Advanced Memory Hierarchy

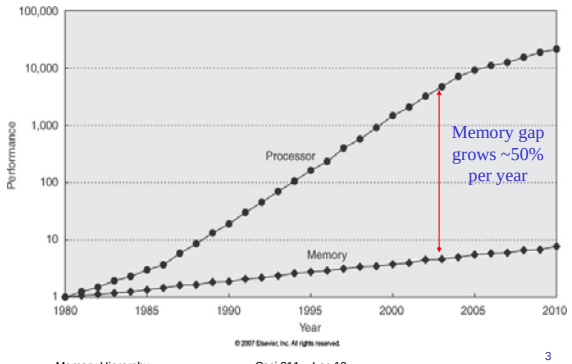
Csci 221 – Computer System Architecture

Lecture 10

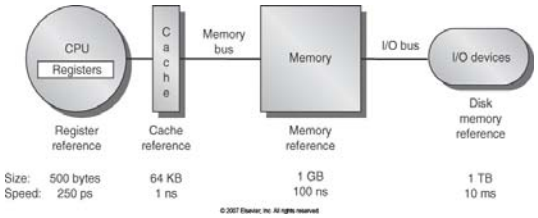
Acknowledgements

- Some slides adopted from EECS 252 at UC Berkeley
  - http://www-inst.eecs.berkeley.edu/~cs252
  - http://www.eecs.berkeley.edu/~pattrsn

Memory Wall



Memory Hierarchy



- Motivation
  - Exploiting locality to provide a large, fast and inexpensive memory

Outline

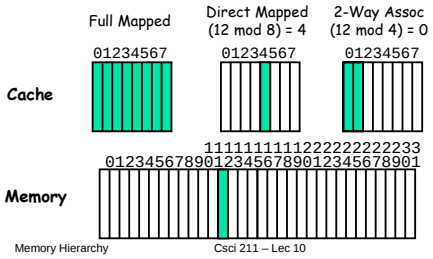
- Cache basics review
- Cache performance optimizations
- Main memory
- Virtual memory

Cache Basics

- Cache is a high speed buffer between CPU and main memory
- Memory is divided into blocks
  - 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)

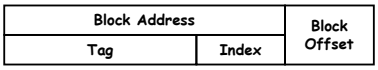
Q1: Block Placement

- Fully associative, direct mapped, set associative
  - Example: Block 12 placed in 8 block cache:
    - Mapping = Block Number Modulo Number Sets



Q2: Block Identification

- Tag on each block
  - No need to check index or block offset
- Increasing associativity ⇒shrinks index ⇒ expands tag



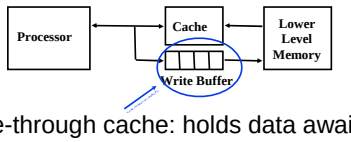
Q3: Block Replacement

- Easy for direct-mapped caches
- Set associative or fully associative:
  - Random
    - Easy to implement
  - LRU (Least Recently Used)
    - Relying on past to predict future, hard to implement
  - FIFO
    - Sort of approximate LRU
  - Note Recently Used
    - Maintain reference bits and dirty bits; clear reference bits periodically; Divide all blocks into four categories; choose one from the lower category
  - Optimal replacement?
    - future

Q4: Write Strategy

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Implement	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Write Buffers



Write-through cache: holds data awaiting write-through to lower level memory

Q. Why a write buffer ?	A. So CPU doesn't stall.
Q. Why a buffer, why not just one register ?	A. Bursts of writes are common.
Q. Are Read After Write (RAW) hazards an issue for write buffer?	A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Outline

- Cache basic review
- Cache performance optimizations
  - Appendix C
  - Ch5.2
- Main memory
- Virtual memory

Cache Performance

- Average memory access time
  - $Time_{total\ mem\ access} = N_{hit} \times T_{hit} + N_{miss} \times T_{miss}$
    $= N_{mem\ access} \times T_{hit} + N_{miss} \times T_{miss\ penalty}$
  - $AMAT = T_{hit} + miss\ rate \times T_{miss\ penalty}$
- Miss penalty: time to replace a block from lower level, including time to replace in CPU
  - Access time: time to lower level(latency)
  - Transfer time: time to transfer block(bandwidth)
- Execution time: eventual optimization goal
  - CPU time = (busy cycles + memory stall cycles) $\times T_{cycle}$
    $= IC \times (CPI_{exec} + N_{miss\ per\ instr.} \times Cycle_{miss\ penalty}) \times T_{cycle}$
    $= IC \times (CPI_{exec} + miss\ rate \times (memory\ accesses / instruction) \times Cycle_{miss\ penalty}) \times T_{cycle}$

