18-447 Lecture 19: Survey of Modern VMs + a Decomposition of Meltdown

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – see the many realizations of “VM”, focusing on deviation from textbook-conceptual norms
  – put everything in 447 together in Meltdown
• Notices
  – Midterm 4/8 in class; covers Lectures 10~19
  – HW 4, due on Friday
  – Lab 4, status check next week
• Readings
  – start on P&H Ch 6
2 Parts to Modern VM

- In a multi-tasking system, virtual memory supports the illusion of a large, private, and uniform memory space to each process.

- Ingredient A: naming and protection
  - each process sees a large, contiguous address space without holes (for convenience)
  - each process’s memory is private, i.e., protected from access by other processes (for sharing)

- Ingredient B: demand paging (for hierarchy)
  - capacity of secondary storage (swap space on disk)
  - speed of primary storage (DRAM)
**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 (\(Y > X\)); also divided as \(Z\) pages (\(Z > Y\))

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

Swap disk divided into \(V\) pages (\(Z > V\), \(V > W\))

**Recall**

Segmented \(EA\): private, contiguous + sharing

Demand paged \(VA\): size of swap, speed of DRAM
EA, VA and PA (almost everyone else)

- **EA₀** with unique ASID=0
- **EA_i** with unique ASID=i

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

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

Swap disk divided into V pages (Z>>V, V>>W)

How do processes share pages?

Recall EA and VA to blur
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 maps only lower 44 bits
- **PA** space size set by implementation
- **64 entry fully associative** I-TLB and D-TLB

Details fixed and exposed by privileged ISA

18-447-S19-L19-S7, James C. Hoe, CMU/ECE/CALCM, ©2019
SPARC TLB Miss Handling

- 32-bit V8 used a 3-level hierarchical page table for HW MMU page-table walk

- 64-bit V9 switched to Translation Storage Buffer
  - a software managed, in-DRAM direct-mapped “cache” of PTEs (think hashed page table)
  - 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

128 2-way ITLB and DTLB

PPN_{20} page offset_{12}

64-bit PowerPC = 64-bit EA -> 80-bit VA → 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 sequentially for tag match
  - if not found in first **PTEG** search a second **PTEG**
  - if not found in 2\(^n\)d **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** $\rightarrow$ 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
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**: Virtual Page Number
- **PO**: Page Offset
- **VA**: Virtual Address
- **PTEBase**: Page Table Entry Base
- **VPN**: Virtual Page Number
- **0s**: 0
- **mem load**: Memory load
- **PPN**: Physical Page Number
- **status**: Status

### VA on TLB Miss, trap
- **VA** of PTE (generated automatically by HW after TLB miss)

### whose address space?
- PTE loaded from mem

### 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 controlled by OS
  - a process can access a page only if one of the 8 PIDs matches the page’s AID
Intel 80386

• Two-level address translation:
  segmented \( \text{EA} \rightarrow \text{VA} \rightarrow \text{PA} \)

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

• Global 32-bit \( \text{VA} \)
  – 20-bit \( \text{VPN} \) + 12-bit \( \text{PO} \)

• An implementation defined paged \( \text{PA} \) space

What is very odd about this?
Living with the mistake

• 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

  Later IA32e added PCID to TLB as fix
Meltdown in 18-447 Terms

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

EA -> TLB lookup

- no
  - hit
    - protection check
      - no
        - ISA says can’t “read” without permission
      - yes
        - PA to cache
  - yes
    - update TLB

- PT walk
  - page in DRAM
    - found
      - page on disk
        - okay
    - don’t exist
      - “page fault” demand paging
      - 10~100 pclk
      - 10 msec

- “seg fault” now what?
  - “protection violation”
How should VM and Cache Interact?

Actually you can “read” without permission; ISA (as an abstraction) only care you can’t “see” the read-value
"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>Iₕ</td>
<td>Iₕ₊₁</td>
<td>Iₕ₊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>Iₕ</td>
<td>Iₕ₊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>Iₕ</td>
<td>Iₕ</td>
<td></td>
</tr>
</tbody>
</table>

- 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 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

[a fundamental tenet that ISA does not care about time]

[Recall 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 \((a)\)
  – block-size \((B)\)

• 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

Infer read-value without “seeing” by running code to cause hit/miss based on unseen value

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

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

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

- MIPS R10K
  - 1 GB mapped (kseg)
  - 0.5 GB unmapped uncached
  - 0.5 GB unmapped cached
  - bottom 2 GB mapped (normal)

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:

\[
\begin{align*}
&l_1: lw\ t0, \ 0(r"X") \\
&l_2: andi\ t0,\ t0,\ 0x1 \\
&l_3: slt\ t0,\ t0, "\text{log}_2(\text{blocksize})" \\
&l_4: add\ t0,\ t0,\ r"Y" \\
&l_5: lw\ x0,\ 0(t0)
\end{align*}
\]

\( l_1 \) is an exception so \( l_1 \sim l_5 \) not observed architecturally; nevertheless addr \( Y \) is cached if LSB of mem[\( X \)] is 0
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!!

Recall

\( \text{Inst}_h \) is a taken branch
Idempotency and Side-effects

- Meltdown vulnerability not a bug but a purposeful performance optimization permitted by ISA
- Same issue doesn’t arise with MMIO—because ISA disallow spurious read if TLB says “uncacheable”

Even not a bug, something is very wrong!!!

- 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?
Midterm2

• Covers lectures (L10~L19), HW, projects, assigned readings (from textbooks and papers)

• Types of questions
  – freebies: remember the materials
  – >> probing: understand the materials <<
  – applied: apply the materials in original interpretation

• **55 minutes, 55 points**
  – point values calibrated to time needed
  – closed-book, one 8½x11-in² hand-written cribsheet + your cribsheet from M1
  – no electronics
  – use pencil or black/blue ink only