ECE 571 – Advanced Microprocessor-Based Design Lecture 14

Vince Weaver

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

`vincent.weaver@maine.edu`

3 March 2016

Announcements

- Spring break, no HW over break
- Branch predictor competition
  `http://www.jilp.org/cbp2016/`

HW#6 Review?

- Should go over floating-point vs integer benchmarks more. Bzip2 vs equake

-
  ```
  objdump --disassemble-all ./bzip2 | grep prefetch
  (nothing)
  objdump --disassemble-all ./bzip2.swprefetch | grep prefetch
  ./bzip2.swprefetch:     file format elf64-x86-64
    401397: 0f 18 0b                prefetcht0 (%rbx)
    401e8c: 0f 18 88 a0 00 00 00  prefetcht0 0xa0(%rax)
  ... * 20 lines
  ```

-
  ```
  objdump --disassemble-all ./equake_l | grep prefetc
  (nothing)
  ```

```
objdump --disassemble-all ./equake_l.swprefetch | grep prefetch
./equake_l.swprefetch:     file format elf64-x86-64
  40192a: 0f 18 0a                prefetcht0 (%rdx)
  401b8d: 0f 18 4d 48             prefetcht0 0x48(%rbp)
  401be8: 0f 18 4d 48             prefetcht0 0x48(%rbp)
  402036: 0f 18 09                prefetcht0 (%rcx)
  40479a: 0f 18 4d 00             prefetcht0 0x0(%rbp)
  4047a9: 0f 18 0b                prefetcht0 (%rbx)
```

- BZIP

		l2-cache-misses	prefetches	time
1a:	bzip2:	33%	165M	3.4s
2a:	SW prefetch:	33%	165M	3.4s
5a:	HWdisable	44%	155k	3.5s
5a:	HWdisable+Sw	44%	151k	3.4s

- Equake

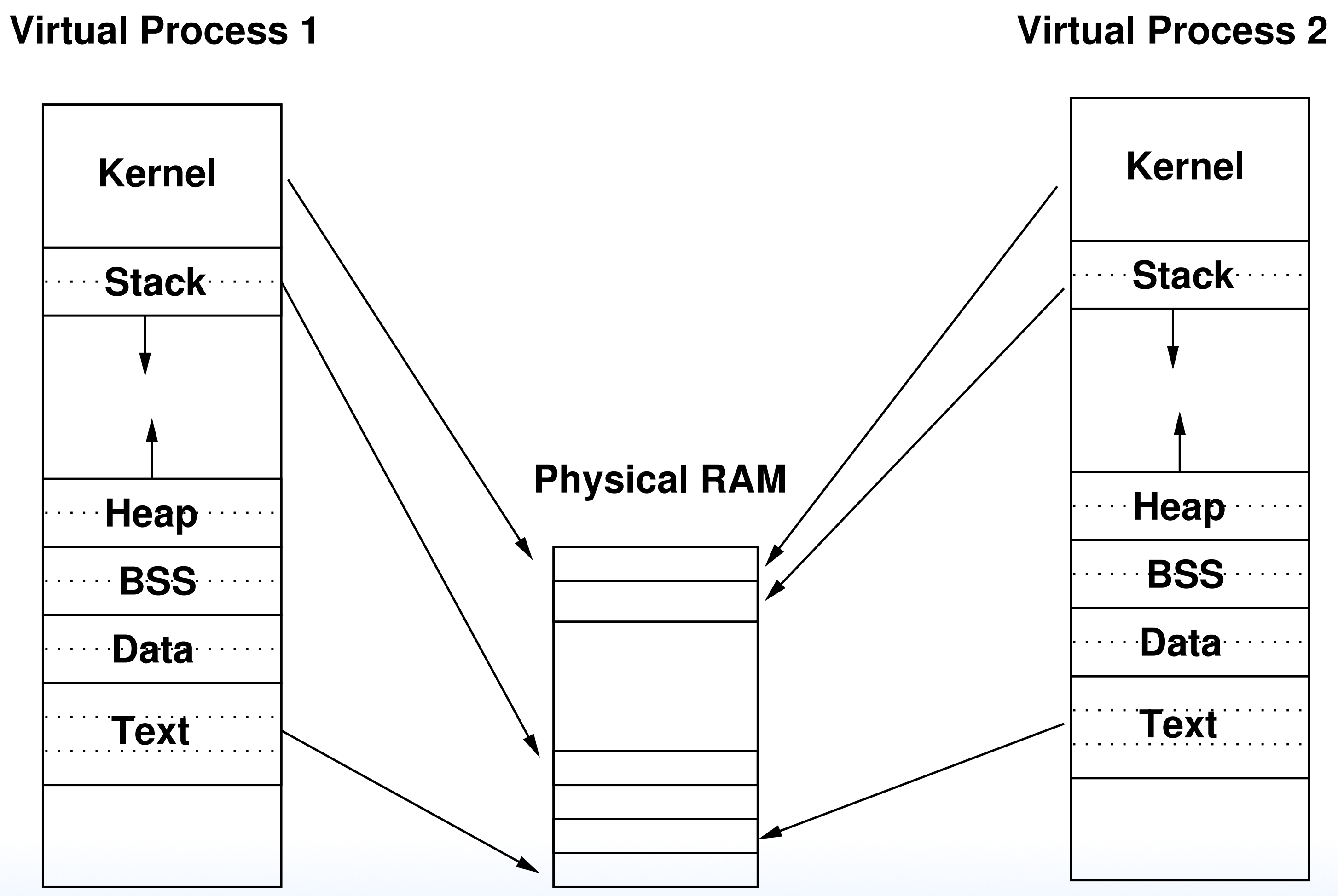
		l2-cache-misses	prefetches	time
3a:	equake_l:	15%	37B	138s
4a:	equale_l swpref	16%	37B	134s
5a:	hwdisable	68%	3M	160s
5a:	hwdisable swpref	68%	3M	160s