Performance Example

- Two caches (assume one clock cycle for hit)
  - I: 8KB, 44% miss rate, 1ns hit time
  - II: 64KB, 37% miss rate, 2ns hit time
  - Miss penalty: 60ns, 30% memory accesses
  - $CPI_{exec} = 1.4$
  - $AMAT_I = 1ns + 44\% \times 60ns = 27.4ns$
  - $AMAT_{II} = 2ns + 37\% \times 60ns = 24.2ns$
  - $CPU\ time_I = IC \times (CPI_{exec} + 30\% \times 44\% \times (60/1)) \times 1ns = 9.32IC$
  - $CPU\ time_{II} = IC \times (CPI_{exec} + 30\% \times 37\% \times (60/2)) \times 2ns = 9.46IC$
  - Larger cache $\Rightarrow$smaller miss rate but longer $T_{hit} \Rightarrow$reduced AMAT but not CPU time

Miss Penalty in OOO Environment

- In processors with out-of-order execution
  - Memory accesses can overlap with other computation
  - Latency of memory accesses is not always fully exposed
  - E.g. 8KB cache, 44% miss rate, 1ns hit time, miss penalty: 60ns, only 70% exposed on average
  - AMAT= 1ns + 44%×(60ns×70%) = 19.5ns

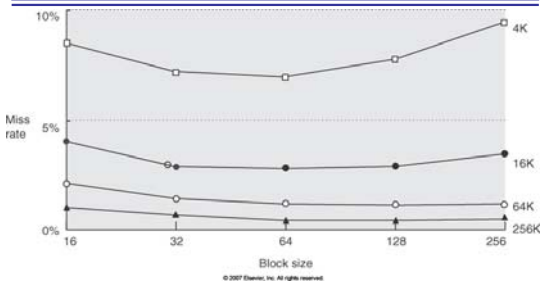
Cache Performance Optimizations

- Performance formulas
  - $AMAT = T_{hit} + miss\ rate \times T_{miss\ penalty}$
  - $CPU\ time = IC \times (CPI_{exec} + miss\ rate \times (memory\ accesses / instruction) \times Cycle_{miss\ penalty}) \times T_{cycle}$
- Reducing miss rate
  - Change cache configurations, compiler optimizations
- Reducing hit time
  - Simple cache, fast access and address translation
- Reducing miss penalty
  - Multilevel caches, read and write policies
- Taking advantage of parallelism
  - Cache serving multiple requests simultaneously
  - Prefetching

Cache Miss Rate

- Three C's
- Compulsory misses (cold misses)
  - The first access to a block: miss regardless of cache size
- Capacity misses
  - Cache too small to hold all data needed
- Conflict misses
  - More blocks mapped to a set than the associativity
- Reducing miss rate
  - Larger block size (compulsory)
  - Larger cache size (capacity, conflict)
  - Higher associativity (conflict)
  - Compiler optimizations (all three)

Miss Rate vs. Block Size



- Larger blocks: compulsory misses reduced, but may increase conflict misses or even capacity misses if the cache is small; may also increase miss penalty

Reducing Cache Miss Rate

- Larger cache
  - Less capacity misses
  - Less conflict misses
    - Implies higher associativity: less competition to the same set
  - Has to balance hit time, energy consumption, and cost
- Higher associativity
  - Less conflict misses
  - Miss rate (2-way, X) ≈ Miss rate(direct-map, 2X)
  - Similarly, need to balance hit time, energy consumption: diminishing return on reducing conflict misses

Compiler Optimizations for Cache

- Increasing locality of programs
  - Temporal locality, spatial locality
- Rearrange code
  - Targeting instruction cache directly
  - Reorder instructions based on the set of data accessed
- Reorganize data
  - Padding to eliminate conflicts:
    - Change the address of two variables such that they do not map to the same cache location
    - Change the size of an array via padding
  - Group data that tend to be accessed together in one block
- Example optimizations
  - Merging arrays, loop interchange, loop fusion, blocking

Merging Arrays

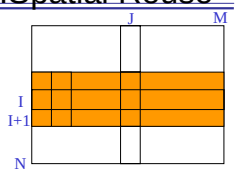
```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
   int val;
   int key;
};
struct merge merged_array[SIZE];
```

