Lecture 12

Memory Design & Caches, part 2

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

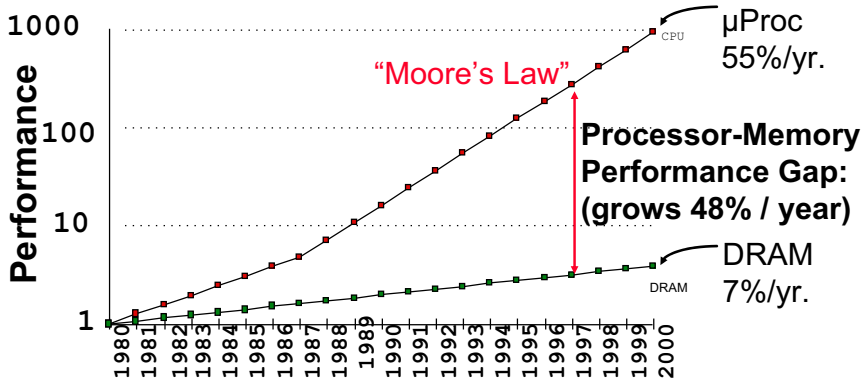
- HW3 is due today
- PA2 is available on-line today
  - Part 1 is due on 2/27
  - Part 2 is due on 3/6
- HW4 available on-line today
  - Merged with HW5
  - Due on 3/13

Review: Memory Technologies

- The cheaper the memory,
  - The slower the access times

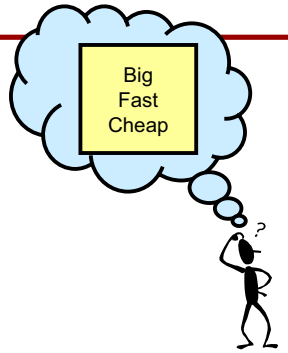
Technology	Access Time	$/GB in 2004
SRAM	0.5-25 ns	$4000–$10000
DRAM	50-120 ns	$100-$200
Disk	5-10 ms	$0.5-$2

Review: The Processor-Memory Performance Gap

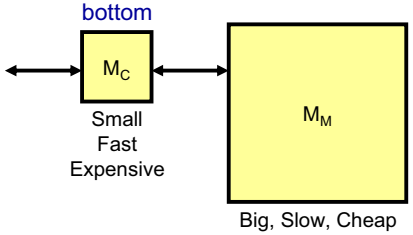


- A 1-cycle access to DRAM memory in 1980 takes 100s of cycles now
- We can't afford to stall the pipeline for that long!

Review: Memory Hierarchy



- Tradeoff cost-speed and size-speed using a hierarchy of memories:
  - small, fast, expensive caches at the top
  - large, slow, and cheap memory at the bottom

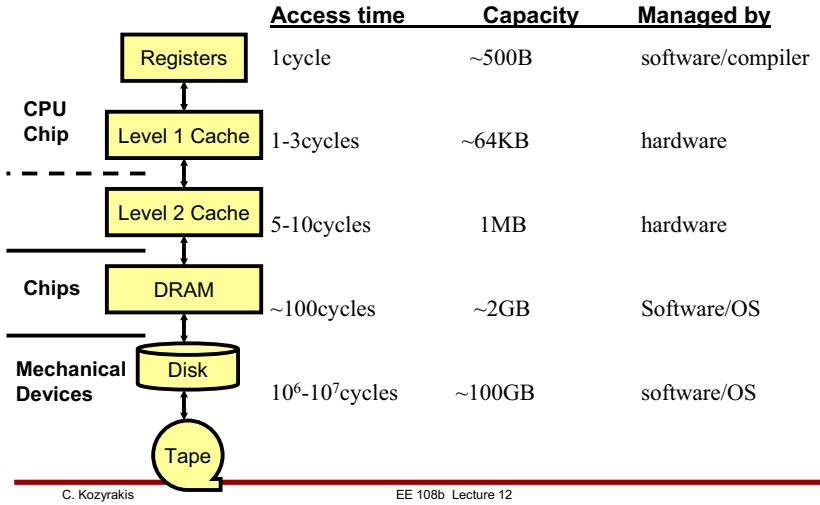


•We can't use large amounts of fast memory
  –expensive in dollars, watts, and space

