Hz, per varie dimensioni dell'array. Si vede che quando l'array è più grande di 25 MiB, gli accessi in memoria possono richiedere circa 70 nanosecondi. Si tratta di un tempo enorme rispetto alla frequenza della CPU. Ricordiamo che il processore è in grado potenzialmente di completare una istruzione per ogni ciclo di clock, quindi circa una istruzione ogni quarto di nanosecondo. Ma le istruzioni si trovano in memoria: se per prelevare una istruzione sono necessari 70 nanosecondi, è del tutto inutile avere un processore velocissimo, in quanto per la stragrande maggioranza del tempo starebbe fermo ad aspettare la prossima istruzione.

In Figura 1, però, notiamo anche che le operazioni di lettura sembrano richiedere molto meno tempo se l'array è sufficientemente piccolo. Per esempio, per array più piccoli di 48 KiB sembra essere sufficiente poco più di un nanosecondo. Questo è l'effetto della memoria *cache*, che ora vogliamo studiare.

2 La memoria Cache

Esistono diverse tecnologie che permettono di costruire una memoria ad accesso casuale. In generale, è possibile costruire memorie grandi ed economiche, ma lente, oppure memorie piccole e veloci, ma costose. Noi vorremmo invece avere memorie grandi, economiche e veloci. Queste non possiamo ottenerle direttamente, ma possiamo avere qualcosa di equivalente usando contemporaneamente i due tipi di memoria e sfruttando alcune caratteristiche dei programmi. Queste caratteristiche prendono il nome di *principi di località*, e sono i seguenti:

- *località spaziale*: se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo accederà ad un indirizzo vicino;
- *località temporale*: se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo vi accederà di nuovo.

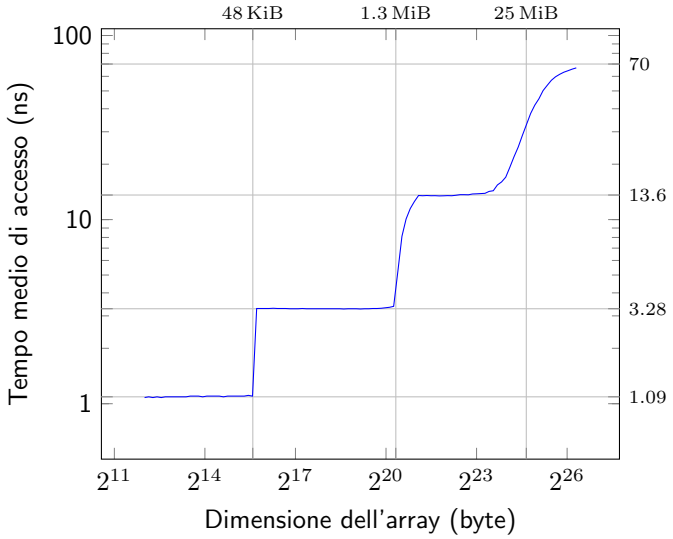

Figura 1: Numero medio di nanosecondi per ogni accesso in memoria per un programma che accede ciclicamente a tutti gli elementi di un array di dimensione variable (Intel Core i7-12700, DRAM DDR4 3200).

Questi sono principi puramente statistici che i programmi tipicamente rispettano. Per esempio, le istruzioni sono eseguite per lo più in sequenza, quindi il prelievo di una istruzione ad un certo indirizzo è molto spesso seguito dal prelievo della successiva (quindi ad un indirizzo vicino). La presenza di cicli nel programma comporterà il prelievo ripetuto delle stesse istruzioni. Analoghe considerazioni valgono anche per i dati, in quanto i programmatori li organizzano spesso in strutture dati in cui le variabili usate insieme si trovano vicine, e vi accedono ripetutamente. Quindi, anche se un programma può avere complessivamente bisogno di molta memoria, se lo osserviamo per un intervallo di tempo sufficientemente breve vedremo che si concentra su una parte molto più piccola (magari diversa man mano che l'esecuzione procede). Se riuscissimo a scoprire qual è questa parte e copiarla nella memoria veloce, il nostro programma potrebbe essere eseguito molto più velocemente.