- Improve spatial locality
  - If val[i] and key[i] tend to be accessed together
- Reducing conflicts between val & key

Motivation Example:Spatial Reuse

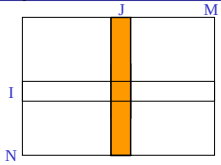
```
DO I = 1, N
   DO J = 1, M
      A(I, J) = A(I, J) + B(I,J)
   ENDDO
ENDDO
```



- Array storage
  - Fortran style: column-major
- Access pattern
  - J-loop: iterate over rows-A(I,J) with I fixed
  - I-loop: iterate over columns
    - Potential spatial reuse
  - Cache misses
    - Could be N*M for A(I,J) if M is large enough

Motivation Example:Spatial Reuse

```
DO J = 1, M
   DO I = 1, N
      A(I, J) = A(I, J) + B(I,J)
   ENDDO
ENDDO
```



- Interchanging I-loop and J-loop
- Access pattern
  - I-loop: iterate over columns-A(I,J) with J fixed
    - Spatial locality exploited: N/b misses given b as the cache line length in words
  - Cache misses
    - N*M/b for A(I,J) assuming a perfect alignment
    - Similar result for B(I,J)

Loop Interchange

- Idea: switching the nesting order of two or more loops

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J)=A(I,J)+B(I,J)
   ENDDO
ENDDO
```

→

```
DO J = 1, M
   DO I = 1, N
      A(I+1,J)=A(I,J)+B(I,J)
   ENDDO
ENDDO
```

  - Sequential accesses instead of striding through memory; improved spatial locality
- Safety of loop interchange
  - Need to preserve true data dependences

Loop Fusion

- Takes multiple compatible loop nests and combines their bodies into one loop nest
  - Is legal if no data dependences are reversed
- Improves locality directly by merging accesses to the same cache line into one loop iteration

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
```

```
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1){
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```


Loop Blocking

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1){
        r = 0;
        for (k = 0; k < N; k = k+1)
            r = r + y[i][k]*z[k][j];};
        x[i][j] = r;
    }
```

- Long-term reuse
  - y[i][k] separated by N k-iterations
  - z[k][j]: ? (N^2 k-iterations)
- How to reduce the no. of intervening iterations?

- Two inner loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity misses a function of N & cache size:
  - $2N^3 + N^2$ words accessed

