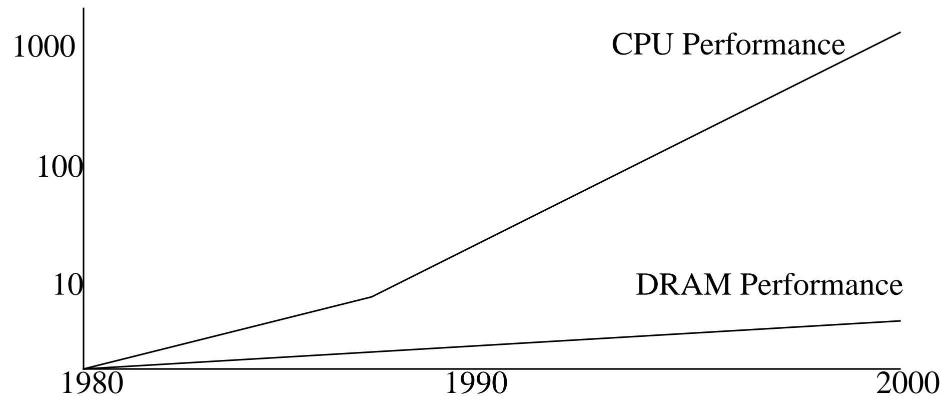
Topic 5: Memory Hierarchies

- **Review: Up to now, concentrating on processor design**
- Reflects history of computer architecture--focused almost exclusively on processor
- As processors have gotten faster, other parts of the system haven't kept up



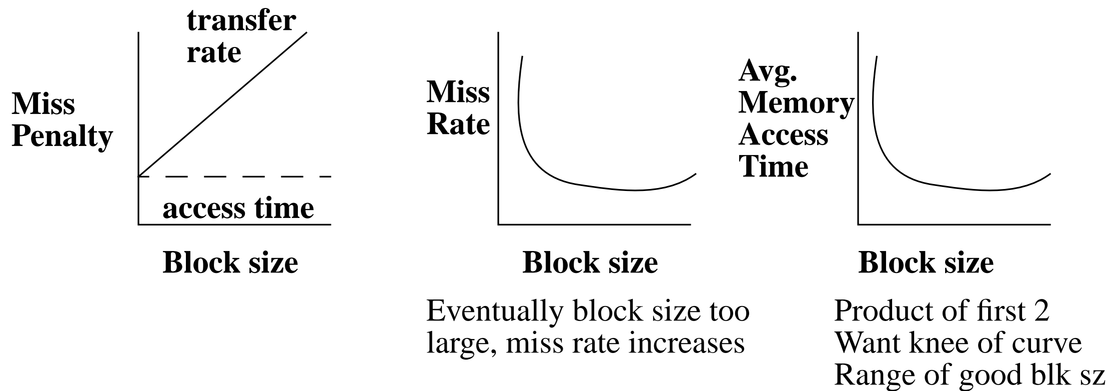
- Need better memory systems to compensate for the gap
- Memory hierarchy: idea that have different levels of memory
- (Draw hierarchy pyramid: cache, memory, solid state disk, magnetic disk)
- Higher levels are faster, smaller, more expensive
- Lower levels are bigger, slower, cheaper
- Inclusive: things at higher levels of hierarchy also exist at bottom level
- 1980: No cache on microprocessor
- 1984-1987: Started to ship microporcessors with caches
- 1995: Alpha 21164 processor includes 2-level cache
- Of 9.3 million transistors, devote 60% to cache

- Principles that make hierarchy work:
- **Locality**
- Temporal: referenced again soon
- Spatial: nearby items referenced soon
- **Smaller is faster**

- **Definitions**
- Upper levels: closer to processor (smaller, faster)
- Lower: further from processor
- **Hit: found data at upper level**
- **Miss: did not find data at upper level**
- **Hit or Miss Rate: fraction of accesses that are hits/misses**
    - Hit rate is so high, usually talk about miss rate
    - Fallacy: think that miss rate is the only important measure
    - Analogous to only looking at instruction count to evaluate CPU performance
    - Average memory access time is more important
- **Hit time: time to access the upper level of the hierarchy,**
    - including time to determine that hit occurred
- **Miss penalty: Additional time to service a miss**
    - Replace a block in upper level with a block from lower level
    - Access time: f(latency to access lower level)
    - Time to trasnfer block: f(bandwidth of upper, lower levels)
    - units: nanoseconds or clock cycles
    - **Average memory access time**
    - **Hit time + Miss Rate * Miss Penalty**

- **Typical cache parameters**
- Caches: first level of memory hierarchy
- Typical parameters:
- Block: minimum unit of transfer between levels of hierarchy
    - Sometimes called "line"
    - 4 to 128 bytes
- Hit time: perhaps one clock cycle
- Miss penalty: 8 to 32 cycles
- Miss rate: 1 to 20 %
- Cache size: 1 to 256 KBytes

- **Increasing block size generally**
  - **increases miss penalty and**
  - **decreases miss rate**



- Implications for CPU
  - Want a fast hit check since done on every memory access
  - Memory access time is unpredictable:
  - 10s of clock cycles -- will probably want to wait (cache or main memory access)
  - 1000s of clock cycles: traditionally interrupt and do something else (disk access)
  - As gap grows between processor and memory performance, looking for more things to do (multithreading)