L'idea è di realizzare la memoria centrale con la tecnologia lenta, ma economica, in modo da poterla avere molto grande. Allo stesso tempo aggiungiamo al sistema una memoria cache come illustrato in Figura 2. La cache è composta da un *controllore cache* e dalla memoria cache vera e propria, realizzata con la tecnologia costosa, ma veloce. La memoria cache è dunque molto più piccola della memoria centrale.

Il controllore intercetta tutte le operazioni di lettura e scrittura nello spazio di memoria eseguite dal processore. Per ogni operazione controlla prima se il dato a cui il processore vuole accedere si trova in memoria cache, nel qual caso l'accesso può essere completato velocemente; altrimenti esegue l'operazione in

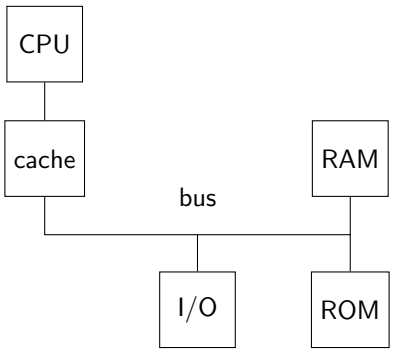

Figura 2: Architettura generale di un calcolatore con memoria cache.

memoria al posto del processore e mantiene una copia del dato in cache (in quanto si spera che il processore lo chiederà di nuovo, per il principio di località temporale). All'inizio la memoria cache non conterrà niente, ma a poco a poco si riempirà con le locazioni a cui il programma sta accedendo. Se la memoria cache si riempie, il controllore dovrà decidere quali locazioni tenere e quali eliminare (*rimpiazzare*), sempre cercando di mantenere quelle che è più probabile che vengano richieste in seguito. Si noti che né il controllore cache, né il processore conoscono le locazioni che verranno richieste in futuro: il controllore vede solo le operazioni che il processore genera e il processore vede una sola istruzione per volta. Le decisioni del controllore sono dunque euristiche.

Tipicamente il controllore legge dalla memoria più di quanto il processore ha chiesto, sia per sfruttare il principio di località spaziale, sia per altri motivi che vedremo tra breve. L'unità di trasferimento utilizzata dal controllore cache è detta *cacheline*. Un esempio tipico è di avere cacheline di 64 byte, allineate naturalmente. Per esempio, se il processore esegue una operazione di lettura all'indirizzo 8, il controllore trasferirà dalla memoria tutti i byte che vanno da 0 a 63 (cioè tutta la cacheline che contiene la locazione richiesta dal processore).

Si noti che tutto quanto abbiamo detto si svolge completamente in hardware, nel controllore cache, ed è trasparente al software: i programmi possono essere scritti ignorando che la cache esista e funzioneranno lo stesso. In presenza della cache, però, verranno in genere eseguiti più velocemente.

Anche il processore può ignorare completamente la presenza della cache, nel senso che non sono richieste modifiche al suo funzionamento, purché disponga di un modo per variare la lunghezza delle operazioni di lettura e scrittura in memoria. Questo perché tali operazioni richiederanno tempi molto diversi, a seconda che il dato richiesto si trovi in cache o debba essere prelevato dalla memoria. A tale scopo è sufficiente che il processore possieda un piedino di ingresso tramite il quale il controllore cache può segnalare quando l'operazione si è conclusa, oppure quando deve essere prolungata rispetto ad un tempo di default.

Si noti infine che il meccanismo della cache non ha alcun senso per le ope-

razioni di I/O, in quanto queste operazioni hanno effetti collaterali che non devono essere cancellati: se un programma vuole leggere il prossimo carattere battuto sulla tastiera è necessario interpellare la tastiera. Non avrebbe alcun senso re-inviare al processore sempre lo stesso carattere conservato in cache. Il controllore cache, dunque, farà passare inalterate tutte le operazioni di lettura e scrittura nello spazio di I/O. Questo però non basta: si ricordi che le interfacce di I/O possono anche avere i loro registri mappati nello spazio di memoria. Per disabilitare la cache anche durante gli accessi a queste interfacce è necessario dunque operare una discriminazione anche in base all'indirizzo usato dal processore, ma per vedere come fare dovremo aspettare di aver introdotto la paginazione.

3 Cache ad indirizzamento diretto

La memoria cache è, dal punto di vista funzionale, una normale memoria ad accesso casuale: le operazioni possibili sono sempre quelle di lettura e scrittura, eseguite una per volta, ognuna ad un certo *indirizzo di cache*.