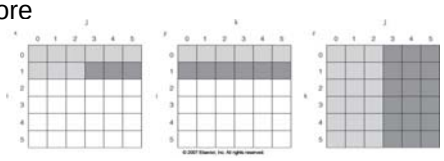
Loop Blocking

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
    for (kk = 0; kk < N; kk = kk+B)
        for (i = 0; i < N; i = i+1)
            for (j = jj; j < min(jj+B-1,N); j = j+1)
            {       r = 0;
                    for (k = kk; k < min(kk+B-1,N); k = k+1)
                        r = r + y[i][k]*z[k][j];};
                    x[i][j] = x[i][j] + r;
            };
```

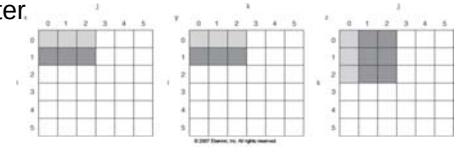
- Divide the matrix into subparts that fit in cache
  - B called blocking factor
- Capacity misses: accessed words from $2N^3 + N^2$ to $2N^3/B + N^2$

Data Access Pattern

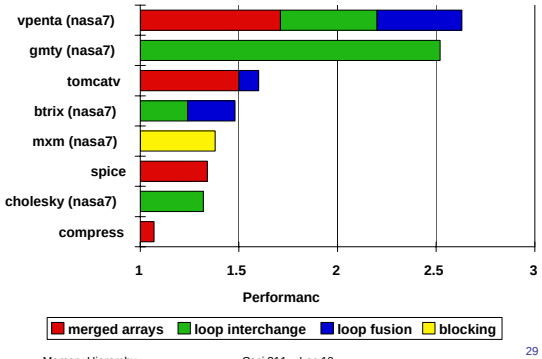
- Before



- After



Performance Impact

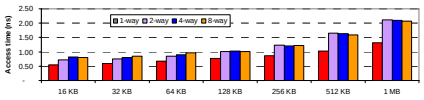


Outline

- Cache basic review
- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
    - Serve multiple accesses simultaneously
    - Prefetching
- Main memory
- Virtual memory

Small and Simple Caches

- Read tag memory and then compare takes time
  - Small cache can help hit time since smaller memory takes less time to index
    - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
    - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
  - Simple ⇒ direct mapping
    - Can overlap tag check with data transmission since no choice



Avoiding Address Translation

- Virtual address→ physical address is necessary if we use virtual memory
  - Translation Look-Aside Buffer (TLB)
    - A small fully-associative cache of mappings from virtual to physical addresses
- Avoiding address translation
  - Index cache using virtual address⇒ only need to translate when cache misses
  - Alternative: virtually-indexed physically-tagged cache
  - Will discuss in depth with virtual memory

Way Prediction

- Set associative cache: check multiple blocks within a set while cache hit
- Way prediction: keep extra bits in cache to predict the “way”, or which block to try on the next cache access
  - Multiplexer is set early to select desired block
  - Perform tag comparison in parallel with reading the cache data
  - Miss ⇒ 1st check other blocks for matches in next clock cycle
    - Accuracy ≈ 85% for a two-way set
    - Drawback: one extra clock cycle of latency when prediction is wrong -- CPU pipeline is hard if hit takes 1 or 2 cycles

Trace Cache

- Targeting instruction cache in Pentium 4
- Trace cache as a buffer to store dynamic traces of the executed instructions
  - Built-in branch predictor
- Cache the micro-ops vs. x86 instructions
  - Decode/translate from x86 to micro-ops on trace cache miss
- Pros and cons
  - Better utilize long blocks
  - Complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
  - Instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

Outline

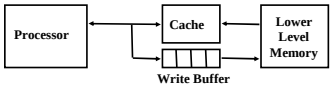
- Cache basic review
- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
    - Serve multiple accesses simultaneously
    - Prefetching
- Main memory
- Virtual memory

Multilevel Caches

- Memory-CPU gap
  - Faster cache to keep pace with CPU?
  - Larger cache to overcome the gap?
- Solution: add another level in the hierarchy
  - L1: small enough to match the CPU speed
  - L2: large enough to capture many accesses
- Average memory access time
  - Hit time$_{L1}$+Miss rate$_{L1}$×Miss penalty$_{L1}$
  - Hit time$_{L1}$+Miss rate$_{L1}$×(Hit time$_{L2}$ + Miss rate$_{L2}$×Miss penalty$_{L2}$)

Giving Priority to Read Misses

- Write Buffer
  - No need to wait for value to go all the way to memory for write instructions to be considered done
    - Place writes in write buffers
    - Latency not critical
- Allow reads to use bus/memory first
  - Need to check write buffer for dependences



Early Restart & Critical Word First

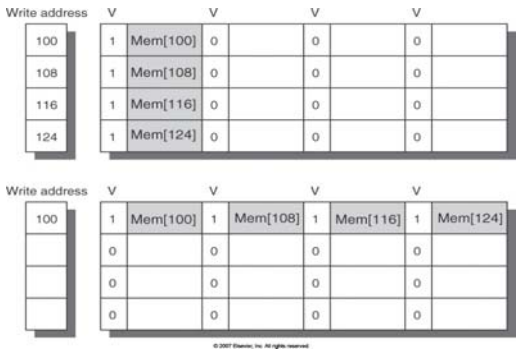
- Don't wait for full block before restarting CPU

- *Critical word first*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - Especially benefit long blocks
