ECE 571 – Advanced Microprocessor-Based Design Lecture 19

Vince Weaver

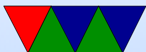
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

21 October 2024

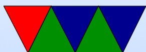
Announcements

- HW#6 was posted
- Project was posted, check it out when you get the chance
- Midterm end of month



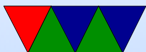
HW#5 Review – Cache Sizing

- This is for a desktop Haswell machine (the Haswell-EP from homework actually has 46-bits of physical address space)
- Side note: 64-bits is 16 Exabytes, 48 bits is 256TB, 44 bits is 16TB
- Haswell machine has a 44-bit physical address space (48 bit virtual), 32-kB L1 data cache, 8-way set associative, 64-bytes per line.
  - Offset = 6 bits



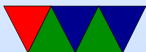
- ◦ Index $= 2^{15}/2^3/2^6 = 2^6 = 6$ bits

  Why does having 12 bits of offset+index makes VIPT caches easier (4096 bytes)
- ◦ 44-6-6=32-bit tag



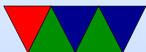
HW#5 Review – Cache Example

- People did more or less OK on this



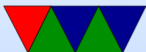
HW#5 – Haswell-EP Memory Paramaters

- Xeon E5 2640-v3
- 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 20MB,64B, writeback
- (What doesn't this say? replacement policy? inclusive/exclusive? write-back?)



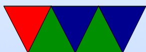
HW#5 – Haswell-EP bzip2

- Bzip: 11MB footprint
  - L1-icache = 107k/19B = 0% miss rate
  - L1-dcache-load = 314M/5.8B = 5.4% miss rate
  - L2 = 208M/414M = 50% miss rate
  - LLC 1k/143M = 0% miss rate

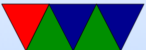


HW#5 – Notes on the events

- the kernel default for l1-cache is loads. this actually changed in Linux 4.1 (17 Feb 2015) So your results will change based on kernel version. Annoying. Before there was a l1d-store events
- Should misses from one level of cache match accesses from the next?
  Shouldn't L1-misses be same as L2-accceses?
  Why would they not match up?
  Bug in counters, not counting stores, bug in counter

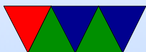


selection, other things going on in system, shared resources, chip errata, prefetching, etc. LLC actually uses offcore-response events



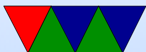
HW#5 – Notes on bzip2 behavior

- Fits well in icache.
- Why is L2 so bad? single threaded? Prefetching location?
- Why is L3 not accessed much? All of benchmark input can fit in L3 cache so once data is loaded no need to go to main memory.



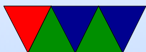
HW#5 Review – equake_l on Haswell-EP

- e-quake mem footprint 700MB
  - L1-icache = 22M/1.4T = 0%
  - L1-dcache = 33B/426B = 7%
  - L2 = 26B/61B = 39%
  - LLC = 2B/15B=14%



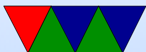
HW#5 – Notes on equake_l behavior

- Again no issues with icache
- L3 much worse than bzip2, at least in part because the working set size doesn't fit in L3



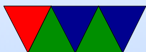
HW#5 – Ampere

- eMAG 8180, interesting history
  Not necessarily a high-end CPU
  data sheet a bit confusing
  - L1 i-cache=32 kB, 8-way?
  - l1 d-cache=32 kB, 8-way?
  - l2 = 256k shared for 2 cores
  - L3 = 32MB, slow

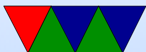


HW#5 Review – bzip2 on Ampere

- L1-icache = 307k/7B = 0%
- L1-dcache-load = 235M/8B = 3%
- L2 = 186M / 1.3B = 14%
- L3? l3c0? = 2M/27M = 9%

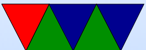


Operating System VM interactions



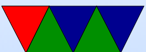
Each process has a page table

- When you context switch, simply update the hardware pointer to this
- On x86, CR3?