•Ideal: the memory hierarchy should be almost as fast as the top level, and almost as big and cheap as the bottom level

•Program locality makes this possible

Review: Typical Memory Hierarchy: Everything is a Cache for Something Else

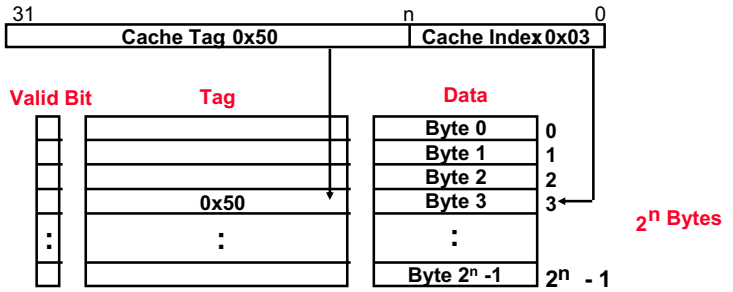


How To Build A Cache?

- Big question
  - I need to map a large address space into a small memory
    - Want associative lookup
- How do I do that?
  - Can build full associative lookup in hardware, but complex
  - Need to find a simple but effective solution
  - Two common techniques:
    - Direct Mapped
    - Set Associative
- Details
  - Block size
  - Write policy
  - Replacement policy

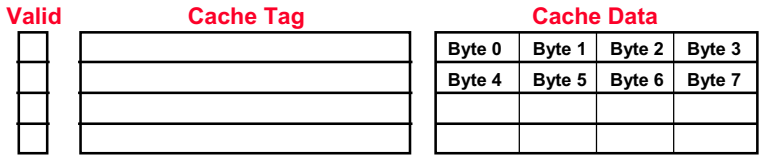
Direct Mapped Caches

- Access with these is a straightforward process:
  - Index into the cache with block number modulo cache size
  - Read out both data and "tag" (stored upper address bits)
  - Compare Tag with address you want to determine hit/miss
  - Need valid bit for empty cache lines



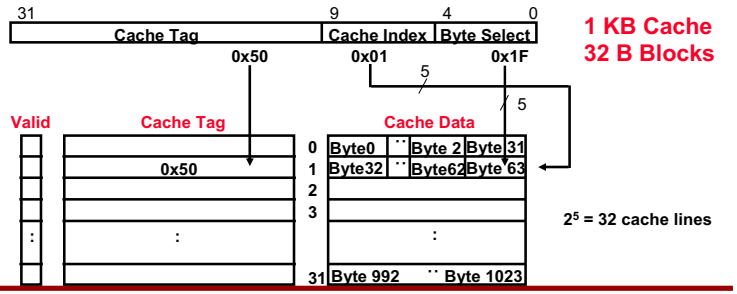
Cache Blocks

- Previous example had only 1 byte blocks
  - Strategy took advantage of temporal locality since if a byte is referenced, it will tend to be referenced soon
  - Did not take advantage of spatial locality
- To take advantage of spatial locality, increase block size
  - Another advantage: Reduces size of tag memory, too!
  - Potential disadvantage: Fewer blocks in the cache



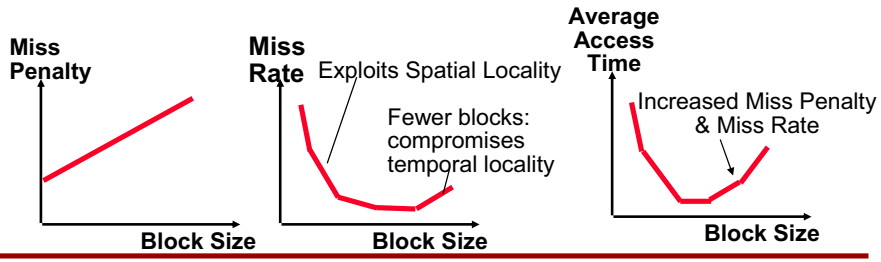
Cache Block Example

- Assume a 2^n byte direct mapped cache with 2^m byte blocks
  - Byte select – The lower m bits
  - Cache index - The lower (n-m) bits of the memory address
  - Cache tag - The upper (32-n-m) bits of the memory address



