Esecuzione speculativa

G. Lettieri

10 Giugno 2025

Nell'architettura costruita fin'ora, l'emissione di una istruzione di salto causa lo stallo dello pipeline nello stadio di emissione, fino a quando l'istruzione non è stata completata e la destinazione del salto non è nota. Questo ci permette di evitare le alee dovute alle diependenze sul controllo, al costo di uno stallo per ogni istruzione di salto. Il BTB (Branch Target Predictor) è ancora presente, e permette agli stadi di prelievo e decodifica di continuare a lavorare anche mentre è in esecuzione una istruzione di salto, eventualmente annullando il lavoro fatto nel caso in cui la previsione si rivelasse sbagliata.

Osserviamo che gli stalli possono durare anche molti cicli di clock: si pensi al caso in cui l'istruzione di salto dipende per i dati da una precedente `ld` il cui operando sorgente non è in cache. Ha dunque senso cercare di ottimizzare anche questo tipo di dipendenze. L'idea è di non mettere in stallo lo stadio di emissione mentre è in corso l'esecuzione di una istruzione di salto, ma di continuare ad emettere le istruzioni che nel frattempo vengono prelevate e decodificate dall'indirizzo predetto dal BTB. Ovviamente, deve essere possibile annullare l'effetto di queste istruzioni quando il BTB sbaglia previsione. La tecnica che permette tutto ciò è detta *esecuzione speculativa*.

1 Stadio di ritiro

L'idea fondamentale è di introdurre un nuovo stadio di *ritiro* dopo lo stadio di esecuzione. Gli effetti delle istruzioni eseguite sono resi permanenti solo nello stadio di ritiro, e fino ad allora possono sempre essere annullati. Inoltre, mentre le istruzioni possono essere eseguite fuori ordine nello stadio di esecuzione, sono sempre ritirate in ordine. L'ordine è quello in cui le istruzioni sono state emesse, che è l'ordine originario nel flusso di esecuzione. In questo modo, quando viene ritirata una istruzione di salto, lo stadio di ritiro può verificare se la predizione del BTB era corretta e, in caso contrario, annullare tutte le istruzioni successive e comunicare allo stadio di prelievo l'indirizzo corretto da cui iniziare nuovamente a prelevare.

Lo stadio di ritiro usa un Reorder Buffer (ROB), organizzato come una coda FIFO. Lo stadio di emissione inserisce le istruzioni emesse in coda al ROB, oltre a copiarle in una stazione di prenotazione. Ogni entrata del ROB ha un flag che dice quando l'istruzione è stata completata. Man mano che le istruzioni

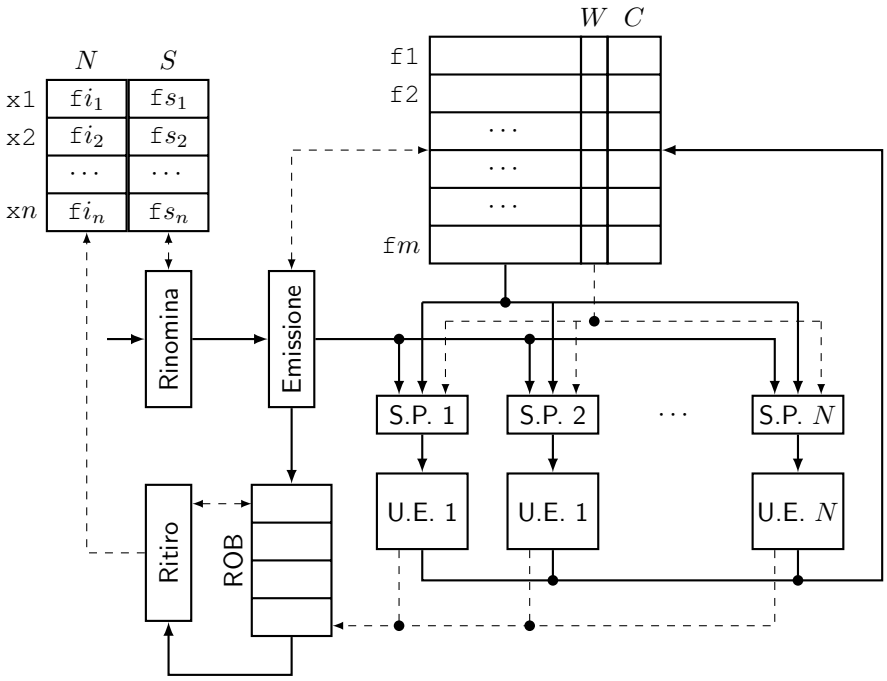

Figura 1: Esecuzione speculativa

vengono completate, in qualsiasi ordine, aggiornano il loro flag nel ROB. Lo stadio di ritiro monitora la testa del ROB e, quando vede che l'istruzione in testa è stata completata, la ritira. Se si tratta di una istruzione operativa, rende permanente il suo effetto (per esempio, scrittura in un registro). Se si tratta di una istruzione di salto predetta correttamente, non fa altro (le istruzioni successive sono state prelevate correttamente). Infine, come abbiamo detto, se si tratta di una istruzione di salto predetta in modo errato, svuota il ROB e fa ripartire lo stadio di prelievo dall'indirizzo corretto.

