Virtual Memory

Use main memory as a “cache” for secondary (disk) storage
- Managed jointly by CPU hardware and the operating system (OS)

Programs share main memory
- Each gets a private virtual address space holding its frequently used code and data
- Protected from other programs

CPU and OS translate virtual addresses to physical addresses
- VM “block” is called a page
- VM translation “miss” is called a page fault
Address Translation

Fixed-size pages (e.g., 4KB)
On page fault, the page must be fetched from disk
- Takes millions of clock cycles
- Handled by OS code

Try to minimize page fault rate
- Fully associative placement
- Smart replacement algorithms

How bad is that?

Assume a 3 GHz clock rate. Then 1 million clock cycles would take 1/3000 seconds or 1/3 ms.

Subjectively, a single page fault would not be noticed... but page faults can add up.

We must try to minimize the number of page faults.
Page Tables

Stores placement information
- Array of page table entries, indexed by virtual page number
- **Page table register** in CPU points to page table in physical memory

If page is present in memory
- PTE stores the physical page number
- Plus other status bits (referenced, dirty, …)

If page is not present
- PTE can refer to location in swap space on disk
Translation Using a Page Table

1. Page table register
2. Virtual address
3. Page table
4. If 0 then page is not present in memory
5. Physical address

Virtual page number
Page offset

Physical page number
Page offset

Valid
Physical page number

31 30 29 28 27 15 14 13 12 11 10 9 8 3 2 1 0
Mapping Pages to Storage

Virtual page number

Page table
- Valid
- Physical page or disk address

Physical memory

Disk storage
Replacement and Writes

To reduce page fault rate, prefer least-recently used (LRU) replacement (or approximation)

- *Reference bit* (aka use bit) in PTE set to 1 on access to page
- Periodically cleared to 0 by OS
- A page with reference bit = 0 has not been used recently

Disk writes take millions of cycles

- Block at once, not individual locations
- Write through is impractical
- Use write-back
- Dirty bit in PTE set when page is written
Fast Translation Using a TLB

Address translation would appear to require extra memory references
- One to access the PTE
- Then the actual memory access

Can't afford to keep them all at the processor level.

But access to page tables has good locality
- So use a fast cache of PTEs within the CPU
- Called a Translation Look-aside Buffer (TLB)
- Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
- Misses could be handled by hardware or software
Fast Translation Using a TLB
TLB Misses

If page **is** in memory
- Load the PTE from memory and retry
- Could be handled in hardware
  - Can get complex for more complicated page table structures
- Or in software
  - Raise a special exception, with optimized handler

If page **is not** in memory (page fault)
- OS handles fetching the page and updating the page table
- Then restart the faulting instruction
TLB Miss Handler

TLB miss indicates whether
- Page present, but PTE not in TLB
- Page not present

Must recognize TLB miss before destination register overwritten
- Raise exception

Handler copies PTE from memory to TLB
- Then restarts instruction
- If page not present, page fault will occur
Page Fault Handler

Use faulting virtual address to find PTE

Locate page on disk

Choose page to replace
  - If dirty, write to disk first

Read page into memory and update page table

Make process runnable again
  - Restart from faulting instruction
If cache tag uses physical address
- Need to translate before cache lookup

Alternative: use virtual address tag
- Complications due to aliasing
  - Different virtual addresses for shared physical address
Memory Protection

Different tasks can share parts of their virtual address spaces
- But need to protect against errant access
- Requires OS assistance

Hardware support for OS protection
- Privileged supervisor mode (aka kernel mode)
- Privileged instructions
- Page tables and other state information only accessible in supervisor mode
- System call exception (e.g., syscall in MIPS)
The Memory Hierarchy

Common principles apply at all levels of the memory hierarchy
- Based on notions of caching

At each level in the hierarchy
- Block placement
- Finding a block
- Replacement on a miss
- Write policy
Determined by associativity

- Direct mapped (1-way associative)
  - One choice for placement
- n-way set associative
  - n choices within a set
- Fully associative
  - Any location

Higher associativity reduces miss rate

- Increases complexity, cost, and access time
Finding a Block

Hardware caches
- Reduce comparisons to reduce cost

Virtual memory
- Full table lookup makes full associativity feasible
- Benefit in reduced miss rate

