18-447 Lecture 19:
Survey of Realworld VM Arch +
Decomposition of Meltdown

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – Part I: see many realworld, non-textbook examples of “VM”
  – Part II: everything in 447 together in Meltdown

• Notices
  – HW 4, due 4/8
  – Midterm regrades, due 4/3
  – Lab 4, due Thursday 4/25 (No late submissions)

• Readings
  – Synthesis Lecture: Architectural and Operating System Support for Virtual Memory (optional)
  – start on P&H Ch 6

emphasis on departures from textbook-conceptual norms
EA, VA and PA (IBM Power view)

64-bit EA\(_0\) divided into \(X\) fixed-size segments

64-bit EA\(_1\) divided into \(X\) fixed-size segments

80~90-bit VA divided into \(Y\) segments (\(Y\gg X\)); also divided as \(Z\) pages

40~50-bit PA divided into \(W\) pages (\(Z\gg W\))

Swap space divided into \(V\) pages (\(Z\gg V, V\gg W\))

Segmented EA: private, contiguous + sharing

Demand paged VA: capacity of disk, speed of DRAM
EA, VA and PA (almost everyone else)

Easy to blur **EA** and **VA** colloquially but full **VA** is \{ASID, EA\}!!!

- **EA}_0 with unique ASID=0
- **EA}_i with unique ASID=i

**VA** divided into **N** “address spaces” indexed by **ASID**; also divided as **Z** pages

**PA** divided into **W** pages (**Z>>W**)

Swap space divided into **V** pages (**Z>>V, V>?W**)

How do processes share pages?
SPARC V9 PTE/TLB Entry

- 64-bit VA + context ID
  - implementation can choose not to map high-order bits (require sign extension in unmapped bits)
  - e.g., UltraSPARC 1 mapped only lower 44 bits
- PA space size set by implementation, $2^{28}$ max pgs
- 64 entry fully associative I-TLB and D-TLB

Unlike caches, TLB specifics and operation have effects visible to kernel SW!!
SPARC TLB Miss Handling

• **32-bit V8** used a 3-level hierarchical page table for HW MMU page-table walk
  
  - L1 Table: 256 descriptors (1024-byte)
  - L2 Table: 64 descriptors (256-byte)
  - L3 Table: 64 PTEs (256-byte)

• **64-bit V9** switched to Translation Storage Buffer
  
  - a software managed, in-DRAM direct-mapped “cache” of PTEs (think hashed pg table or SW TLB)
  
  - HW assisted address generation on a TLB miss
  
  - TLB miss handler (SW) searches TSB. If TSB misses, a slower TSB-miss handler takes over
  
  - OS can use any page table structure after TSB
IBM PowerPC (32-bit)

segments 256MB regions

16-entry segment table

seg ID_{24}  seg offset_{16}  page offset_{12}

128 2-way ITLB and DTLB

PPN_{20}  page offset_{12}

64-bit PowerPC = 64-bit EA -> 80-bit VA \rightarrow 64-bit PA
How many segments in 64-bit EA?
IBM PowerPC Hashed Page Table

- HW table walk
  - VPN hashes into a PTE group (PTEG) of 8
  - 8 PTEs searched for tag match
  - if not found in first PTEG search a secondary PTEG
  - if not found in 2\(^{nd}\) PTEG, trap to software handler
- Hashed table structure also used for 64-bit EA→VA
MIPS R10K

• 64-bit VA
  – top 2 bits set kernel/supervisor/user mode
  – additional bits set cache and translation behavior
  – bit 61-40 not translate at all
    (holes and repeats in the VA??)

