Transient Execution Attacks

Mengjia Yan
Spring 2024
Outline

• What is speculative execution?

• How does Meltdown work?
  • We will connect the dots between a hardware optimization and a software optimization.

• How does Spectre and its variations work?
  • Let’s try to see through these variations and understand the fundamental problem.
Recap: 5-stage Pipeline

I-Fetch (IF)  
Decode, Reg. (ID)  
Fetch (EX)  
Memory (MA)  
Write-Back (WB)
Recap: 5-stage Pipeline

• In-order execution:
  • Execute instructions according to the program order
  • One instruction max per pipeline stage
Build High-Performance Processors

Example #1:

```
FMUL f1, f2, f3 ; 10 cycles
ADD r4, r4, r1 ; 1 cycle -> repeat 10
......
```

Example #2:

```
LD r3, 0(r2) ; 1-100 cycles
ADD r4, r4, r1 ; 1 cycle -> repeat 10 times
......
```

Instruction-Level Parallelism (ILP) when there is NO data-dependency or control-flow dependency
Technique #1: Add More Functional Units

1: **FMUL** f1, f2, f3
2: **ADD** r4, r4, r1
3: **ADD** r4, r4, r1
Technique #1: Add More Functional Units

1: FMUL f1, f2, f3
2: ADD r4, r4, r1
3: ADD r4, r4, r1
Technique #1: Add More Functional Units

1: \text{FMUL} \ f1, f2, f3 ; \ f1=f2*f3
2: \text{FDIV} \ f5, f1, f4 ; \ f5=f1/f4

Need a bookkeeping mechanism to track dependency
# Technique #2: Scoreboard

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Busy?</th>
<th>Dest Reg</th>
<th>Src1 Reg</th>
<th>Src2 Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fmul</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fdiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Technique #2: Scoreboard

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Busy?</th>
<th>Dest Reg</th>
<th>Src1 Reg</th>
<th>Src2 Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fmul</td>
<td>Y</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>Fdiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1: **FMUL** f1, f2, f3
2: **ADD** r4, r4, r1

No dependency, feel free to issue the ADD
**Technique #2: Scoreboard**

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Busy?</th>
<th>Dest Reg</th>
<th>Src1 Reg</th>
<th>Src2 Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fmul</td>
<td>Y</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>Fdiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Read-after-Write (RAW)**

1: FMUL f1, f2, f3
2: FDIV f5, f1, f4

**Write-after-Write (WAW)**

1: FMUL f1, f2, f3 ; 10 cycles
2: FADD f1, f4, f5 ; 4 cycles
Technique #2: Scoreboard

• Upon issue an instruction, check:
  1. Whether any ongoing instructions will generate values for my source registers
  2. Whether any ongoing instructions will modify my destination register

We call such a processor: **in-order issue, out-of-order completion**.

A problem: how to handle interrupts/exceptions?
### Exception in OoO Processors: Example #1

1: **LD** \( r_3, 0(r_2) \); Exception in 3 cycles

2: **ADD** \( r_4, r_4, r_1 \); 1 cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: LD</td>
<td>IF</td>
<td>ID</td>
<td>Issue</td>
<td>ALU</td>
<td>Mem</td>
<td>Mem.</td>
<td>Mem</td>
<td><strong>Exception</strong></td>
</tr>
<tr>
<td>2: ADD</td>
<td>IF</td>
<td>ID</td>
<td>Issue</td>
<td>ALU</td>
<td></td>
<td></td>
<td></td>
<td><strong>WB</strong></td>
</tr>
</tbody>
</table>

Need to delay WB
Exception in OoO Processors: Example #2

1: FMUL f1, f2, f3 ; 10 cycles
2: LD r3, 0(r2) ; Exception in 1 cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: FMUL</td>
<td>IF</td>
<td>ID</td>
<td>Issue</td>
<td>FMUL</td>
<td>FMUL</td>
<td>FMUL</td>
<td>FMUL</td>
<td>...</td>
</tr>
<tr>
<td>2: LD</td>
<td>IF</td>
<td>ID</td>
<td>Issue</td>
<td>ALU</td>
<td>Mem</td>
<td></td>
<td>Exception</td>
<td></td>
</tr>
</tbody>
</table>
Technique #3: In-order Commit

IF → ID → Issue → Regs → ALU → Mem → Reorder Buffer → Commit

In-order

- ALU
- Mem
- Fadd
- Fmul
- Fdiv
Another Way to Draw It

To know more advanced out-of-order (OoO) features, take 6.5900 [6.823]
Re-examine Examples With In-order Commit

1: LD  r3, 0(r2) ; Exception in 3 cycles
2: ADD r4, r4, r1 ; 1 cycle

1: FMUL f1, f2, f3 ; 10 cycles
2: LD  r3, 0(r2) ; Exception in 1 cycle
Recap: Page Mapping

Physical Address Space (limited by DRAM size)

