ECE 571 – Advanced Microprocessor-Based Design Lecture 13

Vince Weaver

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

`vincent.weaver@maine.edu`

17 October 2019

Announcements

- HW#6 Prefetcher was due
- HW#7 will come out soon
- Project coming eventually
- There will be a midterm

HW#5 Review – Cache Sizing

- Haswell machine has a 44-bit physical address space, 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

- 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

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%

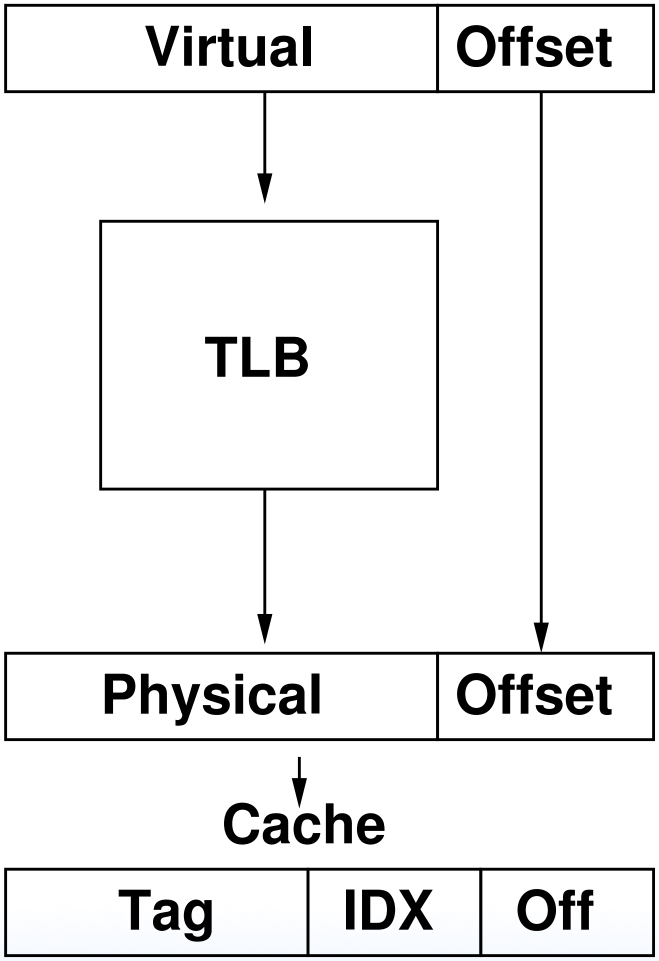
  - L2-dcache-store = 21M/308M = 6.8%
- Why did we have to use raw events? Proper Cortex-A57 event support not added until Linux 4.4. Need to update the kernel, tricky on Jetson.

