Efficient Address Translation

- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table

- Cost of translation =
  Cost of TLB lookup +
  Prob(TLB miss) * cost of page table lookup
TLB and Page Table Translation

Diagram:
- Processor
- TLB
- Page Table
- Physical Memory

Flow:
- Processor sends Virtual Address to TLB.
- TLB checks for Hit.
  - Hit: Frame sent to Page Table.
  - Miss: TLB sends Virtual Address to Page Table.
- Page Table checks for Validity.
  - Valid: Frame sent to Physical Memory for offset addition.
  - Invalid: Exception raised.
- Physical Memory sends Physical Address to Data.
- Data flows back to Processor.
Translation Lookaside Buffer (TLB)
Cache Look-up: Fully Associative
Cache Lookup: Direct Mapped

Why are TLBs not direct mapped?
**TLB on**

**Cache Lookup: Set Associative**

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>value</td>
</tr>
<tr>
<td>0x0053</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>value</td>
</tr>
<tr>
<td>0x120d</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

`hash(address)`

=?

match at hash(address)?

\[\text{yes}\]

return value

---

Most TLB's are (and most caches are) set associative
Question

• What happens on a context switch?
  – Reuse TLB?
  – Discard TLB? (xk resets TLB)

• Solution: Tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process

set CR3
MIPS Address Translation

- Software-Loaded Translation lookaside buffer (TLB)
  - Cache of virtual page -> physical page translations
  - If TLB hit, physical address [As before]
  - If TLB miss, trap to kernel [New!]
  - Kernel fills TLB with translation and resumes execution

- Kernel can implement any page translation
  - Page tables
  - Multi-level page tables
  - Inverted page tables
  - ...

Question

• What is the cost of a TLB miss on a modern processor?
  – Cost of multi-level page table walk
  – MIPS: plus cost of trap handler entry/exit
Hardware Design Principle

The bigger the memory, the slower the memory
## Memory Hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache/first level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache/second level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>100 μs</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory (FLASH)</td>
<td>100 μs</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

**TLB**
- 100 entries
- 1000 entries

**NVRAM**
- 10 ms, 10 GB

**Size**
- 64 KB
- 256 KB
- 2 MB
- 10 GB
- 100 GB
- 1 TB
- 100 PB
- 1 XB
- 100 XB

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core
Question

• What is the cost of a first level TLB miss?
  – Second level TLB lookup

• What is the cost of a second level TLB miss?
  – 64 bit x86: 4-level page table walk

• How expensive is a 4-level page table walk?
Virtually Addressed vs. Physically Addressed Caches

- First level cache has at most a few cycles
  - Delays every instruction fetch and data reference
- Lookup TLB to get physical address, then lookup physical address in the cache?
  - Too slow!
- Instead, lookup virtual address in cache
- In parallel, lookup TLB in case of a cache miss
Virtually Addressed Caches

Processor

Virtual Address

Virtual Cache

Virtual Address Miss

TLB

Virtual Address Miss

Page Table

Invalid

Raise Exception

Hit

Data

Hit

Data

Hit

Frame

Valid

Frame

Physical Memory

Physical Address

Offset

TLB & Virtual Cache: lookup virtual address
Question

- With a virtual cache, what do we need to do on a context switch?
Implementation

**Aliases**

Two virtual addresses

⇒ same physical address

Translation Lookaside Buffer (TLB)

Matching Entry

Virtually
Address

AOPR
(32/64 bytes)

Physical
Memory
Physically Addressed Cache

Processor -> Virtual Cache
      |        | Virtual Address Miss -> TLB
      | Hit/Frame | Virtual Address Miss -> Page Table
      |           | Virtual Address Miss -> Physical Cache
      |           | Physical Address Hit/Update -> Physical Memory

Data Flow: Processor -> Virtual Cache (Hit) -> Data
           |          | Virtual Cache Miss -> TLB (Hit/Framed/Data)
           |          | TLB Hit/Framed/Data -> Page Table
           |          | Page Table Valid/Framed/Data -> Physical Cache
           |          | Physical Cache Hit/Framed/Data -> Physical Memory

Physical cache: lookup physical address
Page Coloring

• What happens when cache size >> page size?
  – Direct mapped or set associative
  – Multiple pages map to the same cache line

• OS page assignment matters!
  – Example: 8MB cache, 4KB pages
  – 1 of every 2K pages lands in same place in cache

• What should the OS do?
Page Coloring

Choose pages for each process to balance usage across page types (colors)
TLB Size (Intel Kaby Lake, 2017)

First level TLB
- Instruction: 128 entries
- Data: 64 entries

Second level TLB
- 1536 entries

Modern server can have 10 TB (!) of DRAM
- 10-20% of server CPU time spent in TLB misses

⇒ 2^33 page frames (!)
When Do TLBs Work/Not Work?

Video Frame Buffer:
32 bits x 1K x 1K = 4MB = 1K pages

2017 laptop: 3K x 2K = 24MB = 6K pages

4K display: 4K x 3K = 48MB = 12K pages
Superpages

- On x86 and ARM, TLB entry can be
  - A page
  - A superpage: a set of contiguous, aligned pages
- x86: superpage is set of pages in one page table
  - One page: 4KB
  - One page table: 2MB
  - One page table of page tables: 1GB
  - One page table of page tables of page tables: 0.5TB

ARM: superpages can be a full page table entry
Superpages

Virtual Address

<table>
<thead>
<tr>
<th>Page#</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Translation Lookaside Buffer (TLB)

Matching Entry

Matching Superpage

Page Table Lookup

Physical Address

Kaby Lake has separate TLB for pages and superpages
When Do TLBs Work/Not Work, part 2

• What happens when the OS changes the permissions on a page?
  – For demand paging, copy on write, zero on reference, ...

• TLB may contain old translation
  – OS must ask hardware to purge TLB entry

• On a multicore: TLB shootdown
  – OS must ask each CPU to purge TLB entry
TLB Shootdown

<table>
<thead>
<tr>
<th>Processor 1 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor 2 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor 3 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>

Copy on write fork of process 0? 

IPI - interprocess interrupt
Virtual Cache Shootdown

- When permissions change for a page, we must shoot down the TLB entry on every CPU
- What about the contents of the virtual cache?
- Lazy shootdown of the virtual cache:
  - Lookup virtually addressed cache and TLB in parallel
  - Use the TLB to verify virtual address is still valid!
  - Evict entry from cache if not
Virtual Cache Aliases

- Alias: two (or more) virtual cache entries that refer to the same physical memory
  - A consequence of a tagged virtually addressed cache!
  - A write to one copy needs to update all copies
- Solution:
  - Virtual cache keeps both virtual and physical address for each entry
  - Lookup virtually addressed cache and TLB in parallel
  - Check if physical address from TLB matches any other entries, and update/invalidate those copies
x86 caches

- 64 byte line size
- Physically indexed
- Physically tagged
- Write buffer