Contents

Sistemi Operativi Avanzati

- Versione originale: Matteo Fanfarillo
- Umile formattazione: Simone Festa

Hardware Insights

Introduzione

In generale, non è possibile dire che lo stato di un'applicazione sia semplicemente dato dal "puzzle" degli stati dei componenti software sviluppati. Di fatto, quando mandiamo in esercizio tali componenti software, stiamo generando dei cambi di stato al livello hardware che concorrono a determinare qual è lo stato effettivo del nostro sistema. Tra l'altro, alcuni dei cambi di stato al livello hardware possono non essere voluti o specificati dal programmatore. Il fatto che l'esecuzione di moduli software sviluppati a qualsiasi livello impatti sui moduli sottostanti (e quindi sull'hardware), come vedremo, può rappresentare un problema importante per quanto riguarda la sicurezza: talvolta, per ottenere un sistema più sicuro e performante a livello hardware, sarà necessario ristrutturare l'applicazione software.

Ma quali sono le entità che si frappongono tra ciò che viene specificato dal programmatore e ciò che realmente avviene nel sistema?

- Il **compilatore**: il programma compilato è un oggetto molto complesso che può interfacciarsi direttamente con l'hardware; il compilatore può decidere ad esempio di inserire particolari istruzioni macchina secondo uno specifico ordine in modo completamente trasparente al programmatore.
- Le **hardware run-time decisions**: una volta che è stato generato il flusso esatto delle istruzioni macchina da eseguire per mandare in esercizio l'applicazione, l'hardware in realtà può prendere delle decisioni a run-time su come gestire tale flusso. A parità di istruzioni macchina, processori di vendor diversi possono prendere delle decisioni a run-time differenti.
- La **disponibilità** (o l'assenza) **di specifiche features dell'hardware**.

In pratica, si ha una sorta di non-determinismo dell'hardware e, quindi, del software.

Esempio: Se implementiamo l'algoritmo del Panificio di Lamport senza l'ausilio di librerie di sistema (e quindi senza l'ausilio di spinlock o semafori), l'algoritmo a un certo punto della sua esecuzione si romperà. Infatti, le macchine moderne (a meno che non siano single core, dove non è assicurata consistenza globale) non garantiscono una visione consistente per tutti i thread di ciò che sta succedendo in memoria. Prima, ad ogni cpu veniva associato un flusso di esecuzione, cosa non vera per i sistemi moderni.

Scheduling e parallelismo nell'architettura di Von Newman

L'architettura di calcolatore più semplice a cui siamo stati abituati a pensare è quella di **Von Newman**, ed è caratterizzata da:

- Un'unica CPU.
- Un'unica memoria.
- Un unico flusso di controllo (*fetch – execute – store*).
- Transizioni nell'hardware time-separated: le istruzioni devono essere eseguite tutte una alla volta.
- Stato della memoria ben definito all'inizio di ciascuna istruzione, la quale quindi deve poter vedere la memoria in uno stato coerente con l'esecuzione delle istruzioni precedenti.

La maniera moderna di pensare le architetture non è basata sull'idea di seguire il flusso di *esecuzione esattamente così com'è* stato codificato nel programma, bensì è basata sul concetto di **scheduling** (e.g. dell'utilizzo dei componenti hardware) che, alla fine della fiera, porterà a eseguire un flusso di esecuzione equivalente al flusso codificato nel programma (gli stati intermedi possono essere diversi, possono cambiare di run in run, non sono deterministici, ma influenzabili da fattori esterni come la temperatura). In particolare, lo scheduling dovrebbe consentire di eseguire più cose in **parallelo**, anche in contesti con un'unica CPU, un'unica memoria e un unico flusso di controllo. Per quanto riguarda lo scheduling, ne esistono diverse tipologie. A livello **hardware** abbiamo:

- Scheduling delle istruzioni all'interno di un singolo program flow.
- Scheduling delle istruzioni in program flow paralleli (**speculativi =** non so se l'esecuzione sia corretta o meno, ad esempio la branch condition non sempre lo è. Il processore sceglie come propagare l'update, non è imposto. Ho garantita l'equivalenza software, non ciò che avviene dentro).
- Propagazione dei valori tra i componenti hardware del sistema.

A livello software invece abbiamo:

- Scheduling dei thread da assegnare alle CPU / ai CPU-core.
- Scheduling delle attività da eseguire sull'hardware (e.g. gli interrupt).
- Supporti di sincronizzazione tra thread software-based.

Anche per quanto riguarda il **parallelismo**, ne esistono diverse tipologie:

- A livello hardware si parla di **ILP** (**Instruction Level Parallelism**), che consiste nell'impiegare le risorse hardware in modo tale da eseguire contemporaneamente istruzioni macchina diverse. (ad esempio posso eseguire al tempo 't' sia A sia B, anche in un unico flusso).
- A livello software, invece, abbiamo il **TLP (Thread Level Parallelism)**, secondo cui un programma può essere pensato come la combinazione di molteplici flussi di esecuzione concorrenti. (Ad esempio ho 3 flussi su 3 processori,singolarmente sarebbero ILP, che cambiano).

Velocità di computazione

È generalmente correlata alla velocità di un processore (espressa in GHz), anche se in realtà esistono istruzioni che possono richiedere un numero arbitrario di cicli di clock a causa di più possibili fattori:

- Possono essere istruzioni più onerose per loro natura.
- Possono dover richiedere a un certo punto una risorsa hardware tuttora occupata da un'altra istruzione, per cui devono rimanere in attesa.
- Possono esservi delle asimmetrie a livello hardware (e.g. un CPU-core può essere più veloce di un altro).
- I pattern per l'accesso ai dati influiscono a loro volta sulle prestazioni: ad esempio, se un dato viene memorizzato in cache, l'accesso a esso sarà più efficiente e viceversa.

Nel corso base di Sistemi Operativi abbiamo parlato di thread CPU-bound e di thread I/O-bound; introduciamo ora una terza categoria di thread (che, di fatto, è una sottocategoria dei CPU-bound): i **memory-bound**. Essi sono dei thread che utilizzano in maniera intensiva la CPU ma, mentre sono in esecuzione, utilizzano in maniera intensiva anche la memoria. I thread (o comunque i programmi) che presentano questa caratteristica possono rappresentare un problema dal punto di vista prestazionale: come riportato dal seguente grafico, il divario prestazionale tra processore e memoria aumenta sempre di più col tempo; questo fenomeno è detto **memory wall**.

Per evitare che i thread memory-bound sperimentino e causino ad altri thread un crollo delle prestazioni, sono necessari dei meccanismi avanzati ad-hoc al livello dell'hardware.

Pipeline

È una **tecnica di scheduling e parallelismo hardware-based**. Infatti:

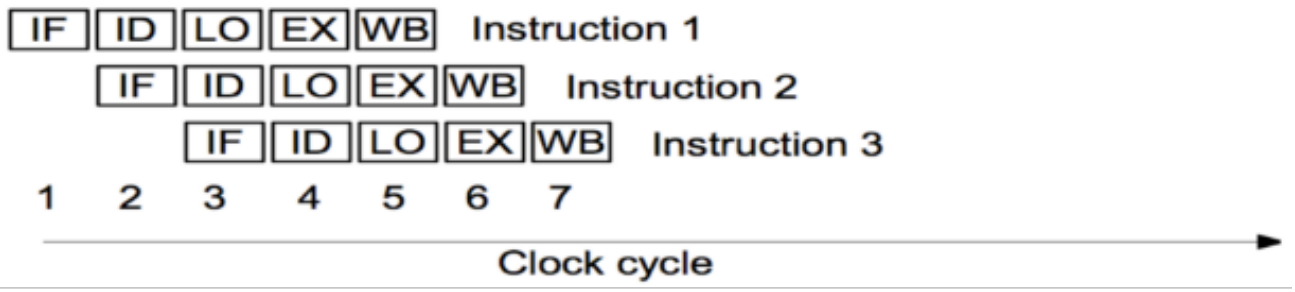
- Non prevede una separazione temporale tra le finestre di esecuzione delle diverse istruzioni: più istruzioni possono essere in esecuzione contemporaneamente (questo è *parallelismo*).
- Le istruzioni che vengono sequenzializzate dal programmatore non sono necessariamente eseguite secondo la stessa sequenza nell'hardware, anche per conseguire la proprietà di parallelismo (questo è *scheduling*).

In ogni caso, ci sono spesso e volentieri coppie di istruzioni (*i1, i2*) in cui *i2*, per essere eseguita, necessita del risultato ottenuto da *i1*. In tal caso, *la causalità deve essere preservata*, per cui *i1* e *i2* non possono essere schedulate in modo del tutto arbitrario. Questo non è altro che un modello **data flow** per l'esecuzione dei programmi.

Ricordiamo che le fasi (stage) delle istruzioni sono:

- **IF** (**Instruction Fetch**): caricamento dell'istruzione nel processore (richiede certamente un accesso in memoria).
- **ID** (**Instruction Decode**): decodifica dell'istruzione, in cui viene stabilito ciò che deve effettivamente essere fatto per eseguire l'istruzione stessa.
- **LO** (**Load Operands**): caricamento degli operandi richiesti per l'esecuzione dell'istruzione (potrebbe richiedere un accesso in memoria).
- **EX** (**Execute**): esecuzione vera e propria dell'istruzione.
- **WB** (**Write Back**): scrittura dell'output dell'istruzione su un registro o in memoria (potrebbe quindi richiedere un accesso in memoria).

Il fatto che un thread o un'applicazione sia memory-bound o meno dipende proprio dagli stage *LO* e *WB*. Inoltre, per permettere l'esecuzione di più istruzioni in parallelo, è necessario che ciascuno *stage coinvolga una componente hardware differente del processore*, in modo tale da non avere collisioni tra più istruzioni che vengono eseguite contemporaneamente. In particolare, un'astrazione di base della pipeline è quella riportata nella seguente figura.



In questo scenario, idealmente, sarebbe possibile completare un'istruzione per ogni ciclo di clock (anche se poi nella realtà non è esattamente così per i motivi esposti nel paragrafo "Velocità di computazione"). Supponiamo comunque di trovarci nello scenario ideale in cui ciascuno stage viene eseguito in un unico ciclo di clock, e supponiamo di voler fornire N risultati (1 per ogni istruzione), di avere L diversi stage per le istruzioni (gli stage sarebbero *IF, ID, LO, EX* e *WB* che devo attraversare) e di avere un ciclo di clock di durata pari a T.

- Senza pipeline si ha un ritardo pari a $N \cdot L \cdot T$
- Con la pipeline si ha un ritardo pari a $(N + L) \cdot T$

Lo speedup è dunque pari a $(N \cdot L)/(N + L)$, che tende a L per N tendente a infinito. Dal punto di vista delle prestazioni, sarebbe magnifico avere un L molto grande (ovvero molti stage diversi per le istruzioni, ovvero riesco a parallelizzare molto di più se aumento il numero di componenti parallelizzabili) ma ciò nella realtà non accade. Il caso estremo richiederebbe avere hardware infinito, visto che ogni stage è un pezzo hw! Nella realtà:

- I processori Pentium prevedevano 5 stage.
- I processori i3 / i5 / i7 prevedono 14 stage.
- I processori ARM-11 prevedono 8 stage.

Questa scelta è dovuta alla necessità di preservare la causalità tra le istruzioni. Più istruzioni si prendono in considerazione insieme, più è probabile che tra tali istruzioni ce ne siano alcune (e.g. *i1, i2*) legate da una relazione di causalità, e sappiamo che *i2*, per essere eseguita, deve attendere che il risultato di *i1* sia pronto; in tal caso, più

L è grande, più la pipeline è lunga, più l'attesa di *i2* sarà lunga. Lo scenario di cui abbiamo appena parlato è una **data dependency**. Oltre a questa esiste anche la **control dependency**, che si può avere nel momento in cui c'è un salto condizionale il cui esito dipende dagli outcome delle istruzioni precedenti al salto. Per quanto concerne la data dependency tra le istruzioni *i1* e *i2*, abbiamo che lo stage **LO** di *i2* tipicamente non può precedere lo stage **WB** di *i1*. Analogamente, per quanto riguarda la control dependency tra le istruzioni *i1*e *i2* (dove *i1* è l'istruzione di salto condizionato), non si conosce l'esito del salto prima della fase **EX*** di *i1*, per cui la fetch di *i2* non dovrebbe precedere lo stage **EX*** di *i1*. Tutte queste condizioni possono comportare dei rallentamenti nell'esecuzione dell'applicazione rispetto al caso ideale di pipeline. Per gestire tali condizioni è possibile ricorrere a svariati meccanismi:

- **Stalli software** (compiler driven): possono essere aggiunti all'interno della pipeline con lo scopo di distanziare due istruzioni *i1*, *i2* in modo tale che uno stage x (e.g. LO) di *i2* venga eseguito dopo uno stage y (e.g. WB) di *i1*.
- **Rischedulazione software** (compiler driven): se devono essere eseguite tre istruzioni *i1, i2, i3* tali per cui *i2* dipende da *i1* ma *i3* non dipende né da *i1* né da *i2*, il compilatore può frapporre *i3* tra *i1* e *i2* in modo tale da svolgere lavoro utile mentre *i2* è in attesa che si completi uno specifico stage di i1.
- **Propagazione hardware**: se l'istruzione *i2* dipende dall'istruzione *i1* e, ad esempio, lo stage LO di *i2* consiste nell'acquisire un valore prodotto dallo stage EX di *i1*, allora è possibile per *i1* propagare il valore a *i2* subito dopo la fase EX senza dover attendere il completamento della write-back.
- **Azzardi hardware supported**: contestualmente a un'istruzione di salto condizionale, il processore può provare a *indovinare* se il salto viene preso o meno per poi anticipare la fetch delle istruzioni successive di conseguenza. Dal punto di vista delle performance, una predizione errata equivale a un inserimento di stalli per attendere passivamente l'esito dell'istruzione di salto; di conseguenza, la tecnica degli azzardi è statisticamente conveniente da adottare.
- **Rischedulazione hardware**: è un meccanismo noto anche come **out-of-order pipeline** (**OOO**), e prevede che le istruzioni vengano completate non necessariamente nel medesimo ordine con cui sono entrate all'interno della pipeline. In particolare, un'istruzione i_k, per accedere allo stage x, non deve necessariamente attendere che tutte le istruzioni a lei precedenti abbiano completato lo stage x. Si può dunque avere un meccanismo di **superamento** delle istruzioni all'interno della pipeline. Tale tecnica è vantaggiosa poiché permette all'istruzione i_k di essere completata prima e, quindi, di liberare un posto all'interno della pipeline. Tuttavia, è una tecnica che *richiede la possibilità da parte dell'hardware di ospitare più istruzioni contemporaneamente all'interno dello stesso stage* (altrimenti non sarebbe possibile effettuare il sorpasso): i processori con questa caratteristica sono detti **superscalari**. Inoltre, ovviamente, affinché si possa avere una out-of-order pipeline, *l'istruzione che effettua il sorpasso non deve dipendere dalle istruzioni che vengono sorpassate.* Non si hanno effetti reali sull'**Instruction Set Architecture** finchè non si deve "mostrare" l'output.

Esempio:

Se in pipeline su un thread ho $A \to B$, e nella pipeline B supera A, ma A è oggetto di *trap* (quindi non può fare quello che stava facendo, come dividere per 0, di conseguenza non potrò averla in ISA), cosa accade a B? Vedremo che in OOO si producono valori registrati in maniera "speculativa" che poi butterò, ma non posso eseguire una UNDO.

Nota: le istruzioni tra due thread sono indipendenti, dipendono solo se usano info condivise, ma ciò è di interesse al programmatore, non al processore. L'ordine della sorgente (programma) non è detto coincida con l'ordine del compilatore, tuttavia abbiamo la sicurezza che il data flow sia compatibile. A livello hardware, se ho operazione x, y, z può capitare quindi di avere z, x, y; ma ciò è realizzato dinamicamente dall'hardware. Ho sorpasso se non ho dipendenza.

Pipeline vs Sviluppo

Come abbiamo visto, i programmatori non hanno il diretto controllo del comportamento di un processore (trattasi di microcodice) ma, comunque sia, il modo con cui viene scritto il software può impattare sulle prestazioni effettive della pipeline. A livello ISA vedo il set di istruzioni e risorse usabili, ma non vedo la pipeline. **Esempio:** Esempi di ISA sono *ADD, COMPARE, JUMP.*onsideriamo le seguenti istruzioni C:

1) `a = *++p`
2) `a = *p++`

L'istruzione 1 prevede che prima debba essere incrementato il puntatore p affinché referenzi la entry successiva e solo dopo si possa accedere alla entry appena referenziata; di conseguenza, l'accesso dipende dall'incremento del puntatore. Al contrario, l'istruzione 2 prevede che prima si debba accedere alla entry attualmente referenziata dal puntatore p e solo poi si debba incrementare p; stavolta, l'accesso non dipende dall'incremento del puntatore. Questo vuol dire

che c'è dipendenza. Consideriamo due programmi, di cui il primo prevede l'istruzione 1 all'interno di un loop e il secondo prevede l'istruzione 2 all'interno di un loop. Nel secondo c'è indipendenza. La differenza prestazionale tra i due programmi è abbastanza significativa (può raggiungere tranquillamente il 20/25%).

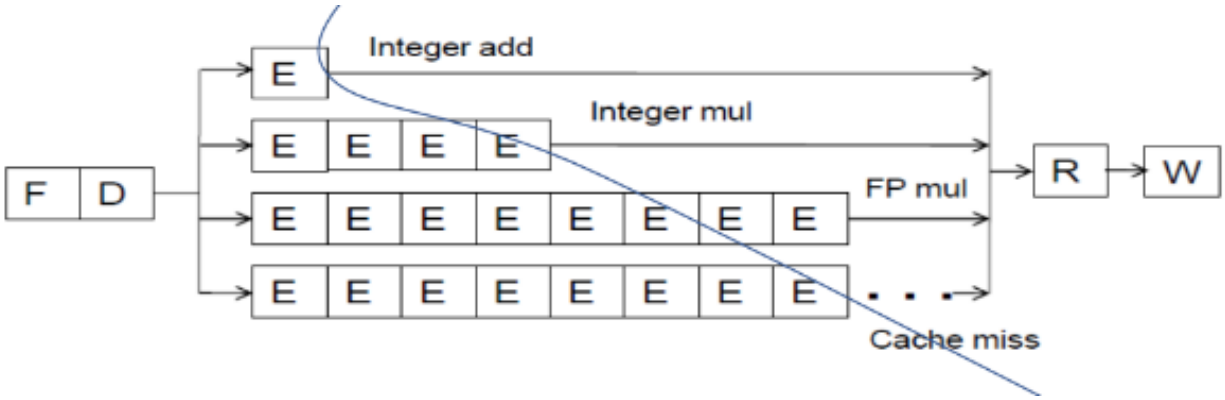
Per giunta, esistono delle istruzioni macchina che, da sole, hanno degli effetti devastanti sulle prestazioni del programma. Un esempio è `cpuid`, che ha lo scopo di restituire l'id del processore (o del CPU-core o dell'hyperthread) che ha processato l'istruzione stessa. Ma, oltre a questo, effettua anche lo **squash** della pipeline: in particolare, nel momento in cui `cpuid` viene realmente eseguita, la pipeline viene svuotata e le altre istruzioni al suo interno vengono buttate. Questo non deve necessariamente rappresentare uno svantaggio: avere istruzioni pendenti all'interno della pipeline significa dire che tali istruzioni possono essere schedulate dinamicamente dall'hardware secondo regole non meglio identificate, e ciò può impattare non solo sulla correttezza, ma anche sulla sicurezza del sistema. Più precisamente, quello di flushare la pipeline è un concetto legato al termine "**serializzazione**". Di fatto, `cpuid` è detta **istruzione serializzante**, poiché riporta la pipeline a lavorare secondo uno schema sequenziale. Non solo: `cpuid`, come tutte le istruzioni serializzanti, garantisce che qualunque modifica apportata a flag, registri e memoria da parte delle istruzioni precedenti sia completata (finalizzata) **prima** che una qualsiasi istruzione a lei successiva venga fetchata ed eseguita. *Ciò implica anche che cpuid non può superare alcuna istruzione davanti a lei nella pipeline.* (*e.g:* Perchè le istruzioni successive devono ancora essere fetchate ed eseguite, quindi come potrei superarle? Ciò che viene dopo questa istruzione serializzante è come se andasse in stallo finchè non completo questa istruzione serializzante, e garantisce anche che le istruzioni precedenti vengano viste da quelle successive.)
Se avessi SOLO istruzioni serializzanti, il sistema sarebbe molto più lento.

Pipeline superscalare

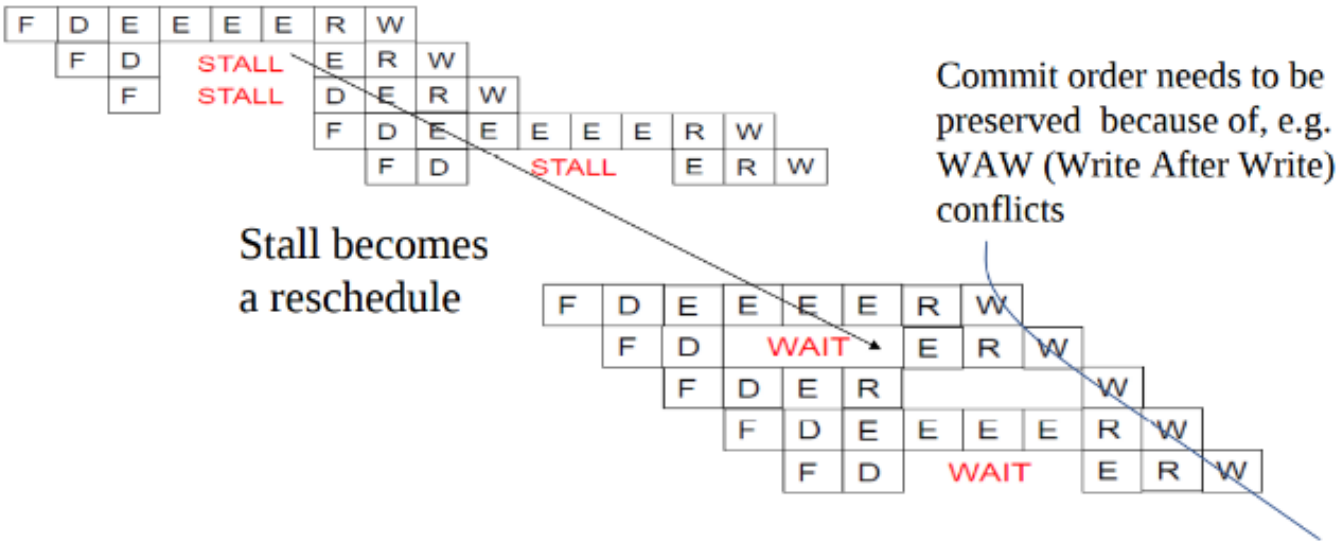
È un insieme di molteplici pipeline che operano simultaneamente all'interno del processore. La si può ottenere aggiungendo *ridondanza alle risorse hardware* in modo tale che più componenti distinti siano adibite a uno stesso stage della pipeline. Come detto in precedenza, questa possibilità permette anche di adottare il modello OOO, la cui idea di base consiste in:

- Effettuare il **commit** (o **retire** o **finalizzazione**) delle istruzioni esattamente nel medesimo ordine in cui sono *entrate nella pipeline*, indipendentemente dagli eventuali sorpassi avvenuti all'interno della pipeline. Ciò significa che le scritture dei risultati delle istruzioni in memoria, su un registro qualunque o su un registro di stato *devono avvenire esattamente nell'ordine prestabilito.*
- **Processare le istruzioni independenti** (sia sui dati che sulle risorse hardware) **il prima possibile**, dove un'istruzione indipendente sulle risorse hardware è un'istruzione che non ha bisogno di attendere che un qualche componente si liberi per processare un certo stage x.

L'immagine riportata di seguito mostra molto bene come lo stage EX delle istruzioni possa avere una durata variabile in termini di numero di cicli di clock, ed è a partire da tale presupposto che risulta utile avere un out-of-order pipeline.



La seguente altra figura mostra invece la differenza prestazionale che si può avere tra il modello non OOO e il modello OOO:



: Con una pipeline che ha ridondanza hardware (come i pc moderni), possiamo ad esempio parlare di "*Pipeline parallele a X canali*", che è come avere una autostrada a X corsie. Con Out-Of-Order Pipeline si evitano "grosse latenze", poichè se "libero una corsia perchè sono veloce" anche gli altri andranno più veloci (liberarando risorse utili ad altri). Lo stallo impatta su tutti invece. Nella O.O.O, se posso andare avanti, lo faccio, non mi importa che devo aspettare il commit, se posso eseguire qualcosa lo eseguo. Per questo parliamo di WAIT e non più di STALL. Nella WAIT, appena ho un risultato (outcome di una "corsia") lo fornisco a chi lo necessita, non mi serve aspettare che si liberi tutta la corsia.

Ora, poiché tra l'**emission** (= iniezione all'interno della pipeline, normalmente di più di una istruzione) e il retire (= **commit**) delle istruzioni si ha una fase di esecuzione in cui le istruzioni stesse possono sorpassarsi a vicenda, si va incontro al cosiddetto problema delle **eccezioni imprecise (OFFENDING)**: supponiamo di avere un'istruzione *i2* che ha superato un'istruzione *i1*, e assumiamo che *i1*, durante lo stage EX, generi un'eccezione (*e.g*: perché magari si tratta di una divisione per 0); a questo punto, anche se il risultato di *i2* non è stato ancora committato e, quindi, esposto a livello di ISA, *i2* può aver comunque toccato delle risorse non esposte a livello di ISA ed effettuato dunque dei cambi di stato (in particolare cambi di **stato micro-architetturale**). Quindi con offending instructions non esponiamo su ISA, ma potrei cambiare lo stato hardware, il che è usabile da malintenzionati. Questo perchè, lavorare in un contesto Out-Of-Order permette il superamento di istruzioni (**SPECULAZIONE**) che lascia delle tracce. Di conseguenza, l'eccezione sollevata da *i1* vede uno stato interno dell'hardware che ingloba anche delle attività relative a *i2*. Un problema analogo lo si ha anche a parti invertite: se l'istruzione che genera l'eccezione è *i2*, l'eccezione vedrà uno stato interno dell'hardware che non comprende il risultato prodotto da *i1*. Questo è un problema dello stato e delle informazioni all'interno del processore (*ma non esposte nell'ISA*). Peggio ancora: *i2* potrebbe sollevare un'eccezione che in realtà, secondo il program flow, non sarebbe dovuta mai esistere, magari perché *i1* è a sua volta un'istruzione che solleva un'eccezione o un'istruzione di salto. In realtà, si possono avere eccezioni imprecise anche in assenza di sorpassi: se l'istruzione *i2* viene dopo l'istruzione *i1* che genera un'eccezione, *i2* può aver comunque attraversato degli stage e, quindi, può aver modificato lo stato interno dell'hardware. Come vedremo, tutto questo rappresenta un grave problema per la sicurezza dei sistemi: **Meltdown** è solo il primo di una valanga di attacchi che hanno sfruttato tale vulnerabilità.

