Module: Virtual Memory

Reading

- Section 5.4 (virtualization to be discussed later!)
Topical Outline

- Review of virtual memory management
  - Motivation
  - Operation

- Speeding up address translation via concurrency

- Speeding up address translation via use of virtually addressed caches

The Memory Hierarchy

- Caching is the mode of operation at each level of the hierarchy
  - tape → disk → memory → cache → registers

- Control by HW or SW guided by
  - Static vs. run-time management of resources
  - Time scales of operation → nanoseconds to seconds
Caching Mechanisms

- Registers: static decisions made by compiler

- TLB/HW cache: runtime decision made on tags, possibly many tags in parallel.

- VM as cache: mapping (w/HW acceleration) followed by SW-managed refill on miss.

- File system cache: SW managed

- Web page cache: SW managed

Caching From Disk

- Programmers have managed use of smaller, faster memories
  - Application controlled overlays
- Disks are mechanical devices \( \rightarrow \) operational speeds are on the order of milliseconds
  - This is on the order of 100,000s of instructions
  - Software management!
  - Unit of management is large blocks to amortize disk overhead
Virtual Memory Management

- Follows the same basic principles as cache management

- Unit of management is a page
  - A page is typically 4Kbytes – 32 Kbytes

- The program sees a virtual address space partitioned into virtual pages
  - Virtual address space typically exceeds physical memory size
  - The program resides on disk

- Physical memory is partitioned into physical pages

Management Policies

- Demand driven operation
  - Pages are brought into memory when referenced

- Placement policy
  - Fully associative placement

- Replacement policy
  - Approximations to least recently used (LRU) are the most common

- Update policy
  - At current disk latencies, write-through is infeasible
  - Write-back update policy is employed
Address Translation: Concepts

- Offsets within the virtual page and corresponding physical page are the same
- We only need to translate the virtual page number (VPN) to the corresponding physical page number (PPN) also called *page frame*

Address Translation Implementation: The Page Table

- Translate the virtual page address to the physical page address
- Keep state information on a page
  - Modified/Not Modified
  - Access rights
  - In memory or on disk
  - Caching policies
Translation Mechanism

- Address translation requires looking up a page table entry (PTE) in a page table (PT)
  - Called a page table “walk”
  - Store page table in memory
  - PT miss handling (i.e. a page fault) is in SW
  - Facilitates code relocation

- Many possible page-table organizations:
  - Hashed implementation
  - Forward-walked, multilevel tables
  - “backward”-walked (via recursion) multilevel tables
  - hashed “inverse” tables

Multi-level (Hierarchical) Page Table

- Divide virtual address into multiple levels

Level 1 is stored in the Main memory

Level 1 page directory (pointer array)

Level 2 page table (stores PPN)

PPN

Page offset

Source: Prof. H. H. Lee, ECE
Caching PTEs: The Translation Lookaside Buffer (TLB)

A four entry TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>V</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>PPN</td>
<td>V</td>
<td>state</td>
</tr>
<tr>
<td>VPN</td>
<td>PPN</td>
<td>V</td>
<td>state</td>
</tr>
<tr>
<td>VPN</td>
<td>PPN</td>
<td>V</td>
<td>state</td>
</tr>
</tbody>
</table>

- Keep a cache of most recently used PTEs
  - Each PTE corresponds to a “relatively” large part of memory
    - For example, a 16Kbyte page may have 4K instructions
  - A small set of PTEs can cover a large code segment
    - For example, 8 PTEs and 16 Kbyte pages corresponds to a program size of 32K instructions
- The TLB access time is comparable or better than cache access time
- Typically operates as a fully associative cache, but can be implemented as a set associative cache

TLB Operation

- TLB size typically a function of the target domain
  - High end machines will have fully associative large TLBs
- PTE entries are replaced on a demand driven basis
- The TLB is in the critical path
The Memory Access Path

- Virtual-to-physical address translation occurs on every access!
  - Adds to the latency of every memory access

- How can we optimize the critical path?