- Summary: disabling prefetch hurt, dramatically so on equake.
  Unclear what exactly the prefetch perf counter is measuring
  Enabling SW prefetch does not seem to do much, even with HW prefetch disabled.

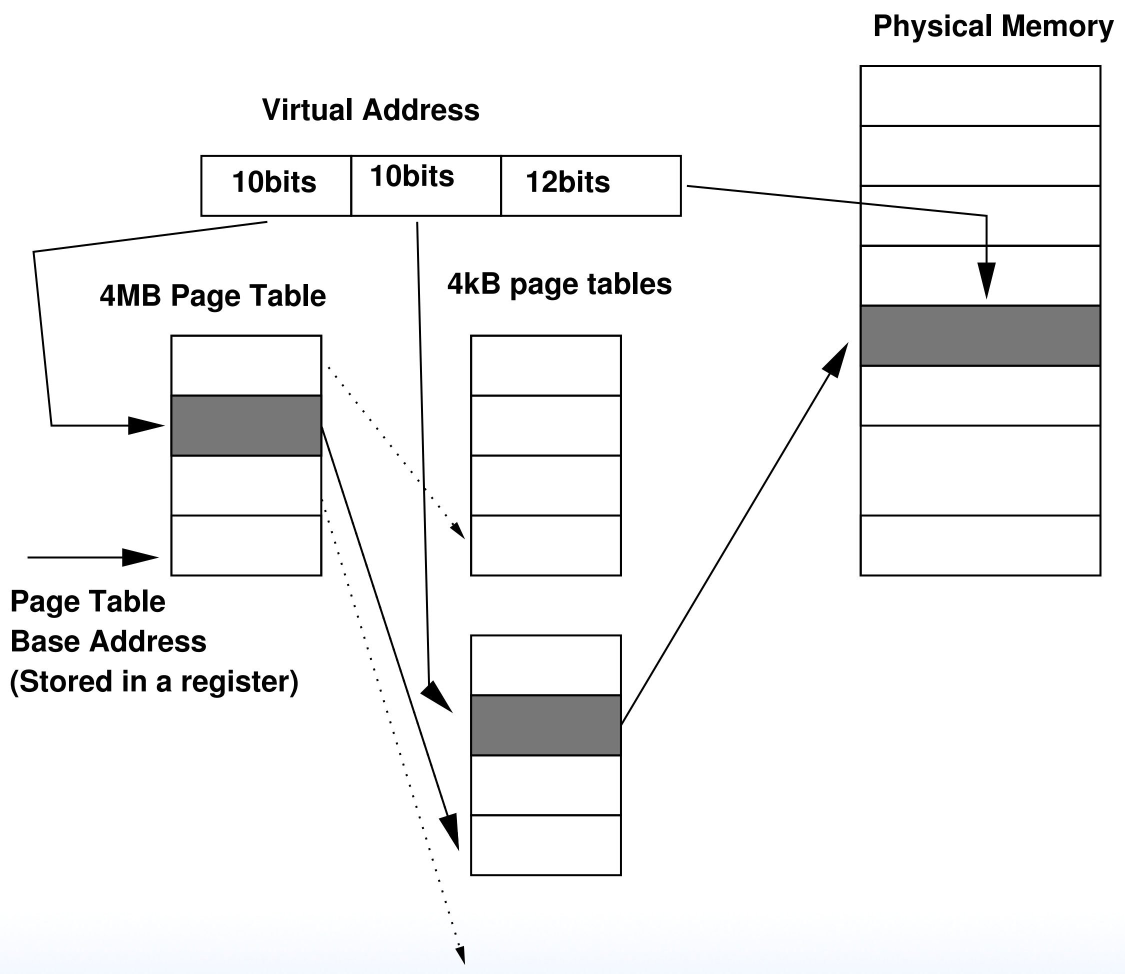
- Why? Lots of possible reasons. compiler bug. hardware bug. hardware engineers not enable SW prefetch (is it incorrect to ignore?) other.

Virtual Memory again

Diagram


Virtual Process 1
Virtual Process 2
Kernel
Stack
Heap
BSS
Data
Text
Physical RAM
Kernel
Stack
Heap
BSS
Data
Text

Hierarchical Page Table Diagram



Walking the Page Table

- Can be walked in Hardware or Software
- Hardware is more common
- Early RISC machines would do it in Software. Can be slow. Has complications: what if the page-walking code was swapped out?

TLB

- Translation Lookaside Buffer
  (Lookaside Buffer is an obsolete term meaning cache)
- Caches page tables
- Much faster than doing a page-table walk.
- Historically fully associative, recently multi-level multi-way
- TLB shootdown – when change a setting on a mapping

and TLB invalidated on all other processors

Flushing the TLB

- May need to do this on context switch if doesn't store ASID or ASIDs run out.
- Sometimes called a "TLB Shootdown"
- Hurts performance as the TLB gradually refills
- Avoiding this is why the top part is mapped to kernel under Linux

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

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

What happens on a page fault

- Walk the page table and see if the page is valid and there

- "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

What happens on a fork?

- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)