Il controllore cache intercetta le operazioni di lettura e scrittura del processore. Ognuna di queste è relativa ad un certo indirizzo di memoria. Si ricordi che il processore genera l'indirizzo di una certa parola quadrupla allineata naturalmente, che abbiamo chiamato *numero di riga*, usando poi i fili di byte enable per selezionare i byte all'interno della riga selezionata. La memoria cache sarà organizzata nello stesso modo, così che le operazioni di lettura e scrittura del processore si mappino facilmente su quelle della cache.

Il controllore, però, lavora con unità più grandi, dette cacheline, che per esempio possono essere di 8 parole quadruple (64 byte), allineate naturalmente. Il controllore può dunque scomporre il numero di riga generato dal processore in una parte meno significativa detta *offset* (di 3 bit se le cacheline sono di 8 parole quadruple) e nel rimanente numero di cacheline. Poiché il controllore trasferisce sempre intere cacheline, il numero di cacheline è sufficiente per determinare se la locazione richiesta dal processore è in cache oppure no.

Dato il numero di cacheline, il controllore cache deve essere in grado di sapere se la corrispondente cacheline di memoria è stata precedentemente copiata in cache e, in caso affermativo, a quale indirizzo di cache. Un modo per realizzare questo meccanismo è di avere una funzione hash da numeri di cacheline a indirizzi di cache. Nelle cache a *indirizzamento diretto* questa funzione è estremamente semplice (e dunque veloce): l'indirizzo di cache (detto *indice*) è dato dai bit meno significativi del numero di cacheline. In particolare, se la cache è grande 2^a cacheline, l'indice sarà di a bit. Si noti che tutti i numeri di cacheline che distano tra loro 2^a cacheline avranno lo stesso indice, e dunque vorrebbero essere memorizzate nella stessa posizione della cache (si dice che c'è un conflitto tra le due cacheline). Il controllore deve sapere quale tra le tante cacheline che potrebbero trovarsi ad un certo indirizzo di cache è quella effettivamente caricata. Queste diverse cacheline si distinguono per la parte del numero di cacheline che non è utilizzata nell'indice e che è detta *etichetta*. Il controllore può

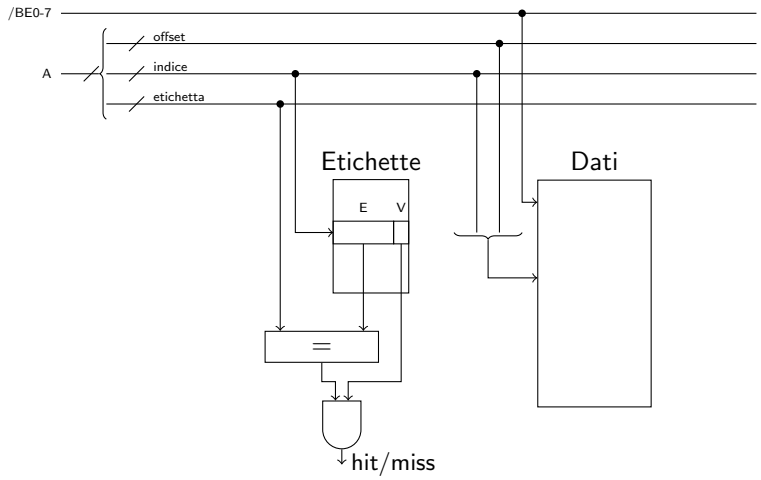

Figura 3: Cache a indirizzamento diretto.

utilizzare una memoria aggiuntiva, detta memoria delle etichette (o *tag*), in cui memorizzare, per ogni indice, l'etichetta della cacheline caricata a quell'indice. Il controllore deve anche sapere a quali indici non è stata caricata ancora alcuna cacheline. Poiché le memorie contengono sempre qualcosa e qualunque sequenza di bit potrebbe essere una valida etichetta, è necessario memorizzare un bit in più per ogni indice, detto *bit di validità*, che valga 1 se e solo se l'etichetta (e dunque la cacheline) contenuta a quell'indice è significativa. All'inizio tutti i bit di validità saranno a 0; passeranno a 1 man mano che il controllore cache caricherà cacheline in risposta agli accessi in memoria del processore. Lo schema è dunque quello di Figura 3.

