Transient Execution Attacks

Mengjia Yan
Spring 2023
Outline

• What is transient execution attack?

• 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
Recap: 5-stage Pipeline

- In-order execution:
  - Execute instructions according to the program order
  - What is the ideal instruction throughput? -- instruction per cycle (IPC)
Build High-Performance Processors

Example #1:

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

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 between instructions
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: FMUL f1, f2, f3 ; f1=f2*f3
2: 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
## 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: FDIV f5, f1, f4

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

Need to delay WB

<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>Exception</td>
</tr>
<tr>
<td>2: ADD</td>
<td>IF</td>
<td>ID</td>
<td>Issue</td>
<td>ALU</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exception in OoO Processors: Example #2

1: \texttt{FMUL} f1, f2, f3 \text{ ; 10 cycles}
2: \texttt{LD} r3, 0(r2) \text{ ; 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>Exception</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Need to delay Exception
Technique #3: In-order Commit

Diagram showing the processing pipeline with stages labeled IF, ID, Issue, ALU, Mem, Reorder Buffer, and Commit. The pipeline includes stages for ALU, Mem, Fadd, Fmul, Fdiv, and other operations, with arrows indicating the flow of data. The diagram highlights the in-order nature of the commit process.
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
Mapping Kernel Pages

Physical Address Space (limited by DRAM size)

Process 1

4KB

Process 2

4KB

Kernel

4KB

Page Table per process

VA

PA

4KB

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

• Put two optimizations together, we have Meltdown
  • Hardware optimization: out-of-order execution
  • Software optimization: mapping kernel addresses into user space

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

......
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

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

• We generally consider Meltdown as a design bug

Will Liu, EntryBleed, https://www.willsroot.io/2022/12/entrybleed.html?m=1
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.
Spectre Variant 2 – Exploit Branch Target

• Most BTBs store partial tags and targets...
  • <last n bits of current PC, target PC>

```plaintext
oxfff110
Br: if (...) {
    ...
}
...

oxfff234
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```

Train BTB properly ➔ Execute arbitrary gadgets speculatively
General Attack Schema

Victim

Access secret

transmit (secret)

Channel

Attacker

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

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

```
r = 1
for i = n-1 to 0 do
    r = sqr(r)
    r = mod(r, m)
    if $e_i == 1$ then
        r = mul(r, b)
        r = mod(r, m)
    end
end
```
Apply the General Attack Scheme

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

```
......
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]
```
• 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

“Easy” to fix
Hard to fix
Next:
Software-Hardware Contract