Process 1

- 4KB

Process 2

- 4KB

Page Table per process

VA

PA

4KB

4KB

4KB
Mapping Kernel Pages

Process 1

- 4KB

Page Table per process

VA

PA

Physical Address Space (limited by DRAM size)

- 4KB

Process 2

- 4KB

- 4KB

Kernel

- 4KB

- 4KB

- 4KB
Jumping Between User and Kernel Space

• Key challenge: need to make sure we use the correct page table
  • CR3 (in x86) or satp (in RISCV) stores the page table physical address
A Performance Optimization

• Context switch overhead:
  • Page table changes, so in many processors, we need to flush TLB

• But sometimes, we only go to kernel to do some simple things
  • E.g., getpid()

• The optimization: map kernel address into user space in a secure way
Map Kernel Pages Into User Space

- What will happen if accessing kernel addresses in user mode?
  - Protection fault
Meltdown

• Meltdown explores the combined effects of two optimizations
  • Hardware optimization: out-of-order execution
  • Software optimization: mapping kernel addresses into user space

• Let’s analyze the timing carefully

• Attack outcome: user space applications can read arbitrary kernel data

```c
...\nLd1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```
Meltdown Timing

Case 1: Fail. Ld2 is squashed before the corresponding memory access is issued.

Case 2: Attack works. Ld2’s request is sent out before the instruction is squashed.

......
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
Meltdown w/ Flush+Reload

1. Setup: Attacker allocates `probe_array`, with 256 cache lines. Flushes all its cache lines

2. Transmit: Attacker executes

   ```
   Ld1: uint8_t secret = *kernel_address;
   Ld2: uint8_t dummy = probe_array[secret*64];
   ```

3. Receive: After handling protection fault, attacker performs cache side channel attack to figure out which line of `probe_array` is accessed → recovers `byte`
Meltdown Mitigations

• Stop one of the optimizations should be sufficient
  • SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (e.g., EntryBleed)
  • HW: Stall speculation; Register poisoning

• We generally consider Meltdown as a design bug

.....
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];

Will Liu, EntryBleed, https://www.willsroot.io/2022/12/entrybleed.html?m=1
Branch Prediction

• Motivation: control-flow penalty
  • Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

![Diagram of branch prediction process]

- **In-order** stages:
  - Fetch
  - Decode
  - Commit

- **Out-of-order** stages:
  - Reorder Buffer
  - Execute

- **Kill** stages:
  - Kill
  - Kill
  - Kill

- **Inject handler PC**

- **Mis-prediction?**
Branch Prediction

• Naïve approach: PC+4

• More advanced, predict two things:
  • Direction of a branch (whether a branch is taken or not)
  • The target address of a branch
Branch Direction Predictor

• 1-bit predictor
  • If taken, set the bit to 1
  • If not-taken, set the bit to 0
  • Predict using this bit

• 2-bit predictor ... N-bit predictor

• More advanced:
  • Use global and local information together
  • Use Neural networks...
Spectre Variant 1 – Exploit Branch Condition

• Consider the following kernel code, e.g., in a system call:

```c
Br: if (x < size_array1) {
Ld1: secret = array1[x]
Ld2: y = array2[secret*64]
}
```

Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; `array1[x]` maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of `array2` was fetched

Always malicious?  
No. It may be a benign misprediction. We do not consider Spectre as a bug.
More Branch Predictors

• How to predict the target address of a branch?
  • jal <label> and blt r1, r2, <label>
  • jalr <r1>
  • ret

• Two structures:
  • Branch Target Buffer (BTB)
  • RAS (Return Address Stack)

2^k-entry direct-mapped BTB
(can also be associative)
Spectre Variant 2 – Exploit Branch Target

Train BTB properly → Execute arbitrary gadgets speculatively
General Attack Schema

Victim

Access secret
transmit (secret)

Attacker
recv()

Channel
Apply the General Attack Scheme

The RSA Square-and-Multiply Exponentiation example.
Attackers aim to leak $e$

Which is **access** operation?
Which is **transmit** operation?

$r = 1$

\[
\text{for } i = n-1 \text{ to } 0 \text{ do }
\]

\[
\begin{align*}
    r &= \text{sqr}(r) \\
    r &= \text{mod}(r, m) \\
    \text{if } e_i &= 1 \text{ then } \\
    r &= \text{mul}(r, b) \\
    \text{end }
\end{align*}
\]

$r = \text{mod}(r, m)$

end
Apply the General Attack Scheme

Which is \textit{access} operation? Which is \textit{transmit} operation?

\begin{verbatim}
......
Ld1: uint8_t secret = *kernel_address;
Ld2: uint8_t dummy = probe_array[secret*64];

Br: if (x < size_array1) {
Ld1: secret = array1[x]
Ld2: y = array2[secret*64]
}

Br: if (…) {
…
}
…
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
\end{verbatim}
General Attack Schema

- Traditional (non-transient) attacks
  - Data in-use

- Transient attacks: can leak data-at-rest
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

Victim

Access secret \hspace{1cm} transmit (secret)

Channel

Attacker

recv()
Next: Mitigations