- **Four questions characterize memory hierarchies**

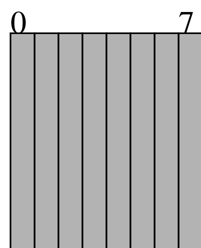
1. **Where can a block be placed in the upper level (Block placement)?**
2. **How is a block found if it is in the upper level (Block identification)?**
3. **Which block should be replaced on a miss (Block replacement)?**
4. **What happens on a write (Write strategy)?**

1. **Where can a block be placed in the upper level (Block placement)?**

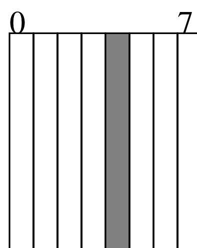
- Example: Want to place block 12 of memory into an 8-block cache
- Fully associative: block can go anywhere in cache
  - to find it later, will have to search every entry
- Direct-mapped: only one place it can go:

- block address modulo number of cache blocks

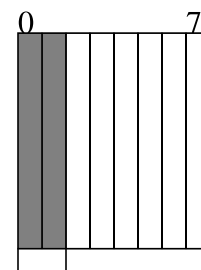
- Hybrid: Set-associative:
    - cache divided into groups of blocks called sets
    - Which set the block is in is deterministic (block address mod number sets)
    - Can be in any location within the set
    - Set size = n: n-way set-associative
- Set-associative is a generalization of other two
    - Fully associate: one set with set size = cache size
    - Direct-mapped: number of sets = number of cache blocks, set size = 1



Fully Associative
Block can go anywhere



Direct-mapped
12 mod 8 = 4



2-way Set Associative
12 mod 4 = 0
Set 0

- **Fully associative caches are very expensive**
    - lots of logic to compare all entries simultaneously
    - tend to be small, used for only highest performance requirements
- **Most processors will use direct-mapped or 2-way set-associative**

2. How is block found if it is in upper level (Block identification)?

- Problem: how to dyou know what memory locations are stored in each cache block?
    - Multiple memory blocks map to same cache block?
- Associate a tag with each cache block that
    - contains part of the memory address of the stored block
- Determine appropriate set

- Then do a search of all the tags within the set to check for match

<table>
<tr><td colspan="2">Block Address</td><td rowspan="2">Block offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

- First part of address will be tag in cache
- Index portion used to determine the set
- Offset gives block within set
- Draw picture of Figure 6.4 at this point!
- Show index, tag, multiplexor
- Compare all tags in parallel; otherwise, too slow
- Increased associativity takes more comparison hardware
- Valid bit says if entry contains valid data

3. Which block replaced on miss (Block replacement)?

- Easy for direct-mapped (only one choice)
- Two options for set-associative and fully-associative
- Random: works surprisingly well in hardware!
    - Often use pseudo-random numbers to make reproducible
    - In O.S., often want to avoid this -- unthinkingly replace something that may have a lot of locality
    - In hardware, may be more important to get a decent guess quicly than to get best guess more slowly
- LRU: replace the least-recently used block
    - exploits temporal locality
    - requires keeping state on the last use
    - more complicated HW
- Effectiveness: use numbers from Dave's slides

4. What happens on a write (Write strategy)?

- Reads are the common case; want to optimize these
- All instructions accesses and all data loads are reads
- Easy to make reads fast: read block at same time tag is read and compared

- Writes are more complicated, even on a hit
- Must check tag for cache hit
    - (cannot occur in parallel with writing, so writes take longer than reads)
    - modify location
    - write block
- **Must also make sure all relevant levels of memory are updated**
- Two main options:
- **Write-through:**
    - Information is written to both the cache block and the block in lower-level memory
- **Write-back:**
    - Information written only to cache block
    - Modified cache block is written to main memory only when it is replaced
    - Bit associated with block: is it clean or dirty?
    - Don't need to write back clean blocks
- **Pros and cons:**
- **Write-through:**
    - slower writes but read misses don't cause memory writes
- **Write-back:**
    - writes occur at cache speeds;
    - read misses can cause memory writes;
    - reduces memory traffic if values are overwritten before they are written back
- **Write Buffer**
    - Additional optimization for write-through:
    - write buffer that holds waiting writes;
    - allows overlap of processor execution of memory updates
- **Options on miss:**
- **write allocate (fetch on write):**
    - load a block on a miss, as in read miss;
    - most common with write-back
    - idea: want cache to have data if rewritten so avoid doing earlier write
- **No write allocate: (write around)**
    - modify block in lower level, don't load into cache;

    - most common with write-through
    - idea: on subsequent write, have to update lower level anyway, so why cache?