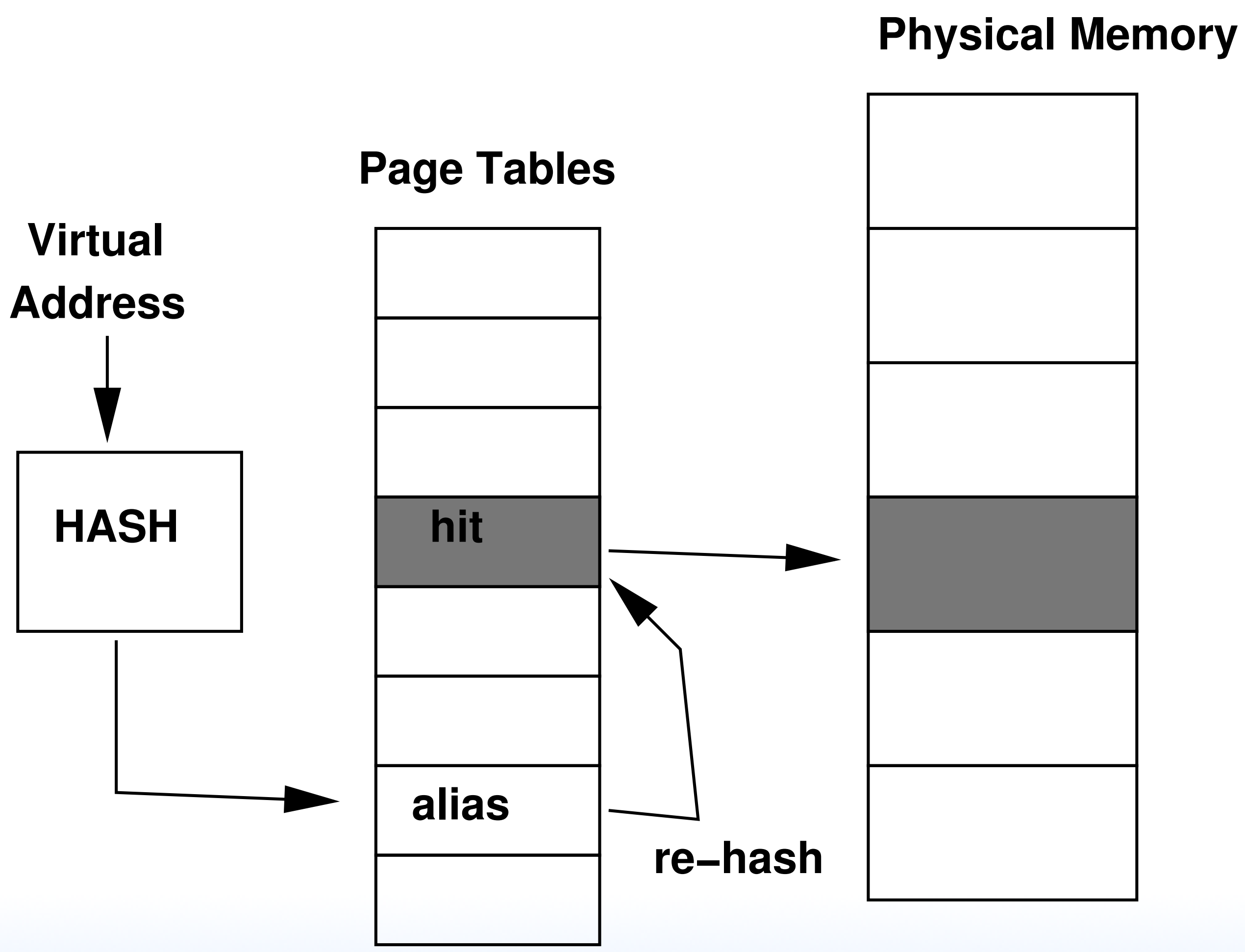
- Page table marked read-only, then shared

- Only if writes happen, take page fault, then copy made
  Copy-on-write

Inverted Page Table

- How to handle larger 64-bit address spaces?
- Can add more levels of page tables (4? 5?) but that becomes very slow
- Can use hash to find page. Better best case performance, can perform poorly if hash algorithm has lots of aliasing.