Algoritmo di Robert Tomasulo

Secondo Robert Tomasulo, in uno scenario di utilizzo di una pipeline speculativa out-of-order (dove speculativa = caratterizzata dall'esecuzione di istruzioni che potrebbero servire solo in un secondo momento), se consideriamo due istruzioni A, B tali che $A \rightarrow B$ nell'ordine di programma, dobbiamo stare attenti nell'evitare i seguenti tre tipi di azzardo:

- **RAW** (**Read After Write**): B deve leggere un dato R necessariamente dopo che A lo ha aggiornato.
- **WAW** (**Write After Write**): B deve scrivere su un dato R necessariamente dopo che A lo ha aggiornato.

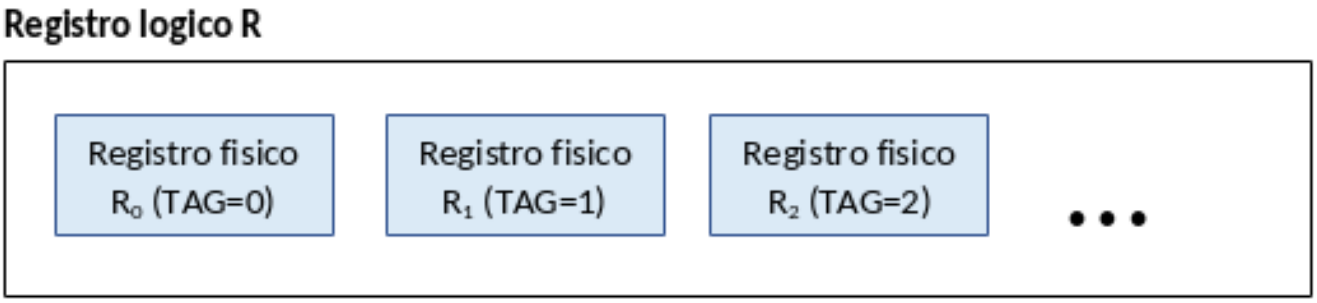
- **WAR** (**Write After Read**): B deve scrivere su un dato R necessariamente dopo che A ne ha letto il valore precedente (altrimenti vorrebbe dire che A è in grado di leggere dal futuro).

Dipendenza RAW

Qui è necessario bloccare l'istruzione B e tenere traccia di quando il dato che deve essere letto da B sarà disponibile (disponibile non vuole dire necessariamente committato).

Dipendenza WAW e WAR

Qui è necessario adottare una tecnica nota come **register renaming**, secondo cui al programmatore vengono esposti dei registri logici, e ciascuno di questi registri logici (che sono di fatto dei multi-registri) ingloba un insieme di registri fisici non visibili al livello dell'ISA. Quindi noi non scriviamo MAI sul registro esposto in ISA, ma su delle '*copie numerate*' o alias, cioè registri multivalore. Ci ritroviamo dunque nella seguente situazione:



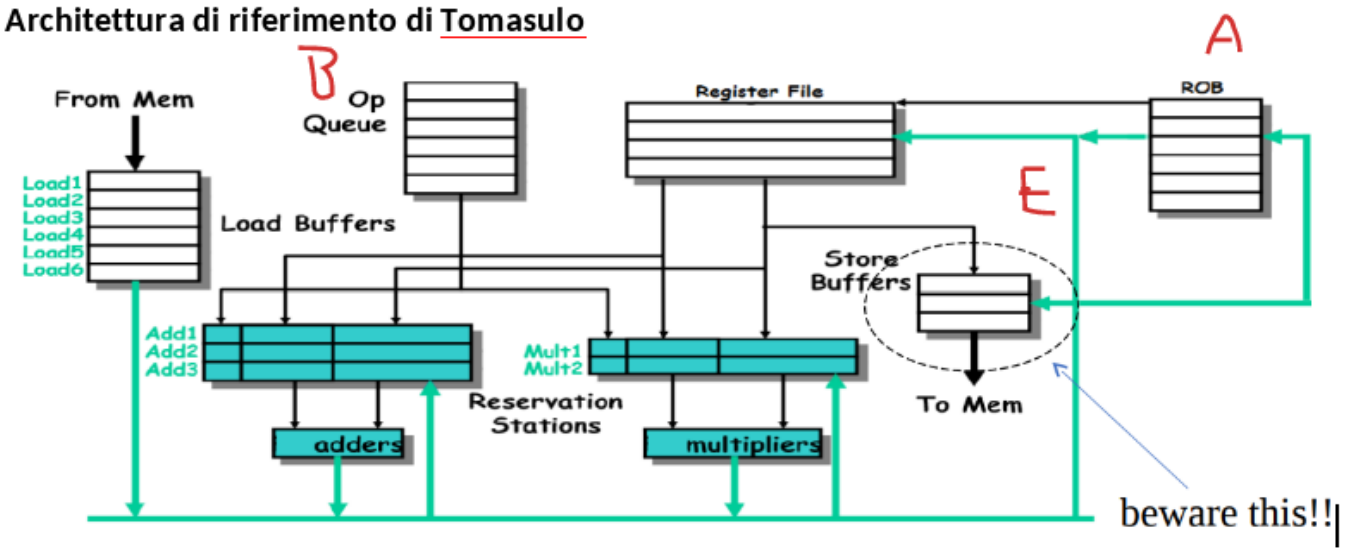
Qui i vari registri fisici rappresentano diverse versioni del medesimo registro logico R. Nel momento in cui all'interno della pipeline entra un'istruzione che vuole scrivere sul registro logico R, si considera il primo registro fisico R_k all'interno del quale viene effettivamente memorizzato il valore (quindi scrivo sempre sul primo tag libero, e leggo dall'ultimo tag in pipeline. Se A scrive in TAG 0, e B scrive in TAG 1, leggo da TAG1). Tipicamente, per la selezione del registro fisico, si segue un approccio round-robin; se però a un certo punto tutti i registri fisici di R sono stati sovrascritti e nessuna delle istruzioni che ha eseguito la scrittura è andata in commit, per un'eventuale altra scrittura su R bisognerà attendere.

Posso superare solo dopo essere entrato nella pipeline (nelle condizioni discusse nelle pagine precedenti), ma non posso superare prima di entrare in pipeline.

In pratica, un *renamed register* materializza il concetto di speculatività di una pipeline: quando vengono effettuate delle write, vengono scritti dei valori che non necessariamente verranno considerati come validi, ma che comunque potranno essere letti dalle istruzioni successive (che sono state introdotte nella pipeline speculativamente).

Relativamente al registro R, vengono memorizzati anche dei metadati di gestione di R che indicano qual è il tag contenente la versione committata; la versione committata è la versione del valore di R scritto dall'ultima istruzione andata in commit che ha toccato proprio R. In questi registri abbiamo il "futuro" che avverrà, perchè non li ho ancora committati.

Architettura di riferimento di Tomasulo



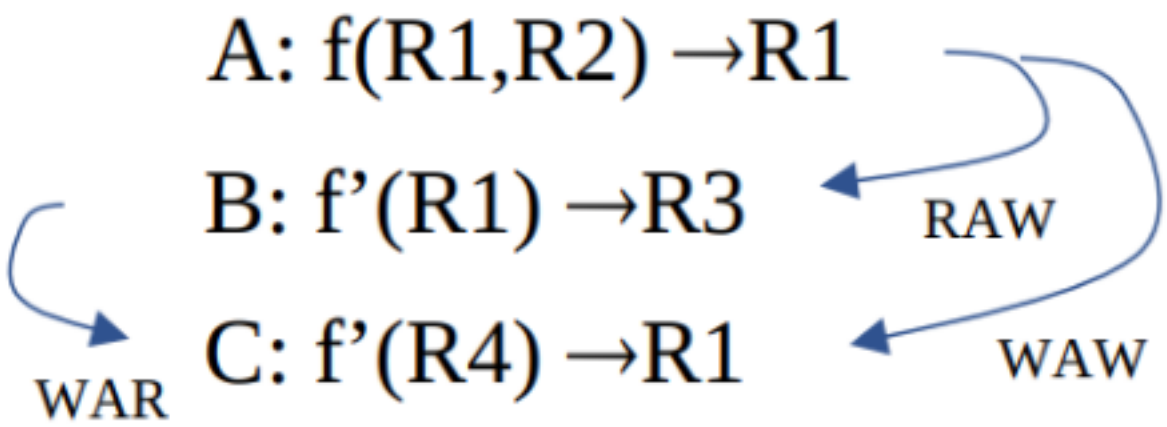
- **A)**: I metadati associati alle istruzioni fetchate vengono memorizzati all'interno del **ROB** (**Re-Order Buffer**), che ci aiuta a scrivere le istruzioni nello store buffer, in quanto posso andare in memoria solo se sono committed, ma io non posso aspettare sempre questa fase prima di utilizzare i dati. Tali metadati possono essere:
  - *L'ordine con cui le istruzioni dovranno andare in commit.*
  - Che cosa dovrebbero fare le istruzioni nella loro esecuzione speculativa.
  - Qual è l'alias (l'istanza fisica) dei registri che dovranno essere usati da ciascuna istruzione; ad esempio, se un'istruzione A deve scrivere su un certo registro R e un'istruzione B successiva deve leggere dallo stesso registro R con una dipendenza RAW, è chiaro che A e B dovranno utilizzare il medesimo alias.
- **B)**: Le operazioni vere e proprie da eseguire (quindi gli OP code delle istruzioni) vengono memorizzate all'interno dell'**OP queue** e, in base alla loro tipologia, verranno date in input a un particolare componente di processamento (add, mult e così via). Nel momento in cui un'istruzione è pronta per essere processata, devono essere recuperati i metadati associati a essa e, in particolare, gli alias dei registri da usare. A tale scopo, c'è un bus comune (detto **Common Data Bus – CDB**) che mette in comunicazione i componenti di processamento col ROB. Se un alias non è pronto per essere utilizzato (perché magari bisogna attendere che venga sovrascritto da un'istruzione precedente), l'istruzione viene posta in attesa all'interno del relativo componente di processamento. I componenti di processamento sono detti anche **reservation station**. *Cosa è Reservation Station?* Per ogni componente in grado di eseguire uno specifico calcolo, si ha una coda/buffer/corsia dove posso mettere varie istruzioni, e le operazioni identificate dai registri usati (sorgenti). Non si ha sorpasso per utilizzare tali componenti in queste code, al massimo se ho due istruzioni dipendenti cerco di metterle nella stessa reservation station.
- Si fa uso di una **cache** per leggere in modo più efficiente i dati provenienti dalla memoria.
- Nel momento in cui viene **committata** un'istruzione, l'eventuale alias aggiornato da tale istruzione viene installato all'interno del **register file** come valore valido, ovvero come **valore esposto all'interno dell'ISA.**
- **E)**: Se l'istruzione committata prevede una scrittura in memoria, il valore di output, prima di essere riportato in memoria, viene inserito nello **store buffer** in modo tale da velocizzare il completamento dell'istruzione. Tuttavia, ciò implica che il valore non sarà in memoria per un po' di tempo, il che può rappresentare un problema dal punto di vista della consistenza.
  Nota: Supponiamo che un'istruzione sia in fase di ritiro, quindi è totalmente conclusa. In questo istante, il dato dovrebbe passare da Store Buffer alla memoria, ma in realtà non è istantaneo, passa un piccolo lasso di tempo. Ciò crea problemi con > 1 thread, perchè se pesco un dato in memoria potrei non vedere alcuni dati. Per questo l'algoritmo della "*pasticceria*" non funziona nei processori moderni, suppone che tale tempo sia nullo. Esistono però delle istruzioni per controllare lo stato dello Store Buffer. Questo approccio è l'unico per lavorare con O.O.O, ovvero non esistono altre tecniche per affrontare le O.O.O pipeline.

Esempi di schemi di esecuzione

Esempio 1 Supponiamo di avere tre istruzioni A, B, C tra cui B e C hanno una stessa latenza d, mentre A ha una latenza $d' > d$.
Assumiamo inoltre che:

- A esegua l'operazione $f(R1, R2)$ e riporti l'output sul registro *R1.* (in particolare, scrive su un alias di *R1*).
- B esegua l'operazione $f'(R1)$ e riporti l'output sul registro *R3.* (legge da STESSO alias di R1)
- C esegua l'operazione $f'(R4)$ e riporti l'output sul registro *R1.* (scrive su UN ALTRO alias di R1).

Ci ritroviamo dunque nel seguente scenario:



È abbastanza evidente che B debba leggere **lo stesso alias** scritto da A, mentre C potrà riportare il valore di output su un **alias differente**. In tal modo è possibile **processare** C **parallelamente ad** A :
In fondo, anche se C genera il suo output prima di A, le dipendenze che legano le tre istruzioni non vengono violate. Infatti, B sarà comunque in grado di leggere dall'alias sovrascritto da A e, ovviamente, rimarrà comunque possibile mandare in commit C solo dopo il completamento di A e B. Il vantaggio che porta questo approccio è la riduzione del tempo necessario per il completamento di C.

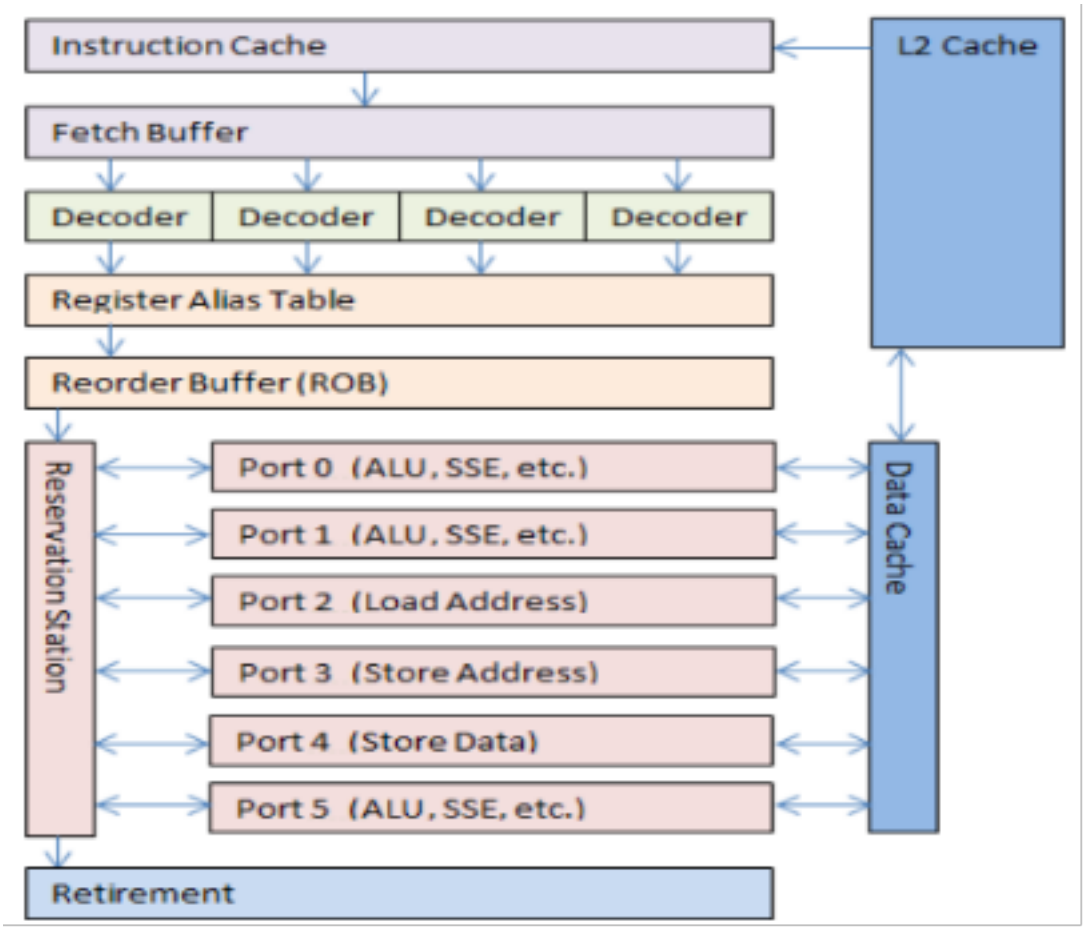
Esempio 2 Vediamo con un secondo esempio riportato qui di seguito come viene associata una entry del ROB a ciascun alias differente:

Before: add r3,r3,4	after	add rob6,r3,4
add r4,r7,r3		add rob7,r7,rob6
add r3, r2, r7		add rob8,r2,r7

E' importante ricordare che, anche se è possibile eseguire in parallelo, non posso pensare di aumentare all'infinito le prestazioni, perchè ad un certo punto eccedo il numero di istruzioni che posso gestire.
Questo porta ad un sistema scarico, ovvero non occupo corsie perchè ci sono caricamenti di elementi dalla memoria, oppure eseguo una predizione/azzardo sbagliata che mi porta a svuotare etc. . .
E' come fare un'autostrada a 20 corsie: è molto difficile saturarla il "giusto".

Che soluzione è possibile adottare? Introduciamo i **Processori HYPERTHREAD**.

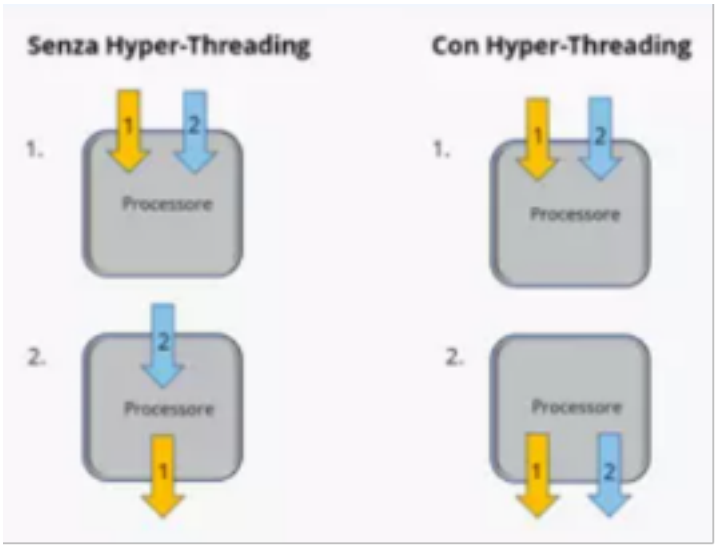
Organizzazione architetturale dei processori x86 OOO



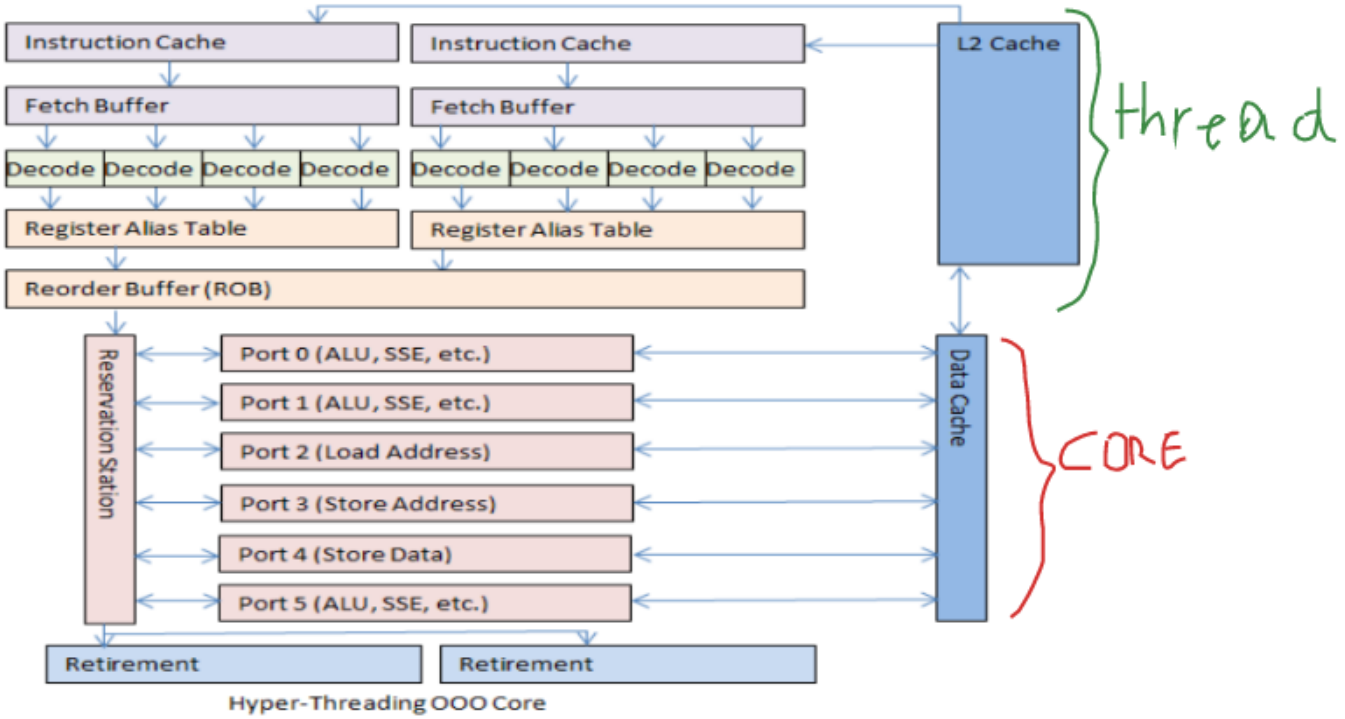
Processori hyper-threaded

Come abbiamo osservato all'inizio della trattazione, le architetture moderne sono soggette al **memory wall**. In particolare, si ha una latenza di esecuzione non solo quando si devono *recuperare i dati dalla memoria*, ma anche quando devono essere estratte le istruzioni stesse. Dunque, si può anche avere una pipeline molto efficiente con un **ILP** elevato, ma il processore non arriva mai a processare in parallelo tutte le istruzioni possibili a *causa dei ritardi causati dalla memoria*, per cui la potenza e le risorse del processore risultano sprecate. Per ovviare all'inconveniente, si può assegnare una *stessa reservation station a più program flow diversi*. Da qui nascono i processori **hyper-threaded**, che hanno un *unico core fisico (i.e. un'unica information station)* che permette di eseguire le reali operazioni dettate dalle istruzioni; d'altra parte, la porzione di processore che memorizza le istruzioni da fetchare, decodifica le istruzioni per uno specifico flusso (Decode) e tiene traccia dei registri logici dell'ISA (mediante Register Alias Table) viene replicata e viene definita **hyperthread**. Una CPU può contenere uno o più core. Un core è un'unità all'interno della CPU che realizza l'effettiva esecuzione. E' un "oggetto engine". Con l'Hyper-Threading, un microprocessore fisico si comporta come se avesse due core logici, ma virtuali. In questo modo si consente a un unico processore l'esecuzione di più thread contemporaneamente. Questo procedimento aumenta le prestazioni della CPU e migliora l'utilizzo del computer.

Esempio: ho una bocca (core) e due mani con cui prendo il cibo (thread). Per migliorare la prestazione devo puntare ad aumentare i thread (più mani = più cibo in entrata). Se aumentassi il numero di bocche, non migliorerei, perchè sempre quel cibo ingerisco, anche se finisco prima devo aspettare la prossima forchettata.



In tal modo, è possibile eseguire più **workflow in parallelo su un unico core fisico**, e l'architettura risultante è riportata nella pagina seguente:



Tale architettura risulta molto vantaggiosa per la maggior parte dei workload, mentre risulta un po' più avversa nel caso in cui tutti i thread che girano sullo **stesso core sono CPU-bound** ma non memory-bound. Infatti, in quest'ultimo caso è più probabile che si verifichi un conflitto tra i thread in esecuzione nell'utilizzo delle risorse della reservation station. Un processore hyper-threaded viene tipicamente marcato con l'indicazione su quanti *core* (="motori") e quanti *hyperthread* (="thread fisici") possiede.
Esempio:
Un processore a due core e quattro thread fisici viene marcato come **2C/4T**.
In un contesto senza Hyperthread avremmo 2 core e 2 thread (1 per ogni core).
In un contesto con Hyperthread possiamo avere 2 core e 4 thread (2 thread per ogni core).
Se ho un caso 2C/4T, vuol dire che il mio programma non può generare più di 4 thread?
No, vuol dire che ogni core può gestire al massimo due thread con un proprio set di registri aventi alias (quindi due workflow in parallelo su unico core), quindi in totale posso gestire al massimo quattro thread/workflow in contemporanea. Ci sarà lo scheduling che realizzerà il parallelismo, ma comunque ne verranno gestiti quattro insieme.
I flussi sui thread possono essere speculativi.

Gestione degli interrupt

Ne sono esempi: muovere mouse, premere sulla tastiera.
L'interrupt è un segnale *proveniente da un certo dispositivo hardware* che indica la necessità di **cambiare il flusso di esecuzione** (e.g. andando a eseguire un handler). Poiché è buona norma processare gli interrupt il prima possibile, quando ne occorre uno, si attende solo che una delle istruzioni correntemente in pipeline vada in commit; dopodiché si effettua lo squash (ovvero si svuota) la pipeline e si iniziano a fetchare le istruzioni dell'handler dell'interrupt. Ciò può portare a un rallentamento dell'esecuzione, ma non vale la pena memorizzare da qualche parte lo stato di esecuzione delle istruzioni che vengono buttate (richiederebbe hardware aggiuntivo, inoltre non ho certezza che dopo l'handler, ritornando al flusso di esecuzione, io parta dall'istruzione subito dopo); tra l'altro, anche mentre viene eseguito il gestore di un certo interrupt può subentrare un ulteriore interrupt, e questo può avvenire iterativamente un numero arbitrario di volte.

Attenzione:
Buttare delle istruzioni dalla pipeline implica prevenire i loro effetti sull'ISA ma, se esse hanno sporcato lo stato micro-architetturale, quest'ultimo non può essere ripristinato: rimangono comunque gli effetti dell'esecuzione speculativa delle istruzioni che sono state buttate. A livello hardware non ho meccanismi di gestione priorità o trap, operazioni offending come una divisione per 0 non vengono svolte dal processore, ma passano ad un handler.

