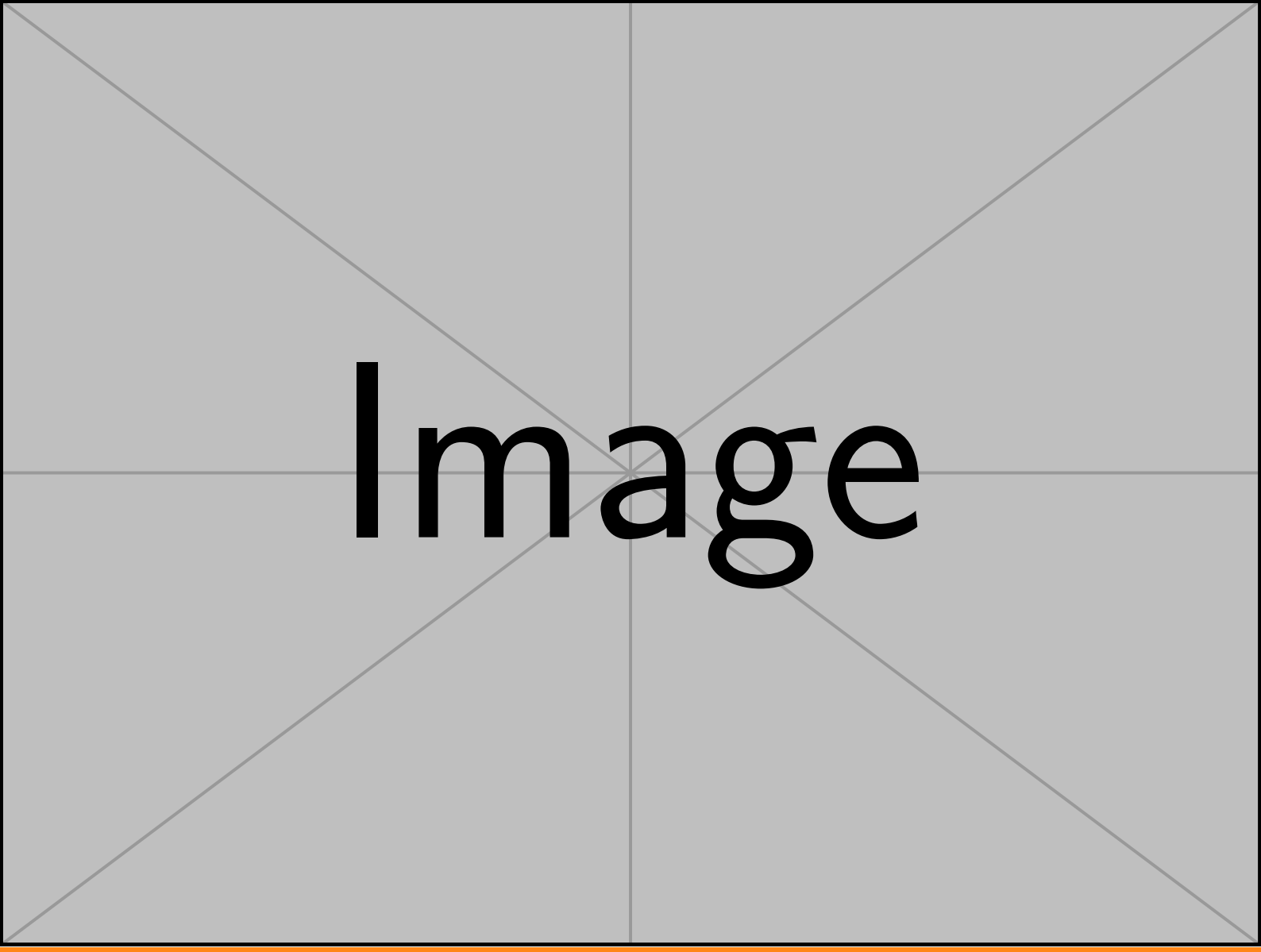

Sistemi Operativi Avanzati

E Sicurezza dei Sistemi

Autore: Andrea Efficace (https://github.com/Effibot)

Istituto: Università di Roma Tor Vergata

Data: 17 gennaio 2023

Indice

Capitolo 1 Hardware Insights

Il mondo dell'IT si è evoluto nel tempo verso linguaggi di programmazione più astratti ed espressivi, al fine di rendere più facile la vita al programmatore e di lasciare la gestione delle risorse hardware a moduli dedicati.