Inverted Page Table Diagram



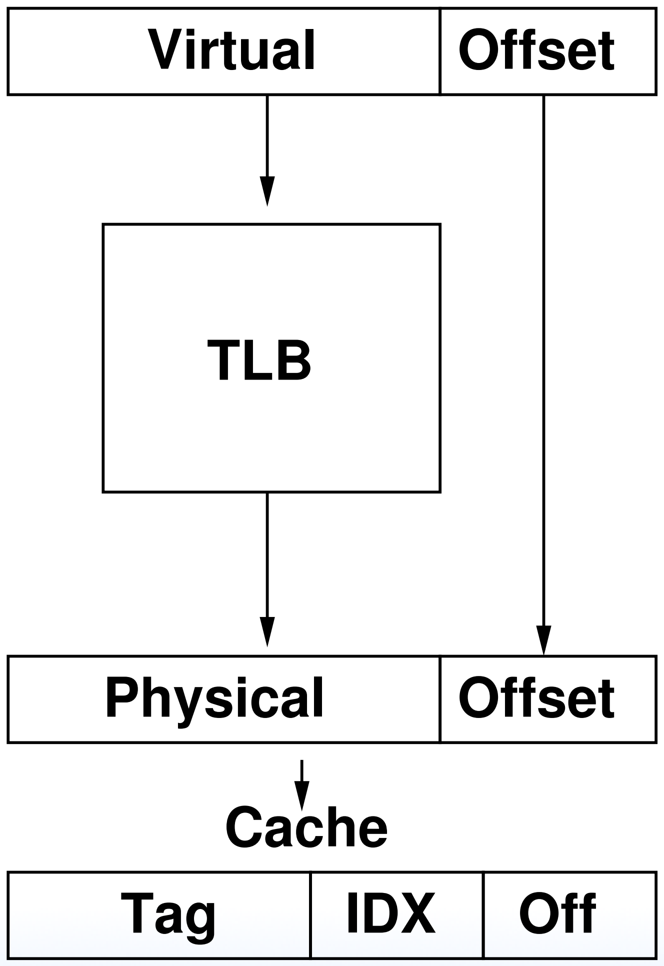
Virtual Memory – Cache Concerns

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

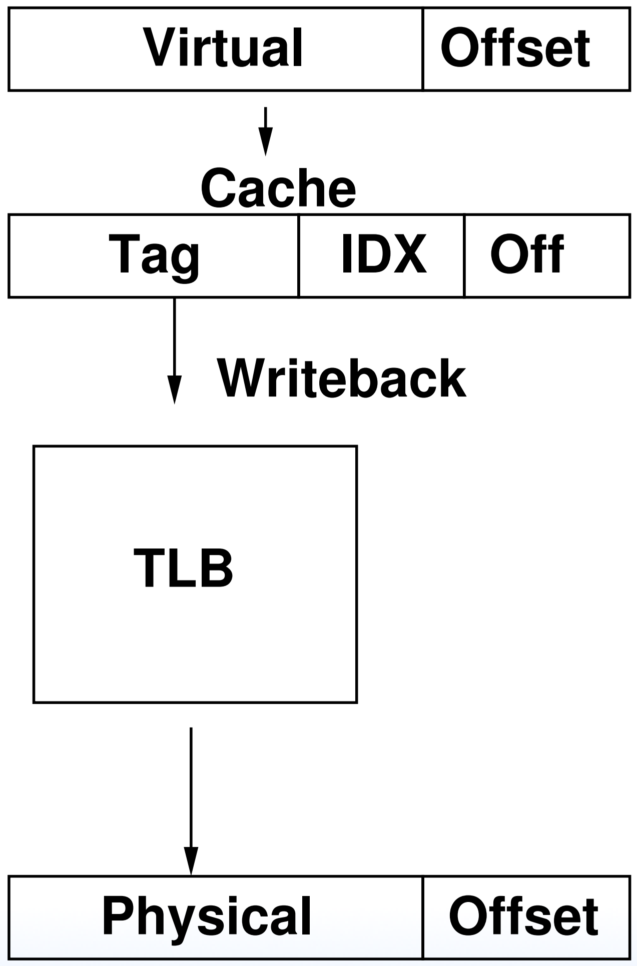


Physical Caches

- 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
- If properly sized, the index bits are the same for virt and physical. In this case no need to do TLB lookup on cache hit.