Virtual Memory – Cache Concerns

Cache Issues

- 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

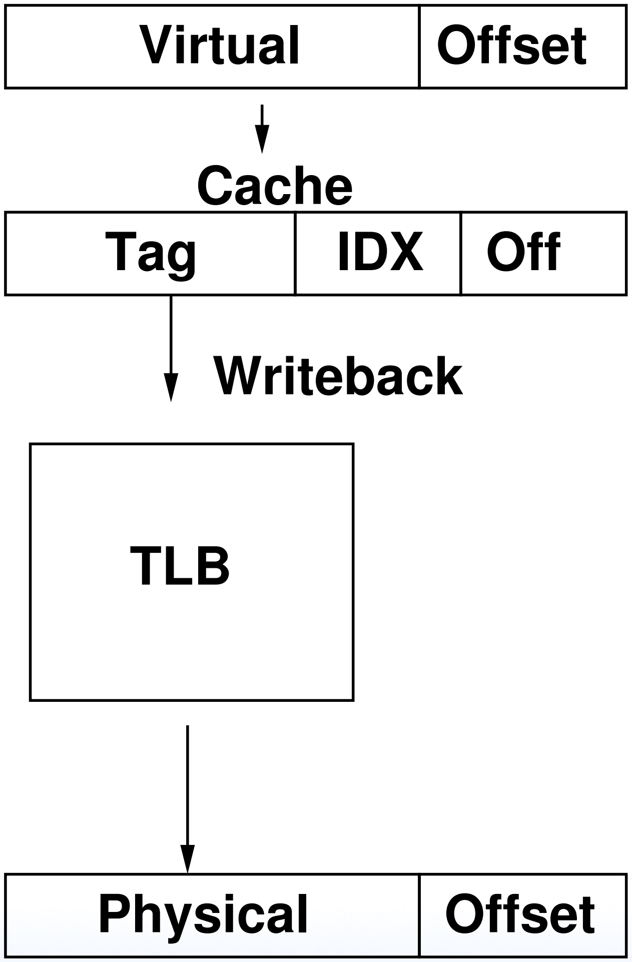
Physical Caches



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

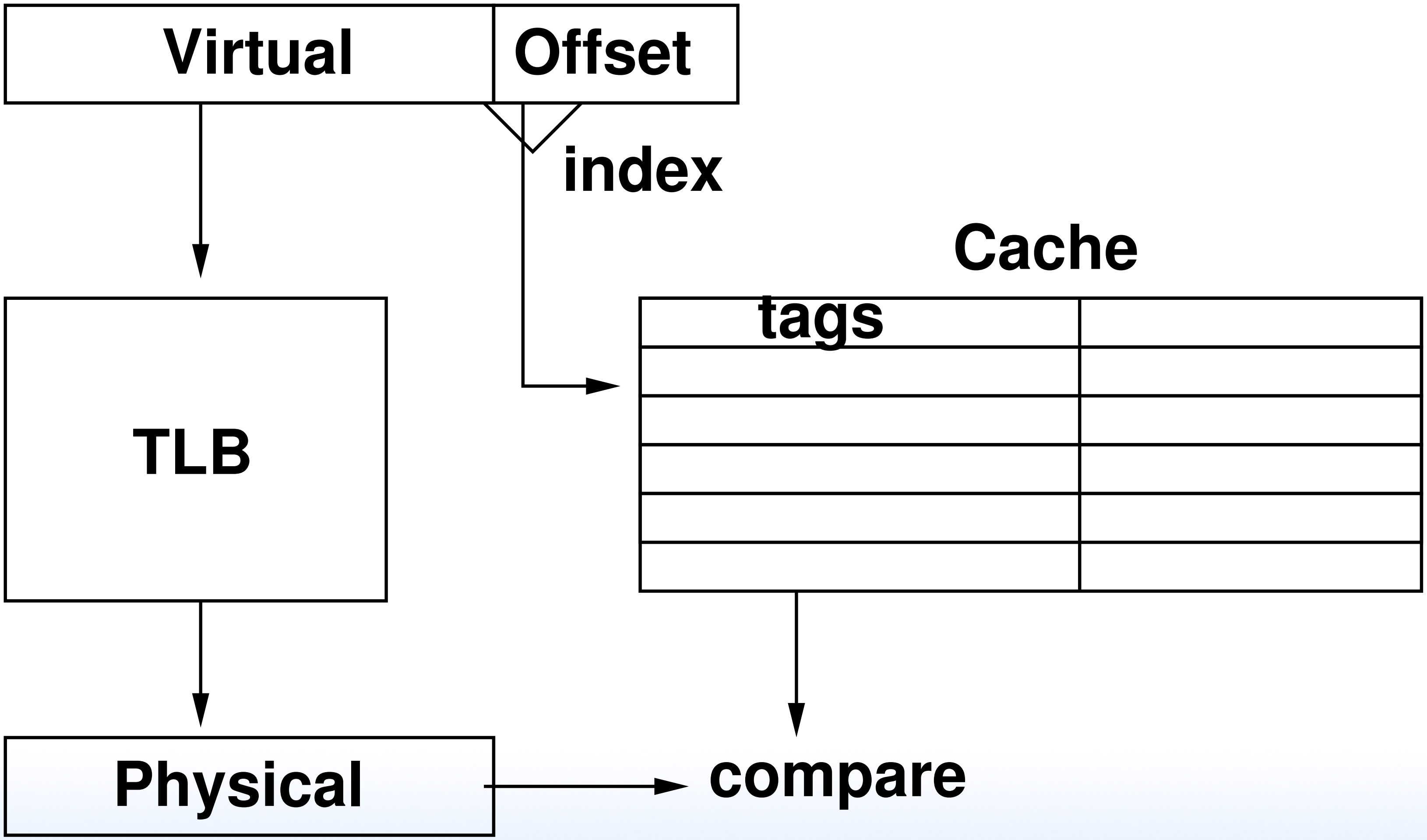


Virtual Caches

- 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?

  - 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



- 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.

Other Virtual Memory Issues

Large Pages

- Another way to avoid problems with 64-bit address space
- Larger page size (64kB? 1MB? 2MB? 2GB?)
- Less granularity. Potentially waste space
- Fewer TLB entries needed to map large data structures
- Compromise: multiple page sizes.
  Complicate O/S and hardware. OS have to find free blocks of contiguous memory when allocating large page.

- Transparent usage? Transparent Huge Pages? Alternative to making people using special interfaces to allocate.