Di norma infatti un programmatore non si preoccupa di alcuni aspetti che sono alla base del funzionamento di un sistema come:

- Le decisioni prese dal compilatore per la gestione delle risorse.
- Le decisioni prese dal processore per un'esecuzione efficiente del **program flow**, come l'*hyper-threading* o l'esecuzione su più core.
- La disponibilità, o l'assenza, delle risorse hardware per l'esecuzione di un programma.

In questo modo si è persa tuttavia anche la capacità di configurare questo hardware in modo da ottimizzare le prestazioni del sistema, e di capire cosa succede quando si esegue un programma.

E' importante quindi per il sistemista riconsiderare la definizione di stato di un programma:

Definizione 1.1 (Program State)

Lo stato di un programma non è costituito soltanto da tutte le risorse software che lo compongono, ma anche da tutte le risorse hardware che lo eseguono accedendo alle risorse esposte dall'ISA (Instruction Set Architecture) del processore e non solo.

Un esempio importante di come l'hardware possa influenzare il comportamento di un programma rispetto al program flow espresso nel codice sorgente è dato dal **Bakery Algorithm** di Lamport, famoso per **risolvere** il problema dell'**accesso concorrente** a una risorsa condivisa:

Esempio 1.1 (Lamport's Bakery Algorithm)

```
# var: choosing = array[1,n] of boolean
# var: number = array[1,n] of integer
while true do {
    choosing[i] := true
    number[i] := 1 + max(number[1], ..., number[n])
    choosing[i] := false
    for j := 1 to n do{
        while choosing[j] do no-op
        while number[j] != 0 and (number[j], j) < (number[i], i) do no-op
    }
    # -- critical section -- #
    number[i] := 0
}
```

Osservazione 1.1

Nello schema precedente l'attesa è attiva: infatti avviene a livello utente tramite controlli su numeri e ID di processi senza mai chiamare il sistema operativo. ■

Infatti, quando un thread vuole entrare nella sezione critica viene alzato un **flag**, settand il valore di *choosing*[*i*] a *true*, e viene assegnato un numero di ordine aumentando di uno il valore di *number*[*i*] rispetto al valore massimo tra tutti i thread che stanno aspettando di entrare nella sezione critica. Si abbassa poi il flag e si cerca di accedere alla sezione critica tramite con due cicli consecutivi all'interno di uno di controllo:

1. Il primo ciclo controlla che nessun altro thread stia aspettando di entrare nella sezione critica, altrimenti il thread si blocca.
2. Il secondo ciclo controlla che possa effettivamente essere servito effettuando un check per controllare che tutti i thread precedenti siano stati serviti e che la coppia (*number*[*j*], *j*) sia minore di quella degli altri.

Supponendo che il programma sia eseguito su una macchina con supporto al multi-threading e che quindi il processore esegua in modo parallelo la stessa funzione espressa nell'algoritmo per più thread, è possibile che l'accesso alla sezione critica venga concesso a più thread contemporaneamente, violando il principio di esclusione mutua.

Implementando l'algoritmo, ad esempio in C, sfruttando un sistema di *logging* facendo consegnare ai thread il loro numero d'ordine al momento di essere serviti, si potrebbe vedere che ad un certo punto il numero depositato avrebbe avuto un incremento doppio rispetto al precedente o che qualche numero d'ordine salterebbe. Questo comportamento non è dovuto al codice sorgente né al processo di compilazione, ma è dovuto al fatto che **nessuna macchina** è **off-the-shelf sequentially consistent** (OSSC), ovvero che nessuna macchina esegue in modo sequenziale i thread che gli vengono consegnati ma a run-time il processore compie delle decisioni che possono influenzare il comportamento del programma e, pertanto, andrebbero considerate nel momento in cui si scrive un programma concorrente.