- **Instruction & data caches vs. merged caches**
- On some cycles, need both instructions and data
- If a single (unified) cache, get structural hazard unless multi-ported
- Often, separate instruction and data caches
- Instruction caches extremely effective

TABLE 1. Miss Rate

Size	Instruction Cache	Data Cache	Unified Cache
1KB	3.06%	24.61%	13.34%
8KB	1.10%	10.19%	4.57%
32KB	0.39%	4.82%	1.99%
128KB	0.02%	2.88%	0.95%

- Advantage of unified cache: not a fixed barrier between instructions and data
    - can fill as needed

- Example using Figure 6.7
    - 2-way set associative version of Alpha data cache
    - 34 bit address
    - 256 blocks total (128 sets)
    - 7-bit index to choose set -- >> 128 sets total
    - 2 blocks per set -->> 256 blocks total
    - 32-byte blocks -->> 8 KBytes altogether
    - 2:1 MUX selects between blocks set based on tags matching incoming address
    - Index (7 bits) plus two bits of offset: select 8-B data (64 bits) from 32-B block
    - Write through, with 4 block write buffer (no write allocate)
    - Write hit: write back to cache location and write buffer
    - Compare to addresses already in buffer; do write-merging if possible to conserve space in buffer
    - On read miss, bring data from lower level
    - On write miss, write around cache to write buffer

- **Classifying misses (the 3 C's)**
- **Compulsory**
    - First access to a block is not in the cache
    - Also called cold start misses or first reference misses
    - Intuition: the misses you would get in an infinite sized cache
- **Capacity**
    - In addition to compulsory misses
    - Real caches have limited size
    - Misses will occur when all blocks of program can't be contained in cache
    - Blocks will be discarded that must later be retrieved
    - Intuition: have limited size of cache but not block placement; finite, fully associative
- **Conflict**
    - In addition to compulsory, capacity misses
    - If set-associative or direct-mapped
    - Blocks may map to same set
    - May discard and later have to retrieve blocks because of these conflicts
    - Also called collision misses or interference misses
    - Intuition: limited size cache and N-way (not infinite) associativity
- **Note: these are helpful intuitions, not set in stone**
- As cache parameters change, misses may change categories
- **Graph (Figure 6.9)**
    - Horizontal axis: cache size (KB)
    - Vertical axis: resulting miss rate
    - At bottom of graph, almost invisible, are compulsory misses -- small portion
    - Capacity misses decrease as cache size gets larger
    - Direct-mapped is always significantly worse than 2-way set associative
    - Other associativities: lower gains as cache gets larger
    - Why? More sets; and spreading out data, so fewer conflict misses
    - Trick: Comparing direct-mapped and 2-way set-associative
    - Plot horizontal, vertical lines: always see that size X 2-way set associative cache has about same performance as size 2X direct-mapped cache
    - True over a wide range of programs

- Second graph: types of misses as percent:
- As caches get bigger, cumpolsory misses become bigger portion
- Direct-mapped always significantly worse than set-associative

- **Improving cache performance**
- Will look at all three portions of memory access time equation
- Avg. mem access time = hit time + miss rate * miss penalty

Three options to reduce access time:

1. **reduce miss rate**
2. **reduce miss penalty**
3. **reduce hit time**

- We will look at reducing each of these in isolation
- Remember that reducing one may increase another
- Ultimate measure is the AMAT average memory access time

1. **Reduce misses via Larger Block size**

- **Show figure 5.11**
- **Larger blocks take advantage of spatial locality -- reduce compulsory misses**
- Reduce number of blocks in cache: may increase conflict misses
    - At very small cache sizes, increasing block size increases miss rates

2. **Reduce misses via Higher Associativity**

- **2:1 Cache Rule**
    - Miss rate of DM cache of size N approx Miss Rate of FA cache of size N/2
- **BUT beware!!! time is the only real measure**
- **May increase cycle time by increasing the associativity**
    - Need more time to do extra comparisons (multiplexors)
- **(Mark Hill) for an internal cache, have to increase cycle time 2% for 2-way vs. 1-way**
    - Show Figure 5.14: for many cases, expecially as caches get large, even for a 2% increase in cycle time, the increased associativity increases AMAT
    - As a result, first-level caches are often direct-mapped

3. Reducing misses via Victim Cache

- **Add a small, fully associative cache between a cache and its refill path**
  - Combine fast hit time of direct-mapped, avoid conflict misses
- **Hold data discarded from cache on miss**
  - Checked on subequent misses to see if they have desired data
  - If found, victim block and cache block are swapped
  - 1 to 5 entries effective at reducing conflict misses
  - 4-entry cache removed 20% to 95% of conflicts for 4KB direct mapped data cache

4. Reducing Misses via Pseudo-Associativity

  - (We will not cover this technique)
  - Two locations per block, but check them sequentially
  - In other words, divide cache in half
  - If in first half of cache, get a "fast hit"
  - Otherwise, "slow hit" or "pseudo hit"
  - Goal: combine:
  - **fast hit time of direct-mapped cache,**
  - **lower conflict misses of 2-way set associative**
  - Disadvantage: CPU pipeline is hard if hit takes variable number of cycles
  - Works better for caches not tied directly to processor (second-level caches)

5. Reducing misses by HW prefetching of instructions and data

- **Idea: get items before they are needed**
  - e.g., instruction prefetch: Alpha 21064 fetches 2 blocks on a miss
- **Extra block placed in stream buffer; stream buffer checked on a miss**
  - Data prefetch: use data stream buffers
  - Effectiveness: Jouppi: 1 data stream buffer got 25% misses from 4KB cache
  - 4 data stream buffers got 43%
- **Disadvantage: prefetching relies on ability to use extra memory bandwidth**

6. Reducing Misses by Compiler-Controlled Prefetching

  - Compiler inserts prefetch instructions to request data before they are needed

- **Register or Data prefetch:**

  Load data into registers
  (HP PA-RISC): load instruction gives hints -- likely to reuse instructions/data

- **Cache prefetch**
  - load into cache but not registers
  - (MIPS IV, PowerPC, SPARC)

  looks like normal load, but actually loads data into cache only

- **Would like to be able to discard prefetch if it would cause a page fault**
- **Non-faulting prefetches: turn into NOOPs if they would cause an exception**
- Issuing prefetch inscrutions takes time

  Is cost of prefetch less than savings in reduced misses?

