Virtual Memory II

Today’s Menu:
- Remaining Virtual Memory Issues
  - Accelerating size of address translation:
    - Translation Lookaside Buffer (TLB)
  - Page Table Size and Multi-level Page Tables
  - Cache and TLB interactions
    - Physical Caches vs Virtual Caches

Memory Hierarchy of a Modern Computer System

By taking advantage of the principle of locality:
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.

TLB Structure & Performance

- TLB structure
  - 32 to 1024 entries (slots)
  - Can be direct mapped, set associative, fully associative
  - Easier to be fully associate here, since TLBs are often pretty small

- TLB miss cost
  - 20 to 500 cycles (much longer if the page is on disk)
  - Hardware and software based

- TLB miss rates
  - 4% to 8% for typical Unix workloads
  - Can be much higher for large applications
  - Operating systems can significantly influence the TLB miss rate

- Page size
  - 4K or 8K Bytes
  - Often support multiple page sizes
  - up to 2GB(!), 2MB is becoming more widely used (“large pages”)

Next Issue: Page Table Size

- Previously determined: Linux page table size
  - 4K page
  - 32 bit virtual address space
  - 22 bits for the page number → 22 virtual pages
  - $2^{22} = 4,194,304$ PTEs
  - Each PTE is at least 4 bytes
  - 1 Million PTEs * 4 bytes = 4 Megabytes

- Hey: you said each process has its own page table!
  - 160 processes = 640 MB of memory for page tables!!!
  - If the page table is not in memory, you can’t find it
  - The page table cannot be swapped out!

- What’s the deal?

Solution: Multi-level page tables

- Example: from Linux
  - Have two levels of page tables
  - Each of these is actually one page (4K) in size!

Only this table needs to stay in memory at all times.
Multi-level Page Tables

<table>
<thead>
<tr>
<th>VPN 1</th>
<th>VPN 2</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 Table (Page Directory)</td>
<td>Level 2 Table (Page Table)</td>
<td>PPN</td>
</tr>
</tbody>
</table>

Multi-level Advantages

- Minimizes page: 4KB+4KB table
- Allows for maximum: 4KB + 4MB
- Page table size scales with memory usage
- Only level 1 MUST be in real memory
  - Level 2 tables can be swapped just like any other page
- Multi-level and TLB integration
  - TLB only stores Level 2 PTEs
  - Don't have to do two TLB accesses to do VA -> PA translation

Alternative Implementation: Reverse Page Tables

- Already implemented as part of OS data structure
  - Identify physical pages for eviction/replacement
  - Need a table of physical page indices
- Hash VPN into a PPN index in the reverse table
  - Effectiveness depends on how well the hash works

Next Issue: Cache & TLBs, How They Interact

- We do memory hierarchies to hide unpleasant facts
  - We don't have as much fast memory as we would ideally like. Solution?
    - Cache hierarchy gives illusion of speed—most of the time. Occasionally it's slow.
  - We don't have as much memory as we would like. Solution?
    - VM hierarchy gives the illusion of size—most of the time. Occasionally it's slow.
- Roughly put: We do cache for speed. We do VM for size.
- So, we have "regular" cache for fast access, and we have a TLB for fast translation VA->PA.
  - How do they interact? They must interact somehow...
  - Do we wait for VA->PA translation before looking in the cache?
  - Is the cache full of virtual or physical addresses?

Simplest Scheme is Sequential: TLB then Cache

- Slowest, but simplest
  1. CPU sends out virtual address
  2. TLB translates to physical, or page faults and we wait for page to load
  3. On TLB hit, translated physical address sent to cache
  4. Cache lookup gives data access fast, or...
  5. Cache miss goes to main mem

Address Translation/Cache Lookup

Simple!

- Virtual Address
  - CPU sends out virtual address
  - TLB translates to physical, or page faults and we wait for page to load
  - On TLB hit, translated physical address sent to cache
  - Cache lookup gives data access fast, or...
  - Cache miss goes to main mem

Slow because you have to wait for the translation before you can check the cache.
**TLBs and Caches: Basic Flow for Access**

- Virtual Address to Physical Address
  - TLB lookup
  - Cache lookup

**Protection and the TLB**

- A process presents a Virtual Address to the TLB for translation
- Ex: either process could present virtual address 0x2000
- How does the TLB know which process’s VA-PA mapping is held in the TLB?
  - There are 2 separate VM address spaces here, one per process
  - There is only 1 TLB, NOT one per process

**Real Example: DECstation 3100**

```
<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>
```

**Protection and the TLB (cont.)**

- Many machines append a PID (process ID) to each TLB entry
- OS maintains a Process ID Register (updated during the switch between processes, called a context switch)
- Another solution is to flush the TLB on a context switch; flush means “empty it”

**Speed & Timing Impacts**

- If we do these accesses sequentially, big impact on speed
  - You have to do lookup in the TLB...
  - ...then you have to do lookup in the cache
  - Involves a lot of memory access time
- One Solution: pipelining
  - Spread the accesses across stages of the pipeline
  - TLB and cache are just like any other resource in the pipeline
  - You gotta be careful to know how long they take (impacts pipe cycle time)
  - You gotta know who is trying use the resource in what pipe stage
  - You can have hazards, need stalls, need forwarding paths, etc

**Ex: MIPS R3000 Instruction Pipeline**

<table>
<thead>
<tr>
<th>Last Fetch</th>
<th>Decode</th>
<th>Reg. Read</th>
<th>ALU / E.A.</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>E.A.</td>
<td>TLB</td>
</tr>
</tbody>
</table>

**Resource Usage**

- Instruction cache
- Data cache
- TLB
- ALU
- E.A.
- Memory
- Write register
Speeding it Up

TLB and then Cache... Why? What else could we do?