1.1 Architetture per la Parallelizzazione

Il modello hardware classico è quello di Von Neumann, che prevede:

- Astrazione di una singola CPU.
- Astrazione di una singola memoria.
- Astrazione di un singolo control-flow, fatto di istruzioni sequenziali (come una pipeline a 1 stage).
- Transizioni di stato nell'HW indipendenti poiché ce n'è solo una in esecuzione ad ogni istante di tempo.
- L'immagine della memoria è definita allo startup di ogni istruzione.

Questo modello non è fatto per l'esecuzione parallela, che richiede architetture più complesse e strategie aggiuntive (come lo **scheduling**) affinché il program-flow sia eseguito come

specificato nel sorgente. Difatti si può notare che il modello di Von Neumann non prevede né che il risultato di un'istruzione si propaghi nel tempo né che l'istruzione successiva possa dipendere da una delle precedenti.

L'approccio moderno di pensare le architetture di calcolo piuttosto che essere basato sul program-flow è basato appunto sul concetto di scheduling per fare qualcosa che sia equivalente al flusso delle istruzioni. Lo scheduling può essere suddiviso in due categorie:

Proposizione 1.1 (Hardware Scheduling)

A livello hardware, lo scheduling definisce:

- L'esecuzuone delle istruzioni di un singolo program-flow.
- L'esecuzione in parallelo (**speculativamente**) di diversi program-flow.
- La propagazione dei valori verso le unità di memoria.

Proposizione 1.2 (Software Scheduling)

A livello software, lo scheduling definisce:

- I **time-frames** per l'esecuzuone dei singoli thread sull'HW.
- Le modalità di gestione di tutte le **attività di sistema** (*task*), come la gestione dei time-frames per gli interrupt.
- Supporti software per la sincronizzazione (sia di task che thread).

Possiamo definire due concetti di parallellismo:

Definizione 1.2 (Istruction Level Parallelism)

Istruction Level Parallelism (ILP) è un concetto di parallelismo basato sul fatto che nella stessa finestra temporale in cui un'istruzione produce l'output, un'altra istruzione può essere eseguita in parallelo. In questo modo è possibile processare 2 o più istruzioni nello stesso ciclo di clock [1].

[1] Anche se una singola istruzione richiede più cicli viene garantito che sia completata prima dell'esecuzione successiva

Definizione 1.3 (Thread Level Parallelism)

Thread Level Parallelism (TLP) è un concetto di parallelismo basato sull'esecuzione multipla di diversi program-flow che portano avanti la logica del programma, che può essere visto come la combinazione di molteplici flussi concorrenti.

Osservazione 1.2

Un'architettura ILP non è per forza anche TLP, ma è sempre vero il viceversa.

La parallelizzazione permette di aumentare la velocità d'esecuzione di un programma e i suoi supporti aumentano la velocità del processore, spesso misurata in Ghz. Tuttavia, questa unità di misura in realtà non misura esclusivamente la velocità di esecuzione, ma anche la quantità di cicli di clock eseguiti e, analogamente, la quantità di istruzioni processate, le interazioni con i diversi componenti hardware ed eventuali asimmetrie e pattern d'accesso ai dati.

Osservazione 1.3 (Categorie di Programmi)

La velocità di un processore non è l'unica cosa che limita la velocità di esecuzione di un programma. Possiamo identificare tre categorie di essi:

- **CPU-Bound**: programmi che richiedono un elevato tempo di CPU per essere eseguiti.
- **I/O-Bound**: programmi che chiamano servizi bloccanti del kernel (come la lettura da disco) che richiedono un elevato tempo di I/O e usano la CPU in maniera intervallata.
- **Memory-Bound**: programmi che richiedono un elevato tempo di accesso alla memoria.