7. Reducing misses by compiler optimizations

- Instructions

Reorder procedures in memory to reduce misses

Profiling to look at conflicts; very effective (reduce missses 75% on 8KB dm cache)

- Data
- One technique: Merging Arrays (show slide)
- Another technique: Loop interchange: change nesting of loops to access data in order stored in mem.

Example: Before:

```
for (k=0; k<100; k=k+1)
    for (j=0; j<100; j=j+1)
        for (i=0; i<5000; i=i+1)
            x[i][j] = 2 * x[i][j];
```

After:

```
for (k=0; k<100; k=k+1)
    for (i=0; i<5000; i=i+1)
        for (j=0; j<100; j=j+1)
            x[i][j] = 2 * x[i][j];
```

Allows sequential accesses instead of striding through memory every 100 words

- Loop fusion: merge 2 independent loops that have same loopign and some variable overlap
- Blocking: Improve temporal locality by accessing blocks of data repeatedly vs. going down whole columns or rows

- **Reducing the Miss Penalty**

1. Reduce penalty of read misses by giving reads priority over writes

- **Write buffers help performance (for both write-through and write-back)**
    - Danger: on read miss, correct block may be in write buffer (not yet in memory)
    - Slow: Wait for write buffer to drain, then read data from memory
    - Fast: Check write buffer; if data is there, can service miss from write buffer
    (Simple logic: let a pending write go to memory, then service miss from memory)

- **Write-back:**
    - Students: what happens on a read miss?
    - Need to bring in a new block, kick out an old block
    - May need to write back a dirty block
    - Logically, must write dirty block to memory before doing read
    - To reduce miss penalty:
    (1) copy dirty block to a write buffer, (2) read new block and put in cache,
    (3) perform write of dirty block from write buffer
    - CPU stall shorter since it can restart as soon as read completes

2. Subblock placement to reduce miss penalty

- **Up to now: Block is smallest unit of transfer between memory and cache**
    - For large blocks, waiting to transfer whole block may be slow
- **Some implementations divide block into subblocks**
    - Draw Figure 5.21 showing sub-blocks
    - Single tag per block
    - Valid bit per sub-block
- Allow one subblock to be fetched on a miss
- Increased complexity

3. Reducing Miss Penalty with Early Restart and Critical First Word

- Similar idea: on a read miss, don’t wait for entire block to be loaded to restart the CPU
    - **Fetch entire block on a read miss, but CPU is waiting for a particular word**
- **Early restart**
    - As soon as the requested word arrives, send it to the CPU and restart

- **Critical word first**
    - Fetch all words in the block but not first to last
    - Request the necessary word FIRST from memory
    - Send to CPU as soon as it arrives
    - Restart CPU while concurrently fetching the remaining words in the block
    - Complex
    - Expensive: requires a memory divided into banks and control that allows you to choose which bank will send data first
- **Schemes generally useful only for large blocks (miss penalty especially high)**
- Spatial locality still holds, so not clear it would offer that much benefit
- Might stall shorter time on the first requested word in block, but may stall on next

4. Non-blocking Caches to Reduce Stalls on Misses

- Idea: just because one CPU request misses, don't stop servicing other requests
- Continue issuing instructions and fetching data (out-of-order issue?)
- **Non-blocking or lockup-free cache allows data cache to continue to supply cache hits during or "under" a miss**
- "Hit under miss": reduces effective miss penalty by being helpful during a miss
- "Hi under multiple miss" or "miss under miss" -- allow overlap of multiple misses
    - So, can have multiple _memory accesses_ (not just cache accesses)
- Implies a memory that supports multiple accesses: banks
- Increases complexity of cache controller
- Pentium Pro allows 4 outstanding memory misses
- Figure 5.22
    - Graph shows that supporting 1 miss (hit under miss) is very effective
    - Supporting 2 misses (hit under 2 miss) is even more effective
    - After that, up to baseline of 64 misses, not much payoff

5. Second-Level Caches

- Dilemma:
    - Performance gap between CPU and main memory
    - Should cache be small and fast to keep up with CPU: maybe direct-mapped
    - Or large to avoid misses that have slow access to main memory: set-associative
- Want both

- **Solution: add another cache to the hierarchy**
    - Small, fast cache next to CPU; larger, somewhat slower second-level cache
    - Draw picture
- **Performance: L1=first level, L2=second level**
    - Average access time = hit time$_{L1}$ + miss rate$_{L1}$ * miss penalty$_{L1}$
    - Miss penalty$_{L1}$ = hit time$_{L2}$ + miss rate$_{L2}$ * miss penalty
- Definitions
- **Local miss rate (a bad term)**
    - Misses in this cache divided by the total number of memory accesses to this cache
    - e.g., miss rate of L2
    - problem: have already gone through filter of L1 cache
    - local miss rate of L2 will vary with size of L1 cache
- **Global miss rate (better)**
    - Misses in this cache divided by total number of memory accesses generated by CPU
    - Global miss rate of second level cache is (miss rate$_{L1}$ * miss rate$_{L2}$)
- **Earlier discussed cache optimizations to reduce miss rate: apply to second level!**
- **Second-level cache characterized by fast hits, few misses**

