ECE 571 – Advanced Microprocessor-Based Design Lecture 13

Vince Weaver

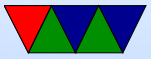
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

6 March 2018

Announcements

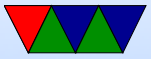
- HW#6 was posted, Prefetcher
- Midterm: Thursday after spring break.
- Projects... this year it's a value prediction tournament



HW#5 Review

1. People had trouble with this.
   Haswell machine has a 44-bit physical address space, 32-kB L1 data cache, 8-way set associative, 64-bytes per line.

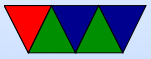
   (a) Offset = 6 bits
   (b) Index = $2^{15}/2^3/2^6 = 2^6 = 6$ bits
   Why does having 12 bits of offset+index makes VIPT caches easier (4096 bytes)
   (c) 32-bit tag



2. Cache Example

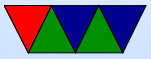
(a) ld 0000 080f = line 0, tag 8 = hit
(b) ld ffff ffff = line f, tag ffffff = miss (cold)
(c) strb 0000 0810 = line 1 tag 8 = miss (unknown type), and LRU says we throw out tag a, which is dirty, so writeback
(d) strb r0, 0xffffffff – hit. Set dirty bit

3. Bzip2 on Haswell
   Haswell memory parameters: L1-icache 32k/8-way/64B L1-cache 32k/8-WAY/64B,4/5 cycles, writeback, shared between threads



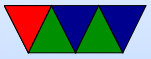
L2 cache 256k/8-way/64B, 12 cycles, writeback
L3 cache 8MB,64B, writeback
(What doesn't this say? replacement policy? inclusive/exclusive? write-back?)
Bzip: 11MB footprint

(a) L1-icache = 13k/19B = 0% miss rate
(b) L1-dcache-load = 311M/6.2B = 5% miss rate
(c) L2 = 208M/411M = 50% miss rate
(d) LLC 601k/139M = 0.4% miss rate



(e) Note, l1-dcache is loads. Issue with l1d-stores, in Linux 4.1 (17 Feb 2015) Kleen posted patch to separate SNB evens from HSW in Linux kernel. So your results will change based on kernel version. Annoying. Before there was a l1d-store events

(f) Why do the results not match up? Shouldn't L1-misses be same as L2-accceses? Why would they not match up? Bug in counters, not counting stores, bug in counter selection, other things going on in system, shared resources, chip errata, prefetching, etc. LLC actually uses offcore-response events



(g) What can we tell about bzip2 behavior? Fits well in icache. Why is L2 so bad? single threaded

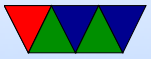
4. equake_l on Haswell

   e-quake mem footprint 700MB

(a) L1-icache = 14M/1.4T = 0%
(b) L1-dcache = 31B/526B = 5.8%
(c) L2 = 22B/52B = 42%
(d) LLC = 8B/13B=58%

5. bzip2 on Jetson

   Jetson TX-1 4 GB LPDDR4



L1 i-cache=48 kB, 3-way,

l1 d-cache=32 kB, 2-way

l2 = 2 MB, 16-way (big?) 512 kB (little?)

(a) L1-icache = 184k/10B = 0%

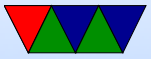
(b) L1-dcache-load = 254M/6B = 4%

(c) L1-dcache-store = 56M/2.2B = 2.5%

(d) L2-dcache-load = 28M/330M = 8.5%

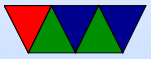
(e) L2-dcache-store = 21M/308M = 6.8%

(f) Why did we have to use raw events? Proper Cortex-A57 event support not added until Linux 4.4. Need to update the kernel, tricky on Jetson.

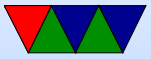


Quick run-through, the path of a load

- OoO, load buffer, etc
- VIPT. So on access it looks up the physical tag in TLB while reading out the tags from each way with the index. Also keep in mind MESI is going on at this level.
- If tag from TLB matches a tag from cache, hit! Good! Cache hit!
- If tag in TLB but not in cache, cache miss.
- If tag not in TLB, TLB miss. Won't know if cache hit until later.

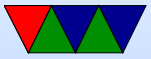


- Now let the hardware walk the page tables.
- If hardware finds the page, great! Return it back up to the TLB
- If hardware can't find the page, time to get the Operating System involved. Page fault.
- Hardware has a list of what should be in memory where (from the executable). Typically these are demand-loaded
  - Text/code – read from disk
  - Data – read from disk
  - BSS – allocate zeros



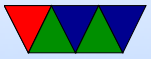
- Stack – if near top growing down, auto-grow
- Heap – similar to stack
- Shared page– could already be in memory (shared lib?) Just need to point to it.
- Zeros – just have one page of zeros you can point to
- Paged out to disk – have offset in page file, need to load it

- Time to bring in the page! Need to find room in Physical RAM. If no room, need to make room. Possibly paging out to disk (this is what LRU/dirty bits are used for). What kind of issues come up when low on RAM and



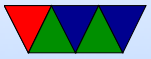
constantly paging same pages in and out (thrashing?)

- Page now in physical RAM, time to go backwards. Update the page table
- Fill in the TLB. Return to memory.
- If page fault occurred, usually re-execute the instruction.
- Issues
  - Could you have race where you re-execute it and the page had gotten swapped out again?
  - Can we page out the page tables? What can go wrong there? Double faults? How many nested page faults can you handle?