I sistemi di calcolo moderni sono evoluti molto dal punto di vista della velocità dei processori, ma meno dal punto di vista della velocità delle memorie. Questo gap ha portato alla necessità di sviluppare architetture hardware ILP che permettano un utilizzo costante della CPU anche quando questa è in attesa di dati dalla memoria.

1.1.1 Pipeline

Le pipeline rappresentano una tecnica basilare di fare ILP attraverso l'**overlapping** di più istruzioni. La tecnica è hardware-based ed unisce scheduling al parallelismo per costruire un modello detto a **data-flow**, ovvero basato sul fatto che:

Definizione 1.4 (Data-Flow Model)

Il **data-flow model** è un modello di calcolo basato sul fatto che la sorgente di un'istruzione dovrebbe essere letta basandosi sull'ultimo aggiornamento fatto durante la sequenza delle istruzioni.

In particolare, abbiamo parallellismo in quanto **non c'è** una netta **separazione temporale** tra le finestre d'esecuzione delle istruzioni e abbiamo scheduling in quanto **non necessariamente** la sequenza di istruzioni processate rispetta quello che è scritto nel programma (in questo caso si intendo proprio l'eseguibile).

L'unico vincolo che una pipeline deve avere è che deve essere rispettato il vincolo di **causalità**: ogni istruzione deve poter essere eseguita solo dopo che tutte le istruzioni da cui dipende sono state eseguite. Ad esempio, se un'istruzione A dipende da un'istruzione B e B dipende da un'istruzione C, allora A può essere eseguita solo dopo che C è stata eseguita e così via.

Osservazione 1.4 (Fasi di una Pipeline)

Le fasi di una pipeline sono 5:

1. **Istruction Fetch**: l'istruzione viene caricata dalla memoria.
2. **Istruction Decode**: l'istruzione viene decodificata e viene generato il controllo per l'esecuzione.
3. **Load Operands**: i dati necessari per l'esecuzione dell'istruzione vengono caricati dalla memoria.
4. **Execute**: l'istruzione viene eseguita.
5. **Write Back**: il risultato dell'istruzione viene scritto nella memoria.

Una pipeline single stage è una pipeline che ha una sola fase di esecuzione. Una pipeline multi-stage è una pipeline che ha più fasi di esecuzione.

Se una pipeline ha più fasi di esecuzione allora è possibie fare in modo che le istruzioni siano eseguite in parallelo, garantendo l'ILP. Ad esempio: se un'istruzione uscita dalla fase di fetch entra in quella di decode, la prossima istruzione può occupare i componenti hardware per il fetch. In questo modo, la pipeline è in grado di eseguire due istruzioni in parallelo, anche se la singola fase d'esecuzione richiede più cicli di clock rispetto alla singola istruzione.

Teorema 1.1 (Pipeline Speedup Analysis)

Supponiamo di voler fornire N risultati, uno per ogni istruzione, e di avere L stage di processamento con un ciclo di clock di lunghezza T. Senza una pipeline, il tempo d'esecuzione espresso come ritardo ingresso uscita è pari a:

$$d = N \cdot L \cdot T \tag{1.1}$$

Con una pipeline, il tempo d'esecuzione è pari a:

$$d = (N + L) \cdot T \tag{1.2}$$

Dove lo speedup è dat da un fattore:

$$\frac{N \cdot L}{N + L} \tag{1.3}$$

Che per N grande è circa pari a L.

Osservazione 1.5

E' vero che incrementando il numero di stage in una pipeline si ottiene una velocità maggiore, ma questo non è automatico: se si aumenta il numero di stage, si aumenta anche il numero di cicli di clock necessari per eseguire un'istruzione e propagarne il risultato.
Infatti, i processori moderni hanno un numero di stage dell'ordine delle decine.