• 8-bit ASID (address space ID) distinguishes between processes

• 40-bit PA

• Translation -
  “64”-bit VA and 8-bit ASID → 40-bit PA
MIPS TLB

- 64-entry fully associative unified TLB
- Each entry maps 2 consecutive VPNs to independent respective PPNs
- Software TLB-miss handling (*exotic at the time*)
  - 7-instruction page table walk in the best case
  - TLB Write Random: chooses a random entry for TLB replacement
  - OS can exclude low TLB entries from replacement (some translations must not miss)

- TLB entry
  - N: noncacheable
  - V: valid
  - D: dirty (*write-enable!!*)
  - G: ignore ASID

<table>
<thead>
<tr>
<th>VPN&lt;sub&gt;20&lt;/sub&gt;</th>
<th>ASID&lt;sub&gt;6&lt;/sub&gt;</th>
<th>0&lt;sub&gt;6&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN&lt;sub&gt;20&lt;/sub&gt;</td>
<td>ndvg</td>
<td>0&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
MIPS Bottom-Up Hierarchical Table

- TLB miss vectors to a SW handler
  - page table organization is not hardcoded in ISA
  - ISA favors a chosen reference page table scheme by providing “optional” hardware assistance

- Bottom-Up Table
  - start with 2-level hierarchical table (32-bit case)
  - allocate all L2 tables for all VA pages (empty or not) linearly in the mapped kseg space
  - VPN is index into this linear table in VA

This table scales with VA size!! Is this okay?
Bottom-Up Table Walk

- **VPN**
- **PO**
- **VA on TLB Miss, trap**
  - whose address space?
- **VA of PTE**
  - (generated automatically by HW after TLB miss)
- **PPN**
- **status**

Can this load miss in the TLB? What happens if it misses?

*notice translation also eats up TLB entries!*
User TLB Miss Handling

mfc0 k0,tlbcxt  # move the contents of TLB context register into k0
mfc0 k1,.epc   # move PC of faulting memory instruction into k1
lw  k0,0(k0)   # load thru address that was in TLB context register
mtc0 k0,entry_lo # move the loaded value (a PTE) into the EntryLo register
tlbwr # write PTE into the TLB at a random slot number
j  k1 # jump to PC of faulting load instruction to retry
rfe # restore privilege (in delay slot)
HP PA-RISC: PID and AID

- 2-level: 64b EA → 96b VA (global) → 64b PA
- Variable sized segmented EA → VA translation
- Rights-based access control
  - user controls segment registers (user can generate any VA it wants!!)
    
    *in contrast, everyone else controls translation to control what VA can be reached from a process*
  - each virtual page has an access ID (AID) assigned by OS
  - each process has 8 active protection IDs (PIDs) in privileged HW registers managed by OS
  - a process can access a page only if one of the 8 PIDs matches the page’s AID *(think lock and keys)*
Intel 80386

• Two-level address translation:
  segmented EA → global VA → PA

• User-private 48-bit EA
  – 16-bit SN (implicit) + 32-bit SO
  – 6 user-controlled registers hold active SNs;
    selected according to usage: code, data, stack, etc

• Global 32-bit VA
  – 20-bit VPN + 12-bit PO

• An implementation defined paged PA space

What is very odd about this?
Living with too small VA space

- 32-bit global VA too small to share by processes
  - per-process EA space oddly bigger than VA space
  - until 1990, no one cared

- Later multitasking OS ignore segment protection
  - time-multiplex **global** VA space for use by 1 process at a time
  - code, data, stack segments always map to entire VA space, 0~(2^{32}-1)
  - set MMU to use a different table on context switch
  - BUT! TLB for VA translation doesn’t have ASID; must flush TLB on context switch

- Much later IA32e/Intel 64 added PCID to TLB
A Contemporary Example: RISC-V

- **Sv32** (also **Sv39, Sv48, Sv48**): hierarchical, HW-walked table for each “hart”

![Virtual Address Diagram](image)

**Figure 4.15: Sv32 virtual address.**

![Physical Address Diagram](image)

**Figure 4.16: Sv32 physical address.**

![Page Table Entry Diagram](image)

**Figure 4.17: Sv32 page table entry.**

- With hypervisor: **VA → Guest PA → Supervisor PA**
- Does not define a TLB (**μarch can cache translation**)
SV57

Figure 4.25: Sv57 virtual address.

Figure 4.26: Sv57 physical address.