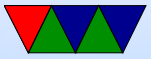


Quick run-through, the path of a store

- Is it much different?

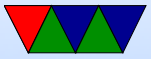


Real World Examples



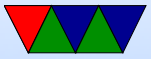
Haswell Virtual Memory

- ITLB
  - 4kB: 128 entry, 4-way, dynamic between Hyperthreads
  - 2MB/4MB: 8, fully assoc, duplicated ht
- DTLB
  - 4kB: 64-entry, 4-way, fixed partition
  - 2MB/4MB: 32 entry, 4-way
  - 1GB: 4-entry, 4-way
- STLB (second level)
  - 4kB/2MB: 1024 entry, 8-way

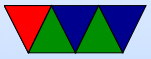


Cortex A9 MMU

- Virtual Memory System Architecture version 7 (VMSAv7)
- page table entries that support 4KB, 64KB, 1MB, and 16MB
- global and address space ID (no more TLB flush on context switch)
- instruction micro-TLB (32 or 64 fully associative)

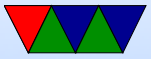


- data micro-TLB (32 fully associative)
- Unified main TLB, 2-way, 2x64 (128 total) on pandaboard
- 4 lockable entries (why want to do that?)
- Supports hardware page table walks



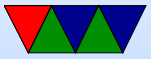
Cortex A9 MMU

- Virtual Memory System Architecture version 7 (VMSAv7)
- Addresses can be 40bits virt / 32 physical
- First check FCSE – linear translation of bottom 32MB to arbitrary block in physical memory (optional with VMSAv7)

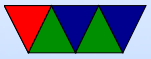


Cortex A9 TLB

- micro-TLB. 1 cycle access. needs to be flushed if ASID changes
- fully-associative lockable 4 elements plus 2-way larger. varying cycles access

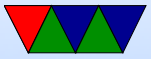


Meltdown Security



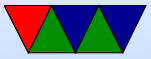
Side Channel Attacks

- Leak info in unexpected ways
- Timing attacks... code takes different number of instructions to execute either side of if/then. If encrypting can maybe tell difference between 0 and 1 unless each way takes exact same cycles
- Leakage: time, performance counters, RF noise, anything shared (caches, stalls on hyperthreads), LEDs on routers, etc



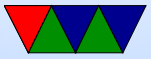
More Meltdown

- Primarily an Intel issue, chips with speculative execution, dating back to Pentium Pro?
- Some very high-end ARM (ARM75) too as well as IBM Power? Not AMD though.
- Problem due to out-of-order processing and speculative execution
- Specifically, what if the results of speculation aren't thrown out and can be accessed somehow
- Virtual Memory: for higher performance, kernel often



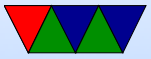
mapped into user address space (but with protection bits to disallow user access)

Often the kernel also has all of physical memory mapped somewhere so it can access all of it. (This was trickier in 32-bit days)

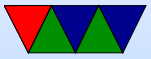


Cache Side-Channel Attacks

- Evict and Time – run and time. Then evict just one line, then run again. If ran slower, it depended on data in that line
- Prime and Probe – load cache full of your data. Wait until code of interest runs. Then see how many of cache lines got kicked out.
- Flush and Reload
  - Single cache line granularity
  - Use `cflush` or similar to kick out a cache line



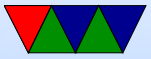
- Reload and time it, can tell if someone else had reloaded it in the meantime by how fast it loads



Meltdown – Toy Code

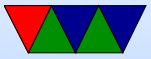
```
raise_exception()
access(prob_array[data*4096])
```

- The access should never happen, as the exception (segfault, etc) will trigger
- If exception slow enough, the access will likely be speculatively executed
- In theory the results of the access are thrown out so you cannot know it happened
- The speculative load might end up in the cache though,



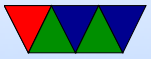
and you can probe to see if it did

- The *bug* is that on Intel chips you can speculatively access kernel data from userspace and it will be cached despite the permission mismatch.
- Why multiply by 4096? Spread across multiple pages so the prefetcher doesn't get in the way.



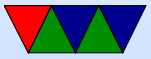
Meltdown – Issues

- Exception Handling – either a signal handler, or else forking a child to cause the exception
- Exception suppression transactional memory
-



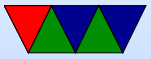
Meltdown – Limitations

- Can get false zeros. Repeat until sure.



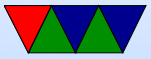
KASLR

- Address Space randomization. If want arbitrarily read out kernel info need to find kernel
- Turns out that with 40 bit address space and large physical memory (8GB) it doesn't take too many tries to find kernel



Performance

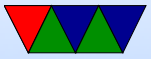
- Up to 500K/s read out



Workarounds

- Hardware
  - Turn off out-of-order (not possible, expensive)
  - Not allow user speculation to access kernel addresses
  - Not put data into cache until permission check completes
- Software
  - KAISER (KPTI) – map kernel in separate address space.

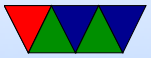
    Large overhead on switch in/out of kernel (syscalls,



context switch)

Up to 30% on someworkloads, almost none on others

- PCID (intel's ASID implementation on Westmere or newer) helps avoid complete TLB flush



Spectre – See next lecture