Per ogni operazione di lettura in memoria da parte del processore, le operazioni svolte dal controllore cache saranno le seguenti:

- se la cacheline è presente (*read hit*), completa la lettura con i dati presenti in cache; per leggere dalla memoria cache dati la parola quadrupla richiesta dal processore è sufficiente usare come indirizzo l'indice e l'offset;
- se la cacheline non è presente (*read miss*), il controllore deve leggere la cacheline della memoria, scriverla nella cache dati a partire dalla posizione dettata dall'indice, e aggiornare l'etichetta; a questo punto i dati sono in cache e si procede come nel caso precedente.

Si noti che, in caso di miss, una eventuale cacheline che si trovava in cache allo stesso indice di quella appena caricata verrà rimpiazzata. Questo è il difetto principale delle cache ad indirizzamento diretto: due cacheline che hanno lo stesso indice non possono essere mantenute contemporaneamente in cache, anche se il resto della cache è vuota.

Per le operazioni di scrittura abbiamo diverse opzioni.

- se la cacheline è assente (*write miss*), il controllore può completare la scrittura in memoria senza caricare la cacheline in cache (*write no-allocate*) oppure caricare la cacheline in cache (*write allocate*) e proseguire come in una *write hit*;
- se la cacheline è presente (*write hit*), il processore può aggiornare solo la cache (*write back*) o sia la cache che la memoria (*write through*).

Si noti che nel caso di *write back* la cacheline dovrà essere riscritta in memoria prima di essere rimpiazzata da un'altra cacheline, perché altrimenti le scritture eseguite dal programma andrebbero perse. La tecnica è comunque conveniente se il programma riesce ad eseguire tante scritture sulla stessa cacheline prima che questa debba essere scritta, in quanto si riduce il numero di scritture in memoria. Per evitare scritture inutili, inoltre, il controllore cache può associare ad ogni cacheline un bit *dirty* e porlo a 1 quando il processore ha eseguito almeno una scrittura in quella linea. Le cacheline che hanno il bit dirty a zero non hanno bisogno di essere riscritte in memoria. Le cache moderne sono tipicamente write-allocate e write-back.

La memoria delle etichette è un costo aggiuntivo ed è bene che sia sufficientemente piccola. Notiamo che abbiamo bisogno di una etichetta per ogni cacheline della memoria dati. Se usiamo cacheline più grandi ne riduciamo il numero, e quindi riduciamo anche la dimensione della memoria delle etichette. Questo è il motivo principale per cui la cacheline contiene più byte di quanti ne può richiedere il processore in una singola operazione. D'altro canto, la cacheline non deve essere nemmeno troppo grande, altrimenti leggere o scrivere una cacheline dalla memoria richiederebbe troppo tempo. La dimensione di 64 byte è un buon compromesso tra queste due esigenze contrastanti.

In presenza di cache non è più il processore ad accedere alla memoria centrale, ma sempre il controllore cache. Dal momento che questo trasferisce sempre cacheline intere, si può ottimizzare questo tipo di trasferimento. In genere si introduce un nuovo tipo di operazione sul bus, specifico per la lettura/scrittura di cacheline. La memoria RAM può essere organizzata in modo tale che, mentre è in corso il trasferimento di una riga, inizi a la lettura della riga successiva all'interno della stessa cacheline. Inoltre, il numero di linee dati può essere reso indipendente dal numero di linee dati del processore[1]. In questo modo il costo della lettura/scrittura di una cacheline è reso inferiore alla somma dei costi dei trasferimenti delle singole righe che compongono la cacheline.

4 Cache associative ad insiemi

Le cache associative ad insiemi sono più costose di quelle ad indirizzamento diretto, ma permettono di alleviare il problema dei conflitti. L'idea è di permettere la memorizzazione in cache di più di una cacheline per ogni indice. Una cache associativa ad insiemi che permette di memorizzare n cacheline per ogni

[1]Per esempio, il Pentium era un processore a 32 bit, ma il bus dati era a 64 bit per velocizzare il trasferimento delle cacheline.