Figure 4.27: Sv57 page table entry.
Meltdown in 18-447 Terms

*How to “know” the value at a memory location without permission to read it?*
VA to PA Translation Flow Chart

1. TLB lookup
   - no → PT walk
     - 10~100 pclk
     - page in DRAM
       - found → PA to cache
       - don’t exist → not in DRAM
         - allocate or bring from disk (10 ms)
         - “page fault” now what?
         - “seg fault” now what?
       - “protection violation”
     - hit → protection check
       - okay → PA to cache
       - violation → now what?

   - yes → protection check
     - okay → PA to cache
     - violation → now what?

2. VA to PATranslation Flow Chart

   ©2024
VA to PA Translation Flow Chart

- TLB lookup
  - no
    - PT walk
      - page in DRAM
        - found
          - hit
            - protection check
              - ISA says can’t “read” without permission
              - okay
                - PA to cache
              - not in DRAM
                - “page fault” allocate or bring from disk (10 ms)
          - not in DRAM
            - “seg fault” now what?
        - don’t exist
          - “seg fault” now what?
      - no
        - create TLB
  - yes
    - protection check
      - ISA says can’t “read” without permission
      - okay
        - PA to cache
      - now what?
        - “seg fault” now what?
How should VM and Cache Interact?

Only a question for L1 caches
How should VM and Cache Interact?

CPU

TLB

cache

lower hier.

virtual

ISA allows μarchitecture to perform “read” without permission; ISA only care program can’t “see” the read-value

physical

from L17 ...

Only a question for L1 caches
“Flushing” a Pipeline

<table>
<thead>
<tr>
<th></th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
<th>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>Iₜ</td>
<td>Iₜ₊₁</td>
<td>Iₜ₊₂</td>
</tr>
<tr>
<td>ID</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>Iₜ</td>
<td>Iₜ₊₁</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>Iₜ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
</tr>
<tr>
<td>WB</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Kill faulting and younger inst; drain older inst
- Don’t start handler until faulting inst. is oldest
- Better yet, don’t start handler until pipeline is empty

Better to be safe than to be fast
“Flushed” a Pipeline

<table>
<thead>
<tr>
<th></th>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
<th>t_8</th>
<th>t_9</th>
<th>t_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>l_0</td>
<td>l_1</td>
<td>l_2</td>
<td>l_3</td>
<td>l_4</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>l_h</td>
<td>l_{h+1}</td>
<td>l_{h+2}</td>
</tr>
<tr>
<td>ID</td>
<td>l_0</td>
<td>l_1</td>
<td>l_2</td>
<td>l_3</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>l_h</td>
<td>l_{h+1}</td>
<td>l_{h+1}</td>
<td></td>
</tr>
</tbody>
</table>

In fact, μarchitecturally
• can read without permission
• can even use read-value in dependent instructions
• as long as at the end can’t “see” any of it

100s of speculative instructions in flight in modern OOO CPUs

Better to be safe than to be fast
Key Idea 3: Inter-Model Compatibility

“a valid program whose logic will not depend implicitly upon time of execution and which runs upon configuration A, will also run on configuration B if the latter includes at least the required storage, at least the required I/O devices ....”

• Invalid programs not constrained to yield same result
  – “invalid”==violating architecture manual
  – “exceptions” are architecturally defined

• The King of Binary Compatibility: Intel x86, IBM 360
  – stable software base and ecosystem
  – performance scalability

[Amdahl, Blaauw and Brooks, 1964]
Key Idea 3: Inter-Model Compatibility

“a valid program whose logic will not depend implicitly upon time of execution and which runs upon configuration A, will also run upon configuration B if the latter includes at least the required storage, at least the required I/O devices ...”

- Invalid programs not constrained to yield same result
  - “invalid” == violating architecture manual
  - “exceptions” are architecturally defined
- The King of Binary Compatibility: Intel x86, IBM 360
  - stable software base and ecosystem
  - performance scalability

from L02 ...

[Amdahl, Blaauw and Brooks, 1964]
What cache is in your computer?

• How to figure out what cache configuration is in your computer
  – capacity (C), associativity (a), and block-size (B)
  – number of levels

• The presence or lack of a cache should not be detectable by functional behavior of software

• But you could tell if you measured execution time to infer the number of cache misses
What cache is in your computer?

