Review of Cache/VM/TLB

Lecture 26

April 30, 1999

Dave Patterson

Outline
° Review Pipelining
° Review Cache/VM/TLB Review slides
° Administrivia, “What’s this Stuff Good for?”
° 4 Questions on Memory Hierarchy
° Detailed Example
° 3Cs of Caches (if time permits)
° Cache Impact on Algorithms (if time permits)
° Conclusion

Review 1/3: Pipelining Introduction
° Pipelining is a fundamental concept
  • Multiple steps using distinct resources
  • Exploiting parallelism in instructions
° What makes it easy? (MIPS vs. 80x86)
  • All instructions are the same length
    ⇒ simple instruction fetch
  • Just a few instruction formats
    ⇒ read registers before decode instruction
  • Memory operands only in loads and stores
    ⇒ fewer pipeline stages
  • Data aligned ⇒ 1 memory access / load, store

Review 2/3: Pipelining Introduction
° What makes it hard?
° Structural hazards: suppose we had only one cache?
  ⇒ Need more HW resources
° Control hazards: need to worry about branch instructions?
  ⇒ Branch prediction, delayed branch
° Data hazards: an instruction depends on a previous instruction?
  ⇒ need forwarding, compiler scheduling

Review 3/3: Advanced Concepts
° Superscalar Issue, Execution, Retire:
  • Start several instructions each clock cycle
    (1999: 3-4 instructions)
  • Execute on multiple units in parallel
  • Retire in parallel; HW guarantees appearance of simple single instruction execution
° Out-of-order Execution:
  • Instructions issue in-order, but execute out-of-order when hazards occur (load-use, cache miss, multiplier busy, …)
  • Instructions retire in-order; HW guarantees appearance of simple in-order execution

Memory Hierarchy Pyramid

Central Processor Unit (CPU)

Increasing Distance from CPU, Decreasing cost / MB

Sizes of memory at each level

Principle of Locality (in time, in space) + Hierarchy of Memories of different speed, cost; exploit to improve cost-performance
Why Caches?

- Processor-Memory Performance Gap:
  - (grows 50%/year)
  - DRAM 7%/yr.
  -μProc 60%/yr.

1989 first Intel CPU with cache on chip;
Today 37% area of Alpha 21164,
61% StrongArm SA110, 64% Pentium Pro

Why virtual memory? (1/2)

- Protection
  - regions of the address space can be read
    only, execute only, . . .
- Flexibility
  - portions of a program can be placed
    anywhere, without relocation
- Expandability
  - can leave room in virtual address space for
    objects to grow
- Storage management
  - allocation/deallocation of variable sized
    blocks is costly and leads to (external)
    fragmentation

Why virtual memory? (2/2)

- Generality
  - ability to run programs larger than size of
    physical memory
- Storage efficiency
  - retain only most important portions of the
    program in memory
- Concurrent I/O
  - execute other processes while
    loading/dumping page

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
- Cache of Page Table Entries makes
  address translation possible without
  memory access in common case

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

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

Virtual Memory Summary

3 Problems:
1) Not enough memory: Spatial Locality
   means small Working Set of pages OK
2) TLB to reduce performance cost of VM
3) Need more compact representation to
   reduce memory size cost of simple 1-level
   page table, especially for 64-bit address
   (See CS 162)

What’s This Stuff (Potentially) Good For?”

Allow civilians to find
mines, mass graves,
plan disaster relief?

4 Questions for Memory Hierarchy

Q1: Where can a block be placed in the
   upper level? (Block placement)
Q2: How is a block found if it is in the
   upper level? (Block identification)
Q3: Which block should be replaced on
   a miss? (Block replacement)
Q4: What happens on a write? (Write strategy)
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)

Miss Rates
Associativity: 2-way  4-way  8-way
Size     LRU  Ran  LRU  Ran  LRU  Ran
16 KB     5.2% 5.7% 4.7% 5.3% 4.4% 5.0%
64 KB     1.9% 2.0% 1.5% 1.7% 1.4% 1.5%
256 KB    1.15% 1.17% 1.13% 1.13% 1.12% 1.12%

Q4: What happens on a write?
- Write through—The information is written to both the block in the cache and to the block in the lower-level memory.
- Write back—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
- WT always combined with write buffers

Comparing the 2 levels of hierarchy
- Cache Version  Virtual Memory vers.
- Block or Line  Page
- Miss  Page Fault
- Block Size: 32-64B  Page Size: 4K-8KB
- Placement:  Direct Mapped, Fully Associative, N-way Set Associative
- Replacement:  Least Recently Used LRU or Random (LRU)
- Write Thru or Back  Write Back