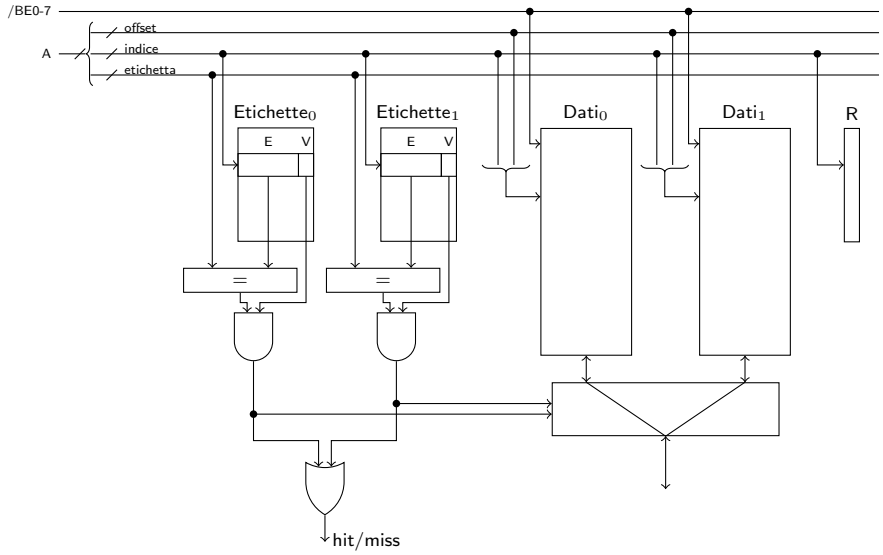

Figura 4: Cache associativa a insiemi (a 2 vie).

indice è detta a n vie. In Figura 4 è mostrato lo schema di una cache associativa a 2 vie. Al suo interno ci sono due repliche di una cache ad indirizzamento diretto, collegate in parallelo. Ogni cacheline ha sempre un indice, ma questo ora individua due possibili locazioni: una in Dati_0 e una in Dati_1. Ogni cacheline può essere memorizzata indifferentemente nell'una o nell'altra. Ogni indice, dunque, individua un *insieme* di possibili locazioni (di cache) per la cacheline.

Si avrà un hit se una delle vie dell'insieme contiene la cacheline cercata, da cui la porta OR che riceve le uscite delle due porte AND. La cacheline a cui accedere dipenderà ovviamente da quale via ha prodotto l'hit. (Si noti che non è possibile che entrambe le vie producano hit, in quanto una cacheline viene caricata solo se non è già presente, e all'inizio la cache è vuota.) In Figura 4 questo è indicato sommariamente dal circuito che si trova sotto le due memorie Dati.

In caso di miss, il controllore può scegliere quale delle due cacheline rimpiazzare. La politica più usata in questo caso è la LRU (Least Recently Used): si rimpiazza la cacheline che non è acceduta da più tempo. Questa politica è quella che si comporta meglio nella maggior parte dei casi, anche se non è ottima in assoluto (in alcuni casi è anzi la peggiore). Purtroppo la politica matematicamente ottima richiede la conoscenza di tutti gli accessi futuri e non è realizzabile in pratica.

La memoria R in Figura 4 contiene le informazioni necessarie ad implementare la politica LRU. Con solo due vie è sufficiente ricordare, per ogni indice, la via riferita dall'ultimo accesso: la via da rimpiazzare in caso di miss sarà evidentemente l'altra. Con più di due vie sarebbe però necessario ricordare

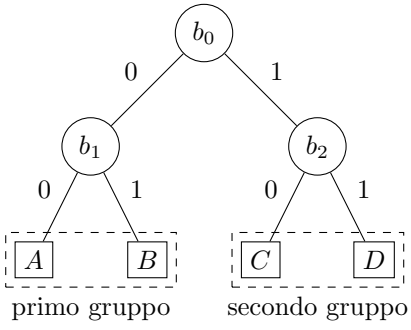

Figura 5: Scelta della via da rimpiazzare nell'algoritmo pseudo-LRU.

l'ordine degli ultimi accessi di ogni via, e aggiornarlo ad ogni accesso, operazione che potrebbe essere troppo costosa. In pratica conviene utilizzare politiche approssimate, se non addirittura effettuare un rimpiazzamento casuale.

