ECE 571 – Advanced Microprocessor-Based Design Lecture 20

Vince Weaver

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

vincent.weaver@maine.edu

23 October 2024

Announcements

- Homework #4 and #5 grades were finally sent out
- Midterm Exam Next Wednesday
  - Can bring one page of notes
  - Benchmarking, skid, power/energy, energy delay, branch predictors (static), caches, virtual memory

Homework Notes

- Cache miss rates typically reported in percent
- On the cache problem, a lot of people misread the line on the last one, it was 1 not 0

Virtual Memory – Cache Concerns

- So where do the page tables live?
- They're too big to live on the processor, they live in RAM
- The operating system allocates/sets them up when creating a process
- When the CPU walks the page tables, the accesses look just like regular load/stores
- This means page tables can end up in the cache too
- Are the page tables able to be paged out to disk?

TLB Cache Coverage

- 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 an array the size of the cache will run out of TLB entries and have to walk the page tables
- This page walk can bring page data into the cache
- This can possibly take up / kick out actual useful data from your cache, causing cache misses earlier than you

might expect

- This has happened in various chips over the years

Quick run-through, the path of a load (CPU)

- Your code does a load in assembly language, from a virtual address
- All of the out-of-order, pipelining, queueing, and such happen
- The load address request is then sent to the caches

Path of a load (caches)

- Let's assume VIPT cache
- The index is determined from the address and used to look up the tags from the cache which are physical addresses
- Simultaneously the virtual address is run through the TLB to get the physical address
- If TLB hit and tag matches the address, **CACHE HIT** and all is good
- If TLB hit but no tag match, **CACHE MISS** and just

as described in class it will try the other cache ways and eventually if all misses, go to main memory

Path of a load (TLB Miss)

- If tag not in TLB, TLB miss. Won't know if cache hit until later.
- Now let the hardware walk the page tables.
- Based on a pointer to the page tables from the OS and the virtual address from the load, the hardware walks the page tables
- If hardware finds the page, great! Return it back up to the TLB. The TLB will be updated (and like a cache might need to make room). Then carry on with checking

for cache hit/miss

- If hardware can't find the page, time to get the Operating System involved. **PAGE FAULT**

Path of a load (Page Fault)

- A page fault is a special type of interrupt, and the Operating System handles it
- For each process in the system the OS has a data structure describing what parts of virtual memory are valid and where to get the data from. These are often demand loaded on first use.
  - ○ Text (code) – read from disk
  - ○ Data (initialized)– read from disk
  - ○ BSS (uninitialized or zeros) – allocate zeros

- Stack – if near top growing down, auto-grow
- Heap – similar to stack, grow up
- Shared page, or mmap() – could already be in memory (shared lib?) Just need to point to it.
- What if page is invalid/not part of process? segfault!
- What if page has been paged out to disk? OS has to bring it back (TODO: is this a page fault? how does the OS record this info)

Path of a load (Bringing in a Page)

- Time to bring in the page!
- Need to find room in Physical RAM (OS has routines for this)
- If no room, need to make room
    - Often "page" to disk
    - Has LRU/dirty bits like cache to try to only page out oldest pages
    - If page came from disk originally (like code from executable) might not need to write to page file, can

possibly just reload it from source if needed again

- What kind of issues come up when low on RAM and constantly paging same pages in and out (thrashing?)

Path of a load (Finishing a Page Fault)

- Page now in physical RAM, time to go backwards
- OS updates the page table to point to new memory
- OS returns from exception
- Typically processor re-executes the faulting instruction, with the hope this time it is found in the page table and everything is good
- 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?

Copy on Write

- Can a physical page have multiple virtual addresses pointing to it (yes, shared memory, libraries, etc)
- Copy on write – for fork and others, when copying pages, instead of copying the page can just mark page read-only and then have multiple virtual addresses point to it. Much faster than copying memory
- If a process tries to write to page, will cause page fault and the code can notice this and actually make the copy and unshare that copy

- Zeros – for pages of zeros, could you get away with only one read-only zeroed out page for entire system?

Quick run-through, the path of a store

- Is it much different?