Two options
1. Overlapped cache & TLB access (in parallel)
   - What are the limitations?
2. Why does our cache use physical addresses?
   - Could it store virtual addresses?
   - What are the problems/considerations?

Overlapped Cache & TLB Access

1. Overlapped cache & TLB access (in parallel)
   - What are the limitations?
2. Why does our cache use physical addresses?
   - Could it store virtual addresses?
   - What are the problems/considerations?

How Overlapping Reduces Translation Time

- Basic plumbing
- High order-bits of the VA are used to look in the TLB...
  - Remember: low order bits are the page offset—which byte address on the page
  - High order bits are what really changes, from virtual to physical address
- ...while low order-bits are used as index into cache
  - Remember: lowest bits are the cache line offset—which byte on the cache line
  - The "intermediate" bits are the cache index: which line in the cache should we check to see if the address we want is actually loaded into the cache mem
  - The highest order bits are the cache tag: when we look in the cache at a line of cache, we must compare these bits with the cache tag, to see if what’s stored in the cache is really the address we want, or just another line of memory that happens to map to this same place in the cache
- The "action" here is on the cache tag high order bits...

Cache vs. TLB Access

- What happens for large caches?
- Should we use VA instead?
- Should we only use VA bits to index?

Two Cache Architectures

Remember:
Cache index used to look up data in the cache
Cache tag used to verify what data is in the cache

1. Virtually-indexed Virtualy-tagged Caches
   - Also known as Virtual-Addressed or Virtual Address Caches
   - The VPN bits are used for both tags and index bits
2. Virtually-indexed Physically-tagged Caches
   - The VPN bits only contribute to the index
   - The tag is physical and requires a TLB lookup, but it can be done in parallel
Virtual Address Cache

- Lookup using VA
- TLB access on miss
- Use PA to access next level (L2)

Multiple Virtual Address Spaces (Multiprogramming)

- Load/store to VA (page 0x02)
- Is it VA from Process 1 or Process 2?
- Is it a hit? miss?

Multiple Address Space Solution

1. Keep "process id" with cache block tags
   - Upon cache lookup check both address tag and process id
2. Flush cache on context switch
   - Expensive (lose contents of the cache every switch)
3. Use single-address space OS's
   - Not practical today (requires too much compiler analysis)

Or, Only Use Virtual Bits to Index Cache

- Don’t need to wait for TLB
- Parallel TLB access (e.g., for larger caches)
- Physically-tagged but Virtually-indexed Cache
- Can distinguish addresses from different processes
- But, what if multiple processes share memory?

Virtual Address Synonyms

- P1 makes a ref to data on page 0x04
- P1 block is a miss
- But, P2 block is in
- Can’t look it up!

Virtual Address Synonyms (Cont.)

- Virtual addresses on page 0x04 of P1 are synonyms of those on page 0x00 of P2
- Synonyms are also referred to as aliases
- The page is shared among the processes
  - Example shared pages are kernel data structures
- Must avoid allowing multiple synonyms to co-exist in the cache
  - Only memory read/written must be resolved
  - Read-only memory (e.g., instructions) can exist in multiple locations
Synonym Solutions

► Avoid: Limit cache size to page size times associativity
  ▶ get index from page offset

► Avoid: Eliminate by OS convention
  ▶ single virtual space
  ▶ restrictive sharing model

► Detect: Search all sets in parallel
  ▶ 64K 4-way cache, 4K pages, search 4 sets (16 entries)

► Reduce search space: Restrict page placement in OS
  ▶ make sure index(VA) = index(PA)

Summary

► Memory access is hard and complicated!
  ▶ Speed of CPU core demands very fast memory access. We do cache hierarchy to solve this one.
  ▶ Gives illusion of speed—most of the time. Occasionally slow.

► Size of programs demands large RAM. We do VM hierarchy to solve this one.
  ▶ Gives illusion of size—most of the time. Occasionally slow.

► VM hierarchy
  ▶ Another form of cache, but now between RAM and disk.
  ▶ Atomic units of memory are pages, typically 4KB to 2MB.
  ▶ Page table serves as translation mechanism from virtual to physical address
  ▶ Page table lives in physical memory, managed by OS
  ▶ For 64b addresses, multi-level tables used, some of the table is in VM
  ▶ TLB is yet another cache—caches translated addresses, page table entries.
  ▶ Saves from having to go to physical memory to do lookup on each access
  ▶ Usually very small, managed by OS
  ▶ VM, TLB, cache have "interleaving" interactions.
  ▶ Big impacts on speed, pipelining. Big impacts on exactly where the virtual to physical mapping takes place.