C'è però un problema legato al pipelining: se deve essere eseguita un'istruzione di salto, il processore è in grado di capire dove stiamo saltando nel program-flow **solo alla finalizzazione** dell'istruzione (ovvero quando in WB viene riportata l'informazione indietro). Tuttavia, durante il processamento di questo salto su uno stage della pipeline nel processore stanno scorrendo altre istruzioni che vengono eseguite in parallelo ma inutilmente.

Osservazione 1.6 (Problemi della pipeline)

- **Control Dependency:** se un'istruzione di salto è in esecuzione, tutte le istruzioni successive non possono essere eseguite.
- **Data Dependency:** un'istruzione richiede un dato che non è ancora stato calcolato da un'istruzione precedente.

1.2 Ottimizzazione della pipeline: OOO-Execution

Al fine di ottimizzare la pipeline sono state proposte diverse soluzioni.

- **Hardware Propagation**: tecnica hardware per rendere disponibili alcuni dati prodotti dalla pipeline prima che siano arrivato allo stage di WB. In particolare il dato viene salvato in appositi registri (o in memoria) alla fine dell'execution stage.
- **Software Stall**: inseriamo, in un flusso di programma, delle istruzioni di stallo fra due istruzioni. Al momento di un salto di cui ancora non si conosce il risultato, si inseriscono degli

stalli finché la destinazione non è nota. Questa soluzione è tipicamente utilizzata dai compilatori.

- **Software Re-Sequencing** (o Scheduling): Al momento di comilazione, se c'è un blocco atomico di programma, si ri-organizza il blococ di istruzioni in modo che le istruzioni che si dipendono siano più distanziate fra loro. Questo permette di evitare stalli.
- **Branch Prediction**: si predice il risultato di un salto prima che l'istruzione venga eseguita. Se la predizione è corretta, allora non si ha bisogno di stalli. Se la predizione è errata, allora si ha bisogno di stalli.

Le proposte precedenti sono state superate infine da una nuova strategia di ottimizzazione chiamata **Out-of-Order Execution** (OOO-Execution).

Definizione 1.5 (OOO-Execution)

L'OOO è una tecnica che permette l'esecuzione efficiente di istruzioni attuando un **riordinamento** delle istruzioni in modo da eseguire prima quelle indipendenti e poi quelle che necessitano del completamento di altre istruzioni. Il superamento di istruzioni indipendenti viene portato avanti anche fino al completamento **senza rendere visibile il risultato all'ISA**, in modo da **non violare** il program-flow.

Osservazione 1.7

L'OOO non è basato su come le istruzioni toccano l'hardware esposto dall'ISA, ovvero i registri. Inoltre, non viene rispettato l'esatto ordine con cui i sorgenti vengono programmati ma viene comunque rispettato il program-flow.

In questo contesto sono importanti due concetti:

Definizione 1.6 (Emission)

L'emissione è il processo di immettere istruzioni all'interno della pipeline.

Definizione 1.7 (Retire)

Il ritiro è l'azione di commit delle istruzioni, e rende i loro side effects "visibili" alle risorse hardware esposte all'ISA.

1.2.1 Algoritmo di Tomasulo

L'algoritmo di Tomasulo è un algoritmo di scheduling che permette di ottimizzare l'esecuzione di istruzioni in un processore pipelined permettendo di risolvere il problema di **data dependency**.

1.3 Intel x86

L'architettura x86 definisce una famiglia di ISA utilizzata poi anche da processori non Intel. Vediamo alcuni chipset che sono stati fondamentali nello sviluppo delle tecniche base di ottimizzazione della pipeline.

Esempio 1.2 (Intel 8086)

Negli anni tra il '76 e il '78 intel propose il chipset 8086, con 14 registri e in grado di processare le istruzioni senza pipeline ma con un ciclo di 4 fasi: *Fetch, Decode, Execute, Retire*.
In particolare l'istruzione di *Retire* veniva utilizzata per effettuare il commit dell'istruzione al fine di rendere visibili a tutte le risorse esposte dall'ISA i risultati e i side-effects dell'istruzione.