Other Virtual Memory Issues

Reusing un-used bits at top of Virtual Address

- People use them, causes problems later.
  See M68k/MacOS, IBM 390, ARM26
- AMD64 canonical addresses to avoid this (top bits have to be all zeros or all ones)
- Though recent systems support this, have a special mode to "ignore" top bits
  - Memory Tagging Extension (MTE) on ARM64
  - Top Byte Ignore on ARM64

- Upper Address Ignore (UAI) on AMD64
- Linear Address Mask (LAM) on Intel

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

Large Pages – Modern Systems

- Alpha and maybe some MIPS supported 8k pages?
- Modern ARM64 has been pushing for 16k pages (class project?)
- Intel by default only options are 4k, 2M, or 1G which isn't as useful

Transparent Huge Pages

- Can you use large pages (2M) for large allocations?
- Can you transparently merge many small pages to large, and vice-versa?
- Complicate O/S and hardware.
- Big problem is with fragmentation, 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)
- Intel: PAE (Physical Address Extension)
- 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

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

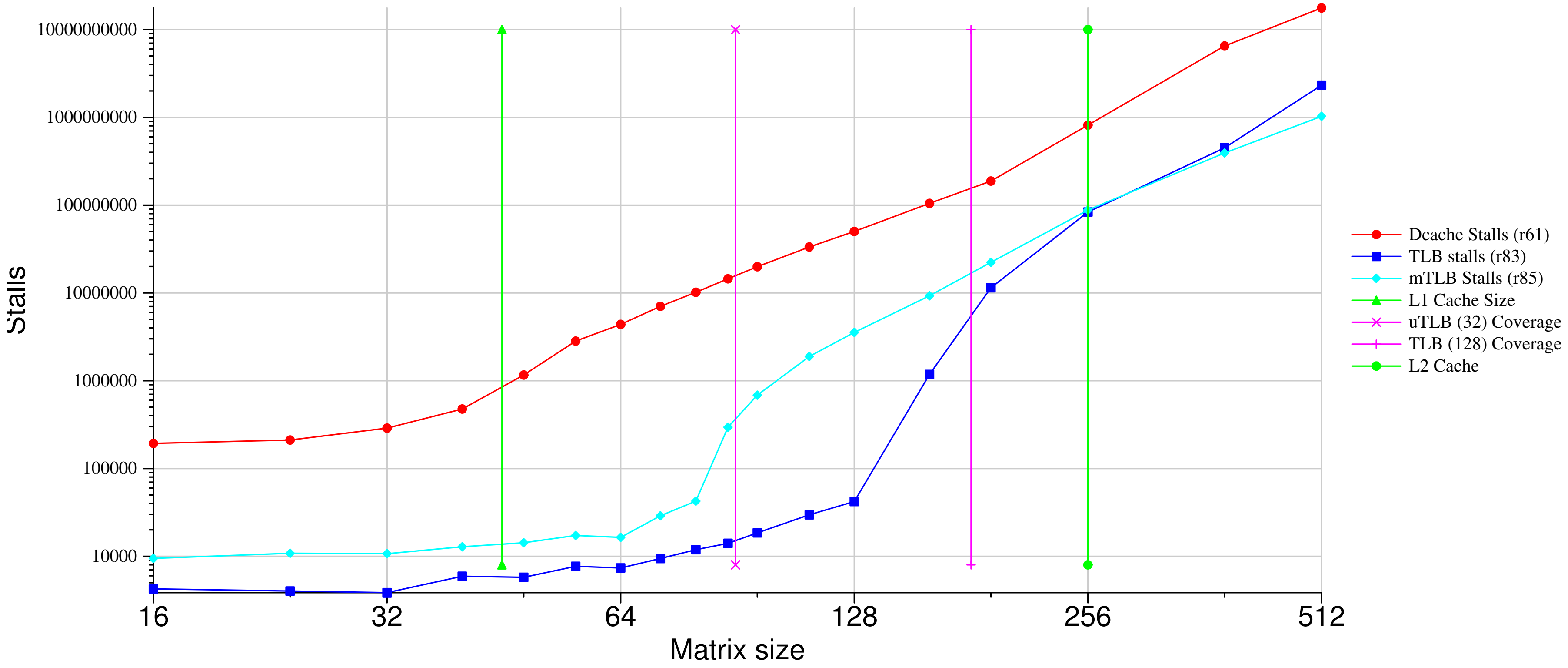
- First check FCSE – linear translation of bottom 32MB to arbitrary block in physical memory (optional with VMSAv7)

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