La cache dell'80486 era a 4 vie e realizzava un interessante algoritmo che approssima LRU ed è molto semplice da realizzare. L'algoritmo si chiama Pseudo-LRU e richiede 3 bit per ogni insieme, chiamiamoli b_0, b_1 e b_2. Chiamiamo A, B, C e D le quattro vie. L'algoritmo raggruppa le vie a due due: A e B da una parte e C e D dall'altra. Il bit b_0 dirige la scelta sul primo gruppo o sul secondo gruppo:

- se $b_0 = 0$ la via da rimpiazzare è una tra A e B. La scelta tra le due è decisa dal valore di b_1, ignorando b_2:
  - se $b_1 = 0$ si sceglie A;
  - se $b_1 = 1$ si sceglie B;
- se $b_0 = 1$, invece, la scelta è tra C e D (secondo gruppo) ed è decisa dal valore di b_2, ignorando b_1:
  - se $b_2 = 0$ si sceglie C;
  - se $b_2 = 1$ si sceglie D;

Si veda anche la Figura 5. Ad ogni accesso vanno aggiornati due bit, lasciando inalterato il terzo, in base alla via interessata dall'accesso. L'idea è che la via appena acceduta deve diventare l'ultima ad essere rimpiazzata, così come accadrebbe nell'algoritmo LRU:

- accesso a A: $b_0 \to 1$ e $b_1 \to 1$;
- accesso a B: $b_0 \to 1$ e $b_1 \to 0$;
- accesso a C: $b_0 \to 0$ e $b_2 \to 1$;
- accesso a D: $b_0 \to 0$ e $b_2 \to 0$.

Il fatto di lasciare sempre inalterato uno tra b_1 o b_2 permette di ricordare l'ordine relativo tra le vie del gruppo non interessato dall'accesso. D'altro canto, la via che si trova nello stesso gruppo di quella interessata dall'accesso può ora trovarsi più in alto in coda (grazie al valore scritto in b_0) rispetto a dove sarebbe stata nel vero algoritmo LRU (da qui l'approssimazione).

Si noti inoltre che, se la memoria R è organizzata in tre banchi indipendenti da un bit ciascuno, ogni aggiornamento può essere implementato con una operazione di scrittura in due dei tre banchi, senza la necessità di dover prima eseguire una operazione di lettura.

L'algoritmo di Pseudo-LRU può essere esteso a cache con più di 4 vie ricorrendo ad alberi più profondi, ma può essere anche ulteriormente approssimato, per esempio eliminando i livelli più bassi dell'albero e sostituendoli con scelte casuali.

A parità di dimensione complessiva della cache, aumentando il numero di vie diminuisce il numero di insiemi. Nel caso estremo avremo una cache con un solo insieme, nel qual caso si parla di cache *completamente associativa*. Una cache completamente associativa lascia piena libertà sul piazziamento delle cacheline all'interno della cache, eliminando così i conflitti. Ovviamente si tratta di una soluzione costosa che è praticabile solo in cache molto piccole.

A LRU e gli accessi ciclici

Cerchiamo di capire meglio cosa succede in Fig. 1. La politica LRU ha questo difetto: nel caso in cui il programma acceda ciclicamente ad un numero di cacheline superiore, anche di una sola cacheline, a quelle che possono essere contenute in cache, LRU causa una miss ad *ogni* accesso. Possiamo facilmente convincerci di questo simulando a mano LRU su una piccola cache completamente associativa, diciamo di 4 cacheline, provando ad accedere ripetutamente alle cacheline numero 1, 2, 3, 4, e 5. Anche se è più complicato da vedere, la stessa cosa accade anche con Pseudo-LRU. Questo difetto può essere usato per scoprire la dimensione di una cache che usi questi algoritmi. Concentriamoci sul punto di Fig. 1 in cui la dimensione dell'array attraversa il valore 48 KiB: per valori immediatamente inferiori il tempo di accesso medio è 1.09 nanosecondi, mentre per valori immediatamente superiori è 3.28. Questa è una forte indicazione della presenza di una cache di 48 KiB con rimpiazzamento LRU o Pseudo-LRU: sotto i 48 KiB tutte le cacheline si trovano in cache (dopo il primo ciclo), mentre dopra i 48 KiB *nessuna* cacheline si trova mai in cache (LRU l'ha sempre rimpiazzata poco prima che il programma vi accedesse). La dimensione delle cache può essere confermata con vari strumenti—per es., in Linux su Intel/AMD, con il comando `lscpu --cache`. La CPU Intel Core i7-12700 contiene in effetti tre livelli di cache, con dimensioni 48 KiB, 1.3 MiB e 25 MiB, che corrispondono ai punti in cui il grafico di Fig 1 ha dei salti (più o meno smussati).