Vediamo ora come si possono rendere permanenti o annullare gli effetti delle istruzioni operative e di accesso alla memoria.

1.1 Istruzioni operative

Per quanto riguarda le istruzioni operative, l'unico effetto che modifica lo stato è la scrittura nel registro di destinazione. Qui possiamo sfruttare la distinzione, già introdotta, tra registri logici e registri fisici. Per ogni registro logico x ricordiamo *due* registri fisici, invece di uno solo:

- un registro fisico *speculativo* $S(x)$, che contiene, o conterrà, il risultato dell'ultima istruzione emessa e non ancora ritirata che ha x come destinazione;
- un registro fisico *non speculativo* $N(x)$, che contiene il risultato dell'ultima istruzione ritirata che aveva x come destinazione.

Supponiamo che lo stadio di rinomina dei registri riceva l'istruzione "*op* src_1, src_2, *dst*. Per prima cosa sostituisce src_1 e src_2 con i loro registri speculativi $S(src_1)$ e $S(src_2)$; quindi sceglie un nuovo registro fisico f non utilizzato e lo fa diventare il nuovo registro speculativo di *dst* ($S(dst)$ diventa f), e infine sostituisce *dst* con f. L'informazione di quale fosse *dst* viene mantenuta in campi aggiuntivi dell'istruzione, e resta disponibile nel ROB. Se l'istruzione viene ritirata, lo stadio di ritiro rende permanente l'effetto dell'istruzione facendo diventare f il registro non speculativo di *dst* ($N(dst)$ diventa f). Se invece l'istruzione viene annullata, lo stadio di ritiro aggiorna la tabella dello stadio di rinomina, in modo che il registro speculativo di *dst* ridiventi il vecchio registro non speculativo ($S(dst)$ diventa $N(dst)$).

1.2 Istruzioni che accedono alla memoria

Per le operazioni che scrivono in memoria (`st`), sfruttiamo la presenza dello store buffer, che avevamo dovuto introdurre per evitare le alee causate dalle dipendenze sui nomi. Ci basta imporre che le scritture non possano essere effettivamente eseguite, e restino dunque nello store buffer, fino a quando l'istruzione non è stata ritirata. Lo stadio di ritiro, in caso di istruzione di salto predetta male, oltre a svuotare il ROB svuoterà anche lo store buffer.

Le istruzioni `ld` non presentano (apparentemente) problemi, e possiamo lasciarle accedere alla memoria anche prima che vengano ritirate. Anzi, uno

dei vantaggi dell'esecuzione speculativa è proprio quello di poter anticipare le istruzioni di lettura, in modo che eventuali cacheline mancanti possano essere prelevate in anticipo rispetto a quando il programma ne avrà effettivamente bisogno.

1.3 Gestione delle eccezioni

Supponiamo che una istruzione generi una eccezione mentre si trova nello stadio di esecuzione. Per esempio, una istruzione di accesso in memoria causa un page fault, o una istruzione di divisione scopre che il divisore è zero. L'eccezione non può essere gestita immediatamente, perché il processore non sa se l'istruzione andava davvero eseguita. Per rimediare a questo problema è sufficiente scrivere che l'istruzione ha causato una eccezione nella corrispondente entrata del ROB. Sarà lo stadio di ritiro, quando e se l'istruzione verrà ritirata, a far partire la gestione dell'eccezione. La gestione dell'eccezione è simile alla gestione di un salto predetto male: tutte le istruzioni successive vengono annullate e il prelievo ripartirà dall'indirizzo della routine che gestisce l'eccezione.

Negli Intel, come sappiamo, la gestione di una eccezione è una operazione piuttosto complicata, che deve svolgere diversi compiti (accesso alla IDT, eventuale cambio pila, salvataggio in pila di 5 parole quadruple, etc.). È probabile che, in caso di fault, lo stadio di ritiro immetta nello stadio di esecuzione una sequenza di microistruzioni che svolgono tutte le operazioni necessarie; l'ultima di queste microistruzioni sarà un salto all'indirizzo della routine di gestione dell'eccezione, letta dalla IDT.