- 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

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
- Can have aliasing issues when processes use same virtual

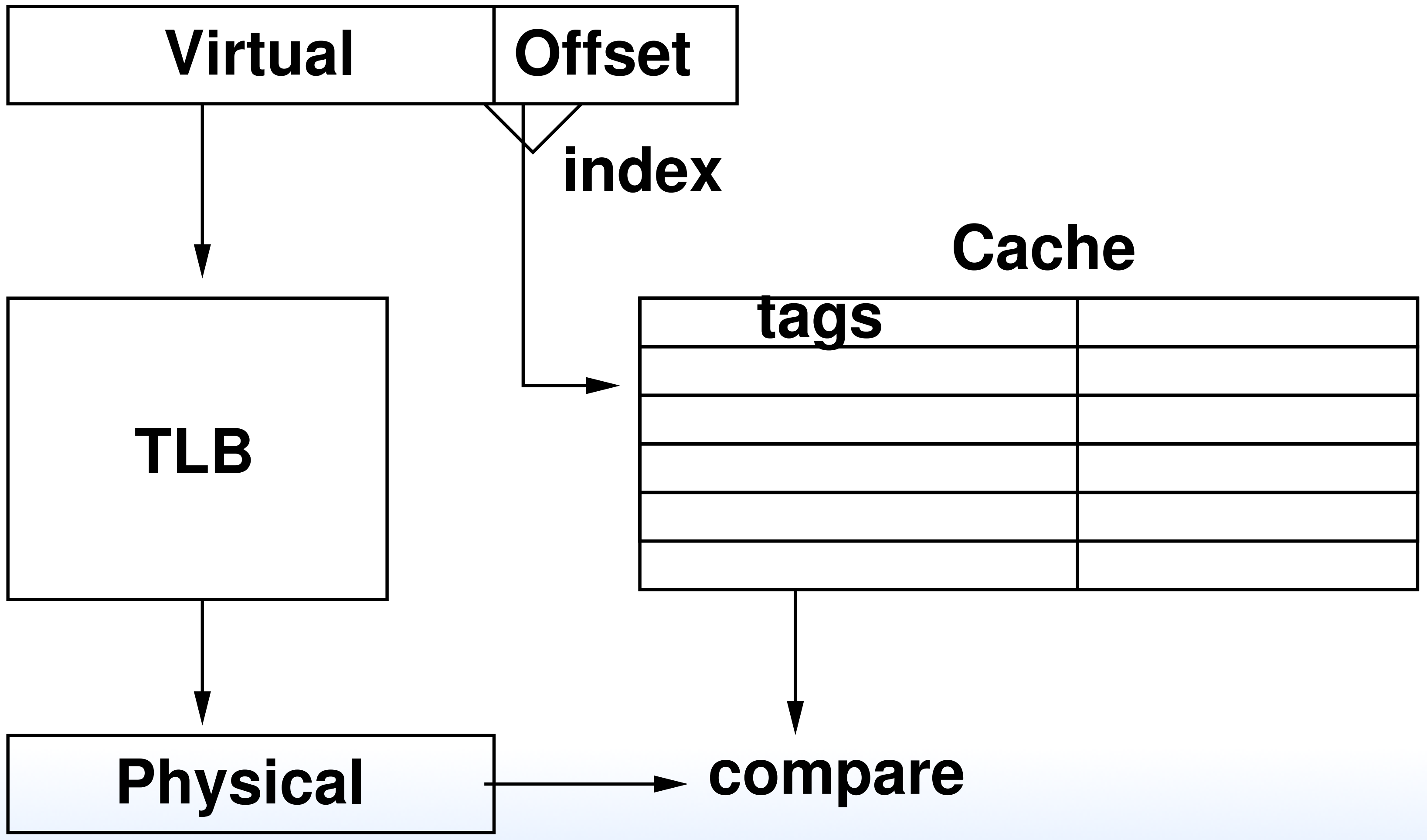
addresses.
Flush cache on context switch?

- How to avoid flushing? Have a process-id. Can also implement sharing this way, by both processes mapping to same virt address.

