ECE 571 – Advanced Microprocessor-Based Design Lecture 15

Vince Weaver

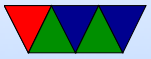
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

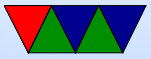
20 October 2022

Announcements

- Homework #6 will finally be posted
- Homework #4 and #5 grades were finally sent out
- Midterm Exam Next Thursday
  - Can bring one page of notes
  - Benchmarking, skid, power/energy, energy delay, branch predictors (static), caches, virtual memory

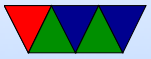


Virtual Memory – Cache Concerns



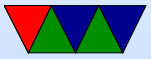
Cache Issues

- 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

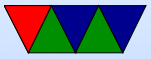


Quick run-through, the path of a load

- 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.

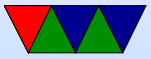


- Now let the hardware walk the page tables.
- If hardware finds the page, great! Return it back up to the TLB
- If hardware can't find the page, time to get the Operating System involved. Page fault.
- Hardware has a list of what should be in memory where (from the executable). Typically these are demand-loaded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



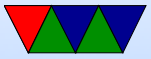
- Stack – if near top growing down, auto-grow
- Heap – similar to stack
- Shared page– could already be in memory (shared lib?) Just need to point to it.
- Zeros – just have one page of zeros you can point to
- Paged out to disk – have offset in page file, need to load it
- What if page is invalid/not part of process? segfault!

- 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).

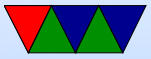


What kind of issues come up when low on RAM and constantly paging same pages in and out (thrashing?)

- 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

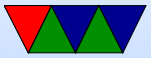


can you handle?

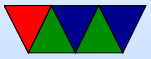


Quick run-through, the path of a store

- Is it much different?

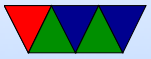


Other Virtual Memory Issues

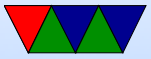


Aside: what if you have unused bits at top?

- People use them, causes problems later.
  See M68k/MacOS, IBM 390
- 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?
  - Upper Address Ignore (UAI) on AMD64

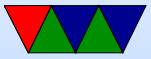


- Linear Address Mask (LAM) on Intel

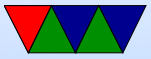


Large Pages

- 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
- 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

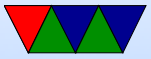


to allocate.



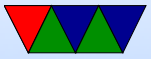
Having Larger Physical than Virtual Address Space

- 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)
- Intel: PAE (Physical Address Extension)
- Linus Torvalds hates this.
- Hit an upper limit around 16-32GB because entire low



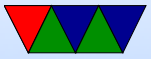
4GB of kernel addressable memory fills with page tables

- On x86 also useful because it provided more bits in PTEs for things like non-execute permissions

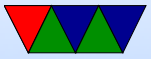


Virtual Machines – Shadow Page Tables

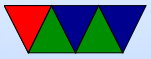
- Virtualization, provide another layer between hardware and OS
- Hypervisor lets you run multiple copies of OS, each thinking they have full control of hardware
- Internal OS have page tables, but so does the real hardware
- Various implementations to try to merge together to



avoid the double layer of abstraction when handling page tables

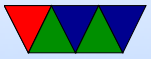


Real World Examples



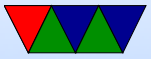
Haswell Virtual Memory

- 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

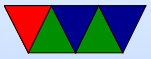


Cortex A9 MMU

- 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)

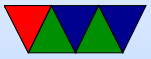


- data micro-TLB (32 fully associative)
- Unified main TLB, 2-way, 2x64 (128 total) on pandaboard
- 4 lockable entries (why want to do that?)
- Supports hardware page table walks



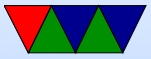
Cortex A9 MMU

- 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)