Optimizing the Critical Path: Principles

- Concurrency between address translation and cache access
  - Overlap cache access and TLB translation

- Making translation the exception
  - Address the cache with the virtual address
Overlapping Cache and TLB Access

- Example: direct mapped, 16 Kbyte cache with 64 byte lines and 16 Kbyte pages

- The virtual address offset field is not translated, and is the same in the physical address

- If all bits of the cache line address come from the offset field, TLB and cache can be accessed in parallel

- What if the cache size is doubled?
Ensuring Concurrency

- Increase the associativity of the cache \(\rightarrow\) reduce the number of bits in the line address

- Increase the page size \(\rightarrow\) increase the number of low order bits of the virtual address that are not translated

- Constrain the placement policy to ensure line address can be known from the virtual address
  - In the preceding example make sure the least significant bit of the virtual page number and physical page number are equal

R10000’s Solution to Synonym

- 32KB 2-Way Virtually-Indexed L1

<table>
<thead>
<tr>
<th>VPN</th>
<th>12 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 bit</td>
</tr>
</tbody>
</table>

- Direct-Mapped Physical L2
  - L2 is *Inclusive* of L1
  - VPN[1:0] is appended to the “tag” of L2
- Given two virtual addresses \(VA1\) and \(VA2\) that differs in \(VPN[1:0]\) and both map to the same physical address \(PA\)
  - Suppose \(VA1\) is accessed first so blocks are allocated in L1&L2
  - What happens when \(VA2\) is referenced?
    1. \(VA2\) indexes to a different block in L1 and misses
    2. \(VA2\) translates to \(PA\) and goes to the same block as \(VA1\) in L2
    3. Tag comparison fails (since \(VA1[1:0]\neq VA2[1:0]\))
    4. Treated just like a L2 conflict miss \(\Rightarrow VA1’s\) entry in L1 is ejected (or dirty-written back if needed) due to inclusion policy

\[a = VPN[1:0]\] stored as part of L2 cache Tag

Source: Prof. H. H. Lee, ECE
Deal w/ Synonym in MIPS R10000

VA1

Page offset

index

a1

VA2

Page offset

index

a2

L1 VIPT cache

miss

TLB

L2 PIPT Cache

Physical index || a2

a2 != a1

a1 Phy. Tag data

Only one copy is present in L1

L1 VIPT cache

Data return

L2 PIPT Cache

a2 Phy. Tag data

Source: Prof. H. H. Lee, ECE
Making Address Translation the Exception

- The cache is addressed using virtual addresses
- When there is a miss in the cache, address translation takes place producing the physical address used to access main memory
  - Page table entries are also updated

Design Space

- Relationship between virtual \(\rightarrow\) physical address translation and the indexing of the cache
- Goal: speed up the critical path
  - Make the common case fast

Tag

```
virtual  physical
Index
virtual  VIVT  VIPT
physical  PIVT  PIPR
```

Conventional Cache
Transitions checked on a cache miss
What are the issues with this type of cache?
- Homonyms
- Synonyms

No concurrency between address translation and cache index operation
Virtually Indexed Physically Tagged

- Some concurrency between address translation and cache index operation

Physically Indexed Virtually Tagged

- Generally not used
Issues for Virtually Addressed Caches

Protection
- Each access to page normally can be checked against permissions recorded in the PTE during translation

Solution:
- Mechanisms must be provided to for checking access rights

<table>
<thead>
<tr>
<th>V</th>
<th>Access rights</th>
<th>PPN/Disk Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Issues (cont.)