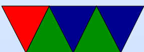
Memory Protection

- Can mark pages as read-only, execute-only, etc
  - Code might want to be read-only
  - Stack might want read/write but no-execute
  - some of data segment (const) might be read-only
- Why?

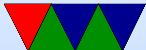


What happens on a fork?

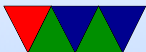
- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)
- Page table marked read-only, then shared
- Only if writes happen, take page fault, then copy made
  Copy-on-write



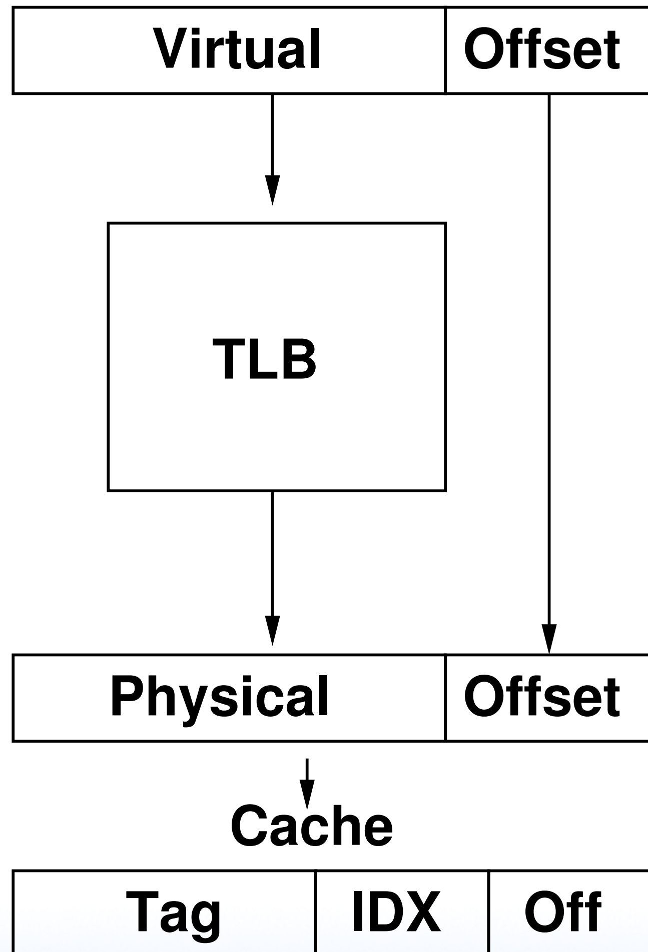
Virtual Memory – Cache Concerns

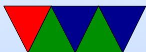


Cache Issues – Do Caches have Physical or Virtual Addresses?