Gestione delle eccezioni

Esse avvengono durante esecuzione di un programma, a livello software. Le eccezioni risultano più complesse da gestire rispetto agli interrupt poiché vengono sollevate dall'esecuzione di particolari istruzioni. Di fatto, un'istruzione A (cosiddetta **offending**) che solleva un'eccezione e_A può essere eseguita speculativamente all'interno della pipeline e, di conseguenza, potrebbe non esistere nel flusso di esecuzione definito dal programmatore. Ad esempio, poco prima dell'istruzione A potrebbe esserci un'istruzione B che solleva a sua volta un'eccezione e_B: in tal caso sarà e_B a esistere realmente e non e_A. Una Istruzione è offending se vuole usare qualcosa che non è usabile. Non genera una trap, perchè non so se arrivo in retire. Se l'istruzione successiva in fondo non viene committata, cancello tutto. A valle di queste considerazioni, un'*eccezione viene presa in carico solo nel momento in cui l'istruzione che l'ha generata va in retire*, anche se ci si può accorgere prima che l'istruzione sia offending. (Vedi sotto, etichettamento offending).
Ma quali sono gli stage in cui un'istruzione può rivelarsi offending?

- **Instruction Fetch / Memory stages**
  (MEM stage = stage in cui avviene l'accesso agli operandi): qui si può avere un page fault (ovvero un tentato accesso a un indirizzo logico di memoria che attualmente non ha un corrispettivo fisico), un accesso in memoria disallineato o una violazione delle protezioni applicate a una pagina di memoria (e.g. si tenta di accedere in scrittura a una locazione read-only).
- **Instruction Decode stage**: qui può emergere che l'istruzione in esercizio sia illegale.
- **Execution stage**: qui può essere sollevata un'eccezione aritmetica (e.g. divisione per zero).
- **Write-Back stage**: qui non possono essere sollevate eccezioni.

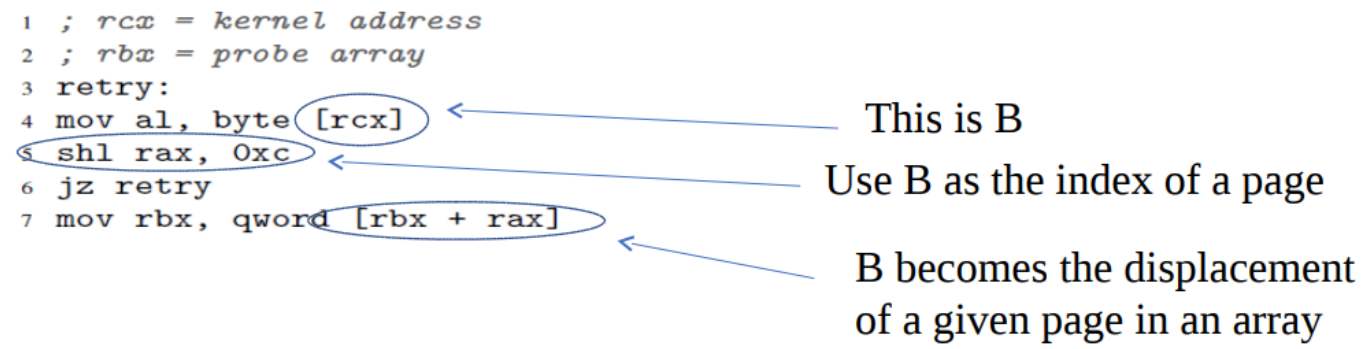
Per etichettare un'istruzione come offending, si imposta a 1 un apposito bit dei metadati. Quando tale istruzione va in *retire*, se il bit vale 1 allora il commit non viene effettuato, bensì viene attivato il gestore di eccezioni opportuno. A tal punto tutte le istruzioni successive a quella offending diventano *phantom* (fantasma).

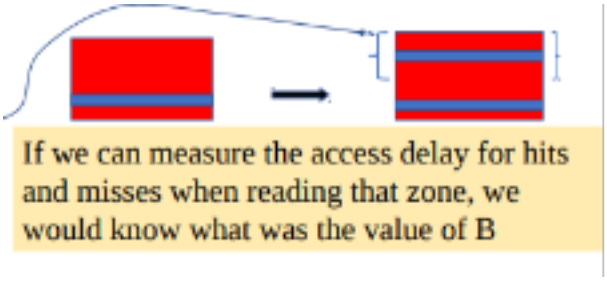
Attenzione: Con le eccezioni si verifica lo stesso problema riscontrato con gli interrupt. In particolare, quando un'istruzione offending va in retire, si hanno altre istruzioni all'interno della pipeline che hanno già modificato lo stato micro-architetturale e, dunque, hanno lasciato tracce di informazione. Questa problematica lascia spazio al cosiddetto attacco **Meltdown**. Sto propagando "attività illecite" nella pipeline.

Attacco Meltdown

Sappiamo bene che *ciascun processo ha il proprio address space* suddiviso in zone di memoria dedicate all'esecuzione *user mode* e zone di memoria dedicate all'esecuzione *kernel mode*. L'attacco ha come scopo ultimo quello di accedere a un'area di memoria situata nel kernel anche se non si hanno i permessi, magari per andare a leggere delle informazioni sensibili di un utente differente del sistema. Andando nel dettaglio, l'attacco viene eseguito nella maniera spiegata qui di seguito. Anzitutto si definisce un array A nella memoria user space e si svuota la cache tramite l'istruzione `cflush`. Dopodiché, mediante una `mov`, si preleva un **particolare byte** x (*e.g*: address) situato nel kernel e si memorizza il byte in un registro; naturalmente questa `mov` è un'istruzione offending. Si accede alla entry con **spiazzamento** x rispetto all'indirizzo base dell'array A (e.g. caricandone il contenuto in un registro) in modo da farla salire in cache:

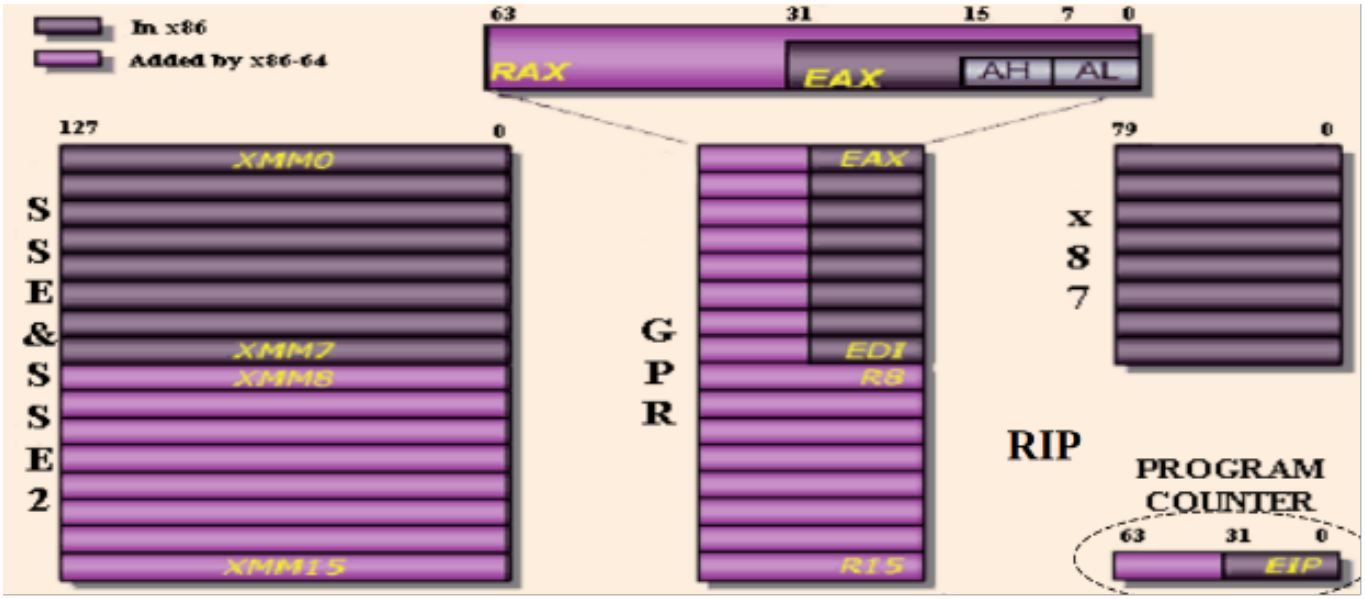
tale aggiornamento della cache rappresenta proprio il *side effect* lasciato sullo stato micro-architetturale da parte dell'istruzione successiva a quella offending che, chiaramente, viene eseguita *solo speculativamente*. A tal punto, poiché l'array A si trova in user space, è possibile accedere a tutte le sue entry e, per ogni entry, cronometrarne il tempo necessario per l'accesso: la entry relativa al tempo di accesso minore sarà chiaramente quella di spiazzamento x, perché si tratta dell'unica entry il cui contenuto era stato memorizzato in cache. Ed ecco qui: **ora è noto lo spiazzamento x**, che era proprio il byte di memoria prelevato inizialmente dal kernel space. Ricapitolando, il valore x è stato ottenuto in maniera indiretta misurando i tempi di accesso alle informazioni presenti nell'array A. Di conseguenza, abbiamo a che fare con un **side-channel** (o **covert-channel**) **attack**, ovvero con un attacco che fa uso di un canale laterale per andare a rubare delle informazioni. Io parto da un byte B presente nella zona kernel, lo salvo in un registro (non potrei), accedo a `array[B]` che va in cache. Successivamente accedo a tutte le entry di array (in un loop continuo, indipendente da B). L'accesso più veloce sarà quello ad `array[B]`, e quindi tra tutti gli accessi effettuati, riesco a capire cosa c'è in `array[B]`, perchè più veloce. Quindi prendendo un byte nella zona kernel, riesco da user a leggere qualcosa a livello kernel.
Meltdown può essere esteso anche al caso in cui si vuole scoprire un'intera stringa all'interno del kernel space, che può essere una password, una chiave segreta e così via. Negli esempi





Insights sui processori x86 e x86-64

X_86 ha 8 registri general purpose, e Program Counter 32 bit.
X_86_64 è backward compatibile, ha 15 registri general purpose a 64 bit e Program Counter a 64 bit.



- **GPR**: registri general purpose.
- **SSE & SSE2**: registri vettoriali, che consentono a singole istruzioni di fare più cose in parallelo.
- **x87**: floating-point stack registers.
- **RIP**: program counter.

Vediamo alcune istruzioni fondamentali per il trasferimento di dati nell'architettura x86-64:

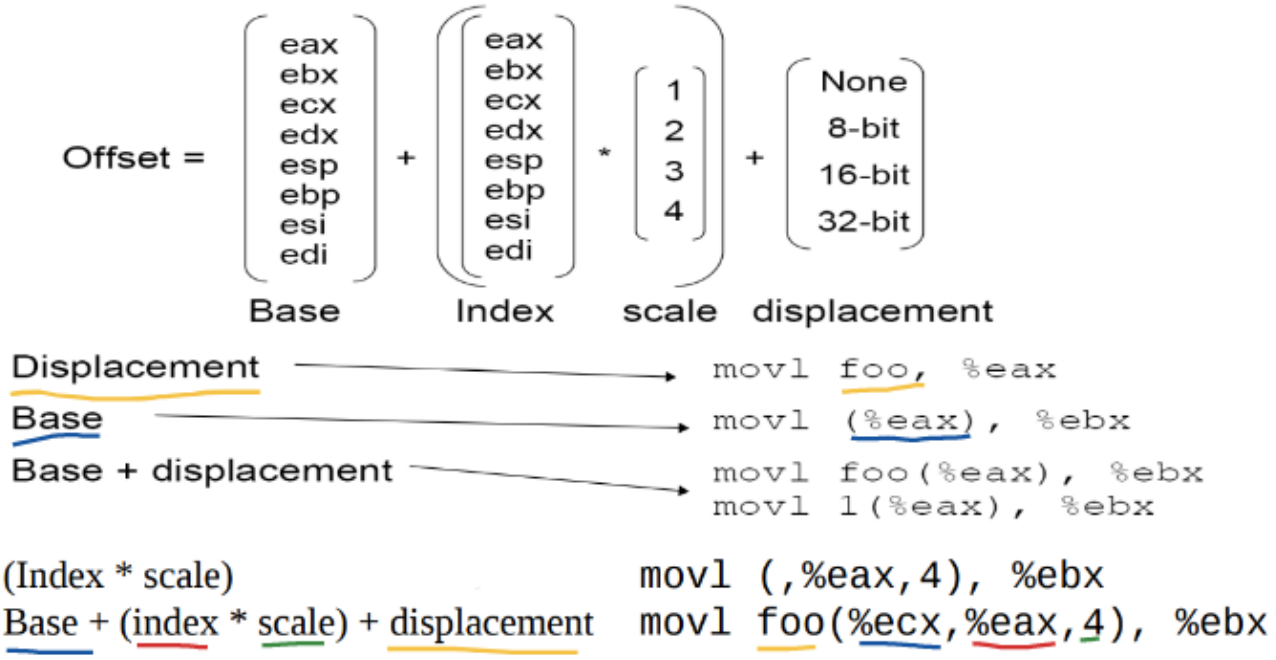
Istruz. (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione	Esempio*
mov	movl	movw	movb	Trasferisce un valore su un registro destinazione.	movw $5, %ax
push	pushl	pushw	\	Aggiunge un elemento sullo stack.	pushl %ebx
pop	popl	popw	\	Elimina un elemento dallo stack e lo salva su un registro.	popl %ebx

*Negli esempi specificati in tabella si è adottata la sintassi AT&T che, a differenza di quella Intel, prevede che la sorgente venga posta prima della destinazione.

La parte più complessa però sta nello specificare la sorgente e/o la destinazione **in memoria logica** (e non su un registro). La figura riportata in seguito mostra il meccanismo utilizzato. In *ISA* non ho i nomi delle variabili, solo quelli dei registri, per arrivare ad una variabile di memoria devo usare l'indirizzo di tale variabile in memoria.
Nota: Le componenti *base, index, scale* sono sempre contenute nelle parentesi, ad esempio *movl (. . . , . . . , . . .)*

- **Displacement**: può ad esempio essere un indirizzo assoluto noto a tempo di compilazione. Possiamo vederlo come il punto dell'address space che specifica la sorgente (es: *foo è displacement diretto*)
- **Base**: può essere relativo al valore di un pointer.
- **Index*scale**: può rappresentare uno spiazzamento rispetto all'indirizzo base; ad esempio, quando si itera su un array di interi, *scale* vale sempre 4 mentre *index* viene incrementato di 1 a ogni iterazione.

(nb: in %ebc è dove carico il tutto).



Vediamo ora le istruzioni logiche e aritmetiche:

Istruzione (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione
and	andl	andw	andb	dest = source && dest
or	orl	orw	orb	dest = source || dest
xor	xorl	xorw	xorb	dest = source ^ dest
not	notl	notw	notb	dest = ^dest
sal	sall	salw	salb	dest = dest << source
sar	sarl	sarw	sarb	dest = dest >> source
add	addl	addw	addb	dest = source + dest
sub	subl	subw	subb	dest = dest – source
inc	incl	incw	incb	dest = dest + 1
dec	decl	decw	decb	dest = dest – 1
neg	negl	negw	negb	dest = ^dest
cmp	cmpl	cmpw	cmpb	Compara due valori; se essi sono uguali, imposta un determinato registro di stato.

Codice assembly dell'attacco Meltdown

Nella pagina seguente viene mostrato il codice assembly che permette di effettuare l'attacco Meltdown. La sintassi utilizzata è quella di Intel. Chiaramente il codice può essere incapsulato all'interno di un programma C.

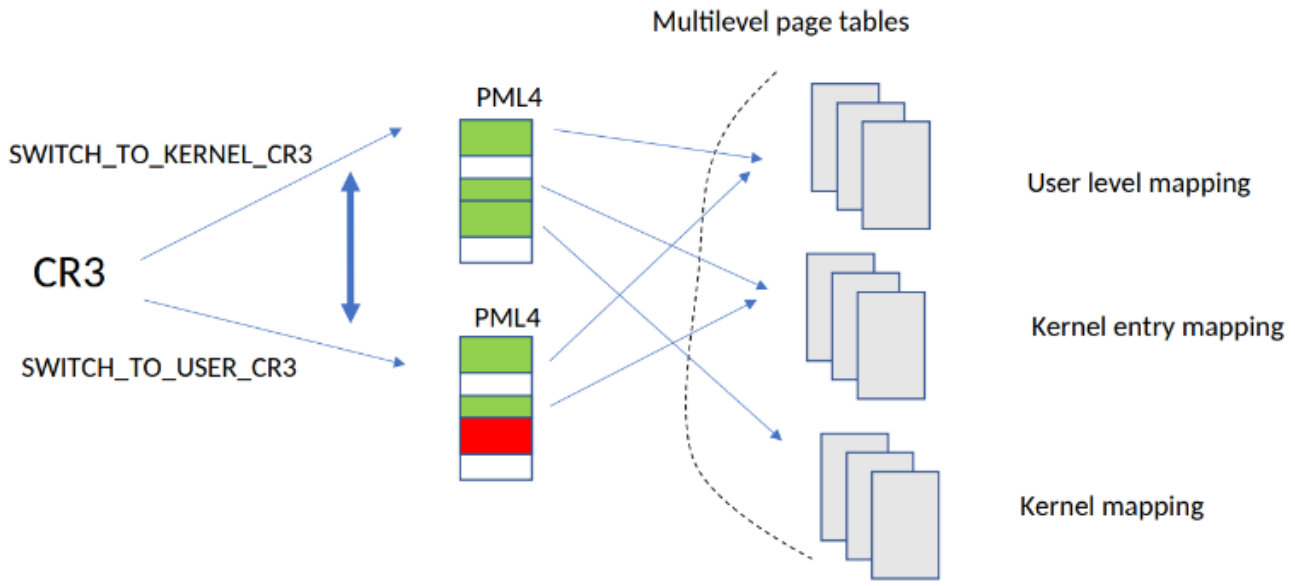
```
; rcx = kernel address
; rbx = probe array (array A)
retry:
mov al, byte [rcx] //istr. offending. Prendo byte[rcx] e lo metto in 'al', ultimo byte del reg. 'eax'
shl rax, 0xc      //shifto rax di 0xc = 12, cioè 12 bit a 0 (verso sx shifto la parte meno significativa) l
jz retry         //se il valore letto nel kernel è nullo, cioè rax=0 riprovare.
mov rbx, qword [rbx + rax] //qui si effettua l'accesso al probe array con spiazzamento pari a rax
```

Vale la pena fare alcune osservazioni:

- 4096 byte è esattamente la dimensione di una pagina di memoria. Ma perché gli spiazzamenti che si prendono in considerazione all'interno dell'array A sono esclusivamente la prima entry di ciascuna pagina di memoria? Per evitare problemi con la cache: di fatto, quando si accede a un certo valore in memoria, non è solo lui a essere caricato in cache, ma come minimo una **linea di cache**, che è lunga *64 byte*. Leggiamo sempre lo 0-esimo byte, perchè la cache lavora a blocchi e potrei trovare '64 byte' rapidi, e quindi non essere in grado di differenziare gli accessi. Per giunta, i processori tipicamente applicano il cosiddetto **pre-fetching**: nel momento in cui viene caricata in cache una linea, vengono conseguentemente caricate anche alcune linee adiacenti per il principio di località. Perciò, per evitare che i valori relativi a più di uno spiazzamento in A vengano caricati in cache a seguito di un unico accesso, si prendono gli spiazzamenti in modo tale che siano distanti tra loro, e una distanza di 4096 byte risulta essere sufficiente.
- Perché se nel kernel space viene letto il byte 0 si fa un nuovo tentativo? Perché un'area di memoria inizializzata a zero o è memoria non valida o è relativa a un terminatore di stringa, per cui si tratta di un'area di memoria non significativa. Resta comunque possibile ritentare ad accedere al medesimo byte all'interno del kernel space perché non è escluso che, in concorrenza, il kernel possa cambiare il valore di quel byte (da zero a un valore significativo e di interesse).

Contromisure per l'attacco Meltdown

- KASLR (Kernel Address Space Randomization): Quando si fa il setup del kernel del sistema operativo, le strutture dati di livello kernel possono cadere ovunque all'interno del kernel space: in particolare, si stabilisce randomicamente un *offset F* rispetto all'indirizzo base del kernel a partire da cui sono definite le varie strutture dati del kernel. In questo modo, si complica la vita all'attaccante poiché lui non conosce a priori l'indirizzo del kernel space in cui andare a effettuare l'attacco; tuttavia, con un approccio brute-force, può comunque fare in modo che, prima o poi, l'attacco vada a buon fine. Per questa ragione, KASLR non è la soluzione definitiva contro l'attacco Meltdown.
- Cash flush:
  Si effettua semplicemente un flush della cache ogni volta che il kernel prende il controllo od ogni volta che un thread viene rischedulato. Tuttavia, così facendo, si avrebbe un calo inaccettabile delle prestazioni: le cache sono condivise tra i vari hyperthread, per cui ciascun cache flush impatterebbe su tutti i thread in esecuzione all'interno dell'host.
- KAISER (Kernel Isolation in Linux):
  Quando un processo gira in modalità *user*, utilizza una page table PT_U all'interno della quale il mapping della maggior parte degli indirizzi del kernel space non è presente: ci sono esclusivamente quegli indirizzi della porzione del kernel che vengono utilizzati ad esempio per gestire gli interrupt (se non ci fossero almeno loro, non saremmo neanche in grado di gestire gli interrupt o comunque di trasferire il controllo al kernel). Dall'altro lato, quando un processo gira in modalità kernel, utilizza un'altra page table PT_K che contiene il mapping degli indirizzi di memoria mancanti in PT_U. Di fatto, gli indirizzi del kernel space il cui mapping è presente nella page table PT_U costituiscono la cosiddetta **entry zone** del kernel, che contiene quasi esclusivamente il codice che permette di effettuare lo switch delle page table (PT_U a PT_K). Perché quest'ultima soluzione risulta funzionante? Nel momento in cui un processo che esegue in user mode incontra un'istruzione offending che vuole accedere a un indirizzo di memoria del kernel space, tipicamente non troverà tale indirizzo nella page table che ha a disposizione. In altre parole, si solleva un'eccezione di tipo **page fault**, che non consente al processo di continuare ad eseguire le istruzioni anche speculativamente. Inoltre, si tratta di una soluzione che, sì, ha dei costi prestazionali, ma mai quanto il cash flush; in particolare, il calo delle performance viene osservato dalla **TLB - Translation Lookaside Buffer** che, a ogni switch da user mode a kernel mode e viceversa, dovrà cachare le informazioni sulla nuova memory view su cui ci siamo spostati (che sia essa relativa alla PT_U o alla PT_K). Entrando un po' più nel dettaglio sulle page table, se ne hanno di livelli differenti (page table di livello 1, page table di livello 2, e così via). Solo le **page table di livello 1** sono replicate tra l'esecuzione user mode e l'esecuzione kernel mode, mentre le altre sono a **istanza unica**; in particolare, al livello 1, la page table PT_K è in grado di raggiungere tutte le informazioni delle page table ai livelli inferiori, mentre la page table PT_U punta solo a un sottoinsieme di informazioni delle page table ai livelli inferiori. Da user possiamo comunque accedere a qualcosa di livello kernel, ma solo per i moduli kernel utili/indispensabili (compile time) per avere il minimo supporto necessario. KAISER sfrutta quindi questa tecnica di **PTI-Page Table Isolation**, disattivabile da GRUB. Il selettore CR3, ad ogni cambio, azzera la TLB e genera cache miss. Però è un'operazione automatica ed abbastanza semplice.



Insights sui branch

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Per i salti **condizionali** e **indiretti**, l'indirizzo da cui riprenderà l'esecuzione è noto soltanto a partire da un certo stage S della pipeline. Chiaramente, però, vogliamo che venga comunque eseguito del lavoro all'interno della pipeline prima che l'istruzione di salto raggiunga lo stage S, altrimenti il calo delle prestazioni sarebbe certo e significativo. A tal proposito, si introducono dei **dynamic predictor** (o **branch predictor**), che hanno lo scopo di predire quale sarà la destinazione verso cui si dovrebbe saltare con la più alta probabilità. Ciò permette di comporre in maniera speculativa un program flow all'interno della pipeline in funzione della predizione che è stata attuata dal predittore. La reale implementazione dei branch predictor è basata sui **Branch History Table** (**BHT**), detti anche **Branch-Prediction Buffer** (**BPB**), che contengono metadati che associano un'istruzione di salto all'ipotesi su dove tale istruzione salterà. E' una zona di cache hardware contenente `[indirizzo istruzione, target da usare se jumpo]`, e l'indirizzo istruzione può anche essere il target stesso. E' un suggeritore, non so nulla dell'affidabilità. L'implementazione minimale (andando avanti verranno aggiunte altre componenti) per una BHT consiste in una cache indicizzata tramite i *bit meno significativi* di ogni istruzione di salto. Ogni qual volta viene fetchata un'istruzione di salto, si può avere una cache hit o una cache miss, dove la *cache hit* la si ha nel caso in cui sono disponibili sufficienti informazioni che permettano di effettuare una predizione sulla destinazione del salto. Queste informazioni consistono in particolari bit di stato: ciò che succede nel passato è rappresentativo di ciò che si prevede che accadrà in futuro. Nei salti **condizionali** ho due destinazioni, saprò quella corretta in fase di esecuzione, e quindi è register dependent, perchè dipende a runtime cosa ci sarà nel registro).
I salti **unconditional** e le **call**, dove salteremo è già noto nello stage di decode, salterò sempre in quel punto.
Anche le **return** vengono sempre eseguite, ma ho più destinazioni note nello stage di esecuzione.
Per i salti **indiretti** salterò sempre (come per le call), ma con più destinazioni. Il target potrebbe essere in un registro.