- *Early restart*—Fetch in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- Spatial locality: benefits of these two depend on block size and the the likelihood another word of the same block will soon be needed

Merging Write Buffer

- Write buffer to allow processor to continue while waiting to write to memory
- Simple technique to increase buffer efficiency: combine multiple writes to the same block
  - Reduces space: has to stall if buffer full
  - Reduces occupancy: wider block transfer more efficiently
- The Sun T1 (Niagara) processor, among many others, uses write merging

Merging Write Buffer



Outline

- Cache basic review
- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
    - Serve multiple accesses simultaneously
    - Prefetching
- Main memory
- Virtual memory

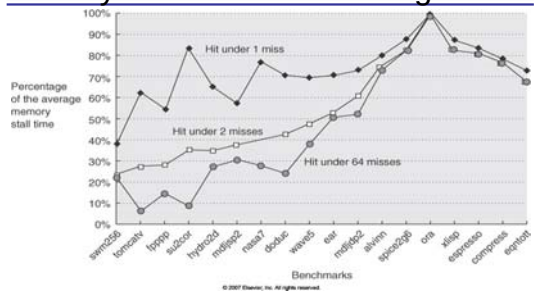
Pipelining Cache

- Multiple cache accesses are overlapped
- Effect
  - Cache hit latency becomes multiple cycles
  - ⇒Increased number of pipeline stages
  - ⇒Greater penalty on mis-predicted branches (pipelined instruction cache)
  - ⇒More clock cycles between the issue of the load and the use of the data
  - High bandwidth, fast clock cycle time, but slow hits
- Instruction cache access pipeline stages:
  - Pentium:1
  - Pentium Pro through Pentium III: 2
  - Pentium 4: 4

Non-Blocking Caches

- Allow cache to continue under misses: overlap miss with other useful tasks
  - Lockup-free cache, non-block cache
  - Hit under miss or even miss under miss
- Hardware support
  - Requires Full/Empty bits on registers + MSHR(Miss Status /Handler Registers) queue for outstanding memory requests
  - Memory that supports multiple requests (if miss under miss allowed): multi-bank memories
  - Significantly increases the complexity of the cache controller; need to keep track of dependencies and ensure precise exceptions
  - Pentium Pro allows 4 outstanding memory misses

Memory Stall vs Non-blocking Cache



- Ratio of average memory stall time over a blocking cache
  - Hit-under 64 misses: one miss per register
  - One miss enough for INT, adding a second helps FP

Multiple Banks

- Divide cache into independent banks that can support simultaneous accesses
  - Sun Niagara L2: 4 banks, AMD Opteron L2: 2 banks
- Banking works best when accesses naturally spread themselves across banks ⇒ mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is "sequential interleaving"
  - Spread block addresses sequentially across banks
  - E.g., if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; …

Prefetching

- Fetching instruction or data in advance
  - Mechanism: hardware or software
    - Complexity vs. flexibility vs. overhead
  - Destination: cache, register, buffer
    - Pollution, register pressure vs. complexity
  - Heuristics: stride or correlation
    - Complexity vs. effectiveness
  - Issues
    - Exceptions, coherence, forwarding
    - Relies on having extra memory bandwidth that can be used without penalty

Hardware Prefetching

- Sequential instruction prefetching
  - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block
  - Requested block placed in instruction cache, and prefetched block placed into instruction stream buffer
- Sequential data prefetching
  - When miss on X, fetch X+1, … X+n blocks into FIFO buffer
  - Can extend to strided prefetch: X+I, X+2i, …
  - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages

Software Prefetching

- Where to load
  - Load data into register (HP PA-RISC loads)
  - Cache prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Exception behavior
  - Faulting: prefetched address causes an exception for virtual address faults and protection violations
  - Non-faulting: prefetched address does not cause an exception, if it does it becomes a no-op
- Overhead concern
  - Issuing prefetch instructions takes time ⇒ saving need to cover overhead
  - Concentrate on pre-fetching data that are likely to be cache misses

Prefetching Example