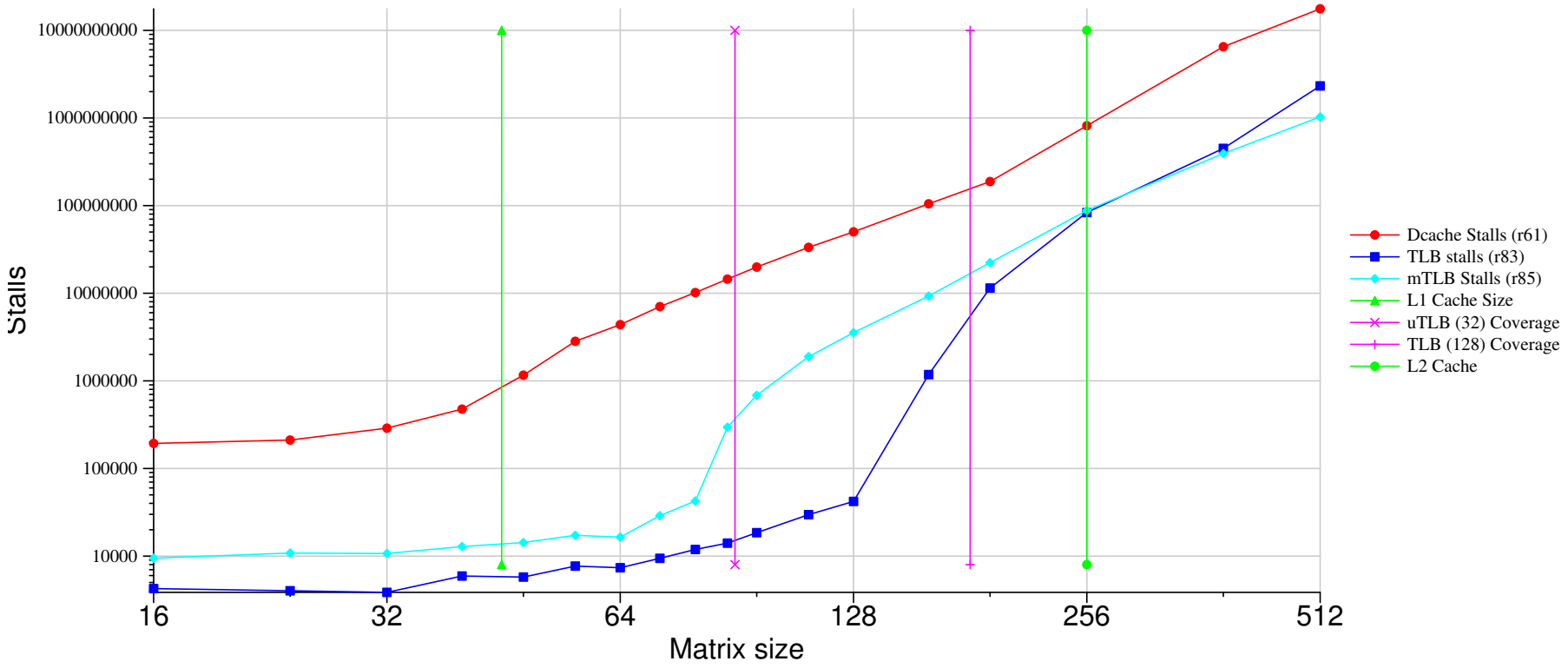


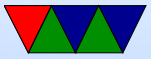
Cortex A9 TLB

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



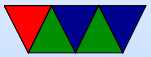
Cortex A9 TLB Measurement





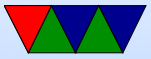
Computer Architecture Security

- We've made fast chips, but at what cost?



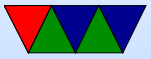
Side Channel Attacks

- 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



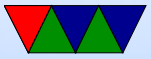
Meltdown

- `https://meltdownattack.com/`
- Problem: speculative execution can load data into caches even if you don't have VM permissions to do so
- Many OSes map kernel into VM address space of each process (this speeds up operating system calls a lot)
  Kernel often includes a full mapping of physical RAM as well
  so in theory you can read out *all* of memory
- This primarily an Intel bug, all chips with speculative



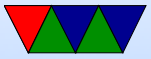
execution, dating back to Pentium Pro?

- Some very high-end ARM (Cortex A75) too as well as IBM Power? *Not* AMD though.



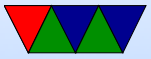
Cache Side-Channel Attacks

- Just loading speculative vaues into cache shouldn't be a problem as you can't actually read the values
- UNLESS there is some sort of side-channel
- If you can use the speculative data as an index to a memory load, you can bring yet another cache line in, the line depending on the value you shouldn't know.
- If you can determine if these lines are in cache you can know the content of the data.

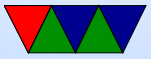


Determining if a Cache line is in Cache

- 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



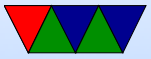
- Reload and time it, can tell if someone else had reloaded it in the meantime by how fast it loads



Meltdown – Toy Code

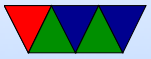
```
raise_exception()
access(probe_array[data*4096])
```

- 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,



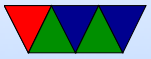
and you can probe to see if it did

- 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.