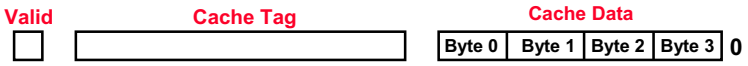
Optimal Block Size

- Larger block sizes take advantage of spatial locality
  - Also incurs larger miss penalty since it takes longer to transfer the block into the cache
  - Large block can also increase the average time or the miss rate
- Tradeoff in selecting block size
- Average Access Time = Hit Time + Miss Penalty × MR



Direct Mapped Problems

- Conflict misses could in turn cause other conflict misses
- Consider an extreme example where both the cache size and block size are 4 bytes



  - Only one entry in the cache
  - Conflict misses will be continuous if we access >1 data item
- Such a situation is referred to as *thrashing* where cache continually loading data into the cache but evicting it before it can be used again

This is a real problem!

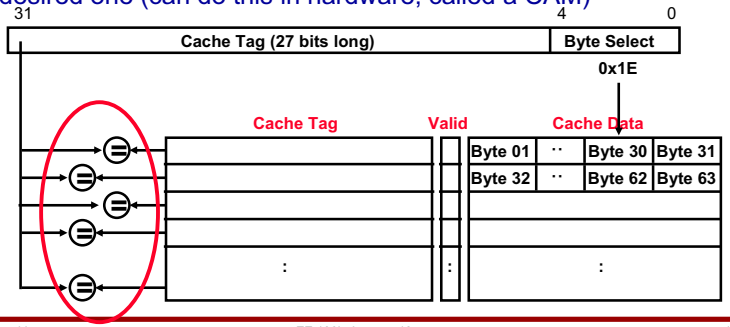
- Consider the following example code:

```
double a[8192], b[8192], c[8192];

void vector_sum()
{
    int i;

    for (i = 0; i < 8192; i++)
        c[i] = a[i] + b[i];
}
```

  - Arrays a, b, and c will tend to conflict in small caches
  - Code will get cache misses with *every* array access (3 per loop)
  - Spatial locality savings from blocks will be eliminated
- How can the severity of the conflicts be reduced?

Fully Associative (FA) Cache

- Opposite extreme, in that it has no cache index
  - Use any available entry to store memory elements
  - No conflict misses, only capacity misses
  - Must compare cache tags of all entries in parallel to find the desired one (can do this in hardware, called a CAM)

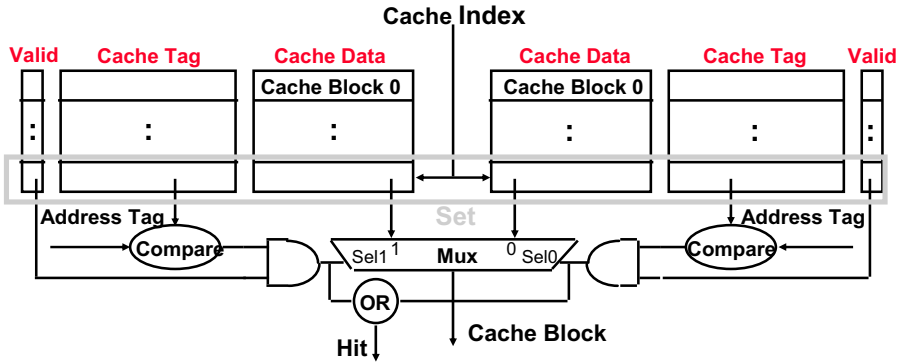


N-way Set Associative

- Compromise between direct-mapped and fully associative
  - Hash-table with N entries/index
  - Set size = "set associativity" (e.g. 7-way, 7 members per set)
  - For fast access, compare all entries (members) at the same time
- One view:
  - N direct mapped caches in parallel
  - Need to coordinate on
    - Who outputs data
    - Issuing a miss (only when all miss)
- Another view:
  - (# Indexes) array of fully associative caches
- Direct Mapped and Fully Associative caches are really just extremes of the N-way Set Associative