```
for (i=0;i<3;i=i+1)
   for (j=0;j<100;j=j+1)
        a[i][j]=b[j][0]*b[j+1][0];
```

How many misses for both cases? How many prefetches?

```
for (j=0;j<100;j=j+1){
        prefetch(b[j+7][0]);
        prefetch(a[0][j+7]);
        a[0][j]=b[j][0]*b[j+1][0]; }
for (i=1;i<3;i=i+1)
        for (j=0;j<100;j=j+1){
                prefetch(a[i][j+7]);
                a[i][j]=b[j][0]*b[j+1][0];
        }
```

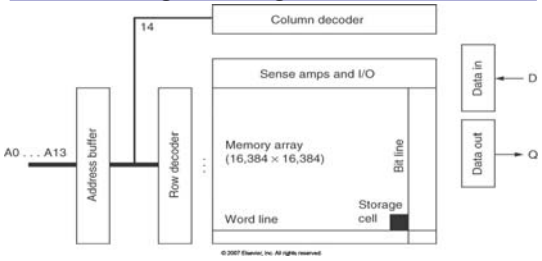

Outline

- Cache basic review
- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
    - Serve multiple accesses simultaneously
    - Prefetching
  - Figure C.17 and 5.11 of textbook
- Main memory
- Virtual memory

Main Memory Background

- Main memory performance
  - Latency: cache miss penalty
    - Access time: time between request and word arrives
    - Cycle time: time between requests
  - Bandwidth: multiprocessors, I/O,
    - large block miss penalty
- Main memory technology
  - Memory is DRAM: Dynamic Random Access Memory
    - Dynamic since needs to be refreshed periodically
    - Requires data written back after being read
    - Concerned with cost per bit and capacity
  - Cache is SRAM: Static Random Access Memory
    - Concerned with speed and capacity
      - Size: DRAM/SRAM - 4-8
      - Cost/Cycle time: SRAM/DRAM - 8-16

DRAM Logical Organization



- Row address strobe(RAS)
  - Sense amplifier and row buffer
- Column address strobe(CAS)

DRAM Performance Improvements

- Fast page mode: reuse row
  - Add timing signals that allow repeated accesses to row buffer without another row access time
  - Each array buffer 1024 to 2048 bits for each access

DRAM Performance Improvements

- Synchronous DRAM (SDRAM)
  - Add a clock signal to DRAM interface, so that the repeated transfers would not bear overhead to synchronize with DRAM controller
- Double Data Rate (DDR SDRAM)
  - Transfer data on both the rising edge and falling edge of the DRAM clock signal ⇒ doubling the peak data rate
  - DDR2 lowers power by dropping the voltage from 2.5 to 1.8 volts + offers higher clock rates: up to 400 MHz
  - DDR3 drops to 1.5 volts + higher clock rates: up to 800 MHz
- Improved bandwidth, not latency

Outline

- Cache basic review
- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
    - Serve multiple accesses simultaneously
    - Prefetching
  - Figure C.17 and 5.11 of textbook
- Main memory
- Virtual memory

Memory vs. Virtual Memory

- Analogy to cache
  - Size: cache << memory << address space
  - Both provide big and fast memory - exploit locality
  - Both need a policy - 4 memory hierarchy questions
- Difference from cache
  - Cache primarily focuses on speed
  - VM facilitates transparent memory management
    - Providing large address space
    - Sharing, protection in multi-programming environment

Four Memory Hierarchy Questions

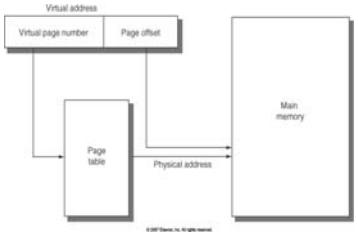
- Where can a block be placed in main memory?
  - OS allows block to be placed anywhere: fully associative
    - No conflict misses; simpler mapping provides no advantage for software handler
- Which block should be replaced?
  - An approximation of LRU: true LRU too costly and adds little benefit
    - A reference bit is set if a page is accessed
    - The bit is shifted into a history register periodically
    - When replacing, find one with smallest value in history register
- What happens on a write?
  - Write back: write through is prohibitively expensive

Four Memory Hierarchy Questions

- How is a block found in main memory?
  - Use page table to translate virtual address into physical address

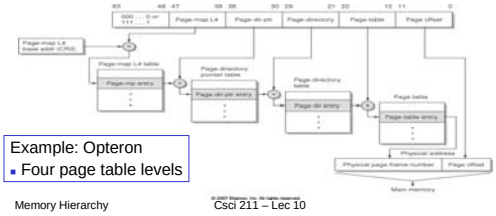
• 32-bit virtual address, page size: 4KB, 4 bytes per page table entry, page table size?

• $(2^{32}/2^{12})\times 2^2 = 2^{22}$ or 4MB



Fast Address Translation

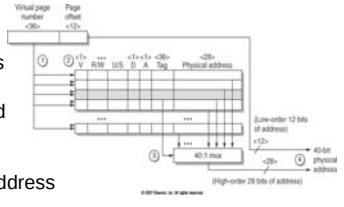
- Motivation
  - Page table is too large to be stored in cache
    - May even expand multiple pages itself
  - Multiple page table levels
- Solution: exploit locality and cache recent translations



Example: Opteron
- Four page table levels

Fast Address Translation

- TLB: translation look-aside buffer
  - A special fully-associative cache for recent translation
  - Tag: virtual address
  - Data: physical page frame number, protection field, valid bit, use bit, dirty bit
- Translation
  - Send virtual address to all tags
  - Check violation
  - Matching tag send physical address
  - Combine offset to get full physical address

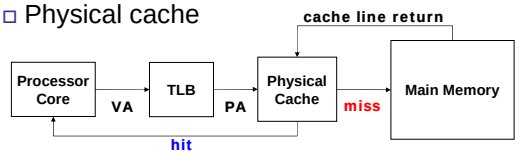


Virtual Memory and Cache

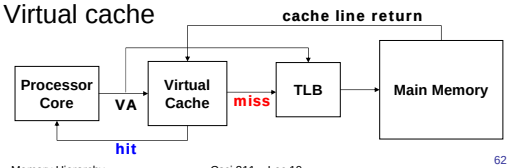
- Physical cache: index cache using physical address