- **Reducing hit times**

1. Small and simple caches

- Remember: small, direct-mapped, on-chip cache is very fast
- Small: less distance for signals to propagate
- Adding size, associativity slow it down -- more comparators, MUXes

2. Avoiding translation of virtual addresses

- Will discuss virtual memory shortly
- Virtual address space(s) map onto physical space
- Must translate virtual address to a physical address
- Translation takes time
- **To eliminate translation step, may have virtually-addressed caches**

- Why doesn't everyone build them this way?
- Potential problems:

1. **On process switch, virtual addresses refer to different physical addresses**

- **Options:**
  - **Flush cache -- otherwise, may map to wrong place**
  - **Add process identifier tag field: cache hit must also match PID**
  - Figure 5.26 shows impact: using PIDs much better than default flush

2. **Aliases (synonyms): 2 virtual addresses may map to same physical address**

  - **O.S. and user program may use 2 virtual addresses for same physical address**

  Problem: may map into a virtual cache in two separate locations

  Change one and the other is inconsistent and out of date

  To avoid this, must force aliases to map to a single physical location

  - Require lower n bits of virtual and physical address be the same (e.g. Sun UNIX)
  - These lower n bits must include entire index
  - In a direct-mapped cache, won't have conflicts
  - Virtual address translation only relates to tag portion of address
  - Called page-coloring

3. **I/O: typically operates on physical addresses**

  - Must translate to virtual addresses to put I/O values into virtual cache

3. **Pipelined writes**

- Write hits : writing data into cache
- **Take longer than reads, since have to check tag before writing data**
  - Check for hit
  - On next cycle, actually change data in the cache entry
- **Separate these into two pipeline stages**
  - On one stage, do tag check
  - On next stage, if it was a hit, actually write to cache
- Include a delayed write buffer for this purpose
- One per cycle

4. Always writing small subblocks -- write through only

- Won't cover this....
- For machines with writes usually 1 word, subblock size 1 word
- Idea: ALWAYS write the word to the subblock regardless of whether tag matches
- ALWAYS turn the valid bit on
- Then send word to memory
- If tags match, have done the right thing
- If tags mismatch, then OK as long as this is a write-through cache
    - correct old block exists at a lower level
    - Just need to chage the address of the write tag and valid bits of other subblocks

- **Cache performance equations**
- Remember: CPUtime = IC * CPI * CT
- Before, only cared about CPI for execution
    - Now, need to take into account the extra CPI for memory accesses
- **CPUtime = IC * (CPI$_{Execution}$ + Memory stall clock cycles) * CT**
- So, only add extra cycles when we stall the processor
- (Hit time is included in the CPIexecution)
- What causes us to stall? Misses
- So, need the fraction of memory accesses (for data and instruction accesses) that MISS
    - **Mem stall cycles = Memory accesses per instr * miss rate * miss penalty**
- CPUtime = IC * (CPI$_{Execution}$ + (Mem accesses/instr * miss rate * miss penalty) * CT

- **Example:**
- Hit time = 1 clock cycle (included in CPIexecution)
- Miss penalty = 25 clock cycles
- Miss rate = 1.0% for 2-way set-associative
- CPUtime = IC * (CPI$_{Execution}$ + MemStallClockCycles) * CT
- Assume: CPI$_{Execution}$ = 2 (i.e., CPI with no cache misses)
- **Mem stall cycles = Memory accesses per instr * miss rate * miss penalt**y
- Assume 1.3 memory references per instruction (I-fetch plus 30% of time D-fetch)
- CPUtime = IC * (2 + (1.3 * 0.01 * 25)) * CT = 2.35 * IC * CT
- NOTE: execution time more than doubles because we miss in cache 1% of time

- **Example:**
- Consider two cache schemes: Unified 32 KByte cache or split 16KB data/ 16KB instr cache
- Which has lower miss rate?
- What is average memory access time?
- From table in book, miss rates are (from SPEC-92 benchmarks on DECstation5000):
    - for 16KB I-cache: 0.64%
    - for 16KB D-cache: 6.47%
    - for 32KB U-cache: 1.99%
- Hit time: 1 clock cycle
- Miss penalty: 50 clock cycles
- Load or store hit (data access) takes 1 extra cycle on a unified cache since only one memoy port and fetching both instruction and data
- 75% of memory accesses are instruction references

- For split cache: overall miss rate
- (0.75)*(0.64%) + (0.25)*(6.47%) = 2.10%
- For unified cache: 1.99%
- So, unified cache has slightly lower overall miss rate

- But, not sufficient to look only at miss rate
- Look at whole access time equation
- AMAT + HT + MR * MP
- Divide between instructions and data:
- AMAT = %instr (Instr HT + (Instr MR) (Instr MP)) +
  %data(Data HT + (Data MR) (Data MP))
- AMATsplit = 0.75 * (1 + 0.64%*50) + 0.25*(1 + 6.47%*50) = 2.05 cycles
- AMATunified = 0.75*(1+1.99%*50) + 0.25*(**2** + 1.99%*50) = 2.24 cycles
- Note: Unified cache has lower miss rate but LONGER AMAT

