Another View of the Memory Hierarchy

Thus far

Next: Virtual Memory

Upper Level

Faster

Larger

Lower Level
Review

° Manage memory to disk? Treat as cache
  • Included protection as bonus, now critical
  • Use Page Table of mappings for each user vs. tag/data in cache
  • TLB is cache of Virtual⇒Physical addr trans

° Virtual Memory allows protected sharing of memory between processes

° Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well

Address Mapping: Page Table

Virtual Address:

Page Table Base Reg

Page Table located in physical memory
Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Cache version</th>
<th>Virtual Memory vers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or Line</td>
<td>Page</td>
</tr>
<tr>
<td>Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4K-8KB</td>
</tr>
<tr>
<td>Placement: Direct Mapped, N-way Set Associative</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Replacement: LRU or Random</td>
<td>Least Recently Used (LRU)</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
Virtual Memory Problem #1

° Map every address ⇒ 1 indirection via Page Table in memory per virtual address ⇒ 1 virtual memory accesses = 2 physical memory accesses ⇒ SLOW!

° Observation: since locality in pages of data, there must be locality in virtual address translations of those pages

° Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?

° For historical reasons, cache is called a Translation Lookaside Buffer, or TLB

Translation Look-Aside Buffers (TLBs)

• TLBs usually small, typically 128 - 256 entries

• Like any other cache, the TLB can be direct mapped, set associative, or fully associative

On TLB miss, get page table entry from main memory
Typical TLB Format

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access Rights</th>
</tr>
</thead>
</table>

- TLB just a cache on the page table mappings
- TLB access time comparable to cache (much less than main memory access time)
- **Dirty**: since use write back, need to know whether or not to write page to disk when replaced
- **Ref**: Used to help calculate LRU on replacement
  - Cleared by OS periodically, then checked to see if page was referenced

What if not in TLB?

- **Option 1**: Hardware checks page table and loads new Page Table Entry into TLB
- **Option 2**: Hardware *traps* to OS, up to OS to decide what to do
  - MIPS follows Option 2: Hardware knows nothing about page table
What if the data is on disk?

- We load the page off the disk into a free block of memory, using a DMA (Direct Memory Access – very fast!) transfer
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an interrupt and update the process's page table
  - So when we switch back to the task, the desired data will be in memory

What if we don’t have enough memory?

- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (disk copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
- And update that program's page table to reflect the fact that its memory moved somewhere else
- If continuously swap between disk and memory, called **Thrashing**
**Address Translation & 3 Concept tests**

Virtual Address

<table>
<thead>
<tr>
<th>VPN</th>
<th>INDEX</th>
<th>Offset</th>
</tr>
</thead>
</table>

**TLB**

<table>
<thead>
<tr>
<th>V. P. N.</th>
<th>P. P. N.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page Number</td>
<td>Physical Page Number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V. P. N.</th>
<th>P. P. N.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>PPN</th>
<th>Offset</th>
</tr>
</thead>
</table>

**Data Cache**

<table>
<thead>
<tr>
<th>Tag:Data</th>
<th>Tag:Data</th>
</tr>
</thead>
</table>

**Physical Address**

<table>
<thead>
<tr>
<th>TAG</th>
<th>INDEX</th>
<th>Offset</th>
</tr>
</thead>
</table>

**Question (1/3)**

° 40-bit virtual address, 16 KB page

<table>
<thead>
<tr>
<th>Virtual Page Number (? bits)</th>
<th>Page Offset (? bits)</th>
</tr>
</thead>
</table>

° 36-bit physical address

<table>
<thead>
<tr>
<th>Physical Page Number (? bits)</th>
<th>Page Offset (? bits)</th>
</tr>
</thead>
</table>

° Number of bits in Virtual Page Number/Page offset, Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 22/14 (PPN/PO)
2: 24/16, 20/16
3: 26/14, 22/14
4: 26/14, 26/10
5: 28/12, 24/12
### Question (2/3): 40b VA, 36b PA