Esempio 1.3 (Intel 80486 - i486)

Introdotto nell'89 l'i486 è un processore con una pipeline a 5 stage, con 2 fasi di decode dovuto al più complesso sistema di indirizzamento della memoria.
Nonostante la pipeline il processore soffriva di un problema di latenza dovuta ad una forte dipendenza dei dati nel program flow. Ad esempio, per **scambiare il contenuto di due registri senza un registro d'appoggio** (lecito in programmazione di sistema dove tutti i registri sono fondamentali) **e senza accesso in memoria** (per non rendere il program-flow memory bound) l'unica cosa da fare è usare una sequenza di tre XOR del tipo:

$$XOR(a,b), XOR(b,a), XOR(a,b)$$

In questo modo si ottiene che la destinazione coincide con la sorgente, rendendo necessario l'inserimento di stalli per evitare che il risultato di una XOR venga utilizzato prima che sia stato calcolato.

Esempio 1.4 (Pentium Pro)

Questo chipset introduce il concetto di **superscalarità** di una pipeline, ovvero la possibilità di esegure più istruzioni che svolgono la stessa operazione in modo parallelo, sfruttando **risorse hardware duplicate**.

Prendiamo il Pentium Pro come esempio per descrivere i concetti dell'OOO-Execution. Questo processore utilizzava una ridondanza a livello hardware per eseguire gli stage di EX di più istruzioni contemporaneamente. Questo portò allo sviluppo di un modo per risolvere il cosiddetto:

Problema 1.1 (Instruction Time Span Problem)

Istruzioni che vengono eseguite contemporamenaente possono avere un tempo di esecuzione diverso, portando le linee di pipeline che processano istruzioni veloci a restare in attesa di linee più congestionate richiedendo più cicli di clock per essere finalizzate.

Infatti quello che in una pipeline tradizionale sarebbe stato risolto inserendo degli stall, in una pipeline superscalare con OOO-Ex viene risolto facendo rescheduling in modo tale da sfruttare le risorse duplicate (come una seconda ALU), per eseguire altre istruzioni e infine **preservare** il commit order (ovvero l'ordine in cui il programmatore vorrebbe che le istruzioni vengano eseguite).

1.4 Exception Handling

Nel contesto delle pipeline out of order il modo con cui vengono gestite le eccezioni ha un aspetto rilevante per la sicurezza. In particolare, a causa del riordinamento delle istruzioni potrebbe succedere che l'esecuzione (concorrente e speculativa) di un'istruzione potrebbe

sollevare un'eccezione anche quando questa **non appartiene** al program flow. Vediamo un esempio:

Esempio 1.5 (Phantom Exception)

Supponiamo di avere in pipe l'istruzione A e l'istruzione B. Supponiamo che sia A che B generino eccezioni e supponiamo che B sorpassi A per qualche motivo (riordinamento o esecuzione speculativa).
In questo contesto il processore vede l'eccezione di B prima che avvenga quella di A, ma essendo A precedente a B nel program-flow **tutte** le istruzioni **successive ad A non** dovevano essere eseguite. Ciò significa che **nemmeno B doveva essere eseguita**, impedendogli di generare l'eccezione.

Quando queste *phantom-exceptions* vengono eseguite in modo speculativo possono portare a **side-effects** non voluti, come ad esempio la modifica di registri che non dovrebbero essere modificati. Questo è un problema di sicurezza, in quanto potrebbe portare ad eseguire codice non voluto. Le vulnerabilità più famose tra queste sono **Spectre** e **Meltdown**.

Bibliografia

[1] «ElegantBook». In: (2022). https://github.com/ElegantLaTeX/ElegantBook.

[2] «The Legrand Orange Book». In: (2008). http://www.latextemplates.com/template/legrand-orange-book.