- **Example (p. 387)**
- What is the impact of two different cache organizations on performance?
    - Cycle time = 2 nsec
    - CPI of a perfect cache = 2.0 cycles (i.e., CPIexecution)
    - 1.3 memory references per instruction (instr fetch plus data fetch 30% of time)
    - 64KB caches
    - 32-byte block size
    - Cache miss penalty = 70 nsec (assumes asynchronous DRAM)
    - Hit time = 1 cycle = 2 nanoseconds
    - Miss rate: DM = 1.4%, 2SA = 1.0%
- **Direct-mapped vs. 2-way SA**
- For SA, must add a multiplexor to choose between blocks in the set
    - **As a result, longer clock cycle: CT of 2-way SA = 1.10*CT of direct-mapped**
- **How are AMAT and CPUtime affected by the cache organizations?**

- AMAT = HT + MR * MP
- For DM: AMAT = 2 + (1.4%)(70nsec) = 2.98 nsec
- For 2SA: AMAT = (2 * 1.10) + (1.0%)(70nsec) = 2.9 nsec
- 2SA has longer clock cycle
- 2-SA still has shorter AMAT

- How about CPU time--time to execute a program?
- CPUtime = IC * (CPIexec + Mem Accesses/Instr * MR * MP) * CT
- In this case, since dealing with nanoseconds not cycles, move CT inside parenthesis:
- CPUtime = IC * [(CPIexec * CT) + (Mem Accesses/Instr * MR * 70nsec)]
  where 70nsec = MP * CT

- For DM: CPUtime = IC * ((2*2.0nsec) + (1.3 * 1.4% * 70nsec)) = 5.27 *IC
- For 2SA: CPUtime = IC * ((2*2.0nsec*1.10) + (1.3 * 1% * 70nsec)) = 5.31*IC

- **So, DM executes program faster, even though 2SA has lower miss rate and faster AMAT because of longer clock cycle!**

Main Memory and Virtual Memory

- **Memory Technology**
- **Memory Latency: access time and cycle time**
    - Access time: time between read reqest and data arrives

- **Memory Chip Organization**
    - Rectangular array of bits
    - In past: often 1-bit data interfaces
    - Now, families of chips with widths of 1, 4, 8 and 16-bits

    - **For large chips, traditionally access first a row of the array and then one of the columns**
    - Requires fewer address lines to access a very large memory
    - Example: 4096K x 1 memory chip (4 Mbit DRAM)
    - How many address lines to address 4096K locations? 22
    - Use 11 bits for row address and 11 bits for column

    - Draw picture with address bits A0 through A10, also RAS and CAS, CS (chip select), WE (write enable), OE (output enable) and 1 output bit D
    - First send row address while asserting RAS
    - Then send column address while asserting CAS
    - Slow, since two address cycles are required

    - **Optimization: send multiple column addresses to access different bits in a row**
    - Why did 1-bit chips become hard to use?
    - for 32-bit wide machine, need 32 of them;
    - for 64-bit machine, need 64 of them.
    - Requires very large minimum memory sizes (4 Mbit chip x 32 chips = 16 MBytes minimum memory size)

- **SRAM: Static Random Access Memory**
    - Typical: 4 Mbits, fast access time 5 nsec
    - D flip-flops: Use four to six transistors per bit; non-destructive reads
    - **SRAMs used for first and second-level cache implementations**

- **DRAM: Dynamic Random Access Memory**
    - Value stored as a charge on a capacitor
    - One transistor to access the bit (either read or write)
    - Single transistor per bit: much denser and cheaper than SRAMS
    - But, capacitors discharge over time--must refresh periodically
    - **Main memories always built of DRAM because of high capacities**

- **Comparison of SRAM/DRAM**
- Capacity of DRAM: 4 to 8 times capacity of SRAM
- Access time of SRAM: 8 to 16 times faster than DRAM (5-25nsec vs. 60-120nsec)
- Cost of SRAM: 8 to 16 times more expensive than DRAM($100-$250/MB vs. $5-$10)

- **Types of RAM chips:**

1. **FPM (Fast Page Mode):**
    - Oldest type still in use
    - Use row and column addresses
    - Step through column addresses sequentially
    - Delay between receiving data and requesting a new bit: access latency
2. **EDO (Extended Data Output):**
    - Same idea but pipelined
    - Allow next memory reference to begin before the previous memory reference has completed
    - Doesn't change latency of one access but improves memory throughput
3. **SDRAM (Synchronous DRAM):**
    - Unlike previous technologies, is driven by a clock
    - Chip transfers a burst of data from sequential addresses in an array or row
    - No additional address lines required; specify a burst length
    - Much better memory bandwidth--becoming technology of choice

- **ROM (Read Only Memory)**
    - Cannot be changed or erased
    - Used for storing essential data: state machines, boot-up code, etc.
    - Problem: inflexible; must replace entire chip if there is an error

- **PROM (Programmable ROM)**
    - Programmable once in the field
- **EPROM (Erasable PROM)**
    - Field-programmable and field-erasable
    - Requires exposure of quartz window to strong UV light for 15 minutes
    - Must remove EPROM chip, put in special chamber for erasure/reprogramming
- **EEPROM**
    - Can be reprogrammed with electrical pulses rather than UV light
    - Byte-erasable
    - Doesn't require removal
- **Flash Memory**
    - A type of EEPROM
    - Block erasable and re-writable
    - Typically produced in small printed circuit cards with tens of megabyts of flash memory
    - Used in digital camers, etc.
    - 100-nsec access times
    - Limitation: can be re-written only about 10,000 times
    - If this improves, may someday replace disks?