Having Larger Physical than Virtual Address Space

- 32-bit processors cannot address more than 4GB
  x86 hit this problem a while ago, ARM just now
- Real solution is to move to 64-bit
- As a hack, can include extra bits in page tables, address more memory (though still limited to 4GB per-process)
- Linus Torvalds hates this.

- Hit an upper limit around 16-32GB because entire low 4GB of kernel addressable memory fills with page tables
- On x86 also useful because it provided more bits in PTEs for things like non-execute permissions

Virtual Machines – Shadow Page Tables

- Virtualization, provide another layer between hardware and OS
- Hypervisor lets you run multiple copies of OS, each thinking they have full control of hardware
- Internal OS have page tables, but so does the real hardware
- Various implementations to try to merge together to

avoid the double layer of abstraction when handling page tables

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

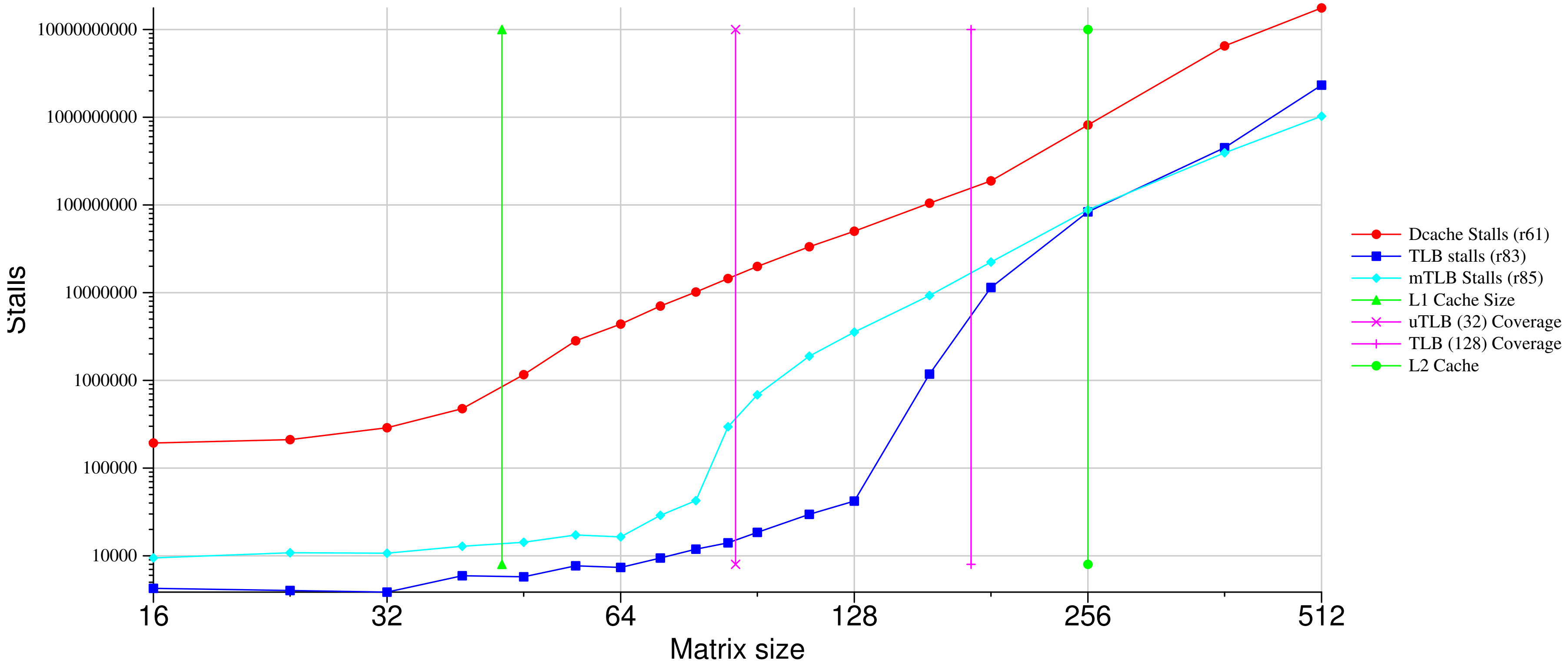
- Virtual Memory System Architecture version 7 (VMSAv7)
- Addresses can be 40bits virt / 32 physical
- First check FCSE – linear translation of bottom 32MB to arbitrary block in physical memory (optional with VMSAv7)

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

Cortex A9 TLB Measurement


Stalls
Matrix size
Dcache Stalls (r61)
TLB stalls (r83)
mTLB Stalls (r85)
L1 Cache Size
uTLB (32) Coverage
TLB (128) Coverage
L2 Cache