2 Vulnerabilità della speculazione

Agli inizi di Gennaio 2018, sono state rivelate una serie di vulnerabilità dei processori, note come Meltdown e Spectre, che hanno mostrato come la tecnica della speculazione nasconda importanti insidie che possono compromettere il meccanismo di protezione del processore. Da allora ne sono state scoperte molte altre, non solo nei processori Intel. Alcuni di queste vulnerabilità sono causate da difetti di progettazione che possono essere corretti, mentre altre (come quelle del genere Spectre) sembrano essere causate da un diffetto intrinseco nell'idea stessa della speculazione.

La domanda che ci dobbiamo porre è questa: è vero che *tutti* gli effetti di una istruzione possono essere cancellati, nel caso di speculazione errata? La risposta è quasi sempre no. Consideriamo, per esempio, una istruzione `ld`: se l'istruzione ha causato una miss in cache, la cache caricherà la corrispondente cacheline e questa non verrà poi rimossa, nel caso in cui l'istruzione venisse in seguito annullata. I progettisti di processori, fino al 2018, sembrano aver lavorato assumendo che ciò non avesse importanza, ma gli autori di Meltdown e Spectre hanno dimostrato il contrario. È molto semplice, da programma, capire se una data cacheline è in cache oppure no, perché i tempi di accesso nei due casi sono molto diversi: pochi cicli di clock contro centinaia. Questa differenza può

essere misurata, in quanto i processori possiedono dei timer con la precisione di qualche nanosecondo, e comunque la misura può essere resa robusta ripetendo l'esperimento tante volte.

Prendiamo in considerazione Meltdow, che è una vulnerabilità del primo tipo (difetto di progettazione). L'idea è di provare ad accedere alla memoria del kernel da livello utente. L'accesso causerà un page fault, che però verrà gestito solo quando l'istruzione arriverà in testa al ROB. Nel frattempo, il processore completerà comunque l'accesso (questo è il difetto di progettazione) e riuscirà ad eseguire diverse altre istruzioni. Queste istruzioni verranno annullate nel momento in cui lo stadio di ritiro riconoscerà il fault, ma i loro effetti in cache resteranno, e il programma potrà poi osservarli. Attenzione: il processo che esegue il programma dovrebbe essere poi terminato a causa del fault, ma in vari sistemi operativi (compresi Linux e Windows) i processi utente possono chiedere al kernel di associare dei propri gestori alle eccezioni, e in questo modo possono continuare ad eseguire anche dopo un fault. L'attaccante, quindi, può osservare lo stato della cache prodotto dalle istruzioni eseguite speculativamente dopo l'accesso vietato. Quali istruzioni potrebbe mettere l'attaccante dopo l'accesso? Osserviamo il seguente frammento:

```
xorl %rax, %rax
mov indirizzo_proibito, %al
shl $6, %rax
mov buf(%rax), %rbx
```

L'attaccante legge un byte dall'indirizzo proibito e poi accede ad un indirizzo che dipende dal byte letto, in particolare all'indirizzo `buf` $+ b \times 64$[1], dove b è il valore letto. Sostanzialmente, accede ad una cacheline diversa per ogni possibile valore del byte segreto. Prima di eseguire questo codice, l'attaccante avrà dichiarato un buffer `buf` di 256×64 byte, e si sarà assicurato di rimuoverlo dalla cache (ci sono istruzioni per farlo). Dopo l'attacco, proverà ad accedere ad ogni cacheline coperta da `buf`, misurando il tempo richiesto da ogni accesso. Se tutto va bene, 255 accessi richiederanno centinaia di cicli di clock (cache miss) e solo uno richiederà pochi cicli di clock (cache hit): questa deve essere la cacheline a cui il processore aveva acceduto durante l'esecuzione speculativa. Supponiamo che l'accesso in questione sia relativo alla i-esima cacheline: vuol dire che il byte letto all'indirizzo proibito valeva i. L'attaccante è riuscito a leggere un byte dalla memoria del kernel; ora può incrementare `indirizzo_proibito` e leggere il successivo, e un byte alla volta riuscirà a leggere tutta lo spazio di indirizzamento riservato al kernel. Cosa troverà mappato in questo spazio? Fino al 2018, Linux aveva una finestra FM, proprio come il nostro nucleo didattico, e quindi in quello spazio di indirizzamento era mappata tutta la memoria fisica, in cui erano contenuti i frame di tutti i processi. Sfruttando Meltdown, un processo di un utente avrebbe potuto leggere la memoria dei processi degli altri utenti.

[1] In realtà è necessario moltiplicare b per 4096 in modo da accedere a pagine diverse. Questo serve a disattivare un meccanismo che non abbiamo visto (il prefetch), che altrimenti confonderebbe le misure.

In risposta a Meltdown, gli sviluppatori di Linux fecero in modo di rimuovere la finestra FM ogni volta che il processore gira a livello utente, per poi ripristinarla ogni volta che il kernel riprende il controllo. Questo, ovviamente, ha un costo in termini di TLB miss.