- Having kernel addresses high also avoids aliasing

- Operating system has to do more work

VIPT



Cache lookup and TLB lookup in parallel. Cache size $+$ associativity must be less than page size.

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.

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.

Haswell Virtual Memory

- L1 (4-way associative)
  - 64 4kB
  - 32 2MB
  - 4 1GB
- L2 (1024 entry 8-way associative, combined 4kB and 2M)
- DCache – 32kB/8-way so VIPT possible

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

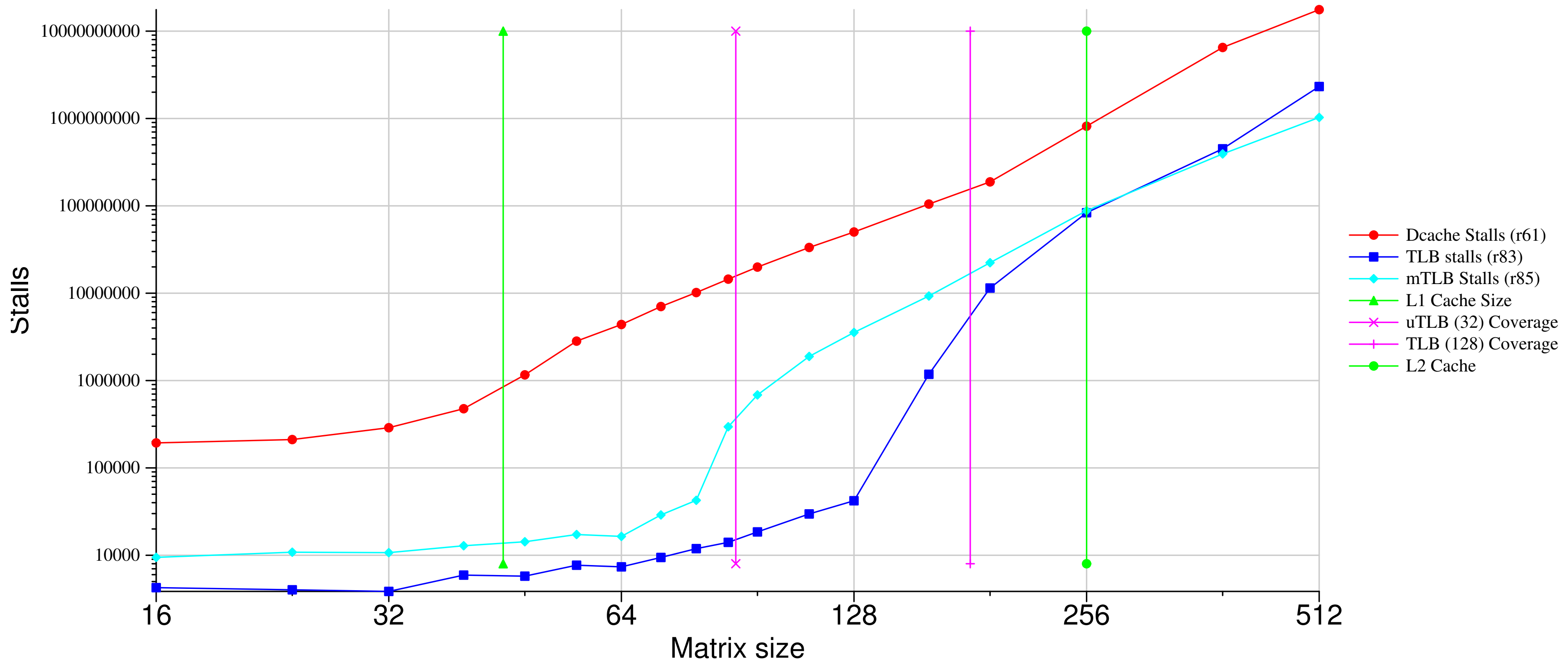
Cortex A9 MMU

- 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



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