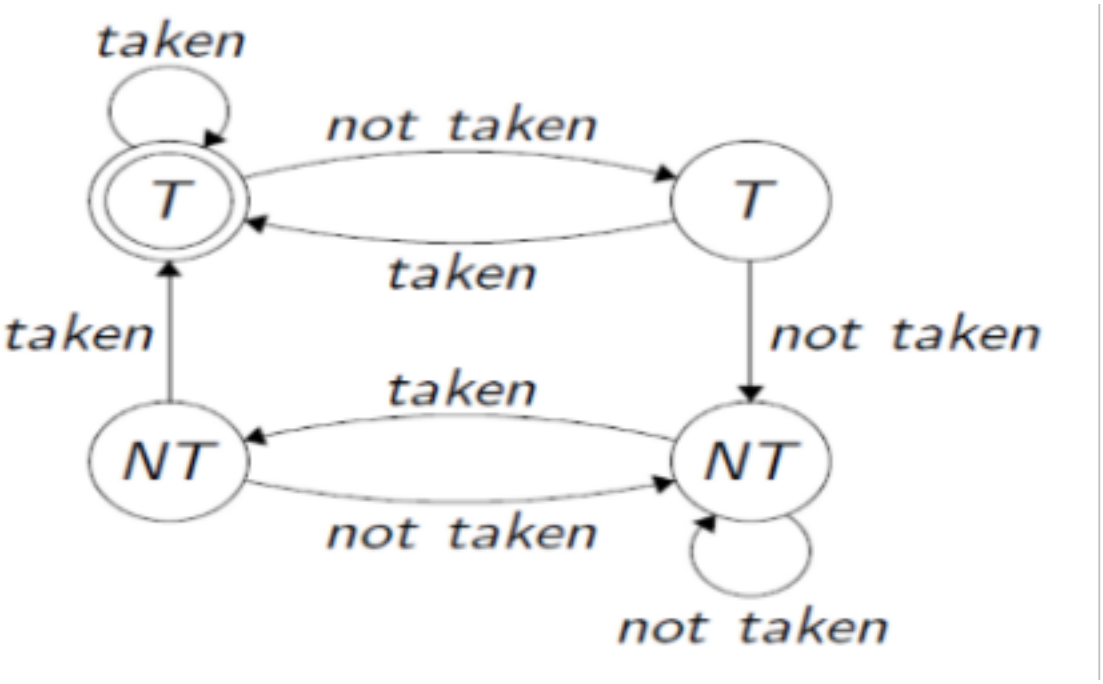
Predittori per i salti condizionali

Predittori coi bit di stato Il modo più facile per implementare un predittore per i salti condizionali è sfruttando un **unico bit di stato**: se l'ultima istruzione di salto ha avuto come esito "taken" (ovvero ha realmente portato a effettuare il salto), allora il predittore prevederà che anche il prossimo salto avrà come esito "taken", e viceversa. Qui la storia passata è rappresentata esclusivamente dall'*ultima istruzione di salto*, per cui non si tratterebbe neanche di un vero e proprio storico.
Esempio Caso semplice:

ciclo for, sbaglio la predizione quando uscirò. Per questo motivo, sono stati introdotti anche i predittori a *due bit* di stato. Di fatto, questi predittori si comportano meglio negli scenari in cui si hanno molte istruzioni di salto con esito "taken" e poche istruzioni di salto con esito "not taken" (e viceversa). Lo scenario classico è quello dei *nested loop*. In particolare, la predizione cambia da "taken" a "not taken" (o viceversa) non più a seguito di un solo errore, ma a seguito di **due errori consecutivi del predittore**. La macchina a stati che rappresenta i predittori a due bit è la seguente, dove:

- T = "predìco che il salto sarà taken"
- NT = "predìco che il salto sarà not taken"

Esempio doppio ciclo: Nel ciclo interno itero, e la predizione mi dice che salterò. Quando finisco le iterazioni nel ciclo interno, la predizione sbaglia e dice che continuerò nel ciclo. Invece aumento l'iterazione sul ciclo esterno (1 errore). Però poi rientro effettivamente nel ciclo interno, quindi in realtà non devo cambiare predizione anche se ho fatto un errore, perchè stavolta ci rientro davvero dentro. *Solo se sbaglio 2 volte cambio previsione.*



Ricordiamo che, in caso di previsione sbagliata, in pipeline vengono caricate istruzioni che alla fine subiranno uno squash, che porterà al refill della cache. Quindi è importante effettuare la predizione con un buon tradeoff affidabilità/supporto hw.

Esistono anche predittori più sofisticati come il **Two-Level Correlated Predictor** e l'**Hybrid Local/Global Predictor** (noto anche come **Tournament Predictor**).

Two-Level Correlated Predictor Consideriamo il seguente codice:

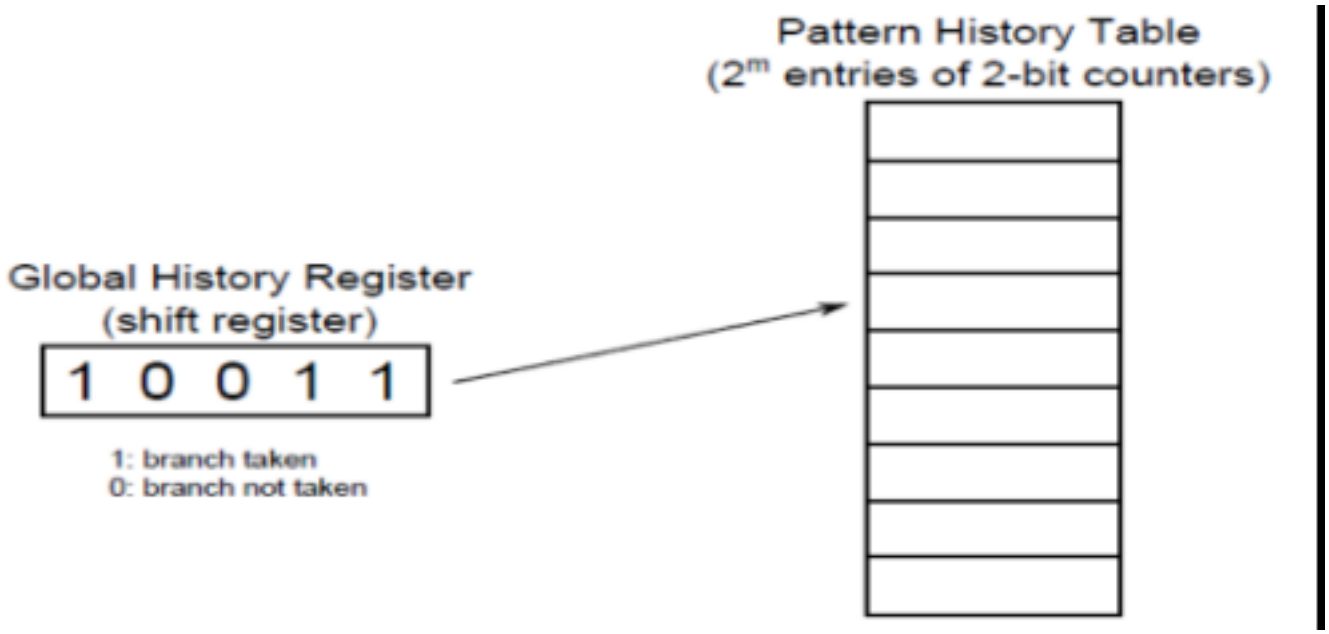
```
if (aa == VAL) aa = 0;
if (bb == VAL) bb = 0;
if (aa != bb) {
                //do the work
              }
```

Ci piacerebbe avere un predittore tale per cui, se i branch relativi ai primi due if vengono presi, predìca che il terzo branch (che dipende dai due precedenti) non venga preso. È quello che fa il **(m,n) Two-Level Correlated Predictor**, dove:

- m = *numero di salti precedenti*, il cui esito determina la predizione che verrà effettuata.
- n = numero di bit del valore che indica quanti **errori** deve commettere il predittore prima di **cambiare** la sua **predizione** (esattamente come nei predittori con bit di stato).

Esempio:

Consideriamo il caso con m=5, n=2. Si ha il seguente scenario:



Il **global history register** è composto da m bit, ciascuno dei quali indica l'esito di uno delle ultime m istruzioni di salto condizionale. Da destra ho i salti più recenti. Tale registro può assumere 2^m valori diversi, e a ciascuno di essi è associato un predittore con bit di stato (= una macchina a stati) differente. Il predittore per la predizione corrente viene scelto sulla base dei risultati degli ultimi m branches, come è codificato nella 2^m bitmask. Sostanzialmente usiamo il Global History Register come indice.

Un esempio semplice con (m = 2, n = 2).
Nel global register posso avere 4 casi: 00, 01 , 10, 11, dove 0 vuol dire "not taken" e 1 "taken". Per ogni casistica associo una entry nel Patter History Table.
00 mappato nell'entry 0, 01 mappato nell'entry 1, 10 mappato nell'entry 2, 11 mappato nell'entry 3.
Il fatto che $n = 2$ vuol dire che ogni entry mantiene 2 bit nell'array, ovvero include l'automa a stati visto precedentemente.

Se ho tre statement if, e salto sempre al terzo in modo condizionato, la mia branch sequence è 001001001001...
Se (m = indifferente, n = 2) mantengo 4 entry con due bit ciascuno (00, 01, 10, 11).

- Entry 00: ho sbagliato due volte, allora il terzo salto sono sicuro.
- Entry 01: ho sbagliato una volta, resto su 0.
- Entry 10: ho sbagliato una volta, resto su 0.
- Entry 11: non capita perchè abbiamo detto di non saltare due volte consecutivamente.

Se avessi (m = 0, n = 2), avrei 0 elementi per identificare l'entry. Sarebbe un predittore basico.

Tournament Predictor

In realtà non è sempre vero che i salti condizionali siano correlati tra loro. Per questo motivo si introduce l'**Hybrid Local/Global Predictor**, che consiste in una "sfida" tra un **predittore locale** (come quelli con bit di stato) e un **predittore globale** (che si basa sulla storia passata e assume che i salti condizionali siano correlati). In pratica, si hanno entrambi questi predittori e in più una macchina a 4 stati che indica se correntemente conviene utilizzare il predittore locale oppure quello globale: se stiamo sfruttando il predittore locale, *switchamo a quello globale dopo due*

errori consecutivi, e viceversa. Ciò è realizzato mediante *Return Stack Buffer RSB*, tipo una semplice cache, con 32 entries (quindi 32 livelli di annidamento), cioè indirizzi associati al return point dell'istruzione call, in cui salvo nella prima entry libera tale indirizzo di ritorno.

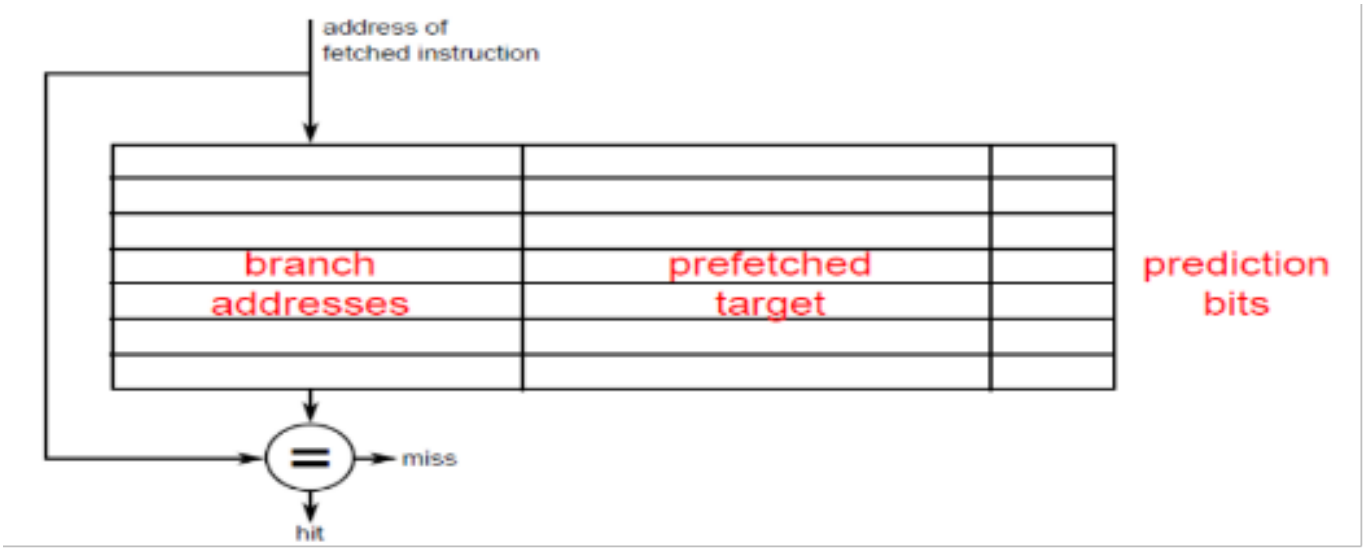
Osservazione su RSB Faccio call di una funzione, salvo indirizzo successivo alla call in RSB, quando esco dalla funzione il return punta all'indirizzo salvato nell'RSB precedentemente. RSB è una cache di tipo LIFO, contenuta nel processore ed è modificabile solo tramite *call* (inserimento valore) *o return* (prelievo valore). Non è esposta nell'ISA.

Questi switch e letture avvengono in elementi mantenuti in cache, allora non ho perdite di prestazioni.

Predittore per salti indiretti

I salti indiretti sono tali per cui l'indirizzo di destinazione viene caricato in un registro. Se l'istruzione deve saltare, questo è funzione del risultato delle istruzioni precedenti, ovvero di cosa queste hanno scritto nei registri, che verranno usati per saltare.

Quindi, il valore di tali registri può *variare sempre nel tempo*, il che rende la predizione più complessa e con più side effects generati. Per questo motivo, le destinazioni possibili possono essere molteplici, e la predizione può risultare più farraginosa. I predittori per i salti indiretti sono dotati della seguente struttura dati:



Ciascuna entry della tabella è relativa all'indirizzo di una particolare istruzione di salto (**branch address**). Ogni branch address è associato a un **prefetched target** (che è il *presunto* indirizzo destinazione del salto, se c'è cache miss non posso prelevarlo, oppure prendo l'istruzione successiva) e a dei **prediction bit** (che, come al solito, determinano numero di errori consecutivi da commettere prima di cambiare la predizione, ovvero prima di cambiare il prefetched target). In particolare, la prima volta che si incontra una certa istruzione di salto indiretto, si effettua il training, in cui il prefetched target verrà inizializzato all'effettivo indirizzo destinazione del salto. In queste tabelle riporto solo i bit meno significativi, per questioni di scalabilità. Tutto questo sta nel core, e se ho hyperthreading, la predizione viene fatta da una comune componente hardware per più thread. Ovvero se ho due thread (supponiamo facciano stesse cose) su due hyperthread e 1 core, questo core comune può essere sfruttato dal thread 2 per avere informazioni sui salti ereditate da thread 1, per ottenere un boost delle performance. A livello di sicurezza è un po' problematico: se thread1 seguisse un comportamento tale da suggerire al thread 2 di fare salti (anche speculativi) che in realtà non dovrebbe fare?

Attacco Spectre

Chiaramente anche la predizione dei target dei salti condizionali può portare il processore a riempire la pipeline con istruzioni eseguite in modo speculativo. In particolare, se la predizione è scorretta, le istruzioni successive al salto dovranno poi essere buttate. Ma anche qui, come nel caso delle eccezioni, le istruzioni considerate in maniera speculativa che poi vengono scartate lasciano dei side effect nello stato micro-architetturale. (Solito discorso: un salto speculativo non installa nulla nell'ISA, ma lascia tracce a livello micro architetturale). Da qui nasce la possibilità di compiere una famiglia di attacchi denominati **Spectre** che sfruttano un covert-channel.
Questi attacchi sono anche più gravi di Meltdown poiché:

- **Non richiedono l'esecuzione di un'istruzione offending.**
- È possibile effettuare un **training scorretto** del branch predictor inserendo nel codice di livello user degli appositi branch / delle istruzioni di salto condizionale.

L'unica soluzione per evitare questi attacchi sarebbe quella di non fare predizioni, ma i sistemi diventerebbero molto più lenti (con o senza hyperthread), a livello di marketing sarebbe una mossa controproducente!

Spectre V1

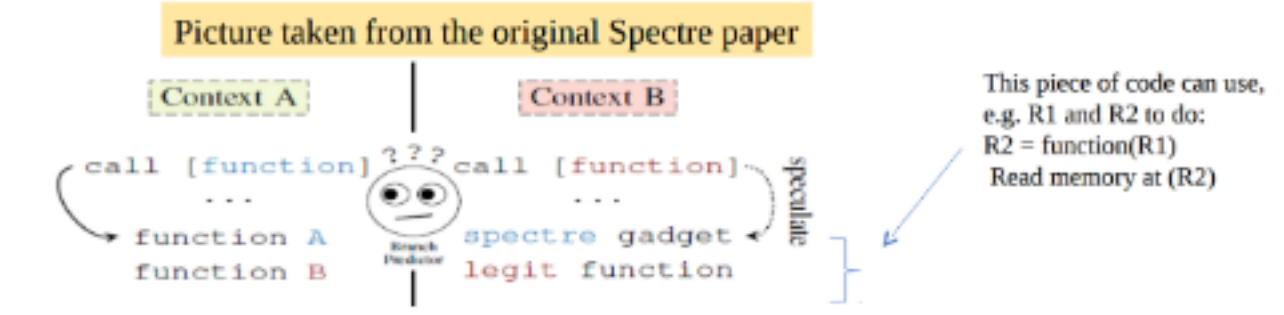
È un attacco che inizialmente effettua il flush della cache (come Meltdown) per poi sfruttare i salti condizionali. In particolare, prevede l'utilizzo del seguente blocco di codice:

```
if (x < array1_size)
  y = array2[array1[x] * 4096]
//il prodotto per 4096 corrisponde a uno shift a sx di 12 posizioni
```

- *array1* è un array che può trovarsi in qualunque punto dell'address space, e `array1[x]` è un particolare valore che verrà moltiplicato per 4096 per ottenere un indice di pagina, che verrà utilizzato per accedere ad *array2*.
- In ogni caso, il valore x viene selezionato in modo tale che sia molto elevato (maggiore di *array1_size*), cosicché il branch non venga realmente preso, mentre magari il predittore prevedeva il contrario; inoltre, x può essere tale che l'accesso ad *array2* porti a reperire (speculativamente) un'informazione segreta (y), ad esempio all'interno del kernel space.
- Quel che si ottiene è che il valore y viene caricato all'interno della cache in modo speculativo. A questo punto, come in Meltdown, si effettuano gli accessi alla prima entry di ogni pagina di *array2* finché non si giunge a quella caricata in cache (di cui si osserva un tempo di accesso ridotto).
- Quindi ciò che carico è in funzione sia dell'array sia di x, se salto molto lontano potrei andare in una zona kernel. Lo scopo è far si che questa condizione venga eseguita speculativamente, quindi il predittore deve essere indotto a credere che tale flusso di esecuzione sarà quello realmente eseguito. **Non devo eseguirlo realmente!**
- No trap perchè sono sempre in user mode e tutto dipende dall'esito dell'IF. Aver *patchato* Meltdown non implica risolvere anche Spectre. La soluzione di Meltdown risiedeva nel fatto che tale attacco creasse una trap poichè si passava da user a kernel, e c'era il cambio del puntatore alla page table. Qui sono sempre in user mode.

Spectre V2

Con più thread passo alla versione v2! È un attacco che sfrutta i salti multi-target (indiretti) ed è di tipo **cross-context**. Ciò vuol dire che l'attaccante è in grado di inferire (aggiungere) delle informazioni sensibili da un contesto di esecuzione diverso dal suo: supponiamo di avere due thread A, B che girano sul medesimo hyperthread in modalità processor sharing o su due hyperthread relativi allo stesso core. Il thread A può portare avanti delle attività che vanno a cambiare lo stato del branch predictor all'interno del CPU-core. Di conseguenza, B può osservare il nuovo stato del branch predictor all'interno del medesimo CPU-core e, quindi, le attività che lui eseguirà in modo speculativo (in funzione dello stato del branch predictor) vengono *praticamente decise, e poi osservate mediante side effect, dal thread A*. Tale side effect, come al solito, può consistere nel caricamento di un'informazione sensibile all'interno della cache attraverso il suo accesso in memoria. L'efficacia dell'attacco diventa particolarmente evidente quando l'attaccante si basa su un contesto che utilizza una libreria condivisa (shared library) col contesto del thread B. Qui, infatti, le pagine di memoria della shared library vista dal thread B e le pagine di memorie della shared library vista dal thread A mappano esattamente sugli stessi indirizzi fisici. Perciò è ovvio che qualunque side effect la vittima lasci speculativamente su tali locazioni fisiche di memoria sia direttamente visibile all'attaccante.



Con riferimento alla figura, il **gadget** è un blocco di codice che è definito nell'address space della vittima e viene sfruttato dall'attaccante affinché la vittima lo esegua in modo speculativo a seguito di un errore del branch predictor; in tal modo, nello stato micro-architetturale, la vittima lascia i side effect desiderati dall'attaccante. Nell'esempio mostrato nella figura vengono utilizzati due registri (*R1* e *R2*), di cui *R1* viene utilizzato per calcolare R2 con una particolare funzione, mentre *R2* viene usato per memorizzare l'indirizzo verso cui accedere in memoria. In realtà, sarebbe sufficiente l'utilizzo di un solo registro *R1*.

NB: **l'utilizzo del medesimo CPU-core tra thread A e thread B è richiesto solo per effettuare un training errato sul branch predictor**. Dopodiché, poiché la cache è condivisa tra più CPU-core, gli accessi in memoria effettuati per stabilire quali dati sono saliti in cache possono essere effettuati anche in un momento in cui il thread vittima (B) gira in un CPU-core differente. Tra l'altro, per portare a termine l'attacco, è possibile anche utilizzare una macchina virtuale su cui possono girare delle applicazioni che, in qualche modo, vanno a cambiare lo stato del branch predictor: in fondo l'hardware sottostante viene utilizzato anche per eseguire le macchine virtuali!

Contromisure per gli attacchi Spectre

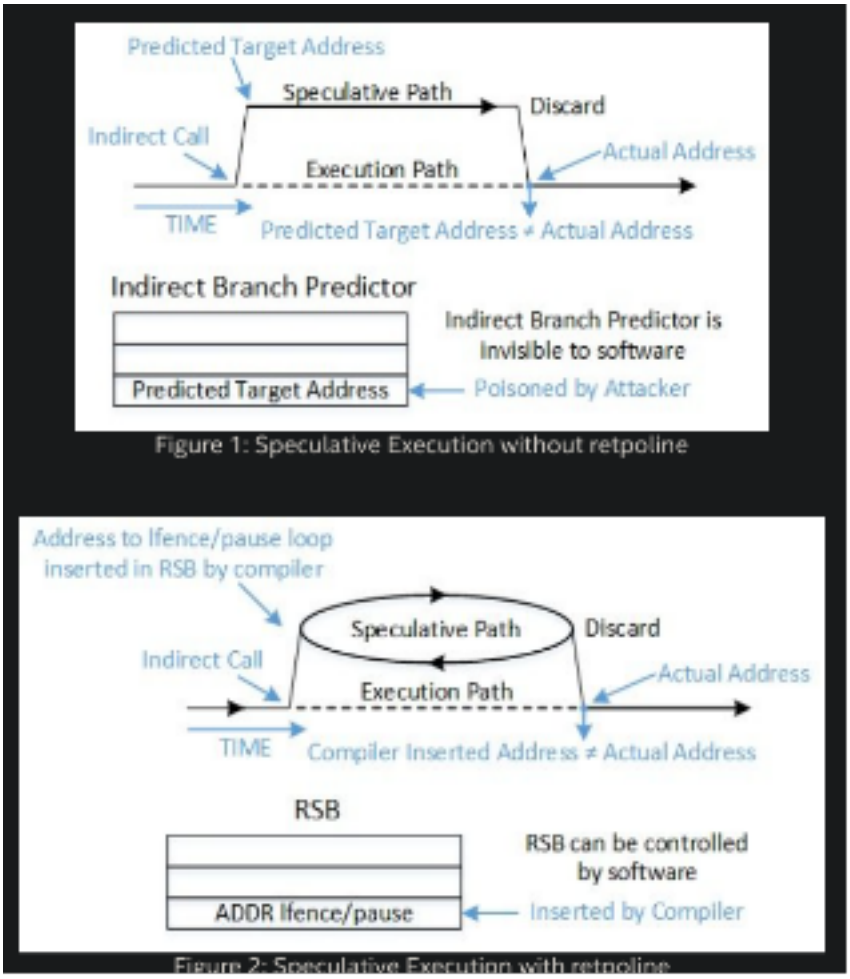
Retpoline - Return trampoline

Al posto di invocare l'istruzione di salto indiretto (che sia essa una jump o una call), si esegue un blocco di istruzioni funzionalmente equivalente che non consente di effettuare una mis-prediction (ovvero un training scorretto del branch predictor) per portare a compimento l'attacco Spectre. Una versione semplificata del blocco di istruzioni è riportata di seguito. Non faccio più salti indiretti, solo diretti!
Invece di *Jump su R*, eseguo *Return su R*, non c'è più predizione. Il *RET* usa il *Return Stack Buffer* per la predizione (di tipo *Lifo*), controllabile via software, a differenza del predittore di Branch Indiretto.

```
push target_address
1:  call retpoline_target
    //put here whatever you would like (with no side effects)
    jmp 1b
retpoline_target:
    lea 8(%rsp), %rsp //we do not simply add 8 to RSP since FLAGS registry should not be modified
    ret   //this will jump to target_address
```

Vediamo nel dettaglio cosa fa questo blocco di istruzioni:

- Carica l'indirizzo destinazione del salto (`target_address`) sullo **stack**. (ora è in cima allo stack)
- Effettua una call di `retpoline_target` (per cui viene caricato sullo **stack** *anche l'indirizzo dell'istruzione successiva alla call*). (Graficamente, nell'address space avremo *target_address* e sopra di lui, l'indirizzo successivo alla call. Questo perché chiamata *retpoline_target*, prevediamo che al suo *return* ripartiamo dall'istruzione successiva ad essa). `rsp` a 64 bit, quindi 8 byte, per questo faccio quello shift.
- In `retpoline_target` si **elimina** l'indirizzo dell'istruzione successiva alla call e, tramite la return, si salta verso l'indirizzo che si trova in cima allo **stack** (ovvero *target_address*). Facendo lo shift di 8 byte, cancelliamo il punto di ritorno della chiamata *retpoline_target.* Dopo ciò, in cima allo stack abbiamo proprio *target_address.*
- Essendoci una **call** (non è register dependent, mi manda verso la funzione chiamata), il predittore non deve essere soggetto a training: prevede che, a seguito della return, si salti verso l'istruzione successiva alla call. Per questo motivo, dopo la call devono essere eseguite delle istruzioni innocue, come una jump verso l'istruzione stessa di call. La *return* vede qualcosa nell' *rsb* che non può essere bypassata. La *ret* esegue speculativamente ciò che gli dice *rsb*. Tuttavia, se la CPU specula, l'RSB la porta ad eseguire *jmp 1b*, intrappolandolo in un loop. Successivamente, la CPU realizza che il valore in RSB è diverso da quello nello stack (*target_address*), e stoppa la speculazione. ***O vado in target_address, o resto in un loop.*** RSB ha senso per singolo processo, mentre il predittore si muove tra più flussi.