- **Memory Organizations**

1. Interleaved Memory (Banks)

- Supports efficient access to sequential words
- Banks usually 1 word wide
- Draw Figure 5.32: four banks, sequential addresses fall in sequential banks
- Send addresses to several banks simultaneously -- all read at same time
- Bus between memory and cache typically one word wide
- If so, must transfer the words on the bus sequentially
- So, 4-word block: send address, wait access time of memory, then send four words of data over the bus

2. Independent Memory Banks

- Supports multiple independent memory accesses
- Interleaved memory: one address to all banks, read sequential words
- Now: multiple memory controllers address independent banks or sets of word-interleaved banks
- Need enough address lines so that can access each bank independently
- Nonblocking caches must use banks to allow multiple concurrent memory accesses

- **Virtual Memory**
- Invented to allow larger address spaces than physical memory without requiring programmer intervention (overlays)

- Divide physical memory into blocks
- Map virtual address spaces onto available physical blocks
- Allows sharing between multiple processes
- **Manages two levels of storage hierarchy:**
    - main memory (physical memory) and disk
- **Move data into main memory on demand**
- Advantages:
    - memory protection: don’t allow a process to access memory other than its own
    - relocation: port code to different physical locations by changing VA mapping

- **Terminology**
- Page or segment (equivalent of cache block)
- Page or address fault (equivalent of cache miss)
- Virtual address: produced by CPU
- Physical address: used to access main memory
- **Typical virtual memory parameters:**
    - Page sizes: 4K to 64K (cache block: 16 to 128 bytes)
    - Hit times: 40-100 clock cycles (cache hit: 1 to 2 cycles)
    - Miss Penalty: 700K to 6M clock cycles (huge--want to avoid misses!) (cache miss: 8 to 100 cycles)
    - Miss rate: 0.00001% to 0.001% (very low) (caches: 0.5% to 10%)
    - Total (disk) memory size -- hundreds of megabytes to gigabytes

- **Pages vs. Segments**
- Pages = fixed size blocks
    - simple, since single fixed-size address
    - Internal fragmentation: unused portions of page
- Segments
    - complex, since variable sized
    - external fragmentation: need to find blocks of main memory to allocate segments; as allocate more and more, will be pieces too small to allocate
- Hybrid approach: paged segments, each an integral number of pages
- Multiple page sizes: Alpha: 8KB, 64KB, 512KB, 4096 KB

- Hierarchy Questions:
- **Q1: Block placement in upper level**
    - Key consideration: keep miss rate low (since high penalty)
    - So, allow block to be placed anywhere (fully associative)
- **Q3: Block replacement:**
    - Again, want to keep miss rates as low as possible
    - Approximation of LRU
    - Have use bits associated with page, set whenever page is accessed
    - Periodically clear the bits
    - Later, record them; use to pick a page among the least recently used
- **Q4: What happens on a write?**
    - Does write-through make sense?
    - No, because access time to magnetic disks is so slow
    - Always use write-back
    - Keep dirty bit so only write back modified pages
- **Q2: Block identification**
    - Page table: area in physical memory that contains mapping from virtual page address to physical address
    - Size? Number of pages in virtual address space * size of physical address
    - Can be very large; may even be paged itself

- **Inverted page table: reduces size by applying hashing function to virtual adrress so table only needs one entry per physical page**
    - So, every access that uses the page table requires at least two memory accesses
    - One to translate to physical; one to access physical

- **To reduce translation time, use TLB**
- **Translation Lookaside Buffer (TLB)**
    - Draw Figure 5.41
    - Idea: use locality of addresses
    - **Keep a small cache of recent address translations**
    - Cache is called TLB (not for a good reason--old IBM terminology)
    - **On a hit in TLB, get physical address from TLB and send to main memory**
    - **On miss in TLB, go to page table for address and replace block in TLB with new translation from page table**
    - Typical TLB parameters: 32 bytes to 8KBytes (small)
    - Often fully associative for fastest access
- Note: no real relationship between hits and misses in cache and TLB; Cache miss means "don't have data"; TLB miss means "don't know how to translate address

- **Typical access: physically-addressed cache:**

1. First must translate address: check TLB for translation (if hit, go to step 3)
2. Translation not in TLB; go to page table (memory access) and update TLB with translation
3. Use physical address to check whether desired data is in cache; if so, return data to processor
4. If cache miss, access memory to get data; update the cache and return data to processor

- **Typical access: virtually-addressed cache:**

1. First use VA to check for data in cache; if hit, return data to processor
2. If cache miss, must do address translation; access TLB; if TLB hit, go to step 4
3. If TLB miss, go to page table to get physical address translation; update TLB
4. Using physical address to access main memory, update cache, return data to processor

- Design issue for virtual memory: what is best page size?
- Reasons for large page size:
    - Page table is inversely proportional to page size; big pages save memory
    - Transferring large page sizes to/from disk maybe over network more efficient
    - Number of TLB entries restricted by clock cycle time; larger page size maps more memory and reducess TLB misses
    - (Note--last reason is why recent processors have supported multiple page sizes; reduce # TLB misses)
- Reasons for smaller page size:
    - Don't waste storage: less internal fragmentation
    - Quicker process start for small processes; smaller page lets you invoke sooner