  - Always address translation before accessing cache
  - Simple implementation, performance issue
- Virtual cache: index cache using virtual address to avoid translation
  - Address translation only @ cache misses
  - Issues
    - Protection: copy protection info to each block
    - Context switch: add PID to address tag
    - Synonym/alias -- different virtual addresses map the same physical address
      - Checking multiple places, enforce aliases to be identical in a fixed number of bits (page coloring)
    - I/O: typically uses physical address

Virtual Memory and Cache

- Physical cache



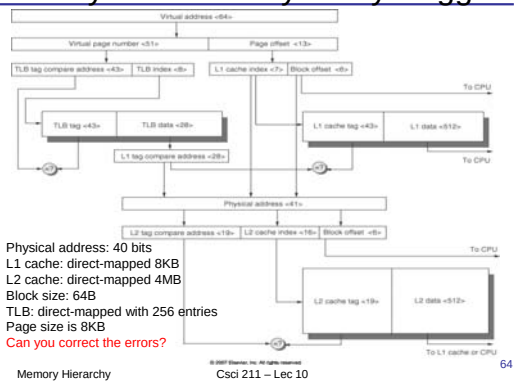
- Virtual cache



Virtually-Indexed Physically-Tagged

- Virtually-indexed physically-tagged cache
  - Use the page offset (identical in virtual & physical addresses) to index the cache
  - Associate physical address of the block as the verification tag
  - Perform cache reading and tag matching with the physical address at the same time
  - Issue: cache size is limited by page size (the length of offset bits)

Virtually-Indexed Physically-Tagged



Physical address: 40 bits
L1 cache: direct-mapped 8KB
L2 cache: direct-mapped 4MB
Block size: 64B
TLB: direct-mapped with 256 entries
Page size is 8KB
Can you correct the errors?

Advantages of Virtual Memory

- Translation
  - Program can be given a consistent view of memory, even though physical memory is scrambled
  - Only the most important part of program ("Working Set") must be in physical memory
- Protection
  - Different threads/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
- Sharing
  - Can map same physical page to multiple users

Sharing and Protection

- OS and architecture join forces to allow processes to share HW resources w/o interference
- Architecture support
  - User mode and kernel mode
  - Mechanisms to transfer between user/kernel modes
  - Read-only processor state
    - Users shouldn't be able to assign supervisor privileges, disable exceptions, or change memory protection
  - Therefore: protection restriction to each page and page table entry
- Related topic: virtual machines

Introduction to Virtual Machines

- VMs developed in late 1960s
  - Remained important in mainframe computing over the years
  - Largely ignored in single user computers of 1980s and 1990s
- Recently regained popularity due to
  - Increasing importance of isolation and security in modern systems
  - Failures in security and reliability of standard OS
  - Sharing of a single computer among many unrelated users
  - Dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable

What is a Virtual Machine (VM)?

- Process level VMs provide application binary interface to applications
  - Provide support to run a single program, which means that it supports a single process
  - Java VM
- (Operating) system level VMs provide a complete system level environment at binary ISA
  - Present illusion that VM users have entire computer to themselves, including a copy of OS
  - Single computer can run multiple VMs, and can support a multiple, different OSes
    - With a VM, multiple OSes all share HW resources
  - J.E. Smith and Ravi Nair, "Virtual Machines: Architectures, Implementations and Applications", MKP, 2004
- An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine -- it cannot break out of its virtual world.

Virtual Machines Basic

- Virtualization software placed between the underlying machine and software
  - Underlying HW platform is called the host, and its resources are shared among the guest VMs

Virtual Machine Monitors (VMMs)

- Virtual machine monitor (VMM) or hypervisor is the software that supports VMs
  - Presents a SW interface to guest software
  - Isolates state of guests from each other, and
  - Protects itself from guest software (including guest OSes)
- Virtualization process involves
  - Mapping of virtual resources to real hardware resources, e.g. registers and memory
    - Resource time-shared, partitioned, or emulated in software
  - Using real machine instructions to emulate the virtual machine ISA
- VMM is much smaller than a traditional OS
  - Isolation portion of a VMM is ≈ 10,000 lines of code

VMM Supporting Multiple OS

- VMM (Virtual Machine Monitor) provides replication and resource management
  - Benefits: flexibility, utilization, isolation
  - VMM intercepts and implements all the guest OS's actions that interact with HW in a transparent way
    - Similar to what an OS does for processes

Implementation of System VMs

- VMM runs in the most highly privileged mode while all the guests run with less privileges