IBRS (Indirect Branch Restricted Speculation)

È possibile aggiornare il registro **MSR** (Model Specific Register, è un registro di controllo nella CPU) per creare un'enclave di esecuzione (= uno spazio di esecuzione chiuso entro determinati confini) dove la storia non è influenzata da cosa avviene al di fuori dell'enclave stessa. In pratica, a livello hardware siamo in grado di utilizzare la branch prediction in maniera differenziata a seconda se stiamo lavorando a livello user (MSR=0) o a livello kernel (MSR=1): nel primo caso il branch predictor dei salti indiretti si basa sulla storia passata della sola esecuzione a livello user, mentre nel secondo caso si basa sulla storia passata della sola esecuzione a livello kernel. Sostanzialmente, a tempo t decido che lo stato del predittore non dovrà più essere usato per un certo tempo. E' un guscio che mi protegge dall'esterno. Utile se ho app che lavorano a livelli diversi (user e kernel), se ad esempio volessi che le scelte kernel non venissero influenzate dall'user.

IBPB (Indirect Branch Prediction Barrier)

Anche qui ci si basa sull'utilizzo del MSR. In particolare, nell'istante in cui si va a scrivere all'interno di tale registro, stiamo cancellando tutto ciò che il branch predictor dei salti indiretti ha imparato finora, creando così una sorta di barriera. Al tempo t resettiamo il branch predictor.

IBRS e **IBPB** sono operazioni software, spesso si usa la syscall `prcrc()` per lavorare col Processor Control. Se lavoriamo con `exec()` (programma che chiama un altro programma) funzionano uguale, perchè facenti parte dello stesso programma iniziale.

Sanitizzazione

Introduzione

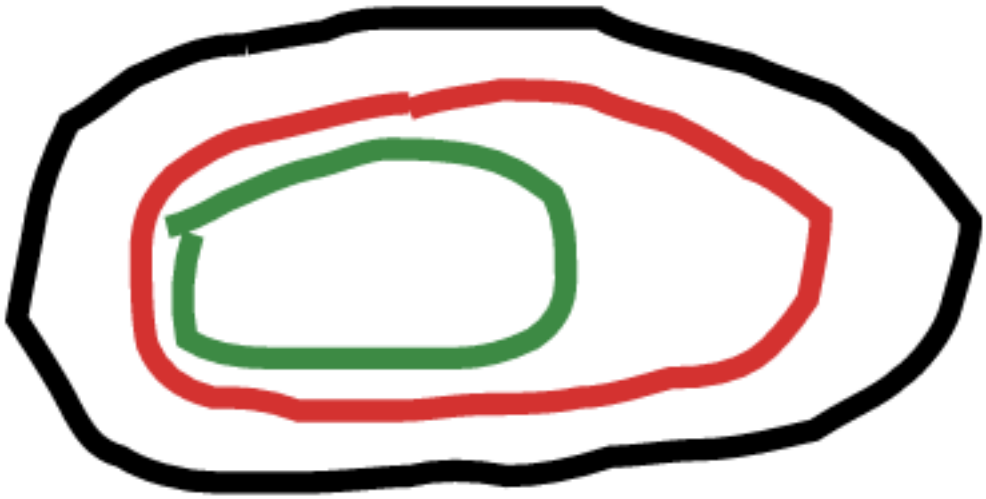
Per Spectre V1 abbiamo visto la possibilità di riusare le patch di Meltdown. Spectre V1 mi permetterebbe di leggere dati kernel? Sì, perchè la predizione a livello kernel potrebbe essere basata su predizioni livello user.
A livello kernel abbiamo una tabella, che chiamiamo **driver**, in cui per ogni indice ho un servizio che richiama una

system call. L'indice è passato dall'user, ma è compatibile con la taglia della tabella? Dipende, con l'indice facciamo un salto verso una entry, ma se eccedo la dimensione? **Speculativamente** starei usando un indice errato come pointer per una zona codice Kernel, perchè se le precedenti `call` erano corrette, il predittore si sarà settato in un certo modo. (Se ha sempre saltato, speculativamente lo rifarà). Soluzione? Sanitizzazione.

Come funziona

È una contromisura per i salti condizionali. In particolare, ciascuna condizione `if` che coinvolge una certa variabile V prevede dei valori per V ammissibili e dei valori per V non ammissibili. Quello che si fa è ridurre all'osso l'insieme dei valori non ammissibili, facendo sì che tutti gli altri valori non ammissibili non possano in alcun modo essere assunti da V. Questa è una tecnica estremamente utile per gli indici delle tabelle. Infatti, supponendo che per una tabella gli indici ammissibili vadano da 0 a $j-1$, se per qualche motivo l'indice dovesse assumere un qualunque valore maggiore di $j-1$, viene forzato ad assumere il valore j. In pratica, abbiamo imposto che j sia l'unico valore non ammissibile che può essere materializzato. A questo punto, se l'indice vale j, viene imposto ad esempio un accesso in memoria innocuo (i.e. alla prima locazione di memoria subito dopo la tabella vengono inserite delle informazioni non sensibili in modo tale che, anche speculativamente, viene acceduta o una entry della tabella o l'unica locazione di memoria ammissibile e innocua al di fuori della tabella).

Come funziona - for dummies



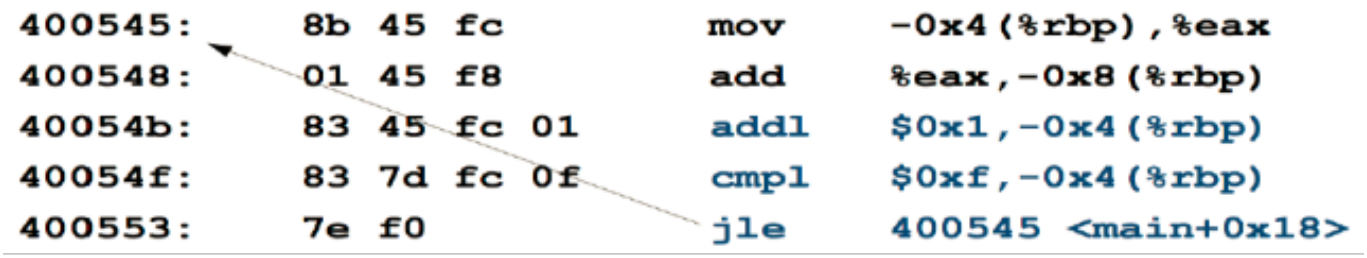
Normalmente avremmo "indici OK" (verde) ed "indici NON OK" (rosso). Però se usassi questi indici NON OK, potrei andare in zone delicate. Introduco la "zona nera", più ampia. Tutto quello che cade li dentro lo butto nella zona rossa, che sarà la più piccola possibile, per limitare i danni.
Come lo faccio? Applicando una maschera di bit.
Perchè devo differenziare zona rossa e nera?
Perchè se lascio solo zona rossa posso andare in zone brutte. Se questa è piccola (ad esempio un indirizzo non critico) e faccio si che tutti gli altri indici *NON OK* vadano lì dentro, limito i danni.
Perchè non voglio che applicando questa maschera ad un indice NON OK, questa mi porti ad una zona verde. Quindi prima applico la maschera, e poi faccio il controllo.

Loop unrolling

Ricordiamo che i salti sono azzardi, sbagliare salto compromette performance (squash pipelin) e sicurezza. E se riducessimo i salti?

```
int s=0;
for (int i=0; i<16; i++)
    { s+=i;}
```

Questo ciclo si traduce nella seguente sequenza di istruzioni assembly (dove la sintassi adottata è AT&T):



Come si può notare, a ogni iterazione del ciclo vengono eseguite cinque istruzioni, di cui tre di controllo (in blu scuro) e solo due di lavoro effettivo, in cui viene incrementata la variabile `s` (in nero): in pratica, in questo particolare esempio, abbiamo un overhead di computazione pari ai 3/5 delle istruzioni totali, il che porta inevitabilmente a un degrado delle prestazioni. Per questo motivo, corre in aiuto il **loop unrolling**, che è una tecnica per "srotolare" i cicli: da una parte si riduce il numero di iterazioni che devono essere eseguite, dall'altra, all'interno di ciascuna iterazione, si esegue il lavoro utile più volte. In tal modo, si riduce il numero di volte in cui devono essere eseguite le istruzioni di controllo. Per effettuare l'unroll in modo automatico, è possibile ricorrere alle seguenti direttive di C:

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
//region to unroll
#pragma GCC pop_options
```

È anche possibile specificare esplicitamente il fattore di unroll mediante `#pragma unroll (N)`. Ma affinché queste direttive siano effettivamente attive, è necessario compilare il file C col *flag -O*.

Attenzione: il loop unrolling porta alla necessità di utilizzare un maggior numero di registri per eseguire tutte le operazioni della medesima iterazione. Inoltre, porta ad avere le istruzioni macchina del ciclo su un range di indirizzi più ampio, per cui si ha una minore località. Per questi motivi, non è una tecnica che può essere sfruttata in modo spregiudicato. Per sfruttarlo bene, bisogna capire architettura e obiettivi!

Power wall

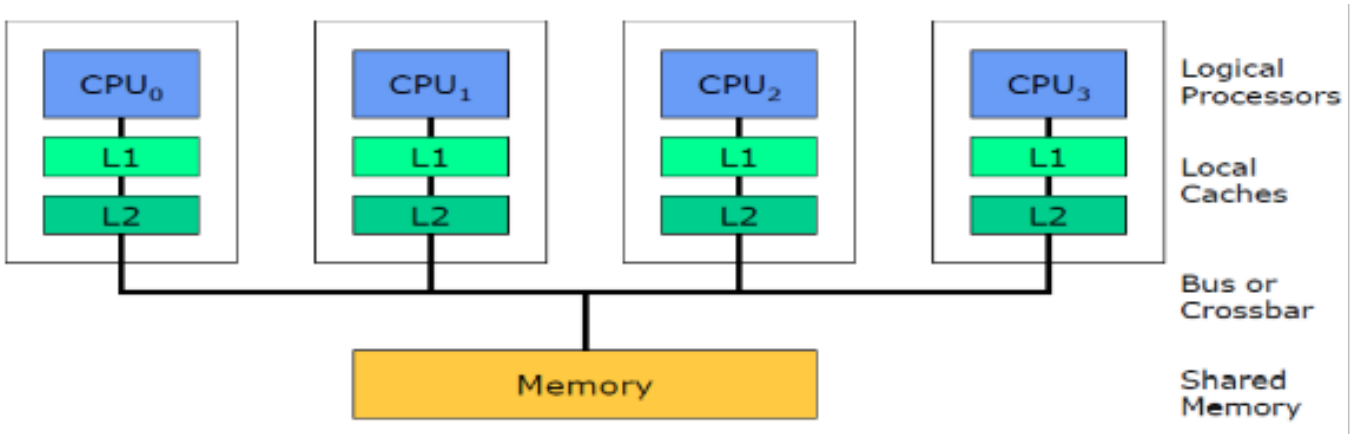
Per aumentare le prestazioni dei processori, è possibile seguire due approcci:

1. Aumentare la frequenza del clock.
2. Incrementare i componenti hardware a supporto del processore.

Per quanto riguarda la strategia 1, c'è una limitazione: la dissipazione massima che un processore può tollerare senza che si bruci è 130 W. Ma la dissipazione è pari a $V \cdot V \cdot F$, dove V è il voltaggio ed F è la frequenza del clock. V non può essere al di sotto di una certa soglia minima, altrimenti i componenti hardware non funzionerebbero proprio. Di conseguenza, esiste un upper-bound per F che è stato già raggiunto. Questa limitazione è detta **power wall**. A causa del power wall, ad oggi siamo obbligati a seguire la strategia 2 per rendere più efficienti i processori. In particolare, nel tempo, sono state adottate le soluzioni architetturali descritte di seguito.

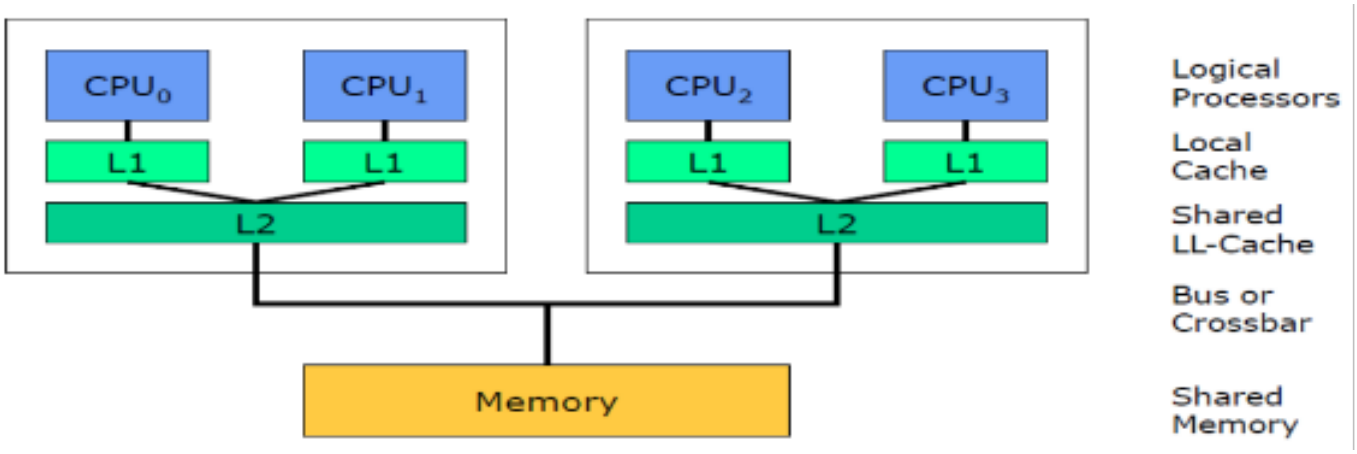
Symmetric Multiprocessors

Si hanno semplicemente molteplici processori, ciascuno dei quali, per accedere alla memoria, impiega la stessa quantità di tempo. Abbiamo per ogni core un thread.



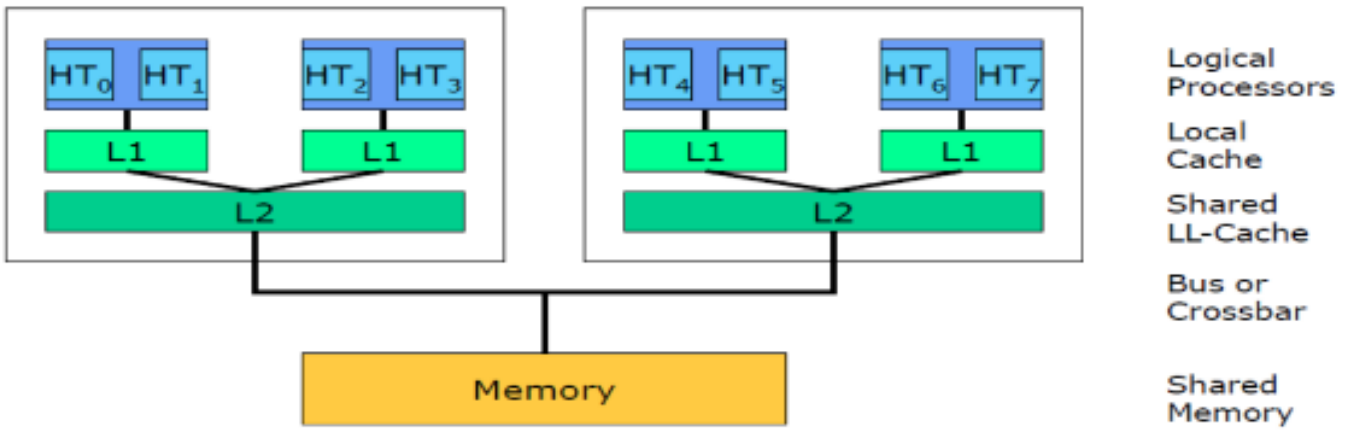
Chip Multi Processor (CMP) o Multicore

All'interno di ciascun processore si hanno più motori (più core). Un hypethread per core.



Symmetric Multi Threading (SMT) o Hyperthreading

All'interno di ciascun core possono esserci **più hyperthread distinti**. Dal punto di vista del sistema operativo, è esattamente come se ci fosse un numero di processori pari al numero totale di hyperthread. Il problema qui è che la memoria rappresenta un collo di bottiglia importante, anche perché viene acceduta in modo concorrente da tutti gli hyperthread. Dobbiamo vederla come architettura distribuita. La memoria è esposta in ISA (ma non le componenti cache $L_1, L_2 \dots$).

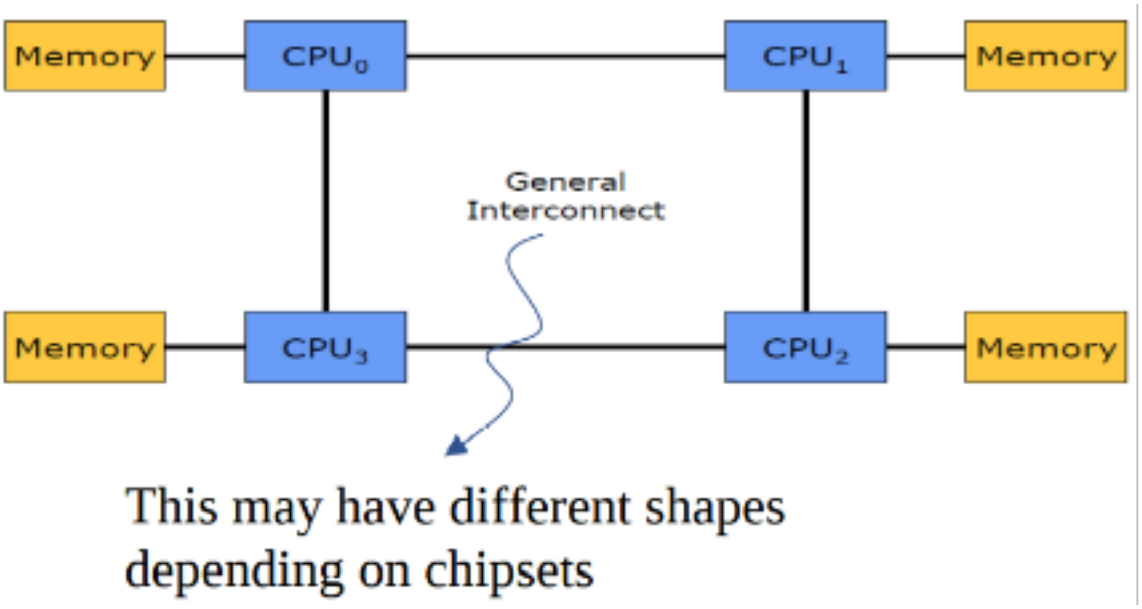


Successivamente si è passati all'**UMA**: memoria unica ad accesso uniforme, cioè stesse componenti hardware accedute in parallelo. Poi **NUMA**.

Non Uniform Memory Access (NUMA):

Qui la memoria è divisa in banchi, ciascuno dei quali è direttamente collegato a uno specifico core (o a un insieme di core). Se un thread che gira in CPU-core i accede al banco di memoria a esso vicino, l'interazione con la memoria risulta molto efficiente, mentre se deve accedere a un altro banco di memoria, impiegherà più tempo. Tale soluzione risulta vantaggiosa nel momento in cui si riesce a fare in modo che ciascun CPU-core acceda prevalentemente al *proprio* banco di memoria (ovvero effettui prevalentemente degli accessi locali). Ciascuna **coppia** (CPU-core i, *banco di memoria* i) o (insieme i di CPU-core, banco di memoria i) costituisce un **nodo NUMA**. I thread possono leggere da indirizzi diversi (quello che abbiamo detto prima), tuttavia l'interconnect è time shared, si genera traffico, che deve essere ben gestito.
Osservazione:

- Un accesso locale (CPU verso memoria adiacente) richiede un tempo pari a *50(hit)+200(miss)* cicli, quindi è anche facile capire, osservando il tempo, se si ha hit o miss.
- Se l'accesso è NON locale e SENZA TRAFFICO, si passa a *200(hit)+300(miss)*. Tempi totalmente diversi, che possono creare **problemi di coerenza nella cache.**



NB: ISA definisce come il processore interpreta e esegue le istruzioni, ma non specifica i dettagli sulla gerarchia di memoria o la presenza delle cache.

Cache Coherency

Com'è possibile osservare, nelle architetture di memoria elencate precedentemente la cache è **replicata**: da qui sorge il problema della **cache coherency**, secondo cui bisogna stabilire qual è il v*alore corretto da restituire a un thread che effettua una determinata lettura in memoria.* Disponiamo infatti di un processore e di una zona di memorizzazione (cache e RAM). Osservare solo l'interfaccia memoria-processore ci fornisce una visione limitata. Se una istruzione scrive una `mov`, questa passa prima per lo *store buffer*, non viene subito esposto nell'ISA. Se un altro programma dovesse leggere, non è detto che il valore sia già stato scritto. La coerenza viene **definita** in tre punti fondamentali:

- **Causal consistency**: se un processore p_1 scrive un valore v nella locazione di memoria X e poi effettua una lettura da X, dovrà leggere il valore v, a meno che non ci siano altri processori che nel frattempo hanno finalizzato delle scritture concorrenti su X (coerenza di tipo RAW – Read After Write).
- **Avoidance of staleness**: se un processore p_1 scrive un valore v nella locazione di memoria X (memoria qui intesa come RAM, visibile su ISA) e *dopo una quantità sufficiente di tempo* un altro processore p_2 effettua una lettura da X, p_2 dovrà leggere il valore v, a meno che non ci siano altri processori che nel frattempo hanno finalizzato delle scritture concorrenti su X.
- **Avoidance of inversion of memory updates**: le varie scritture su uno stesso dato X devono essere viste da tutti i processori nello stesso ordine. Se ho una sequenza di scrittura dalla cache verso la RAM, l'ordine di scrittura in cache deve essere riproposto anche nella RAM. Queste proprietà valgono sulla singola locazione.

Le due principali tecniche utilizzate per mantenere la consistenza, ricordando che la copia master è in memoria, e le copie in cache, sono:

- **Write through cache:** Prima aggiorniamo in cache e poi in memoria. Questo porta al seguente problema: una CPU potrebbe leggere due volte lo stesso dato dalla cache, e tra le due letture un'altra CPU può aver toccato il dato e aggiornato in memoria. Quindi per la CPU in esame, abbiamo una incoerenza tra quello che c'è in cache e quello che c'è in memoria.
- **Write back cache:** Aggiornare la cache e procedere alla memorizzazione in un secondo momento che decidiamo noi. Il problema è che lasciar decidere a noi potrebbe portare a scritture in memoria non nello stesso ordine di come sono state effettivamente eseguite.

Protocolli CC Cache Coherency

Permettono di conseguire la cache coherency e sono il risultato delle scelte di:

- Un insieme di transazioni (e.g. aggiornamento di una replica della cache) supportate dal cache system distribuito.
- Un insieme di stati per i cache block (= unità minime della memoria che possono essere portate all'interno della cache).
- Un insieme di eventi che capitano all'interno del cache system distribuito.
- Un insieme di transizioni di stato.

Il design dei protocolli CC dipende da svariati fattori come:

- La topologia dei componenti (e.g. single bus, gerarchica, ring-based).
- Le primitive di comunicazione (i.e. unicast, multicast, broadcast).
- Le cache policy (e.g. write-back, write-through).
- Le feature della gerarchia di memoria (e.g. numero di livelli della cache, inclusiveness).

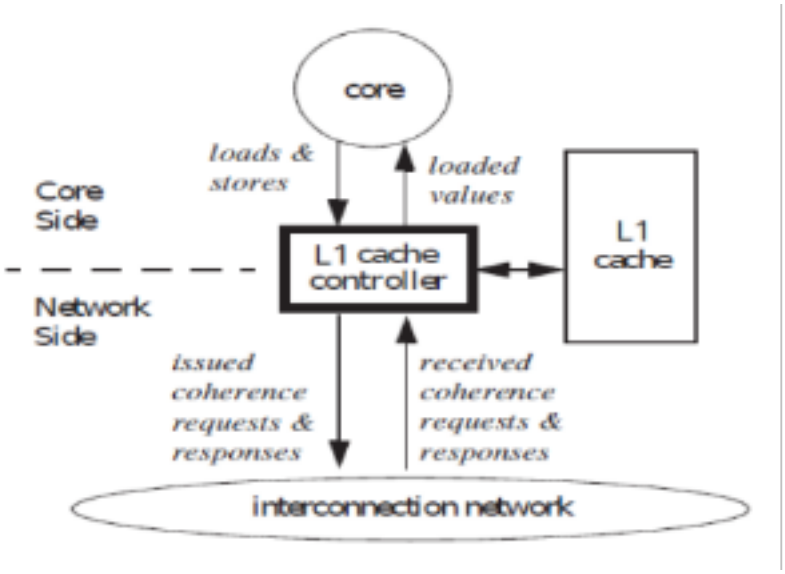
Implementazioni di cache coherency differenti possono presentare prestazioni diverse in termini di:

- **Latenza**: tempo per completare una singola transazione.
- **Throughput**: numero di transazioni completate per unità di tempo; aumenta se si fa in modo che scritture su aree di memoria diverse (e sufficientemente distanti) da parte di thread differenti avvengano in modo concorrente.
- **Overhead spaziale:** numero di bit richiesti per mantenere lo stato di un cache block.

Esistono due famiglie principali di protocolli CC:

- **Invalidate protocols**: quando un CPU-core scrive su una copia di un blocco, qualsiasi altra copia del medesimo blocco (i.e. della medesima informazione) viene *invalidata*: soltanto chi ha scritto l'ultima informazione ha la versione aggiornata del blocco. In tal caso, si utilizza poca banda ma si ha una latenza elevata per gli accessi in lettura: infatti, un CPU-core che si trova vicino a una replica invalidata, per andare a leggere il dato, ha la necessità di sfruttare l'unica replica valida, che è lontana. Sostanzialmente se aggiorno una linea di cache, spariscono tutte le altre repliche. Leggeranno tutti da me, perchè ho l'unica copia valida.
- **Update protocols**: quando un CPU-core scrive su una copia di un blocco, la nuova informazione viene propagata su tutte le altre copie del medesimo blocco. In tal caso, si ha una bassa latenza per gli accessi in lettura ma utilizza molta più banda.

Per poter invalidare / modificare una replica di un blocco che è stato aggiornato, si ricorre al cosiddetto **Snooping cache**: le componenti della cache e della memoria sono agganciati a un **broadcast medium** (= interconnection network, è un canale di comunicazione broadcast) tramite un controller, che ha la responsabilità di osservare le transizioni di stato degli altri componenti e di far reagire il componente locale di conseguenza (appunto invalidando o modificando la replica locale del blocco aggiornato). Naturalmente, il controller utilizza il broadcast medium anche per trasmettere gli aggiornamenti che avvengono in un blocco di cache locale. Mediante il broadcast medium siamo in grado di serializzare le transizioni di stato. *Inoltre, una transizione di stato non può occorrere finché il broadcast medium non viene acquisito in uso dal controller.*



Nel mondo reale viene adottato lo *Snooping cache* accoppiato con un protocollo basato sull'invalidazione. Vediamo dunque nel dettaglio alcuni di questi protocolli basati sull'invalidazione. Ad esempio, una componente di cache L_1 ha una replica r, vuole eseguire operazione su tale replica mediante il richiamo di broadcast medium, in cui annuncio alle altre componenti che, chi ha una copia della replica r, deve aggiornare. Nel mentre, nessuno può eseguire altri aggiornamenti se ho io il broadcast medium. Ciò crea un problema di scalabilità, se lavoro con tanti dati e repliche.

Osservazione: devo per forza aggiornare le repliche? E se non mi servissero? Ad esempio, un thread su CPU_0 con cache L_1 vi esegue un aggiornamento su linea di memoria e comunica in maniera distribuita di cancellare quella linea di cache. Ma se quella linea di cache fosse già invalida per altri?
Si mantiene traccia di queste operazioni mediante:

Protocollo MSI (Modified-Shared-Invalid)

È un protocollo in cui qualsiasi transazione di scrittura su una **copia di un cache block** invalida tutte le altre copie del cache block. Quando dobbiamo servire delle transazioni di **lettura**:

- Se la policy della cache è **write-through**, recuperiamo semplicemente l'ultima copia aggiornata dalla *memoria*.
- Se la policy della cache è **write-back**, recuperiamo l'ultima copia aggiornata o dalla memoria o da un altro componente di *caching*.

In questo protocollo è necessario tenere traccia dello stato della copia di un blocco:

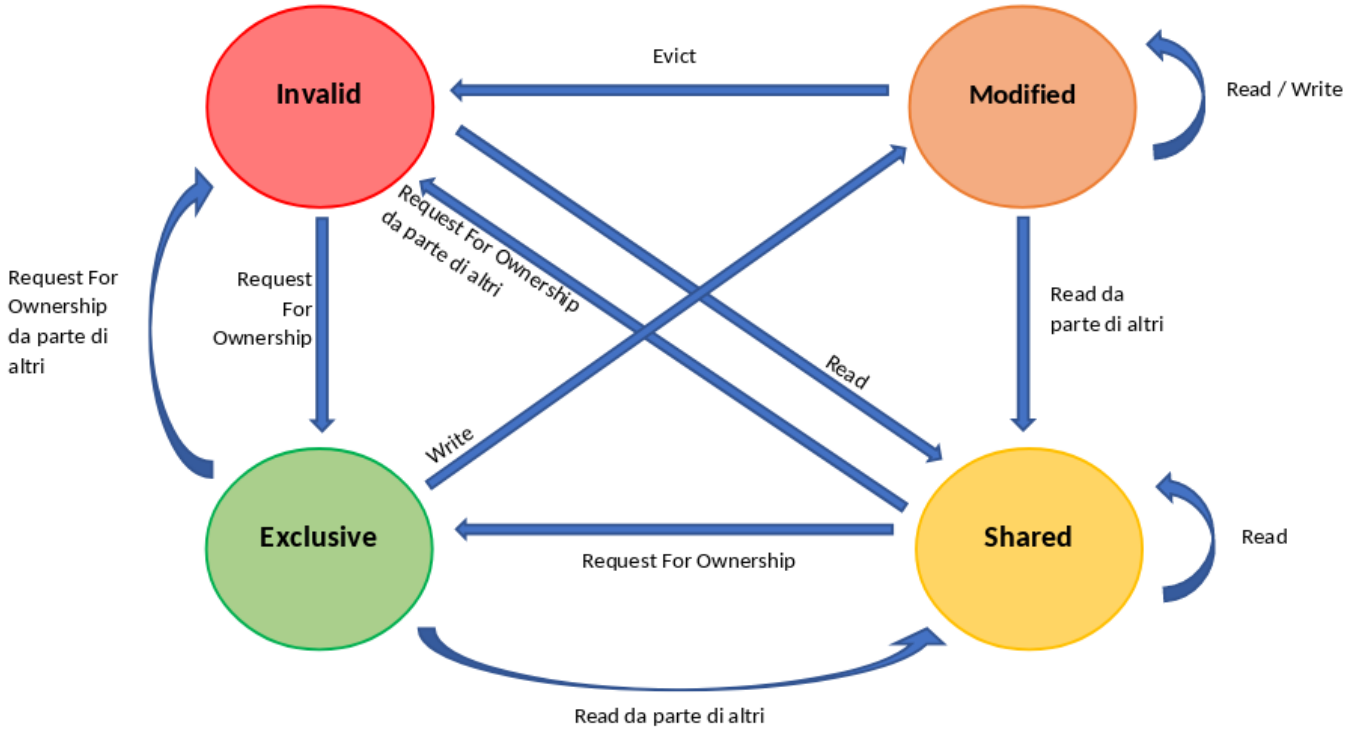
- Si trova nello stato *modified* se è appena stata sovrascritta.
- Si trova nello stato *shared* se esistono dei CPU-core che stanno leggendo altre copie del medesimo blocco (per cui esistono più copie *valide* di tale blocco).
- Si trova nello stato *invalid* se un'altra copia del medesimo blocco è appena stata sovrascritta.

Il **problema** di MSI risiede nel fatto che richiede l'invio di un messaggio di invalidazione in broadcast (tramite il broadcast medium) ogni volta che una copia di un blocco viene modificata (e questo anche in caso di scritture successive sulla medesima copia). Questo può portare a impegnare massivamente il broadcast medium anche quando non ce n'è bisogno, perché magari tutte le altre copie erano nello stato invalid già da prima. Per ovviare a tale inconveniente, si ricorre ai protocolli descritti successivamente.

Protocollo MESI (Modified-Exclusive-Shared-Invalid):

Rispetto a MSI, prevede uno stato in più, che è **exclusive**. In pratica, un CPU-core i che vuole apportare delle modifiche alla propria copia di un blocco deve richiedere l'**ownership** (*l'esclusività*) di quel blocco; questa è l'unica occasione in cui CPU-core i utilizza il broadcast medium per comunicare agli altri nodi che devono *invalidare* la propria copia del blocco. Una volta che la copia è nello stato *exclusive*, CPU-core i può modificarla tutte le volte che vuole *senza dover comunicare più nulla* a nessuno, finché un altro CPU-core non richiede di effettuare una lettura da un'altra copia dello stesso cache block (momento in cui la copia passa allo stato *shared*). L'automa raffigurato di seguito descrive in modo completo il funzionamento del protocollo MESI. Schematizzando:

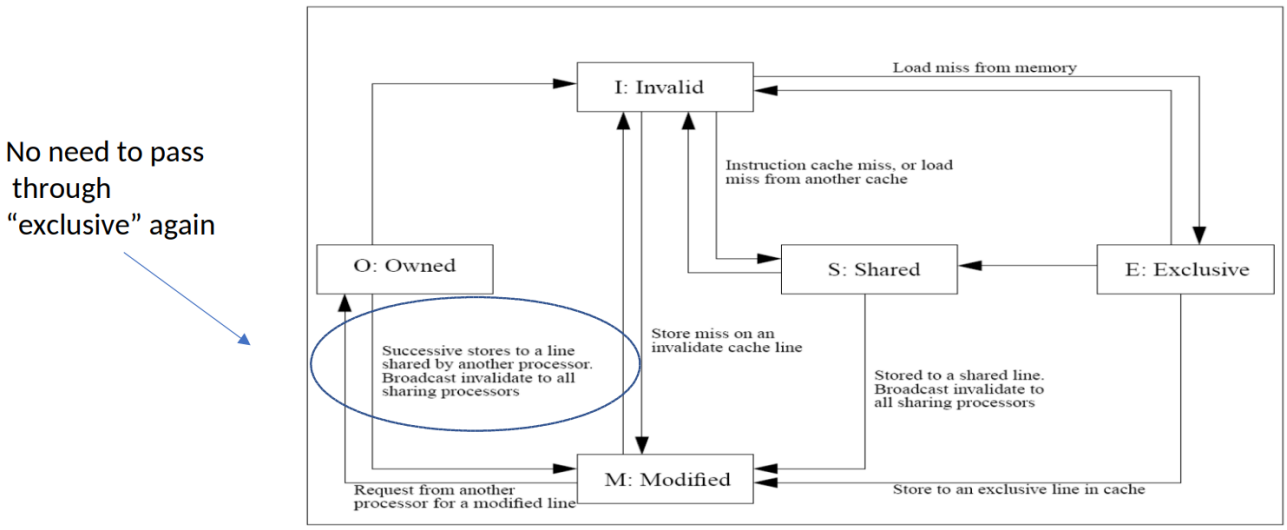
- **Modified**: Ho modificato un dato shared.
- **Exclusive**: Sono il solo proprietario del dato, posso modificarlo liberamente, senza bus message (passando a Modified).
- **Shared**: Ho una copia del dato che un altro processo possiede.
- **Invalid**: La mia copia del dato non è aggiornata.



Ogni volta che si esce dallo stato modified, si effettua il *write back* in memoria delle nuove informazioni che sono state scritte. Se volessi solamente leggere? Chi ha l'informazione si trova in *exclusive* e poi passa a *shared*. Potrei introdurre un quinto stato per evitare queste due transizioni. La linea exclusive è a mio uso esclusivo. Solo lo stato Invalid crea interazioni distribuite. Lo stato Modified è importante per cache write back, ma non devo informare tutti.

Protocollo MOESI (Modified-Owned-Exclusive-Shared-Invalid):

Rispetto a MESI, prevede un ulteriore stato in più, che è **owned**. In particolare, una copia di un cache block passa dallo stato modified allo stato *owned* (anziché shared) nel momento in cui stava per essere sovrascritta ma un'altra copia dello stesso blocco viene letta. *Owned* significa che la copia è tuttora in fase di aggiornamento ma, nel frattempo, altre copie dello stesso blocco vengono lette: se devono occorrere nuovi aggiornamenti quando si è nello stato *owned*, non ci si deve preoccupare di chiedere nuovamente l'esclusività del blocco, bensì si passa direttamente allo stato modified, invalidando tutte le altre copie. Di seguito è rappresentato l'automa completo del protocollo MOESI.



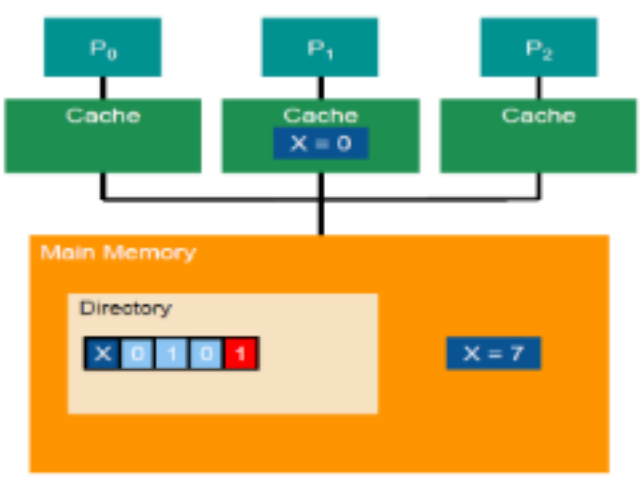
Nello stato **Owned** io ho l'uso esclusivo di un dato. Se mi chiedono di condividerlo, gli altri **non** possono scriverci. Tale stato ci permette di transitare tra più stati, in quanto non devo riacquisire l'esclusività.

Protocolli directory based:

In realtà, i protocolli basati su snooping cache non scalano poiché le transazioni portano a una comunicazione broadcast. Per questo motivo sono stati introdotti i protocolli directory based, in cui i vari aggiornamenti possono essere point-to-point tra i vari CPU-core / processori. In particolare, supponiamo di avere un sistema con N processori P_0, P_1,..., P_{N-1}. Per ciascun blocco di memoria si mantiene una directory entry, composta da:

- N bit di presenza (l'i-esimo bit è impostato a 1 se il blocco si trova nella cache di P_i).
- 1 dirty bit, che indica se il blocco è stato **modificato** senza che gli aggiornamenti siano stati riportati in memoria.

Se dirty bit = 0 e ho più possessori, linea cache condivisa.
Dirty bit = 1 se qualcuno lo scrive, però per passare a condiviso deve ritornare a 0. E' una machera che mi dice chi può fare cosa.



- **Caso 1**: P_0 vuole *leggere* il blocco X, ha un cache miss e il dirty bit è pari a 0. X viene letto dalla memoria, il bit `presence[0]` viene settato a 1 e i dati vengono consegnati al lettore.
- **Caso 2**: P_0 vuole *leggere* il blocco X, ha un cache miss e il dirty bit è pari a 1. X viene richiamato dal processore P_j tale che `presence[j]=1`. Dopodiché il dirty bit viene settato a 0, il blocco viene condiviso e il bit `presence[0]` viene settato a 1. Infine, i dati vengono consegnati al lettore e propagati in memoria.
- **Caso 3**: P_0 vuole *scrivere* sul blocco X, ha un cache miss e il dirty bit è pari a 0. P_0 invia un messaggio di invalidazione non a tutti gli altri processori, bensì solo a quelli col bit `presence[j]` pari a 1. Dopodiché tali `presence[j]` vengono settati a 0, e `presence[0]` e il dirty bit vengono impostati a 1.

- **Caso 4**: P_0 vuole *scrivere* sul blocco X, ha un cache miss e il dirty bit è pari a 1. X viene richiamato dal processore P_j tale che `presence[j]=1`. Dopodiché `presence[j]` viene settato a 0 e `presence[0]` viene impostato a 1.

Implementazioni x86

Intel:

- Principalmente **MESI**.
- **Cache inclusive** (tutte le informazioni che si trovano in un particolare livello della memoria sono riportate anche in tutti i livelli *inferiori* della memoria).
- **Write back** (gli aggiornamenti effettuati su un blocco di cache B verranno riportati in memoria solo quando B dovrà essere rimosso dalla cache - "evicted").
- Cache L1 composta da **linee** (blocchi) **di 64 byte**.

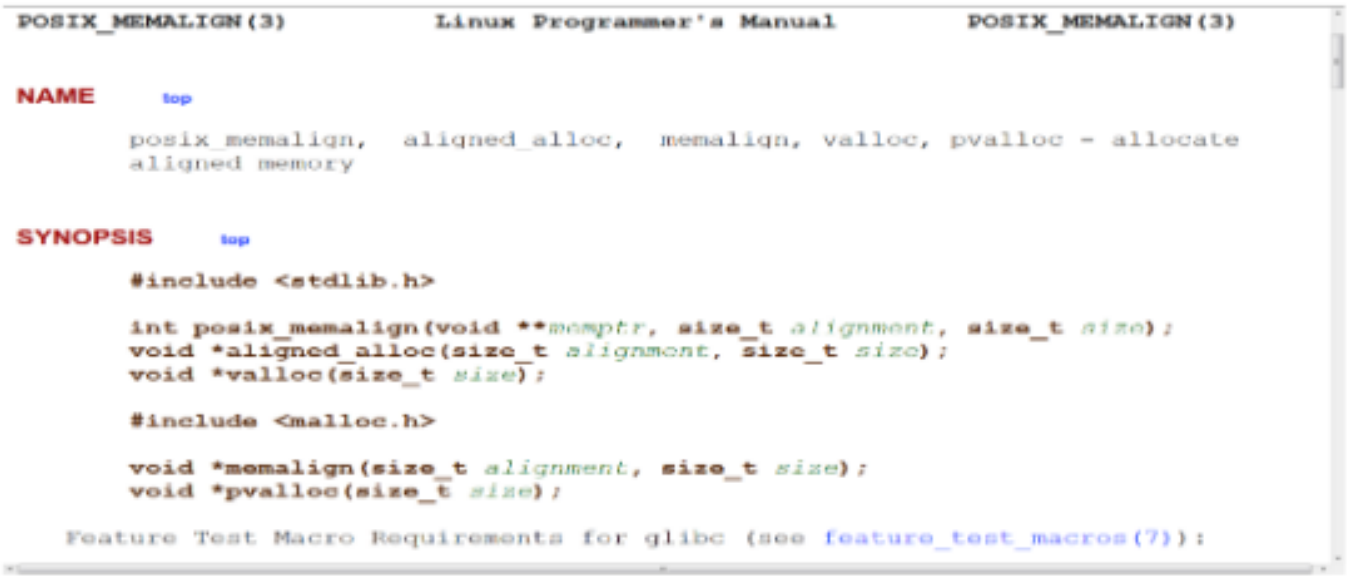
AMD:

- Principalmente **MOESI**.
- **Cache non inclusive** (o esclusive), in particolare al livello L3 (non è detto che la cache L3 contenga tutti i blocchi presenti in cache L1 e in cache L2, magari dato sale in L1 ma non è detto che sia in L3); qui la cache L3 è utile per ospitare i blocchi che devono essere rimossi dai livelli superiori della cache. Sicuramente avere cache non inclusive rende il sistema meno vulnerabile ad attacchi di tipo side-channel che si basano sull'utilizzo della cache.
- **Write back**.
- Cache L1 composta da **linee** (blocchi) **di 64 byte**.

False cache sharing

Supponiamo di avere un thread A che deve eseguire delle operazioni di scrittura su un dato X, e supponiamo di avere un thread B che deve eseguire delle operazioni di scrittura su un dato Y, dove X e Y sono *independenti* tra loro. Idealmente, i due thread devono poter effettuare le loro scritture in modo concorrente. Se però X e Y si trovano sulla **stessa linea di cache**, abbiamo che A, per scrivere su X, deve richiedere in uso esclusivo tutti i 64 byte che costituiscono il blocco e, quindi, anche Y; d'altra parte, B, per scrivere su Y, deve richiedere in uso esclusivo l'intero blocco e, quindi, anche X. Questo scenario, noto come **false cache sharing**, porta all'inondazione di transazioni distribuite dovuta a un ping-pong tra A e B di *Request For Ownership* per la medesima linea di cache: da qui consegue un crollo delle performance. Tale effetto deleterio lo avremmo avuto anche nel caso in cui A avesse dovuto eseguire delle operazioni di scrittura su X mentre B debba eseguire delle operazioni di lettura su Y.

Comunque sia, dobbiamo essere attenti anche allo scenario duale: se il thread A deve eseguire delle operazioni correlate tra loro sia sul dato X che sul dato Y, stavolta è opportuno avere X e Y sulla stessa linea di cache: in tal modo, diminuiamo la probabilità di ritrovarci dei dati scorrelati all'interno del blocco in cui si trovano X e Y.



POSIX_MEMALIGN(3) Linux Programmer's Manual POSIX_MEMALIGN(3)

NAME top

posix_memalign, aligned_alloc, memalign, valloc, pvalloc - allocate aligned memory

SYNOPSIS top

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
void *valloc(size_t size);

#include <malloc.h>

void *memalign(size_t alignment, size_t size);
void *pvalloc(size_t size);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

Se `ALIGNMENT = 64`, sto allineando esattamente a una linea di cache. Parto sicuramente da li, poi in base alla size potrei eccedere.

Attacco Flush + Reload

È l'antenato degli attacchi Meltdown e Spectre visti precedentemente ed è stato documentato per la prima volta nel 2013. In quest'attacco si hanno due thread: A (la vittima) e B (l'attaccante); che hanno la possibilità di accedere alle medesime informazioni in memoria. B può eseguire il **flush** della cache per poi eseguire degli accessi cronometrati in memoria (**reload**): se per leggere un dato impiega poco tempo, vuol dire che, nel frattempo, quello stesso dato è stato **ricaricato** in cache da parte di A (e quindi viene usato da A). Questo meccanismo può essere applicato anche sulle istruzioni macchina: anch'esse, quando vengono accedute per essere eseguite, vengono caricate in cache. Ciò vuol dire che B può essere in grado di capire anche quali sono le attività che A sta svolgendo. **Nell'ISA, l'unica operazione esposta per la cache è il FLUSH. Non posso fare altro!**

L'implementazione di Flush+Reload su x86 è basata su due building block:

- Un timer ad alta risoluzione. (**RDTSC**, *32 bit edx, 32 bit eax* (tot 64)per ogni processore).
- Un'istruzione non privilegiata che effettui il flush della cache. (quindi togliere contenuto cache con cflush, dando l'indirizzo della memoria).

Le immagini riportate di seguito mostrano i dettagli di queste due istruzioni.

RDTSC

Read Time-Stamp Counter

Opcode	Mnemonic	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX.

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 15 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3 for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDTSC instruction is always enabled.) The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the IA-32 Architecture in the Pentium processor.

Operation

```
if(CR4.TSD == 0 || CPL == 0 || CR0.PE == 0) EDX:EAX = TimeStampCounter;
else Exception(GP(0)); //CR4.TSD is 1 and CPL is 1, 2, or 3 and CR0.PE is 1
```

Flags affected

None.

CLFLUSH

Flush Cache Line

Opcode	Mnemonic	Description
0F AE /7	CLFLUSH m8	Flushes cache line containing m8.

Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see Section , CPUID-CPU Identification). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to insure that previous stores are included in the writeback.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

- L'istruzione `clflush` accetta come unico parametro una locazione di memoria, che identifica il blocco di memoria da rimuovere dalla cache.
- È bene utilizzare `clflush` subito dopo l'invocazione all'istruzione `mfence` (che approfondiremo più avanti): per ora basti pensare che `mfence` fa da **barriera** tra tutte le istruzioni che la precedono e tutte le istruzioni che la seguono. In tal modo, siamo sicuri che `clflush` mandi in write back e *rimuova davvero le informazioni che erano state portate in cache*. **Infatti, anche se committo l'istruzione, il dato non va in cache, ma nello store buffer. "Forzo" con *mfence* ASM, che porta il dato dallo store buffer in cache**. In caso contrario, anche se siamo sicuri che `clfush` venga committata dopo l'istruzione i di accesso alla memoria, `clflush` potrebbe intervenire prima che la memoria cambi realmente stato per effetto dell'istruzione i. (Quindi svuotiamo una cache già vuota, perchè dentro non è stato ancora messo nulla!)
  Questo è un problema di **memory consistency**, che analizzeremo successivamente.

ASM inline

È un modo per utilizzare la tecnologia assembly all'interno di un programma scritto in C. È molto utile se si vogliono incapsulare alcune istruzioni machine-dependent all'interno del programma. La sintassi è la seguente:

```
__asm__ [volatile][goto] (AssemblerTemplate
              [ : OutputOperands ]
              [ : InputOperands ]
              [ : Clobbers ]
              [ : GotoLabels ]);
```

- **AssemblerTemplate** = blocco di istruzioni assembly che si vuole inserire all'interno del flusso di esecuzione di una funzione C. Scritto in ISA, lavoro con memoria e registri, quindi niente comandi C, come *int c*.
- **OutputOperands** = blocco dell'ASM inline in cui è possibile specificare in quali variabili devono essere caricati i valori di determinati registri dopo l'esecuzione dell'AssemblerTemplate (post-movimento dati).
- **InputOperands** = blocco dell'ASM inline in cui è possibile specificare in quali registri devono essere caricati i valori di determinate variabili prima dell'esecuzione dell'AssemblerTemplate (pre-movimento dati). Entrambi servono per collegare il codice C all'assembly.
- **Clobbers** = registri di CPU che si vogliono memorizzare sullo stack prima dell'esecuzione dell'AssemblerTemplate e che si vogliono **ripristinare** dopo l'esecuzione dell'AssemblerTemplate. Mi basta specificare quali, senza preoccuparmi di altro.
- **GotoLabels** = istruzioni di jump presenti all'interno dell'AssemblerTemplate e che hanno come destinazione altri punti del programma (fuori dall'AssemblerTemplate). Questo blocco dell'ASM va specificato sempre assieme al prefisso `goto` (che viene posto subito prima delle parentesi tonde). Sarebbero i costrutti del tipo `fun_x: ...` Che troviamo in assembly.
- **Volatile** = prefisso opzionale che indica che il compilatore non dovrà attuare alcuna *ottimizzazione* sul codice assembly specificato nell'AssemblerTemplate (quindi non deve esserci il re-ordering delle istruzioni e così via).

Direttive di compilazione C per gli operandi

Il simbolo `=` all'interno dell'ASM inline significa che l'operando deve essere utilizzato come output. Se invece un operando non è associato al simbolo `=`, allora verrà utilizzato come input.

Possibili operandi per l'ASM

- **r** = registro generico che verrà scelto dal compilatore. Ha un indice associato, quindi è "recuperabile".
- **m** = locazione di memoria generica che verrà scelta dal compilatore.
- **i/I** = operando a 64/32 bit immediato.
- **a** = eax (o una sua variante come rax, ax, al dipendentemente dalla dimensione degli operandi).
- **b** = ebx (o una sua variante come rbx, bx, bl dipendentemente dalla dimensione degli operandi).
- **c** = ecx (o una sua variante come rcx, cx, cl dipendentemente dalla dimensione degli operandi).
- **d** = edx (o una sua variante come rdx, dx, dl dipendentemente dalla dimensione degli operandi).
- **S** = esi.

- **D** = edi.
- **0-9** = indici degli operandi (e.g. se a è il primo operando a essere utilizzato all'interno dell'ASM inline, può essere identificato con l'indice 0).
- **q** = registro che può essere utilizzato per indirizzare un singolo byte (e.g. `eax` che incapsula `al`).

Esempio di utilizzo di ASM inline

Vediamo un esempio di funzione che può essere invocata all'interno di un loop e ha lo scopo di misurare il tempo impiegato per effettuare un accesso in memoria in caso di cache miss.