- How to figure out what cache configuration is in your computer
  - capacity ($C$), associativity ($a$), and block-size ($B$)
  - number of levels
- The presence or lack of a cache should not be detectable by functional behavior of software
- But you could tell if you measured execution time to infer the number of cache misses

Cache invisible architecturally, but performance “side-effect” easily detectable using timer

Timing side-channel attack: infer read-value without “seeing” by running code to cause hit/miss based on unseen value
MIPS R10K

- 64-bit virtual address
  - top 2 bits set kernel/supervisor/user mode
  - additional bits set cache and translation behavior
  - bit 61-40 not translate at all
    (holes and repeats in the VA??)

- 8-bit ASID (address space ID) distinguishes between processes

- 40-bit physical address

Translation -
4"-bit VA and 8-bit ASID \(\rightarrow\) 40-bit PA
MIPS R10K

- 64-bit virtual address
  - top 2 bits set kernel/supervisor/user mode
  - additional bits set cache and translation behavior
  - bit 61-40 not translate at all (holes and repeats in the VA)
- 8-bit ASID (address space ID) distinguishes between processes
- 40-bit physical address

Translation:
- "64"-bit VA and 8-bit ASID → 40-bit PA

<table>
<thead>
<tr>
<th>X:</th>
<th>1 GB mapped (kseg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y:</td>
<td>0.5 GB unmapped uncached</td>
</tr>
<tr>
<td></td>
<td>0.5 GB unmapped cached</td>
</tr>
<tr>
<td></td>
<td>bottom 2 GB mapped (normal)</td>
</tr>
</tbody>
</table>

simplified example from 32-bit VA in R2000/3000

Read addr \( Y+C, Y+2C, Y+3C \ldots \) so addr Y is not in cache; then attempt to execute:

\[
\begin{align*}
I_1: &\ lw\ t0,\ 0(r"X") \\
I_2: &\ andi\ t0,\ t0,\ 0x1 \\
I_3: &\ sle\ t0,\ t0,"\log_2(\text{blocksize})" \\
I_4: &\ add\ t0,\ t0,\ r"Y" \\
I_5: &\ lw\ x0,\ 0(t0)
\end{align*}
\]

\( I_1 \) is an exception so \( I_1 \sim I_5 \) not observed architecturally; nevertheless addr Y is cached if LSB of mem[X] is 0
Control Speculation: PC+4

When inst$_h$ branch resolves
- branch target (Inst$_k$) is fetched
- flush instructions fetched since inst$_h$ ("wrong-path")
Control Speculation: PC+4

When $\text{Inst}_h$ branch resolves
- branch target ($\text{Inst}_k$) is fetched
- flush instructions fetched since $\text{Inst}_h$ ("wrong-path")

Train BTB so the previous executes as "wrong path" —
architecturally nothing illegal happened!!

control flow "restitched"
Idempotency and Side-effects

• Loading from real memory location $M[A]$ should return most recent value stored to $M[A]$
  $\Rightarrow$ writing $M[A]$ once is the same as writing $M[A]$ with same value multiple times in a row
  $\Rightarrow$ reading $M[A]$ multiple times returns same value
  This is why memory caching works!!

• LW/SW to mmap locations can have side-effects
  – reading/writing mmap location can imply commands and other state changes
  – consider a FIFO example
    • SW to 0xffff0000 pushes value
    • LW from 0xffff0000 returns popped value

What happens if 0xffff0000 is cached?
Idempotency and Side-effects

- Meltdown vulnerability not a bug but an ISA-allowed simplification—*no fast kill after exception as with BP miss*
- Same issue doesn’t arise with MMIO—ISA disallows spurious read if PTE says “uncacheable” or “side-effect”

*Not a “bug” but something is very wrong!!!* 
*How to fix this . . . .*

- LW/SW to mmap locations can have side-effects
  - reading/writing mmap location can imply commands and other state changes
  - consider a FIFO example
    - SW to 0xffff0000 pushes value
    - LW from 0xffff0000 returns popped value

What happens if 0xffff0000 is cached?