Il comando `lscpu` ci permette anche di sapere che la prima cache è in realtà associativa a 12 vie, mentre la seconda e la terza sono associative a 10 vie. Per la prima cache avremmo potuto scoprire il numero di vie osservando più preci-

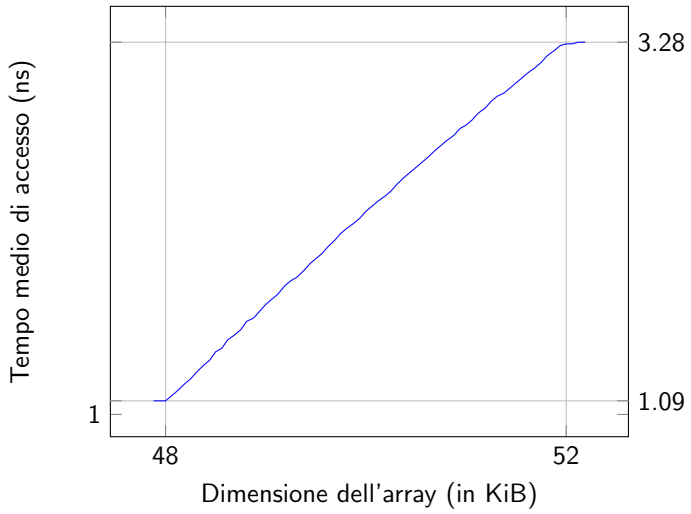

Figura 6: Ingrandimento di Fig 1 nei pressi del primo salto.

samente cosa accade intorno al primo salto. La Fig 6 mostra il grafico di Fig. 1 ingrandito intorno a quel punto, con gli assi non più in scala logaritmica: vediamo che l'aumento del tempo di accesso medio è in realtà lineare. Come mai questo accada è presto detto, se teniamo a mente che (Pseudo-)LRU si applica ad ogni *insieme* indipendentemente dagli altri. Quando l'array è grande 48 KiB può essere ancora contenuto completamente nella prima cache e tutti gli accessi si risolvono lì. Il tempo medio di accesso misurato è dunque interamente dovuto alla prima cache. Pensiamo ora a cosa accade se aumentiamo la dimensione dell'array di una cacheline. L'insieme in cui la cacheline dovrebbe essere caricata, chiamiamolo i, è sicuramente pieno (tutti gli insiemi sono pieni), quindi è necessario un rimpiazzamento. In pratica, 13 cacheline dell'array vorrebbero andare tutte nello stesso insieme i, che però ne può contenere solo 12—diciamo che i è "sovraffollato". Il programma accede ciclicamente a tutto l'array, e dunque accede ciclicamente anche a queste 13 cacheline: (Pseudo-)LRU causerà dunque una miss (nell'insieme i) ogni volta che il programma accede ad una di queste 13 cacheline, ma non quando accede alle altre. Osserveremo allora che il tempo medio di accesso è aumentato leggermente. Quando la dimensione dell'array è 48 KiB + 128 byte entra in gioco una nuova cacheline, che andrà a causare un sovraffollamento su un altro insieme, e così via. Ogni nuova cacheline aumenta il tempo medio di accesso di un valore costante, fino a quando tutti gli insiemi saranno sovraffollati e quindi tutti gli accessi causeranno miss. A quel punto il tempo medio di accesso che misuriamo sarà interamente dovuto alla seconda cache, come se la prima non ci fosse.

Siccome la cache ha $v = 12$ vie, gli insiemi sono grandi ciascuno $v \times 64$ byte

e la prima cache contiene

$$n = \frac{48 \times 1024}{v \times 64} = 64$$

insiemi. Tutti gli insiemi saranno sovraffollati quando l'array raggiunge la dimensione di

$$48\,\mathrm{KiB} + n \times 64\,B = 52\,\mathrm{KiB},$$

come confermato in Fig 6. Se non avessimo saputo il numero di vie avremmo potuto ricavarlo da queste relazioni, osservando i punti in cui il grafico si appiattisce.

I salti intorno a 1.3 MiB e 25 MiB in Fig. 1 non rispettano questo modello. Ciò può essere dovuto ad una implementazione approssimata di Pseudo-LRU, oppure (soprattutto per il secondo salto) all'effetto di altre cache che vedremo in seguito (TLB).