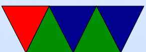
Physical Caches



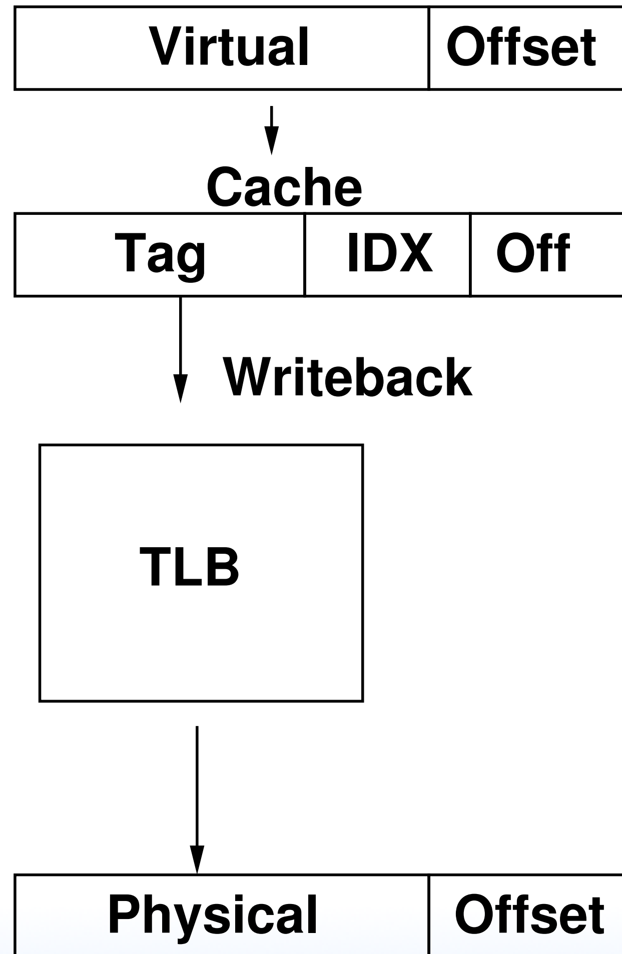


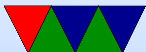
Physical Caches, PIPT

- Location in cache based on physical address
- Can be slower, as need TLB lookup for each cache access
- No need to flush cache on context switch (or ever, really)
- No need to do TLB lookup on writeback



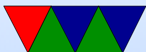
Virtual Caches





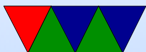
Virtual Caches

- Location in cache based on virtual address
- Faster, as no need to do TLB lookup before access
- Will have to use TLB on miss (for fill) or when writing back dirty addresses
- Cache might have extra bits to indicate permissions so TLB doesn't have to be checked on write
- Aliasing: Homonyms: Same virtual address (in multiple processes) map to different physical page
  - Must flush cache on context switch?



- Having kernel addresses high also avoids aliasing

  - How to avoid flushing? Have a process-id (ASID). Can also implement sharing this way, by both processes mapping to same virt address.
  - Having kernel addresses high also avoids aliasing
- Aliasing: Synonyms: Phys address has two virtual mappings
  - Operating system might use page or cache coloring
- Operating system has to do more work.



VIPT

Virtual | Offset

index

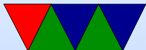
Cache

tags

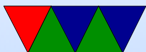
TLB

Physical

compare

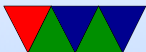


- Cache lookup and TLB lookup in parallel. Cache size + associativity must be less than page size.
- If properly sized (so that the page offset fits completely in the index) then index bits are the same for virt and physical.
- If not sized, the extra index bits need to be stored in the cache so they can be passed along with the tag when doing a lookup
- No need to flush or track ASID on context switch



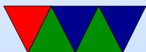
Combinations

- PIPT – older systems. Slow, as must be translated (go through TLB) for every cache access (don't know index or tag until after lookup)
- VIVT – fast. Do not need to consult TLB to find data in cache.
- VIPT – ARM L1/L2. Faster, cache line can be looked up in parallel with TLB. Needs more tag bits.
- PIVT – theoretically possible, but useless. As slow as PIPT but aliasing like VIVT.

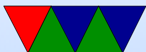


Cache Issues – Page Tables are Cached

- Page table Entries are cached too
- What happens if more memory can fit in the cache than can be covered by the TLB?
- If you have 128 TLB entries * 4kB you can cover 512kB
- If your cache is larger (say 1MB) then a simple walk through the cache will run out of TLB entries, so page lookups will happen (bringing page table data into cache) and so you do not get maximal usefulness from the cache
- This has happened in various chips over the years

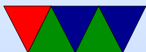


Wrap-up Summary

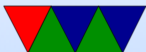


Quick run-through, the path of a load

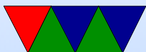
- OoO, load buffer, etc
- VIPT. So on access it looks up the physical tag in TLB while reading out the tags from each way with the index. Also keep in mind MESI is going on at this level.
- If tag from TLB matches a tag from cache, hit! Good! Cache hit!
- If tag in TLB but not in cache, cache miss.
-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

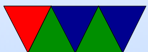


- ◦ Stack – if near top growing down, auto-grow
  - ◦ Heap – similar to stack
  - ◦ Shared page– could already be in memory (shared lib?) Just need to point to it.
  - ◦ Zeros – just have one page of zeros you can point to
  - ◦ Paged out to disk – have offset in page file, need to load it
- Time to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