False hits in a multi-process system
- Multiple processes typically have the same virtual address spaces
  - The same virtual address in two distinct address spaces refer to two distinct physical addresses

Solutions
- Flush on a context swap
  - This is a problem in SMT machines
- Add an address space identifier (ASID) or process ID (PID)
  - Now all processes effectively have distinct address spaces
• Synonym or Aliasing Problem
  ❖ Multiple copies of the same physical memory line exist at distinct locations in the cache
  ❖ This can occur when
    0 Two processes may share a physical page
    0 This page may be mapped to different parts of the individual virtual address spaces → A shared physical location has two virtual addresses
    0 Page size is smaller than the cache
• Solution:
  ❖ Placement constraints on shared pages

---

• How do we ensure than any line in the shared (physical) page maps to exactly one line in the cache
  ❖ Constrain the necessary locations in address spaces of A and B
    0 Ensures all lower order bits required for addressing the cache are identical
Address Spaces, PID, DLLs and SMT*

- Assume page size = cache size and that synonyms have been avoided through the use of PID.
- A shared DLL will lead to
  - DLL thrashing in direct mapped caches
  - DLL duplication in set associative caches


Alpha Memory Management

- Paged, segmented address space
- Each page table fits in a page
  - 8KB pages
- L1 & L2 tables use physical addresses
- L3 is virtually mapped
- 64 bit PTE
  - 32 bit frame address
  - Status/protection bits
- Protection
  - Pages tables protected
The TLB is not flushed on a context switch since the ASN is used. TLB is flushed when ASNs are recycled.

Actual address size is a function of the processor mode.

---

**Some Example Memory Hierarchies**

<table>
<thead>
<tr>
<th>Intel P6 Core</th>
<th>DEC Alpha 21264</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core of PPro, PII, PIII</td>
<td>Split I + D (64K+6K)</td>
</tr>
<tr>
<td>Split I + D L1 (8K+8K)</td>
<td>Split I + D TLBs (128)</td>
</tr>
<tr>
<td>Split I + D TLBs (32+64)</td>
<td><strong>two-cycle</strong> L1 access</td>
</tr>
<tr>
<td>L2 originally on MCM</td>
<td>2-way set associative</td>
</tr>
<tr>
<td>PIII has 2x cache + L2</td>
<td>8-entry victim cache</td>
</tr>
<tr>
<td>“Xeon” has big L2</td>
<td>L2 up to 16MB</td>
</tr>
<tr>
<td></td>
<td>L2 access 12 cycles</td>
</tr>
<tr>
<td>Aside: note 5 FUs and</td>
<td>Aside: 6 FUs</td>
</tr>
<tr>
<td>3-wide fetch unit,</td>
<td>FUs in <strong>clusters</strong></td>
</tr>
<tr>
<td>40-entry ROB</td>
<td>4-wide fetch</td>
</tr>
<tr>
<td></td>
<td>20+15 queues</td>
</tr>
</tbody>
</table>
Study Guide: Glossary

- Aliasing
- Homonym
- Multilevel page table
- Page
- Page frame
- Physical page
- Page table walk
- Physically tagged
- Physically indexed
- PID
- Synonym
- Translation look-aside buffer (TLB)
- Virtually tagged
- Virtually indexed
- Virtual memory
- Virtual page

Study Guide

- Be able to trace through the page table and cache data structures on a memory reference (see sample problems)
- Understand how to allocate virtual pages to page frames to minimize conflicts in the cache
- Relationships between address translation, page size, and cache size.
  - For example, when is concurrency possible between cache indexing and address translation
  - Given a memory system design (page sizes, virtual and physical address spaces, cache parameters) how would you modify the design to improve performance
    - For example, address translation time, page the page table
• Virtually indexed caches
  ❖ Problems & solutions
  ❖ Using 32 bit virtual addresses create an instance of synonyms
  ❖ Using 32 bit virtual addresses show how a false hit can take place on a context swap if the cache is not flushed or if PIDs are not used
• Trace a memory reference through the complete Alpha memory hierarchy
• Given a cache design and virtual address space and page size, define by VPN the pages that conflict in the cache