ECE 571 – Advanced Microprocessor-Based Design Lecture 19

Vince Weaver

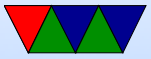
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

16 October 2020

Announcements

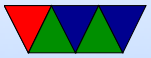
- HW#6 was due
- Project will be coming up
- Midterm end of month



HW#5 Review – Cache Sizing

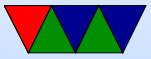
- Haswell-EP machine has a 46-bit physical address space (48 bit virtual), 32-kB L1 data cache, 8-way set associative, 64-bytes per line.
  - Offset = 6 bits
  - Index = $2^{15}/2^3/2^6 = 2^6 = 6$ bits

    Why does having 12 bits of offset+index makes VIPT caches easier (4096 bytes)
  - 46-6-6=34-bit tag



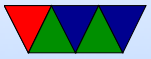
HW#5 Review – Cache Example

- ld 0000 080f = line 0, tag 8 = hit
- ld ffff ffff = line f, tag ffffff = miss (cold)
- strb 0000 0810 = line 1 tag 8 = miss (unknown type), LRU says throw out tag a (dirty) so writeback
  NOTE: not necessarily a conflict miss. Yes there was a conflict and something got kicked out, but the miss itself is only a conflict if it had been in cache before (which we don't know)
- strb r0, 0xffffffff – hit. Set dirty bit

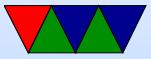


HW#5 Review – Bzip2 on Haswell-EP

- Haswel-EP 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

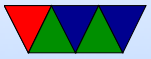


- L1-icache = 13k/19B = 0% miss rate
- L1-dcache-load = 311M/6.2B = 5% miss rate
- L2 = 208M/411M = 50% miss rate
- LLC 1k/139M = 0% miss rate
- Note, l1-dcache is loads. Issue with l1d-stores, in Linux 4.1 (17 Feb 2015) Kleen posted patch to separate SNB evens from HSW in Linux kernel. So your results will change based on kernel version. Annoying. Before there was a l1d-store events
- Why do the results not match up? Shouldn't L1-misses be same as L2-accceses? Why would they not



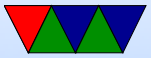
match up? Bug in counters, not counting stores, bug in counter selection, other things going on in system, shared resources, chip errata, prefetching, etc. LLC actually uses offcore-response events

- What can we tell about bzip2 behavior? Fits well in icache. Why is L2 so bad? single threaded? Prefetching location?



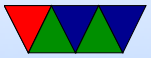
HW#5 Review – equake_l on Haswell-EP

- e-quake mem footprint 700MB
  - L1-icache = 14M/1.4T = 0%
  - L1-dcache = 31B/526B = 5.8%
  - L2 = 22B/52B = 42%
  - LLC = 2B/14B=14%

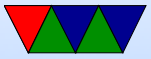


HW#5 Review – bzip2 on Jetson

- 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?)
- Results
  - L1-icache = 184k/10B = 0%
  - L1-dcache-load = 254M/6B = 4%
  - L1-dcache-store = 56M/2.2B = 2.5%
  - L2-dcache-load = 28M/330M = 8.5%

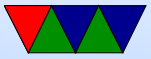


  - L2-dcache-store = 21M/308M = 14%
- Why did we have to use raw events? Proper Cortex-A57 event support not added until Linux 4.4. Need to update the kernel, tricky on Jetson.



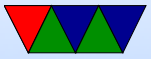
What happens on a memory access

- Cache hit, generally not a problem, see later. To be in cache had to have gone through the whole VM process. Although some architectures do a lookup anyway in case permissions have changed.
- Cache miss, then send access out to memory
- If in TLB, not a problem, right page fetched from physical memory, TLB updated
- If not in TLB, then the page tables are walked



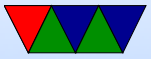
(by the hardware on x86)

- It no physical mapping in page table, then page fault happens



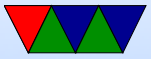
What happens on a page fault

- The OS process structure has info on what memory regions are valid and what should be there (text/data comes from executable on disk, bss zeroed pages, heap/stack might be auto-allocated zeroed pages)
- "minor" – page is already in memory, just need to point a PTE at it. For example, shared memory, shared libraries, etc.
- "major" – page needs to be ~~created or~~ brought in from disk.



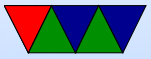
- Demand paging.
- Needs to find room in physical memory.
- If no free space available, needs to kick something out.
  Disk-backed (and not dirty) just discarded.
  Disk-backed and dirty, written back.
- Memory can be paged to disk. Eventually can OOM.
- Memory is then loaded, or zeroed, and PTE updated.
- Can it be shared? (zero page)

- "invalid" – segfault

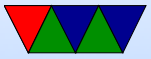


Quick run-through, the path of a load

- 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.

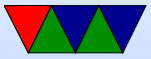


- Now let the hardware walk the page tables.
- If hardware finds the page, great! Return it back up to the TLB
- If hardware can't find the page, time to get the Operating System involved. Page fault.
- Hardware has a list of what should be in memory where (from the executable). Typically these are demand-loaded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



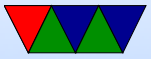
- Stack – if near top growing down, auto-grow
- Heap – similar to stack
- Shared page– could already be in memory (shared lib?) Just need to point to it.
- Zeros – just have one page of zeros you can point to
- Paged out to disk – have offset in page file, need to load it

- 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



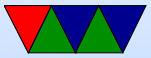
constantly paging same pages in and out (thrashing?)

- 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?



What happens on a fork?

- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)
- Page table marked read-only, then shared
- Only if writes happen, take page fault, then copy made
  Copy-on-write