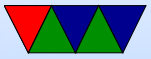
Performance

- Up to 500K/s read out



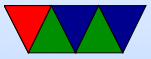
Meltdown – Issues

- Exception Handling – either a signal handler, or else forking a child to cause the exception
- Exception suppression transactional memory
- Limitation: Can get false zeros. Repeat until sure.



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
  - KASLR (Kernel Address Space Layout Randomization). If want arbitrarily read out kernel info need to find kernel

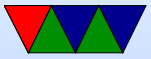


Turns out that with 40 bit address space and large physical memory (8GB) it doesn't take too many tries to find kernel

- KAISER (KPTI) – map kernel in separate address space.

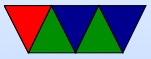
  Large overhead on switch in/out of kernel (syscalls, context switch)

  Up to 30% on someworkloads, almost none on others
- PCID (intel's ASID implementation on Westmere or newer) helps avoid complete TLB flush



Spectre Vulnerability

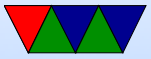
- Unlike Meltdown, pretty much *any* processor with speculative execution affected
- Doesn't leak info from kernel, but from one part of program to another
- Why a problem? Well if javascript can read anything in rest of browser (passwords, history, etc)
- SPECulative execution, will haunt us for ages



Spectre Variant 1 – Bounds Check

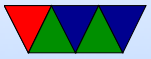
```
if (x<array1_size)
        y = array2[array1[x]*256];
```

- Ideally finds this code already existing in user code
- If mispredicts the check, will speculatively access the out-of-bounds value
- Attacker controls X
- Attacker trains the branch predictor that value is true with lots of runs
- Then passes in a value that is wrong but branch is



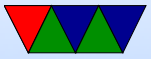
predicted the previous way.

- array1_size is not cached, so it stalls and execution goes beyond
- Probe the cache much like meltdown



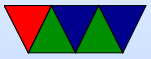
Indirect Branches

- Instead of relying on user code, train up the BTB
- Doesn't have to be the same address space, just has to alias in the BTB
- On many machines only 30 or fewer bits of BTB used to index

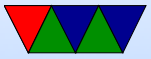


Spectre Variant 2 – Branch Target Injection

- Note for next year: review how this works
- X maliciously chosen
- Branch prediction manipulated to predict wrong
- arrays all kicked out of memory
- array1size was kicked out of RAM, so cache miss and slowly get value for RAM
- meanwhile predicts branch is good and so fetches array2[k*256]
- Eventually figures out and squashes wrong branch, but

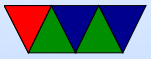


the fetch already underway into cache



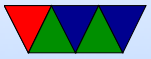
Finding a gadget

- Need to find code that runs with adversarial values are in register
- Not hard, often unused values leak across function calls (if a function doesn't use them)
- Need to find way to trigger a branch in a way that acts on these as pointers.
- Then find existing indirect jump
- Train the BTB to want to jump to our gadget
- clear out cache, perform attack



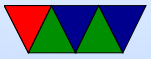
Notes

- some i7 up to 188 instructions can execute speculatively between
- can be triggered from Javascript. No clflush, but can evict all of cache by reading through an array.
- branch predictors on cpus are independent?



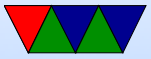
Workarounds

- Software
  - Disabe hires timers in javascript
  - Memory barriers – can halt speculation with special instructions, but have to insert them all through code where it might be an issue.
  - Kaiser/KPTI not help
  - Retpoline and IBRS, see next slides



Mitigation: New barriers

- Added by Intel with firmware update, new MSR
- IBRS – indirect branch restricted speculation
  flush branch predictor on entry to kernel, disable brpred on hyperthread
- STIBP – single-thread indirect branch prediction – disable brpred on sibling thread (currently they share brpred)
- IBPB – indirect br pred barrier – flush branch predictor state



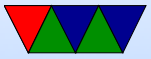
Mitigation: retpoline

- Indirect branches can be used for spectre attack
- Can you (at a performance cost) change all indirect branches to disable branch prediction?
- Original indirect call

```
jmp     *%r11
```

- Replace with this:

```
        call    set_up_target   ; skip ah
capture_spec:
        pause; lfence           ; trap to
```



```
        jmp     capture_spec      ; can't s

set_up_target:
        mov     %r11, (%rsp)      ; overwri
        ret                       ; call us
```

- Return trampoline
- Convert indirect branch into a `ret` in a common location, makes it hard to train branch predictor.
- Also adds a code-trap so that if code speculates past the branch it gets trapped in an infinite loop
- Downside: all indirect branches now slower retpoline.