#### 2-way set-assoc. TLB, 512 entries, 40b VA:

<table>
<thead>
<tr>
<th>TLB Tag (? bits)</th>
<th>TLB Index (? bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
</table>

#### TLB Entry: Valid bit, Dirty bit, Access Control (say 2 bits), Virtual Page Number, Physical Page Number

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Access (2 bits)</th>
<th>TLB Tag (? bits)</th>
<th>Physical Page No. (? bits)</th>
</tr>
</thead>
</table>

#### Number of bits in TLB Tag / Index / Entry?

1. 12 / 14 / 38 (TLB Tag / Index / Entry)
2. 14 / 12 / 40
3. 18 / 8 / 44
4. 18 / 8 / 58

### Question (3/3)

#### 2-way set-assoc, 64KB data cache, 64B block

<table>
<thead>
<tr>
<th>Cache Tag (? bits)</th>
<th>Cache Index (? bits)</th>
<th>Block Offset (? bits)</th>
</tr>
</thead>
</table>

#### Data Cache Entry: Valid bit, Dirty bit, Cache tag + ? bits of Data

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Cache Tag (? bits)</th>
<th>Cache Data (? bits)</th>
</tr>
</thead>
</table>

#### Number of bits in Data cache Tag / Index / Offset / Entry?

1. 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)
2. 20 / 10 / 6 / 86
3. 20 / 10 / 6 / 534
4. 21 / 9 / 6 / 87
5. 21 / 9 / 6 / 535
4 Qs for any Memory Hierarchy

° Q1: Where can a block be placed?
  • One place (direct mapped)
  • A few places (set associative)
  • Any place (fully associative)

° Q2: How is a block found?
  • Indexing (as in a direct-mapped cache)
  • Limited search (as in a set-associative cache)
  • Full search (as in a fully associative cache)
  • Separate lookup table (as in a page table)

° Q3: Which block is replaced on a miss?
  • Least recently used (LRU)
  • Random

° Q4: How are writes handled?
  • Write through (Level never inconsistent w/lower)
  • Write back (Could be "dirty", must have dirty bit)

Q1: Where block placed in upper level?

Block 12 placed in 8 block cache:
• Fully associative
• Direct mapped
• 2-way set associative
  - Set Associative Mapping = Block # Mod # of Sets
Q2: How is a block found in upper level?

- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag

Q3: Which block replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Miss Rates</th>
<th>Associativity: 2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Q4: What to do on a write hit?

° Write-through
  • update the word in cache block and corresponding word in memory

° Write-back
  • update word in cache block
  • allow memory word to be “stale”
  => add ‘dirty’ bit to each line indicating that memory be updated when block is replaced
  => OS flushes cache before I/O !!!

° Performance trade-offs?
  • WT: read misses cannot result in writes
  • WB: no writes of repeated writes

Three Advantages of Virtual Memory

1) Translation:
  • Program can be given consistent view of memory, even though physical memory is scrambled
  • Makes multiple processes reasonable
  • Only the most important part of program (“Working Set”) must be in physical memory
  • Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later
Three Advantages of Virtual Memory

2) Protection:
   • Different processes protected from each other
   • Different pages can be given special behavior
     - (Read Only, Invisible to user programs, etc).
   • Kernel data protected from User programs
   • Very important for protection from malicious programs ⇒ Far more “viruses” under Microsoft Windows
   • Special Mode in processor ("Kernel mode") allows processor to change page table/TLB

3) Sharing:
   • Can map same physical page to multiple users ("Shared memory")

Why Translation Lookaside Buffer (TLB)?

° Paging is most popular implementation of virtual memory (vs. base/bounds)
° Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection
° Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make fast
And in Conclusion…

○ Virtual memory to Physical Memory Translation too slow?
  • Add a cache of Virtual to Physical Address Translations, called a **TLB**

○ Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well

○ Virtual Memory allows protected sharing of memory between processes with less swapping to disk