  - VMM must ensure that guest system only interacts with virtual resources
- The emulation of virtual ISA using host ISA is performance critical
  - Unlike real machine implementation, guest and host are often specified independently⇒a good match is difficult to find
  - ISA support: if plan for VM during design of ISA, easier to improve speed and code size
    - ISA is virtualizable; the system is co-designed VM
    - E.g. IBM Daisy processor and Transmeta Crusoe

VMM Overhead

- Depends on the workload
- User-level processor-bound programs (e.g., SPEC CPU) have zero-virtualization overhead
  - Runs at native speeds since OS rarely invoked
- I/O-intensive workloads ⇒ OS-intensive ⇒ execute many system calls and privileged instructions ⇒ can result in high virtualization overhead
  - For system VMs, goal of architecture and VMM is to run almost all instructions directly on native hardware
- If I/O-intensive workload is also I/O-bound ⇒ low processor utilization since waiting for I/O ⇒ processor virtualization can be hidden ⇒ low virtualization overhead

Requirements of a VMM

- Guest software should behave on a VM exactly as if running on the native HW
  - Except for performance-related behavior or limitations of fixed resources shared by multiple VMs
- Guest software should not be able to change allocation of real system resources directly
- Hence, VMM must control ≈ everything even though guest VM and OS currently running is temporarily using them
  - Access to privileged state, address translation, I/O, exceptions and interrupts, …

Requirements of a VMM

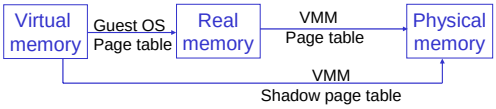
- VMM must be at higher privilege level than guest VM, which generally run in user mode
  - Execution of privileged instructions handled by VMM
- E.g., Timer interrupt: VMM suspends currently running guest VM, saves its state, handles interrupt, determine which guest VM to run next, and then load its state
  - Guest VMs that rely on timer interrupt provided with virtual timer and an emulated timer interrupt by VMM
- Requirements of system virtual machines are ≈ same as paged-virtual memory

ISA Support for Virtual Machines

- If plan for VM during design of ISA, easy to reduce instructions executed by VMM, speed to emulate
  - ISA is virtualizable if can execute VM directly on real machine: VMM retain ultimate control of CPU
  - Since VMs have been considered for desktop/PC server apps only recently, most ISAs were created ignoring virtualization, eg. 80x86, most RISC arch.
- VMM must ensure that guest system only interacts with virtual resources
  - Conventional guest OS runs as user mode program on top of VMM
  - Guest OS access/modify information related to HW resources via a privileged instruction⇒trap to VMM

Impact of VMs on Virtual Memory

- Virtualization of virtual memory
  - Each guest OS in every VM manages its own set of page tables
- VMM separates real and physical memory
  - Makes real memory a separate, intermediate level between virtual memory and physical memory
  - Guest OS maps virtual memory to real memory via its page tables, and VMM page tables map real memory to physical memory
  - Shadow page table maps directly from the guest virtual address space to the physical address space of the hardware



Impact of VMs on Virtual Memory

- IBM 370 architecture added additional level of indirection that is managed by the VMM
  - Guest OS keeps its page tables as before, so the shadow pages are unnecessary
  - (AMD Pacifica proposes same improvement for 80x86)
- To virtualize software TLB, VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM
  - Any instruction that accesses the TLB must trap
  - TLBs with Process ID tags support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch

Impact of I/O on Virtual Memory

- I/O most difficult part of virtualization
  - Increasing number of I/O devices attached to the computer
  - Increasing diversity of I/O device types
  - Sharing of a real device among multiple VMs
  - Supporting many device drivers that are required, especially if different guest OSes are supported on same VM system
- Give each VM generic versions of each type of I/O device driver, and let VMM to handle real I/O
  - Method for mapping virtual to physical I/O device depends on the type of device: virtual disks, virtual network

Summary

- Cache performance optimizations
  - Reducing cache miss rate
  - Reducing hit time
  - Reducing miss penalty
  - Parallelism
- Main memory
  - Basic technology and improvement
- Virtual memory
  - Fast translation
  - Sharing and protection
  - Additional reference: [Smith and Nair, 04]