```
unsigned long probe (char* adrs) {
 volatile unsigned long cycles;
 asm ( "mfence \n" //barriera per qualunque accesso in memoria
       "lfence \n" //barriera per gli accessi in memoria di tipo load
       "rdtsc \n" //preleva il timer da un registro specifico e
                  //carica i 32 bit meno significativi in eax e
                  //i 32 bit più significativi in edx
       "lfence \n" "movl %%eax, %%esi \n" //carica i 32 bit meno
                                          //significativi del timer in esi
       "movl (%1), %%eax \n"  //effettua l'accesso in memoria
                              //(in particolare alla variabile adrs,
                              // salvata in %1=ecx)
       "lfence \n"
       "rdstc \n"  //preleva il nuovo timer
       "subl %%esi, %%eax \n" //sottrae i 32 bit meno significativi
                              //dei due timer, che corrisponde al
                              //tempo impiegato per l'accesso in memoria
       "clflush 0 (%1) \n" //effettua il flush di adrs dalla cache
       : "=a" (cycles)  //dopo l'esecuzione dell'ASM inline, il contenuto
                        //di eax (registro 0) andrà nella variabile cycles
       : "c" (adrs)  //prima dell'esecuzione dell'ASM inline, il contenuto
                     //di adrs va nel registro ecx (registro 1)
       : "%esi", "%edx" ); //esi, edx sono i clobbers
      return cycles;
}
```

Eliminando la sola istruzione `clflush`, è possibile misurare il tempo impiegato per effettuare gli accessi in memoria in caso di cache hit.

NB_1 : gli attacchi Meltdown e Spectre utilizzano proprio questo costrutto qui.

NB_2: esistono delle API di C che implementano l'invocazione di rdtsc e clflush, e sono rispettivamente `__rdtscp()` e `_mm_clflush()`.

NB_3 : i registri `eax`, `ecx` non sono stati inseriti tra i clobbers perché sono **caller save**: sono registri che, a ogni chiamata di funzione, vengono salvati dalla funzione chiamante, cosicché la funzione chiamata possa utilizzarli a piacimento.

Altri dettagli su Flush + Reload È possibile rendere l'istruzione `rdtsc` *privilegiata*, ovvero si può fare in modo che venga invocata esclusivamente a livello kernel. Tuttavia, l'attacco **Flush+Reload**, per andare a buon fine, non ha la stretta necessità di sfruttare `rdtsc` per tenere traccia del trascorrere del tempo: esiste un modo altresì efficace per emulare un timer a grana sufficientemente fine. In pratica è sufficiente creare un nuovo thread che dovrà eseguire la seguente funzione:

```
void *time_keeper (void *dummy) {
  while(1){
            timer = (timer+1); //timer è una variabile globale
          }
}
```

NB: una variabile volatile è toccabile da più entità, non è nel registro del singolo thread. Evitiamo quindi ottimizzazioni, prendo e scrivo in memoria.

Un'altra cosa che vale la pena osservare è che questo tipo di attacco ha una probabilità di gran lunga minore di avere successo nel caso in cui si abbiano le *cache non inclusive*. Supponiamo che il thread T_A della vittima giri sul CPU-core$_A$ e il thread T_B dell'attaccante giri sul CPU-core$_B$. Se le cache sono *non inclusive*, qualunque blocco di memoria acceduto da parte di T_A viene caricato sulla cache di livello 1 propria di CPU-core$_A$ ma magari non sulle cache di livelli inferiori. A tal punto, T_B non può vedere in alcun modo sulla cache il valore utilizzato dal thread T_A: al livello L1 non è presente perché ciascun CPU-core ha la sua copia privata della cache di livello L1; ai livelli sottostanti, invece, *il dato non è proprio presente*. Se lavoro su una cache in uso *esclusivo*, cioè ho ad esempio due core sulla stessa cpu che condividono risorse, le tempistiche sono più veloci in quanto non mi sposto tra stati diversi.

Memory Consistency

Con riferimento al *memory wall*, se imponessimo che tutte le istruzioni che prevedono un'interazione con la memoria effettuino l'accesso vero e proprio alla memoria immediatamente, avremmo un degrado delle prestazioni, dovuto al fatto che l'*accesso alla memoria richiede molti cicli di clock*;
Per esempio, se abbiamo un'istruzione A che scrive un dato sulla memoria e un'istruzione B che legge il medesimo dato dalla memoria, B deve attendere che il dato sia disponibile in cache o in RAM prima di essere finalizzata, allora le ottimizzazioni effettuate sulla pipeline vanno vanificate. Di conseguenza, è necessario un meccanismo che disaccoppi le visioni di:

- Come (e in quale ordine) le istruzioni di accesso alla memoria vengono viste dal **processore che le esegue** (**program order**)
- Come (e in quale ordine) le istruzioni di accesso alla memoria vengono viste dagli **altri processori** / program flow mediante l'utilizzo delle risorse condivise nell'architettura di memoria, come la cache o la RAM (**visibility order**).

Tale meccanismo prevede l'utilizzo degli **store buffer**, che sono delle piccole aree di **memoria private per una CPU-core**, che mantengono temporaneamente i dati prodotti dalle istruzioni finalizzate ma ancora non riportate in cache o in RAM. In particolare, l'ordine con cui saranno viste le operazioni in memoria da parte degli altri program flow non corrisponde necessariamente con l'ordine in cui le istruzioni sono uscite dalla pipeline, e tale ordine dipende dal modello di consistenza della memoria utilizzato. *Non stiamo parlando di Cache coherency, perchè qui ancora non sappiamo se la scrittura in cache sia stata effettuata.*

Consistenza sequenziale

Definizione di Lamport, 1979

Un sistema multiprocessore è sequenzialmente consistente se il risultato di una qualunque esecuzione è lo stesso rispetto a se le operazioni di tutti i processori fossero eseguite in qualche ordine sequenziale e le operazioni di ciascun processore preso singolarmente apparissero in tale sequenza esattamente nell'ordine specificato dal suo programma.

In altre parole, per avere consistenza sequenziale, *non è possibile avere un'inversione delle istruzioni di uno specifico program order all'interno della sequenza globale delle operazioni.*

Esempio Supponiamo che CPU-core$_1$ debba eseguire le operazioni a_1, b_1 (dove a_1 viene prima di b_1 nel *program order*), e supponiamo che CPU-core$_2$ debba eseguire le operazioni a_2, b_2 (dove a_2 viene prima di b_2 nel program order). Allora, una sequenza che rispetta la consistenza sequenziale è data da:
a_1, a_2, b_1, b_2
Invece, una sequenza che non rispetta la consistenza sequenziale è data da: b_1, a_2, b_2, a_1.
Infatti, tale sequenza vede b_1 prima di a_1, il che rappresenta un'inversione rispetto all'ordine di programma proprio di CPU-core$_1$.

Nel mondo reale, la stragrande maggioranza delle macchine hanno dei chipset che non sono realizzati per soddisfare la consistenza sequenziale. Ma come mai questa scelta se la consistenza sequenziale è fondamentale per la correttezza di molte applicazioni concorrenti? Per *motivi di scalabilità* dell'architettura di memoria e di *costi* legati alla sequential consistency; tra l'altro, la consistenza sequenziale impone un ordinamento fisso per tutte le operazioni, anche per quelle completamente scorrelate tra loro. Algoritmi come *Bakery* o *Dekker* non vanno bene per le architetture odierne perchè *non lavorano in questo modo.* Come si comportano quindi i computer moderni?

Total Store Order TSO

È il modello di consistenza preferito nelle architetture reali, come x86 e SPARC. È basato sull'idea per cui *effettuare lo store dei dati non è equivalente al riportare effettivamente tali dati n**ell'architettura di memoria***. Quest'ultima operazione viene tipicamente **ritardata** rispetto al commit dell'istruzione di store; ad esempio, si potrebbe dover attendere che la linea di cache su cui si dovrà scrivere il nuovo valore passi allo stato exclusive.

È proprio questo il modello di consistenza che prevede lo **store buffer**. In particolare, nel momento in cui un'istruzione di `store` viene committata, il contenuto da scrivere poi in memoria viene nel frattempo inserito all'interno dello *store buffer*, che risulta essere un componente intermedio tra il CPU-core e l'architettura di memoria (cache + RAM). Ricordiamo che, dopo aver committato un'istruzione, questa *non passa subito da Store Buffer a Cache*, bensì passa un certo tempo. Solo dopo che va in cache, il risultato di tale istruzione risulta *visibile*.

Non sarebbe meglio andare direttamente in cache, in modo che i dati prodotti siano subito disponibili?

Il problema di questa idea risiede nell'automa **Mesi**. Come sappiamo, la linea di cache su cui scriviamo deve essere esclusiva, e questo richiede step intermedi come l'uso del *broadcat medium*, la richiesta di esclusività etc... Solo quando ho effettivamente queste condizioni passo a copiare in cache.
Capiamo subito che operare sulla cache è un'operazione abbastanza delicata. Lo Store Buffer, invece, è a uso dedicato del flusso in corso, non ho conflitti con altri thread. L'unica limitazione è che altri non vedono ciò che produco. Come vedremo dopo, gli Store Buffer, in x86 hanno una gestione FIFO, e possiamo usare alcune funzioni per forzare la scrittura in memoria.

Il TSO è un modello di consistenza che offre anche dei grossi vantaggi:

- Se nel program order di un CPU-core si hanno due istruzioni A, B, di cui A effettua una store di un dato X e B va a rileggere quel dato X, allora B ha la possibilità di recuperare il dato aggiornato andando semplicemente a leggere nello store buffer. In tal modo si risparmiano diversi cicli di clock per finalizzare le istruzioni perché chiaramente un **accesso allo store buffer è molto più veloce di un accesso alla cache.**
- È possibile effettuare il packaging delle informazioni da riportare nell'architettura di memoria. Se si hanno molteplici operazioni di scrittura che insistono sul medesimo blocco di memoria, si hanno altrettanti accessi allo store buffer ma poi lo store buffer dovrà *interagire con l'architettura di cache solo una volta.* Questo chiaramente riduce i costi di accesso alla memoria.

Tuttavia, il TSO non offre consistenza sequenziale perché, se abbiamo due istruzioni A, B in cui A viene prima di B nel program order, è possibile riportare nell'architettura di memoria prima gli effetti di B e poi gli effetti di A. In particolare, la presenza dello store buffer causa il **load-store bypass**, che è un fenomeno che si verifica quando si hanno due istruzioni A, B di cui A è una `store` di un dato X eseguita da un certo thread T_A, mentre B è una `load` del medesimo dato X eseguita da un altro thread T_B in un'istante successivo rispetto ad A. Di fatto, in tal caso, quando l'istruzione A viene finalizzata, il dato da essa prodotto potrebbe trovarsi soltanto nello store buffer. Di conseguenza, se il thread T_B gira su un CPU-core *diverso* rispetto a T_A, nel momento in cui deve andare ad accedere al dato X, *può farlo solo mediante la cache / RAM, per cui esegue la* `load` *di una versione vecchia di* X.

L'effetto finale è che l'istruzione B viene resa visibile prima dell'istruzione A.

Un'altra considerazione da fare è che *non necessariamente lo store buffer segua la disciplina FIFO.* (in *x86* si, in altre architetture no). Nel caso in cui non lo si segue, si ha lo **store-store bypass**, secondo cui due istruzioni di `store` successive possono essere invertite nel visibility order.

Memory Fencing

Il programmatore ovviamente deve avere la possibilità di prevenire i fenomeni di `load-store` bypass e `store-store` bypass per rendere corretto il software che sta sviluppando. Corrono così in aiuto tre categorie di istruzioni di **memory fencing** (= sincronizzazione delle attività che vengono eseguite sulla memoria), che sono:

- **SFENCE** (Store Fence): serializza le operazioni di *store* verso la memoria; tutte le store *precedenti* a lei vengono riversate in memoria prima dell'esecuzione di qualunque altra istruzione di store successiva a lei.
- **LFENCE** (Load Fence): serializza le operazioni di *load* dalla memoria; tutte le load precedenti a lei vengono completate prima dell'esecuzione di qualunque altra istruzione di load successiva a lei.
- **MFENCE** (Memory Fence): serializza tutte le operazioni in memoria; è un tipo di istruzione che unisce le capability di *sfence* e di *lfence*.

NON devo abusare di queste istruzioni, ma usarle con criterio per contesti delicati, altrimenti è come se stessi flushando la pipeline in ogni operazione.

Queste istruzioni sono garantite essere ordinate rispetto a qualsiasi altra istruzione serializzante (serializing instruction), come `cpuid` (che, se ricordiamo, effettua, tra le varie cose, lo squash della pipeline).

Inoltre, esistono delle istruzioni che, se precedute dal prefisso **lock**, diventano istruzioni serializzanti. Con *lock* si ha un approccio del tipo *blocco, lavoro, rilascio*. Quando lavoro è linearizzabile, grazie al lock vedo l'effetto delle altre attività. Queste sono: `add`, `adc`, `and`, `btc`, `btr`, `bts`, **`cmpxchg`**, `dec`, `inc`, `neg`, `not`, `or`, `sbb`, `sub`, `xor`, `xand`.

Esempio cmpxchg Significa *compare then exchange*, ha due operandi (o_1, o_2, dove o_1 può essere una locazione di memoria) e compara il contenuto di o_1 col registro `rax`/ `eax` / `ax` / `al`.

- Se i due valori sono **uguali**, allora il contenuto di o_2 viene riversato in o_1;
- Altrimenti, il contenuto di o_2 viene copiato in `rax` / `eax` / `ax` / `al`.

Si tratta di un'istruzione che può effettuare un primo accesso in memoria, una comparazione e poi un secondo accesso in memoria, per cui **non è atomica**. Se la si vuole eseguire in modo atomico, si utilizza appunto il prefisso **lock**, quindi ho una linea di cache esclusiva per me. Eseguire `cmpxchg` in modo **atomico** vuol dire riversare gli aggiornamenti in memoria già al completamento dell'istruzione stessa, per cui *non ci si può appoggiare allo store buffer*. Di conseguenza, è richiesto che, prima della `lock cmpxchg`, il contenuto dello *store buffer* venga riportato all'interno dell'architettura di memoria. È tale caratteristica che rende l'istruzione *serializzante* (ma, a differenza di `cpuid`, non effettua lo squash della pipeline).

La `lock cmpxchg` viene utilizzata anche per implementare i *lock* (e in particolare gli *spinlock*) per gli accessi in sezione critica. Per fare un esempio, di seguito è riportata una funzione C contenente un ASM inline, che implementa un trylock mediante cmpxchg
NB: trylock = se non riesco a ottenere il lock, non rimango in attesa e lascio perdere).