- **Magnetic Disk:**
- Draw picture: platters, tracks, sectors
- Collection of platters (1 to 20)
- Rotate on a spindle at : 3600 to 7200 RPM
- **Platters** = metal disks covered with magnetic material on each side
- Disk diameters: 1.8 inches to 8 inches
- **Tracks** = concentric circles
- 500 to 2500 tracks per surface
- Outer tracks are longer, can store more bits than inner tracks
- Disks rotate with **constant angular velocity**, so higher bit rate on outer tracks
- Tracks divided into sectors (perhaps 64 sectors per track)

- **Sector** is smallest unit of data transfer to disk
- Movable **arm** contains read/write **heads; one head pair per surface**
- Cylinder: all the tracks under the arms at a given point
- **Seek time**: Time to move the arm to the proper track
- Average seeks: 8 to 12 msec (advertised)
- Because of locality of reference, may actually be substantially less
- **Rotation time**: time for correct sector to rotate under the disk head
- Say 3600 RPM: average rotation time is time to rotate half a track
- (1/2 revolution)/(3600 revolutions per minute) = 8.3 milliseconds
- **Disk access time**: seek plus rotation time
- **Disk transfer time**: 2 to 8 megabytes per second
- **Disk controller**: sits between the disk and the (SCSI) bus; handles bus protocol, sends data out at agreed-upon rate from disk buffers

- **Trends in Magnetic Disk**
- Aerial density: bits per square inch: currently increasing at 60% per year
- Capacity increasing, cost dropping at this rate of 60% per year
- Products: 644 million bits per square inch; Lab: 3 trillion bits per square inch
- Disks get smaller: easier to spin lighter, smaller disks
- Save power and volume
- Fewer cylinders, so lower seek times
- Disk about 100 times cheaper than DRAM
- DRAM about 100,000 times faster than disk

- **Disk controller**: sits between the disk and the (SCSI) bus; handles bus protocol, sends data out at agreed-upon rate from disk buffers
- **Example:**
- 512 byte sectors, 20 msec average seek time, 1 MB/sec transfer rate, 2 msec controller overhead
- Average access time when disk is free:
- 20 msec + (0.5 rotation)/(3600 RPM) + 512 bytes/1 MByte/sec + 2 msec = 30.8 msec to transfer one sector
- Note that seek is 20 msec or 65% of the time

- **DISK ARRAYS**
- **Disk Trends:**
    - **Capacity increasing at 60% per year**
    - **$/MB improving at 60% per year**
    - **Evolving to smaller physical sizes (14”->5.25” -> 3.5” -> 2.5”-> ...?)**
- **Create arrays of disks to close performance gap between disks and CPUs**
- **Large number of small disks**
- **High bandwidth on large accesses**
- **High rate of random I/Os for small accesses**

- **Problem: Reliability**
- **Reliability of N disks = Reliability of 1 disk divided by N**
- **MTTF of 50,000 hours for 1 disk**
- **70 disks in array: 50,000/7 = 700 hours ---> from 6 years to 1 month**

- **Two Main Techniques in Disk Arrays:**

1. **Striping for High Performance**
2. **Redundancy for High Reliability**

- **Striping:**
- **Interleave data from a single file across multiple disks**
- **Fine-grained interleaving: every file spread across all the disks; any access involves all the disks**
- **Course-grained interleaving: interleave in large blocks; small accesses may be satisfied by a single disk**

- **Redundancy:**
- **Maintain extra information in disk array**
- **When a disk fails, use redundancy information to reconstruct data on the lost disk**
- **Examples: duplication, parity, Reed-Solomon error correction codes**

- **Combinations of Striping and Redundancy options define RAID “levels”**

1. **RAID Level 1: Mirroring or Shadowing**

- Maintain a copy of each disk
- very reliable
- high cost--must buy twice as many disks
- great performance--on a read, may go to disk with faster access time

2. RAID Level 2: Memory-Style Error Correction and Detection

- Not really implemented in practice
- Based on DRAM-style codes that detect and correct errors (Hamming codes)
- Example: need 4 check disks for 8 data disks
- Somewhat reduced cost compared to mirroring
- Not necessary in practice because disks identify themselves as broken
- Need to be able to correct, but don't need to detect

3. RAID Level 3: Fine-Grained Interleaving and Parity

- Correction only, not detection
- Calculate parity bit-wise across the disks in the array (using exclusive-or)
- Maintain a separate parity disk; Updated on write operations
- When a disk fails, can use other data bits and parity bits to reconstruct bits on lost disk
- Fine-grained interleaving, so all disks are involved in any access to the array

4. RAID Level 4: Large Block Interleaving and Parity

- Similar to Level 3 except do large block interleaving
- Now small accesses may be satisfied by a single disk
- Can have many independent small accesses going on
- Good for workloads like transaction processing -- lots of small updates to accounts

5. RAID Level 5: Large Block Interleaving and Distributed Parity

- Similar to Level 4
- Notes that parity disk is likely to become a performance bottleneck
- Distribute parity blocks throughout all the disks in the array
- Better overall performance

6. RAID Level 6: Reed-Solomon Error Correction Codes

- Protection against two disk failures ; "P + Q redundancy"