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Location method</th>
<th>Tag comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>n-way set associative</td>
<td>Set index, then search entries within the set</td>
<td>n</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Search all entries</td>
<td>#entries</td>
</tr>
<tr>
<td></td>
<td>Full lookup table</td>
<td>0</td>
</tr>
</tbody>
</table>
Choice of entry to replace on a miss
- Least recently used (LRU)
  - Complex and costly hardware for high associativity
- Random
  - Close to LRU, easier to implement

Virtual memory
- LRU approximation with hardware support
Write Policy

Write-through
- Update both upper and lower levels
- Simplifies replacement, but may require write buffer

Write-back
- Update upper level only
- Update lower level when block is replaced
- Need to keep more state

Virtual memory
- Only write-back is feasible, given disk write latency
Sources of Misses

Compulsory misses (aka cold start misses)
- First access to a block

Capacity misses
- Due to finite cache size
- A replaced block is later accessed again

Conflict misses (aka collision misses)
- In a non-fully associative cache
- Due to competition for entries in a set
- Would not occur in a fully associative cache of the same total size
### Cache Design Trade-offs

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decrease capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decrease conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decrease compulsory misses</td>
<td>Increases miss penalty. For very large block size, may increase miss rate due to pollution.</td>
</tr>
</tbody>
</table>
Multilevel On-Chip Caches

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache
## 2-Level TLB Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtual addr</strong></td>
<td>48 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td><strong>Physical addr</strong></td>
<td>44 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td><strong>Page size</strong></td>
<td>4KB, 2/4MB</td>
<td>4KB, 2/4MB</td>
</tr>
<tr>
<td><strong>L1 TLB</strong></td>
<td>L1 I-TLB: 128 entries for small pages, 7 per thread (2×)</td>
<td>L1 I-TLB: 48 entries</td>
</tr>
<tr>
<td></td>
<td>L1 D-TLB: 64 entries for small pages, 32 for large pages</td>
<td>L1 D-TLB: 48 entries</td>
</tr>
<tr>
<td></td>
<td>Both 4-way, LRU replacement</td>
<td>Both fully associative, LRU replacement</td>
</tr>
<tr>
<td><strong>L2 TLB</strong></td>
<td>Single L2 TLB: 512 entries</td>
<td>L2 I-TLB: 512 entries</td>
</tr>
<tr>
<td></td>
<td>4-way, LRU replacement</td>
<td>L2 D-TLB: 512 entries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Both 4-way, round-robin LRU</td>
</tr>
<tr>
<td><strong>TLB misses</strong></td>
<td>Handled in hardware</td>
<td>Handled in hardware</td>
</tr>
</tbody>
</table>
### 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong></td>
<td><strong>L1 I-cache:</strong> 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td><strong>L1 I-cache:</strong> 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td>(per core)</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td></td>
<td>32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong></td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td>(per core)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L3 unified cache</strong></td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
<tr>
<td>(shared)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n/a: data not available</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nehalem Overview

Virtual Memory 25

Intel Nehalem microarchitecture

- Quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
- Prefetch Buffer (16 Bytes)
- Branch Prediction global/bimodal, loop, indirect jmp
- Instruction Queue 18 x64 Instructions
- Alignment
- MacroOp Fusion
- Complex Decoder
- Simple Decoder
- Simple Decoder
- Simple Decoder
- Decoded Instruction Queue (28 μOP entries)
- MicroOp Fusion
- Micro Instruction Sequencer
- 2 x Register Allocation Table (RAT)
- Reorder Buffer (128-entry) fused
- Reservation Station (128-entry) fused
- Part 4
- AGU Store Addr. Unit
- AGU Load Addr. Unit
- Integer/MMX ALU, Branch
- SSE ADD Move
- SSE ADD Move
- Integer/MMX ALU
- Integer/MMX ALU
- 2x AGU
- FP ADD
- FP MUL
- Integer/MMX ALU
- Integer/MMX ALU
- SSE MUL/DIV Move
- SSE MUL/DIV Move
- Memory Order Buffer (MOB)

Result Bus

- octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M

GT/s: gigatransfers per second

Uncore
- Quick Path Interconnect
- DDR3 Memory Controller
- Common L3-Cache 8 MByte

256 KByte
- 8-way, 64 Byte Cacheline, private L2-Cache

512-entry L2-TLB-4K

4 x 20 Bit 6.4 GT/s
3 x 64 Bit 1.33 GT/s

Computer Organization II