Picking Optimal Page Size
- Minimize wasted storage
  - small page minimizes internal fragmentation
  - small page increase size of page table
- Minimize transfer time
  - large pages (multiple disk sectors) amortize access cost
  - sometimes transfer unnecessary info
  - sometimes prefetch useful data
  - sometimes discards useless data early
- General trend toward larger pages because
  - big cheap RAM
  - increasing mem / disk performance gap
  - larger address spaces
Starting an Alpha 21064: 1/6
° Starts in kernel mode where memory is not translated
° Since software managed TLB, 1st load TLB with 12 valid mappings for process in Instruction TLB
° Sets the PC to appropriate address for user program and goes into user mode
° Page frame portion of 1st instruction access is sent to Instruction TLB: checks for valid PTE and access rights match (otherwise an exception for page fault or access violation)

Starting an Alpha 21064: 2/6
° Since 256 32-byte blocks in instruction cache, 8-bit index selects the I cache block
° Translated address is compared to cache block tag (assuming tag entry is valid)
° If hit, proper 4 bytes of block sent to CPU
° If miss, address sent to L2 cache; its 2MB cache has 64K 32-byte blocks, so 16-bit index selects the L2 cache block; address tag compared (assuming its valid)
° If L2 cache hit, 32-bytes in 10 clock cycles

Starting an Alpha 21064: 3/6
° If L2 cache miss, 32-bytes in 36 clocks
° Since L2 cache is write back, any miss can result in a write to main memory; old block placed into buffer and new block fetched first; after new block loaded, and old block is written to memory if old block is dirty

Starting an Alpha 21064: 4/6
° Suppose 1st instruction is a load; 1st sends page frame to data TLB
° If TLB miss, software loads with PTE
° If data page fault, OS will switch to another process (since disk access = time to execute millions of instructions)
° If TLB hit and access check OK, send translated page frame address to Data Cache
° 8-bit portion of address selects data cache block; tags compared to see if a hit (if valid)
° If miss, send to L2 cache as before

Starting an Alpha 21064: 5/6
° Suppose instead 1st instruction is a store
° TLB still checked (no protection violations and valid PTE), send translated address to data cache
° Access to data cache as before for hit, except write new data into block
° Since Data Cache is write through, also write data to Write Buffer
° If Data Cache access on store is a miss, also write to Write Buffer since policy is no allocate on write miss

Starting an Alpha 21064: 5/6
° Write buffer checks to see if write is already to address within entry, and if so updates the block
° If 4-entry write buffer is full, stall until entry is written to L2 cache block
° All writes eventually passed to L2 cache
° If miss, put old block in buffer, and since L2 allocates on write miss, load missing data from cache; Write to portion of block, and mark new L2 cache block as dirty;
° If old block in buffer is dirty, then write it to memory

Page 5, 4/29/99 10:57 PM
Classifying Misses: 3 Cs

- **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. 
  *(Misses in an Infinite Cache)*

- **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. 
  *(Misses in a Fully Associative Size X Cache)*

- **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. 
  *(Misses in N-way Associative, Size X Cache)*

3Cs Absolute Miss Rate (SPEC92)

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>Compulsory</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.14</td>
<td>0.12</td>
</tr>
<tr>
<td>2</td>
<td>0.11</td>
<td>0.10</td>
</tr>
<tr>
<td>4</td>
<td>0.08</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>0.04</td>
<td>0.02</td>
</tr>
</tbody>
</table>

3Cs Relative Miss Rate

3Cs Flaws: for fixed block size
3Cs Pros: insight ⇒ invention

Impact of What Learned About Caches?

- 1960-1985: Speed = f(no. operations)

- 1990s
  - Pipelined Execution & Fast Clock Rate
  - Out-of-Order execution
  - Superscalar

- 1999: Speed = f(non-cached memory accesses)

Quicksort vs. Radix as vary number keys:

**Instructions**

Quicksort

Radix sort

Quick (Instr/key) Radix (Instr/key)

Quick (Instr) Radix (Instr)

Quick (Clocks/key) Radix (Clocks/key)

Quick (Clocks) Radix (Clocks)

Quicksort vs. Radix as vary number keys:

**Instructions and Time**
What is proper approach to fast algorithms?

Cache/VM/TLB Summary: #1/3

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- 3 Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity.

Cache/VM/TLB Summary: #2/3

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions: 1) Where can block be placed? 2) How is block found? 3) What block is replaced on miss? 4) How are writes handled?
- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance

Cache/VM/TLB Summary: #3/3

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?