```
int try_lock (void *uadr) { //uadr = indirizzo dove si trova il lock
     unsigned long r = 0;
     asm volatile (
          "xor %%rax, %%rax\n" //rax = 0, sto azzerando il contenuto.
          "mov $1, %%rbx\n" //rbx = 1
          "lock cmpxchg %%rbx, ($1)\n" //if (uadr == 0) uadr = 1
          "sete (%0)\n" //if ("cmpxchg had success") r = 1
          : : "r" (&r), "r" (uadr) //prima dell'esecuzione dell'ASM inline,
                                   //due registri qualsiasi vengono popolati con &r, uadr
          : "%rax", "%rbx" //rax, rbx sono i clobbers
       );
     return (r) ? 1 : 0; //if ("cmpxchg had success") return 1; else return 0 }
}
```

Le istruzioni come la `cmpxchg` che eseguono la `load` di una locazione L di memoria in un registro, modificano il valore di tale registro ed eseguono la store del nuovo contenuto del registro all'interno della locazione L di memoria sono le cosiddette istruzioni **Read-Modify-Write** (**RMW**).
Ci permettono di rendere le operazioni globalmente visibili.

Anche qui esistono delle API di C che implementano l'invocazione di sfence, lfence, mfence e cmpxchg, e sono rispettivamente:

`_mm_sfence()`,`_mm_lfence()`, `_mm_mfence()` e `sync_bool_compare_and_swap()`

In ogni caso, utilizzare le istruzioni **RMW** per implementare i lock non è una soluzione particolarmente scalabile per realizzare degli schemi di coordinazione. Infatti, con lo spinlock, si ha un thread in sezione critica e tanti thread in *busy waiting*, per cui se disgraziatamente il thread in sezione critica viene *deschedulato* (ciò avviene comunemente se giro su una VM), si ha non solo un ritardo ma anche uno spreco di risorse e di energia da parte di chi è in *busy waiting*. Con lock, la linea di cache è a mio uso esclusivo.
Per questo motivo, si preferisce proporre degli algoritmi che prevedono un utilizzo alternativo delle istruzioni RMW e, a tal proposito, si hanno due possibilità principali che **non richiedono Lock**:

- **Non-blocking coordination** (lock-free / wait-free synchronization). Nel caso *lock-free*, posso ottenere degli abort, e quindi ritento (alg. Linearizzabile). Nel caso *wait-free*, non ho mai abort, ed è consistente.

- **Read Copy Update** (**RCU**). Include *Read intensive + aggiornamento*. Tecnica che *non è bloccante solo per chi legge* la struttura dati, mentre per chi aggiorna deve serializzare. Qui stiamo totalmente cambiando approccio, è un diverso modo di fare.

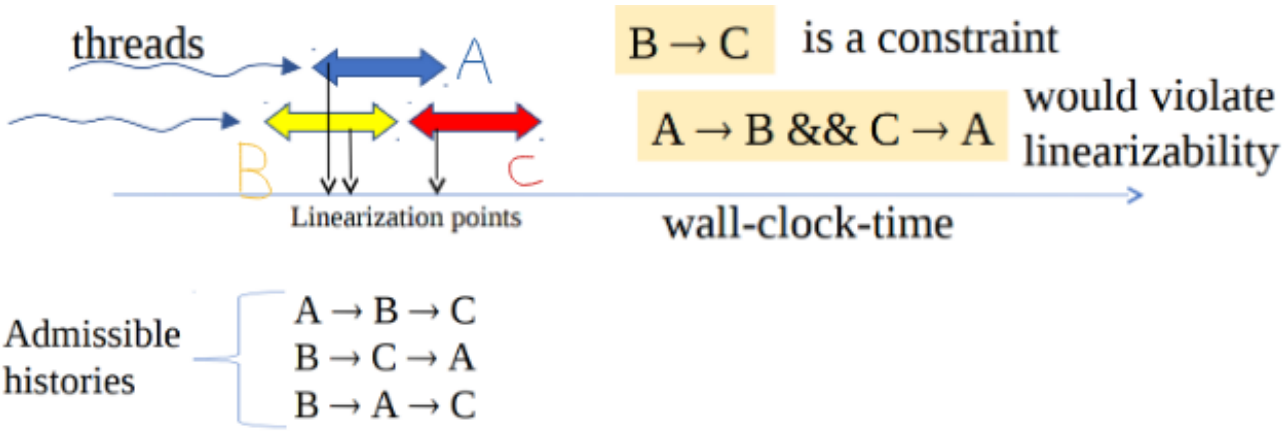
Cosa usare è influenzato dal "con cosa sto lavorando"!

Linearizzabilità

Supponiamo di avere una struttura dati S e delle funzioni che accedono a S; diciamo che un'esecuzione concorrente di tali funzioni è corretta se è **linearizzabile**, ovvero *se è come se le operazioni fossero eseguite in modo sequenziale* (dove quella successiva viene eseguita solo dopo il completamento della precedente). Questo è vero se:

- Gli accessi concorrenti a S, nonostante possano durare diversi cicli di clock, possono essere visti come se i loro effetti si materializzassero in un unico punto del tempo.
- Tutte le operazioni che si sovrappongono nel tempo possono essere ordinate in base al loro istante di materializzazione selezionato.

Esempio Supponiamo di avere tre funzioni A, B, C che accedono a una stessa struttura dati condivisa, e supponiamo di avere due thread T_1, T_2, di cui T_1 invoca la funzione A mentre T_2 invoca la funzione B e successivamente la funzione C.
Allora vale ciò che è riportato nella figura seguente:



Siamo solamente sicuri che B venga prima di C, non possiamo dire altro.
Tuttavia, lo studio di algoritmi concorrenti che utilizzano la materializzazione istantanea delle operazioni non è così banale. Ad esempio, in presenza di un salto condizionale, una determinata operazione può essere materializzata in istanti differenti a seconda se il salto viene preso oppure no.

Osservazione: Lavorare con istruzioni *atomiche* vuol dire *bloccare una linea di cache*, quindi *non ho concorrenza su un qualcosa che è solo mio*, al massimo posso averla se opero su linee diverse.
Prendiamo l'esempio sopra:
Siamo sicuri che il thread T_2, quando esegue l'operazione C, riesca a vedere tutto ciò che è stato fatto in B? (sopra è quasi stato dato per scontato!).
In un caso di *Sequential Consistency*, sì (anche se noi non operiamo in questo contesto), altrimenti non è scontato. Ad esempio, il thread potrebbe essere stato spostato su una CPU diversa, e prima di andarci lo Store Buffer nella vecchia CPU non è stato flushato. Quindi nella nuova CPU non posso recuperare il "vecchio Store Buffer".

Linearizzabilità vs operazioni RMW

Le operazioni RMW (Read-Modify-Write), nonostante implementino accessi in memoria non banali, appaiono in modo atomico sull'intera architettura hardware. Di conseguenza, possono essere sfruttate per definire dei punti di linearizzazione delle operazioni, in modo tale da ordinare le operazioni in una storia linearizzabile. Inoltre, le operazioni RMW possono fallire; di conseguenza il loro esito, come per i salti condizionali, può influenzare l'esecuzione o la non-esecuzione di una particolare istruzione e l'istante in cui essa viene eventualmente materializzata.

Sappiamo che con le operazioni RMW è possibile implementare un meccanismo di locking. I lock basati su RMW incentivano ancor più la linearizzazione, poiché portano le operazioni a essere eseguite in modo sequenziale.

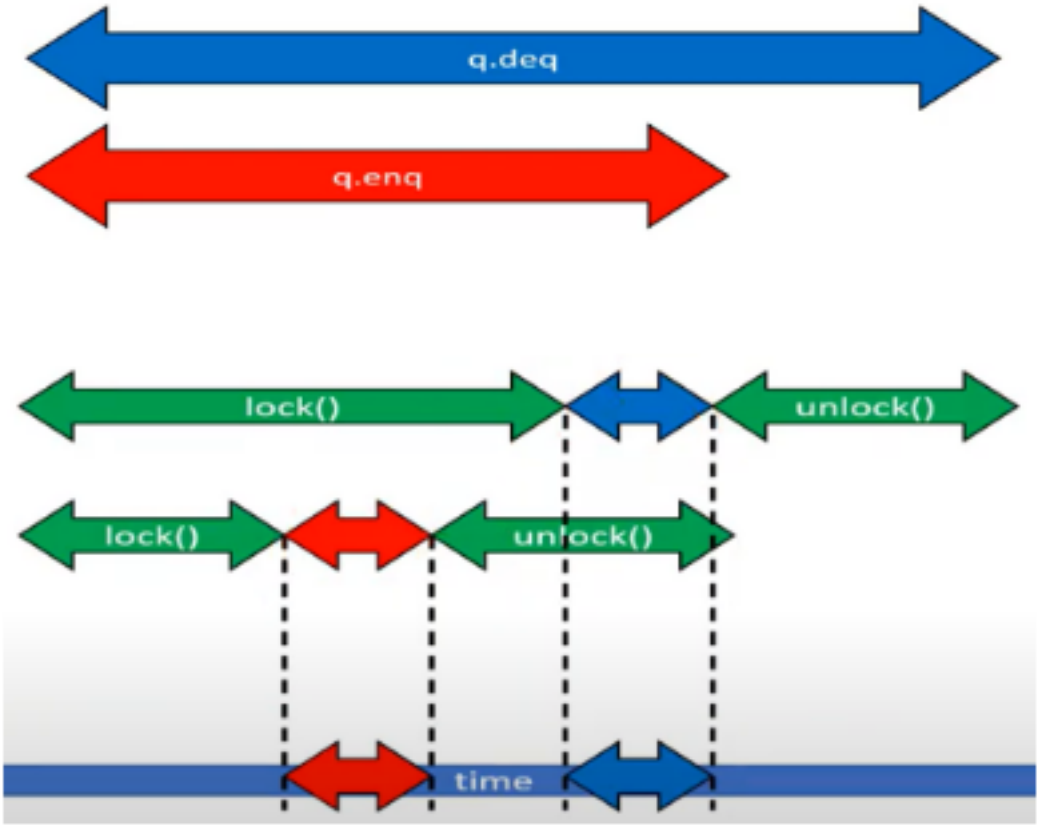


Figure 1: 43.png

Ma, come dicevamo in precedenza, preferiamo delle tecniche alternative all'utilizzo dei lock per sincronizzare i thread mediante operazioni RMW. Analizziamole ora. Con *lock* si ha un approccio del tipo *blocco, lavoro, rilascio* come è già stato detto. Utile per operazioni in tempo reale su strutture condivise, ma poco scalabile. Poi, con *fence, le rendo effettivamente visibili.* Vediamo delle alternative:

Non-blocking coordination

È un tipo di sincronizzazione **non bloccante**, che prevede due possibili approcci:

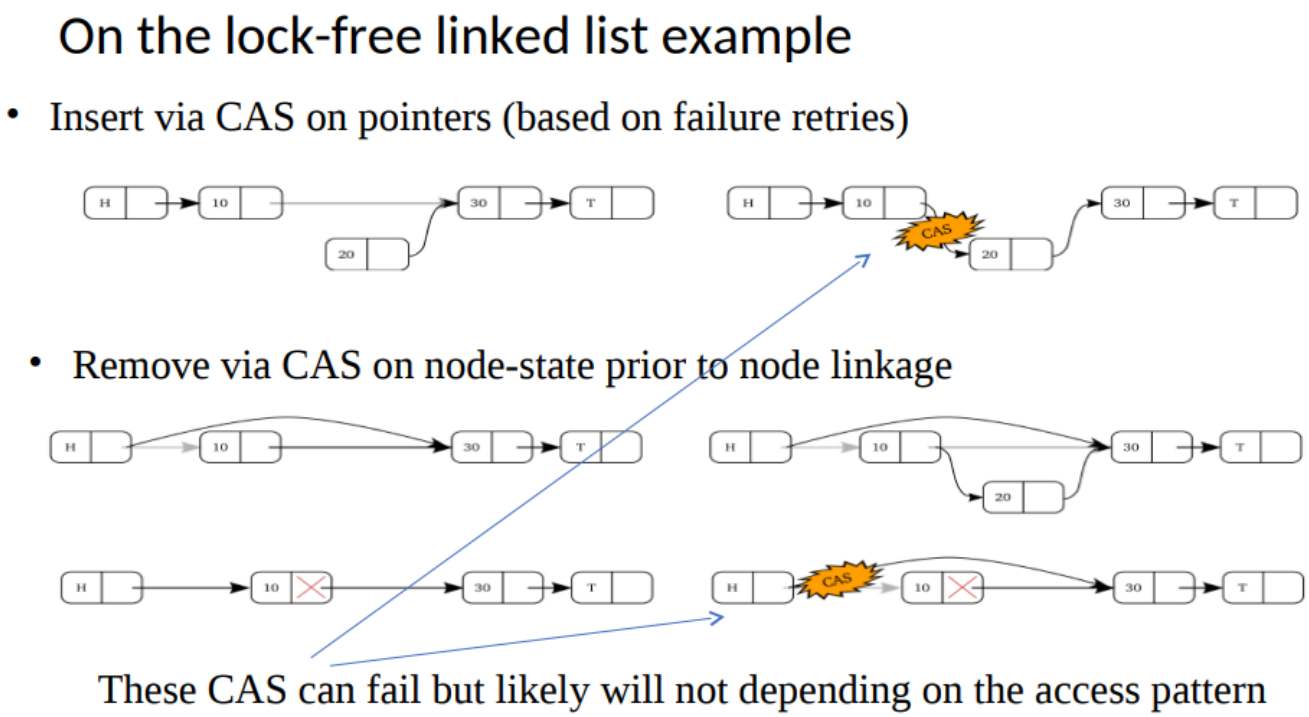
1) **Lock-freedom:** almeno un'istanza di una chiamata a funzione termina con successo in tempo finito AND tutte le istanze di chiamata a funzione terminano in tempo finito (o con successo o no). *Se fallisco, posso ricominciare da zero.*
2) **Wait-freedom** tutte le istanze di una chiamata a funzione terminano con successo in tempo finito. *Tutto quello che faccio va "sempre bene".* Dipende però con che struttura lavoro.

Sincronizzazione lock-free

Prevede la seguente logica: se due operazioni ordinate sono incompatibili (ovvero portano a fallire una qualche operazione RMW), allora una di loro può essere accettata ma l'altra deve essere rifiutata ed eventualmente rieseguita con una nuova try. Perciò, sono algoritmi basati sulla logica **abort / retry**: se una qualche operazione non va a buon fine, viene semplicemente abortita e in caso si effettua un nuovo tentativo. Chiaramente, per essere un approccio efficiente, deve essere caratterizzato da un basso numero di abort, in modo tale da non sprecare troppo lavoro eseguito in CPU e nell'hardware in generale.

Un grosso vantaggio della sincronizzazione *lock-free* è che non dà problemi nel caso in cui un thread dovesse crashare. Infatti, qui il comportamento di ciascun thread è *indipendente* da quello che succede a tutti gli altri thread; nel caso in cui si utilizzi un lock, invece, il crash del thread che detiene correntemente il lock porta all'impossibilità per tutti gli altri thread di acquisire successivamente il lock. (Vedi descheduling di un thread su Macchina Virtuale).

Esempio In questo schema, supponiamo di avere in memoria condivisa un certo *val*. Quando lo leggo, e poi lo voglio usare per una `compare and swap`, il confronto lo faccio tra il valore in una certa locazione e quel *val*. Se sono uguali eseguo lo swap. Se qualcuno lo ha cambiato, ho un fallimento.



- **Insert**: supponendo di inserire il nodo 20 tra i nodi 10 e 30 all'interno della lista collegata, si procede nel seguente modo: si ispeziona la lista fino al punto di aggiunta, si imposta 30 come successore di 20 e poi, tramite un'operazione di `Compare And Swap` (**CAS**, che è un'istruzione atomica. Sarebbe come il `compare and exchange`: o aggiorno io, o aggiorna un altro. Vince chi lo fa prima, il secondo farebbe un confronto errato. Se fallisco, ripeto!), si imposta 20 come successore di 10.

  Con l'approccio **lock-free**, tale operazione non dà problemi; l'unico caso in cui si ha un fallimento (e quindi un abort) è quello in cui c'è un altro inserimento concorrente nello stesso punto della lista collegata. Ad esempio, se concorrentemente a 20, un altro thread tenta di inserire un nodo 21 tra i nodi 10 e 30, solo uno dei due inserimenti avrà successo, mentre l'altro fallirà; viceversa, se si hanno inserimenti concorrenti in posizioni differenti della lista, avranno tutti successo nonostante non ci siano lock / attese.

- **Remove**: supponiamo di voler rimuovere il nodo 10 dalla lista collegata. Non possiamo farlo semplicemente deallocandolo ed eseguendo una `Compare And Swap` sul puntatore al successore del nodo testa (come nella figura in alto a sinistra). Infatti, può succedere che concorrentemente venga tentata l'esecuzione di un inserimento di un nodo 20 proprio subito dopo il nodo 10; se l'operazione di rimozione nel frattempo dealloca il nodo 10, poi l'inserimento è impossibilitato a cambiare il puntatore al successore di 10 perché si ritrova a lavorare su un *path* non più valido (vedere figura in alto a destra). Quello che si fa, dunque, è marcare il nodo 10 come da eliminare, per poi deallocarlo effettivamente in un secondo momento (vedere figura in basso a sinistra). Di conseguenza, vengono utilizzati dei bit all'interno di ciascun nodo che indicano lo stato del nodo stesso; questo a discapito dei puntatori, che si ritroveranno quindi con meno bit a disposizione. Ne consegue che i puntatori non possono più essere utilizzati con la massima granularità. Dopo aver marcato il nodo come da eliminare, si può procedere con una `Compare And Swap` per correggere il puntatore del nodo precedente (come nella figura in basso a destra).

Attenzione

In tale contesto è molto pericoloso eliminare un determinato nodo per poi riutilizzarlo per inserirvi delle informazioni differenti.
In breve: se un thread stava lavorando sul nodo 10, viene deallocato, e nel mentre il nodo 10 viene eliminato per far posto ad altro, quel thread, quando ritorna, avrebbe un indirizzo a qualcosa che non fa più parte della lista, come ad esempio ar ee sensibili! Il thread esegue **traversing**, si muove solo sulla lista, non conosce altro!.
In lungo: Supponiamo che un thread A debba effettuare una lettura sul nodo n, e supponiamo che venga deschedulato

subito dopo aver recuperato l'indirizzo di memoria di quel nodo ma prima di leggere il valore contenuto; supponiamo anche che un thread B deallochi proprio il nodo n e riutilizzi l'indirizzo di memoria di n per scrivere una nuova informazione da inserire nella lista collegata (o in una qualsiasi struttura dati). Allora, il thread A, quando verrà rischedulato, leggerà la nuova informazione anche se non era previsto dal flusso di esecuzione. Questo può anche rappresentare un problema di sicurezza nel momento in cui nei nodi sono riportati dei dati sensibili oppure, ad esempio, dei dati relativi agli utenti che stanno effettuando l'accesso a un determinato sistema.

Sincronizzazione wait-free

Qui non abbiamo né una logica di abort né una logica di retry: tutti i thread svolgono sempre lavoro utile, indipendentemente da quello che stanno facendo altri thread in concorrenza. Chiaramente si tratta di un approccio molto più "challenging" del precedente. Come risolviamo il problema del non poter riusare quegli spazi di memoria nella linked list?

Un esempio di struttura dati wait-free è il **registro atomico (1,N) wait-free**, che prevede un *solo scrittore* e N lettori ed è un caso particolare del registro atomico (M,N) wait-free. È un registro arbitrariamente grande in cui le operazioni di scrittura e lettura devono essere effettuate in modo atomico e potrebbero richiedere un alto numero di cicli di clock per essere eseguite. In una situazione del genere non ci piace l'utilizzo dei lock, perché farebbe crollare vertiginosamente le prestazioni; piuttosto, si procede nel seguente modo:

- Si hanno molteplici istanze del registro atomico e un puntatore che referenzia l'istanza più aggiornata.
- Lo scrittore, quando deve aggiornare il registro, ne alloca una nuova istanza; nel momento in cui ha terminato la scrittura, aggiorna con una `Compare And Swap` il puntatore per farlo referenziare verso la nuova istanza.
- I lettori sfruttano il puntatore per accedere all'area di memoria dov'è allocata l'istanza correntemente *più aggiornata* del registro atomico. Una volta che la lettura è iniziata, il lettore è sicuro che non ci saranno mai interferenze con l'istanza da parte di scrittori (per cui l'operazione andrà certamente a buon fine).

Questa soluzione presenta però una difficoltà in particolare: non è banale stabilire quando è possibile effettuare la garbage collection delle varie istanze del registro atomico. Comunque sia, la letteratura stabilisce che servono almeno $N + 2$ buffer (istanze) per far funzionare correttamente il meccanismo:

- N buffer sono (al limite) per gli N lettori.
- 1 buffer è adibito per contenere l'ultimo eventuale valore che non è stato ancora acceduto da alcun lettore.
- L'ultimo buffer serve allo scrittore per inserire un eventuale nuovo valore.

Quindi sostanzialmente, ad ogni modifica creo un nuovo buffer, dico che quello è il più aggiornato, e chi lavora col vecchio lo usa finchè non ha terminato. Un limite da gestire è che non posso mantenere infiniti buffer, però qualcuno potrebbe puntarci ancora!

Scendendo in qualche ulteriore dettaglio dell'implementazione di tale meccanismo, si ha un'unica variabile di sincronizzazione composta da due campi, 32 per identifiare l'istanza e 32 per il contatore:

- nel primo è indicato qual è lo slot (tra gli N+2 totali) che è stato aggiornato più recentemente dallo scrittore.
- Nel secondo è riportato il numero di lettori che si sono attestati esattamente a quello slot; questo secondo campo viene aggiornato dai lettori mediante una chiamata a `fetch_and_add` **atomica**, un'istruzione che esegue la lettura, e l'incremento di una determinata variabile in modo atomico. Conto chi ha preso il riferimento sostanzialmente. Ovviamente le operazioni devono essere finite, sennò non posso contarle. Incremento solo se mi sposto tra aree di memoria, altrimenti non aumento se sto sempre sulla stessa.
- Il reader decrementa il counter se ha finito di leggere.
- Quando lo scrittore deve effettuare una scrittura, esegue una`atomic_exchange` sulla variabile di sincronizzazione, ovvero legge e mette da parte il contenuto attuale della variabile di sincronizzazione e riscrive tale variabile con:
- L'indirizzo di memoria del nuovo buffer (nel primo campo).
- Il valore 0 per indicare che inizialmente non ci sono lettori che hanno acceduto al nuovo buffer (nel secondo campo).

Chiaramente ciascun lettore dovrà essere in grado di visualizzare sempre la versione della variabile di sincronizzazione associata allo slot in cui l'operazione di lettura era iniziata. Nel momento in cui in uno slot non ci sono più letture pendenti (per cui *#letture iniziate = #letture terminate*), tale slot può essere sfruttato dallo scrittore per inserirvi dei

nuovi dati aggiornati; tra l'altro, con $N + 2$ slot totali, esiste sempre almeno uno slot che soddisfa questa proprietà, per cui lo scrittore non rimarrà mai bloccato per eseguire le scritture.

Esistono anche delle *ottimizzazioni* del protocollo: ad esempio, è possibile fare in modo che ciascun lettore esegua un'unica `fetch_and_add` sul contatore dei lettori per ogni diversa istanza di registro che va a leggere. In tal modo, si riduce il numero totale di operazioni atomiche eseguite e questo è un beneficio anche per la *cache coherency*: sappiamo che le istruzioni atomiche di tipo RMW portano un particolare CPU-core a prendere un blocco di cache nello stato exclusive, lasciando così gli altri CPU-core bloccati nel caso in cui vogliono accedere al medesimo blocco di cache.

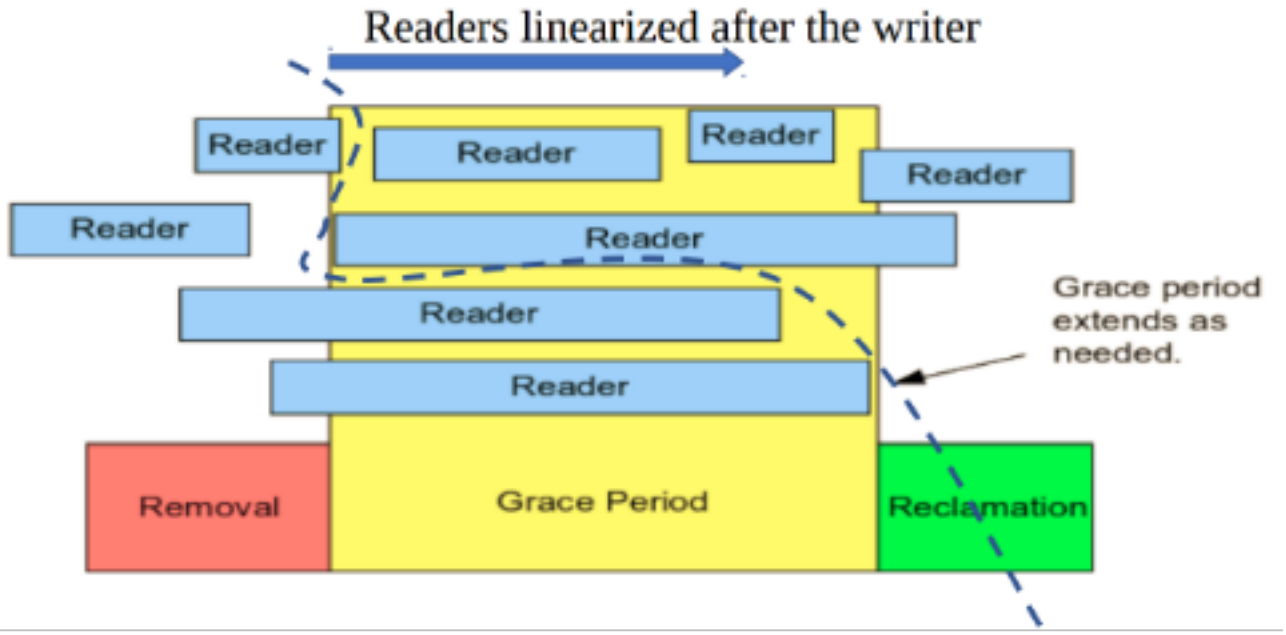
Read Copy Update RCU

È un tipo di sincronizzazione che fa da trade-off tra l'utilizzo dei lock e la lock-freedom. Infatti, non offre le stesse garanzie della lock-freedom (in cui tutte le istanze di chiamate a funzione terminano in tempo finito) ma, di contro, semplifica il meccanismo di garbage collection; questo rende RCU più scalabile. Qui possiamo ammettere, su una struttura dati concorrente, *un solo writer* e n di reader per volta:

- I reader, per eseguire le *letture*, non devono attendersi né a vicenda né col writer.
- Il writer, per eseguire una *scrittura*, deve attendere solo eventuali altri writer.

Si tratta dunque di un approccio vincente per le strutture dati **read-intensive**, che nella pratica sono tantissime; infatti, Linux implementa RCU per molte strutture dati usate a livello kernel. (Ad esempio, la lettura di una *hash table* per vedere il thread control block (*TCB*) di un thread!)

RCU prevede che un buffer o un nodo di una struttura dati non può essere deallocato finché non siamo sicuri che sia diventato inutilizzato. In particolare, tra l'istante in cui viene invocata la deallocazione (e il buffer/nodo viene marcato come "da deallocare") e l'istante in cui il buffer/nodo viene effettivamente deallocato, si ha il cosiddetto **grace period**.



Nella figura qui sopra si hanno tre *reader* (escludiamo quello che conclude prima di *grade period*) che eseguono una lettura concorrente alla removal e che quindi devono essere attesi prima della rimozione vera e propria (*reclamation*): il grace period dura fin tanto che tutti e tre questi reader non hanno concluso la loro lettura. Il Grace period è un periodo di grazia in cui, anche se sono pronto a cambiare indirizzo, lascio "attivo" il vecchio indirizzo per far concludere le letture. In particolare, è necessario attendere tutti e tre i reader indistintamente perché lo scrittore non sa qual è l'istante esatto in cui avviene la linearizzazione della removal, per cui deve assumere che tutte le letture concorrenti siano iniziate antecedentemente a tale istante. Non posso modificare *removal* finchè qualcuno la usa. Quindi, se reader concorrenti e su *cpu diverse*, non vedono che un "pezzo" è stato sganciato, allora devo aspettarli. Questo perchè chi linearizza prima di me, non potrebbe rientrare se togliessi la struttura.

Funzionamento

Il lettore

1) Segnala la sua presenza.
2) Legge la struttura dati.
3) Segnala che se ne sta andando.

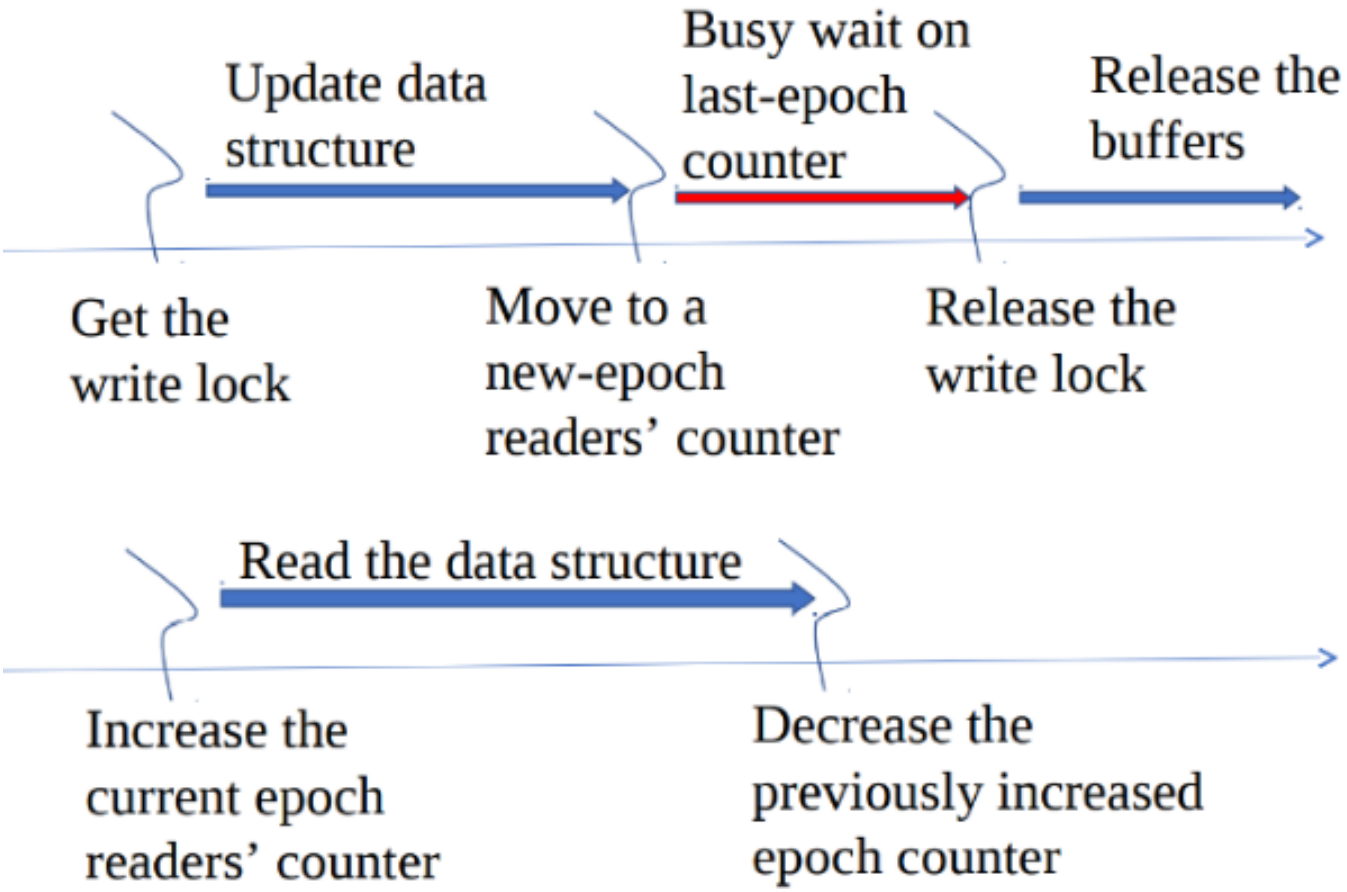
Lo scrittore

1) Acquisisce il lock di scrittura.
2) Aggiorna la struttura dati.
3) Attende che i lettori "*standing*" terminino le loro operazioni; notiamo che i lettori che operano sull'istanza della struttura dati già modificata sono dei *don't care reader*.
4) Dealloca la vecchia istanza della struttura dati.
5) Rilascia il lock di scrittura.

Non-preemptable RCU vs preemptable RCU

A questo punto è necessario risolvere il seguente problema: come fa lo scrittore a distinguere un lettore standing da un lettore che opera sull'istanza della struttura dati già modificata? Di seguito vengono proposte due possibili soluzioni.

- **Non-preemptable RCU**: qui è necessario che i reader disattivino la possibilità di essere interrotti ("prelazionati" dalla CPU, cioè thread su cpu non interrompibili, non rilasciano la cpu su richiesta) durante la loro esecuzione. In tal caso, lo scrittore, subito dopo aver aggiornato la struttura dati, invoca: `for_each_online_cpu(cpu)`, `run_on(cpu)`. In pratica, lo scrittore si fa un giro su tutte le CPU della macchina e ne richiede l'utilizzo. Appena riesce a essere schedulato sulla CPU_i, significa che sulla CPU_i non è più in esecuzione (o non lo è mai stato) un lettore che era standing al completamento dell'aggiornamento. Di conseguenza, quando lo scrittore verrà schedulato su tutte quante le CPU, sarà possibile concludere il grace period senza problemi. Il grosso svantaggio di questo approccio sta nell'impossibilità di interrompere l'esecuzione dei lettori: se a un certo punto dovesse subentrare un thread con priorità arbitrariamente elevata, dovrà in ogni caso aspettare che un qualche lettore termini prima di essere schedulato.
- **Preemptable RCU**: Qui potrei subire un *interrupt*, ma stando al "ragionamento" di prima, lasciare la CPU vuol dire aver finito. Si usa il concetto di **epoca**:
  Si ricorre all'utilizzo di un **presence-counter atomico** che va a indicare quanti lettori stanno correntemente insistendo su una determinata versione (**epoca**) della struttura dati; è atomico perché, come al solito, vengono acceduti in lettura e in scrittura con un'unica istruzione atomica (i.e. `fetch_and_add`). Nel momento in cui lo scrittore aggiorna la struttura dati, redireziona il puntatore al presence-counter verso una nuova istanza di presence-counter (*quello relativo alla nuova epoca*), in modo tale che i lettori successivi aggiornino quest'ultimo; d'altra parte, i lettori standing, quando completano le loro operazioni, **decrementano** il presence-counter della vecchia epoca (last-epoch), in modo tale che sia tutto consistente. Chiaramente, il grace period termina quando il last-epoch counter diviene uguale a zero. Nella pagina seguente è riportato uno schema riassuntivo sul funzionamento del preemptable RCU.



Vettorizzazione

È un meccanismo in cui le singole istruzioni hanno un vettore di sorgenti e un vettore di destinazioni; in altre parole, vengono coinvolti molteplici registri contemporaneamente. Di conseguenza, si ha un miglioramento delle performance. La vettorizzazione è una forma di **SIMD** (**Single Instruction Multiple Data**) alla base delle GPU; in alternativa al SIMD esistono:

- **MIMD** (**Multiple Instruction Multiple Data**): è la forma di calcolo realmente utilizzata nelle macchine reali; rispetto al SIMD prevede l'utilizzo di molteplici processori/core. Chip con hyperthread, ad esempio un hyperthread (MI) e speculazione (MD).
- **MISD** (**Multiple Instruction Single Data**): è una forma di calcolo più rara, anche se qualcuno sostiene che un processore speculativo sia MISD; di fatto, i processori speculativi introducono la possibilità di eseguire più istruzioni in parallelo su uno stesso dato, sfruttando anche molteplici istanze del medesimo registro logico.
  Nb: i renamed register non sono esposti in ISA.
- **SISD** (**Single Instruction Single Data**): è la forma di calcolo più banale, dove è previsto un unico CPU-core non speculativo. Sarebbe l'architettura di Von Neumann.

Per poter implementare la vettorizzazione, la macchina chiaramente deve fornire appositi registri, apposite istruzioni macchina e appositi meccanismi dell'hardware, come ad esempio:

- L'**hardware data gather**, per raccolta di dati dalla memoria verso un registro vettoriale.
- L'**hardware data scatter**, per il riversamento di dati dal registro vettoriale verso la memoria.

Tramite flag `-O` posso implementare calcoli vettoriali senza API, esistono anche `-O2` o `-O3`, più cresce più ottimizza, ma deve essere opportunamente gestito, cioè sezioni critiche vanno regolate con `mfence` per evitare problemi ad accessi in memoria. Alcune ottimizzazioni abilitabili sono il *vectorize* e *loop-unrolling*. Il *volatile* è invece una non-ottimizzazione, che però può essere utile per cose più delicate.

In x86, la vettorizzazione è detta **SSE** (**Streaming SIM Extension**), mentre in x86-64 è detta **SSE2**.

In SSE si hanno 8 registri vettoriali a 128 bit, che sono $XMM0$, $XMM1$,..., $XMM7$. Tali registri sono in grado di ospitare:

- ✓ 2 64-bit double precision floats
- ✓ 4 32-bit single precision floats
- ✓ 1 128-bit integer (unsigned)
- ✓ 2 64-bit integers (signed or unsigned)
- ✓ 4 32-bit integers (signed or unsigned)
- ✓ 8 16-bit integers (signed or unsigned)
- ✓ 16 8-bit integers (signed or unsigned)

In SSE2, invece, si hanno 16 registri vettoriali a 128 bit, che sono $YMM0$, $YMM1$,..., $YMM15$.

Le istruzioni di `mov` che coinvolgono i registri *XMM* e *YMM* (caricamento di dati nei registri vettorizzati / store dei dati dai registri vettorizzati in memoria) potrebbero richiedere che le informazioni siano allineate in memoria (e.g. agli 8 byte, ai 16 byte). L'utilizzo delle istruzioni che richiedono l'allineamento sui dati non allineati causa un **general protection error**.
Esistono più modi per allineare le informazioni in memoria:

- `__attribute__ ((aligned (16)))`
- `mmap()`, che alloca delle pagine di memoria allineate ai 4 KB.

Vettorizzazione esplicita ed implicita

- **Vettorizzazione esplicita**: il programmatore utilizza esplicitamente le istruzioni che coinvolgono i registri vettorizzati.
- **Vettorizzazione implicita**: il programma viene compilato col flag `-O` in modo tale che il compilatore `gcc` scandisca il codice sorgente e lo mappi, ove possibile, su istruzioni vettoriali.

Per quanto riguarda la vettorizzazione esplicita, in realtà in C sono offerti degli instrinsics (delle funzioni ad hoc) per sfruttare la vettorizzazione:

- Vectorized addition - 8/16/32/64-bit integers

MMX(64-bit)	`d = _mm_add_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_pi32(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi32(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_si64(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi64(a, b)`	`__m128i a, b;`	`__m128i d;`

- Vectorized addition - 32-bit floats

SSE(64-bit)	`d = _mm_add_ss(a, b)`	`__m128 a, b;`	`__m128 d;`
SSE(128-bit)	`d = _mm_add_ps(a, b)`	`__m128 a, b;`	`__m128 d;`

- Vectorized addition - 64-bit doubles

SSE2(64-bit)	`d = _mm_add_sd(a, b)`	`__m128d a, b;`	`__m128d d;`
SSE2(128-bit)	`d = _mm_add_pd(a, b)`	`__m128d a, b;`	`__m128d d;`

Kernel Programming Basics