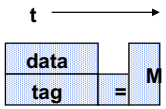
Two-way Set Associative

- Cache index selects a "set"
- The two tags in the set are compared in parallel
- Data is selected based on the tag



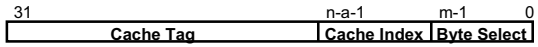
Set Associative Caches

- Work quite well
  - 2 ways is generally much better than direct mapped
    - Get rid of many conflicts
  - 4 way is a little better than 2 way
  - More than 4 ways give little additional benefit
- Set Associative Disadvantages
  - N comparators vs. 1
  - Critical path is longer, since need to have extra logic to combine the data from all the caches
    - it requires an extra MUX that delays the data result
  - Data comes after determine Hit/Miss
    - Direct mapped assumes a hit and recovers later if a miss



Tag & Index with Set-Associative Caches

- Assume a 2^n-byte cache with 2^m-byte blocks that is 2^a set-associative
  - Which bits of the address are the tag or the index?
  - m least significant bits are byte select within the block
- Basic idea
  - The cache contains $2^n/2^m=2^{n-m}$ blocks
  - Each cache way contains $2^{n-m}/2^a=2^{n-m-a}$
  - Cache index: (n-m-a) bits after the byte select
    - Same index used with all cache ways…
- Bonus: Odd #'s of ways allow odd-size caches (i.e. non-power-of-2)
  - Ex.: 3 MB cache = 3 ways of 1 MB caches, 6 of 512K, etc.



Intuitive Model of Cache Misses (3 Cs)

- Now, let's talk about *missing* in the cache . . .
- Several different categories of misses
  - Compulsory/Cold Start in FA of infinite size
    - First access to a block
    - Only (partial) solution is to increase the block size
  - Capacity in finite-size FA cache
    - Cache cannot contain all blocks accessed by program
    - Solution is to increase cache size
  - Conflict (collision) in DM cache
    - Multiple memory locations map to the same cache location
    - One solution is to increase the cache size
    - A second solution is to increase the associativity

Replacement Methods

- Which line do you replace on a miss?
- Direct Mapped
  - Easy, you have only one choice
  - Replace the line at the index you need
- N-way Set Associative
  - Need to choose which way to replace
  - Random (choose one at random)
  - Least Recently Used (LRU) (the one used least recently)
    - Often difficult to calculate, so people use approximations. Often they are really not recently used

What About Writes?

- So far we have talked about reading data
- But a processor also writes data
- Where do we put the data we want to write?
  - In the cache?
  - In main memory?
  - In both?
- Caches have different policies for this question
  - Most systems store the data in the cache
    - Why?
  - Some also store the data in memory as well
- What happens on a cache miss depends on:
  - Whether main memory is always up to date
- Processor does not need to "wait" until the store completes

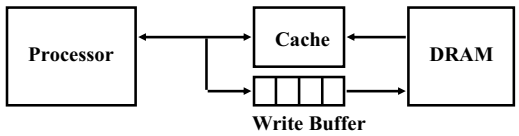
Cache Write Policy

- Write-through (writes go to cache and memory)
  - Main memory is updated on each cache write
  - Replacing a cache entry just replaces the existing entry with the new entry
  - Memory write causes significant delay if pipeline must stall
- Write-back (write data only goes to the cache)
  - Only the cache entry is updated on each cache write so main memory and the cache data are inconsistent
  - Add "Dirty" bit to the cache entry to indicate whether the data in the cache entry must be committed to memory
  - Replacing a cache entry requires writing the data back to memory before replacing the entry if it is "dirty"

Write Policy Trade-offs

- Write-through
  - Misses are simpler and cheaper since block does not need to be written back
  - Easier to implement, though most systems need an additional buffer, called a write buffer, to be practical
  - Uses a lot of bandwidth to the next level of memory
- Write-back
  - Words can be written at the cache rate
  - Multiple writes within a block require only one "writeback" later
  - Efficient block transfer on write back to memory at eviction

Buffering Writes



- Use Write Buffer between cache and memory
  - Processor writes data into the cache and the write buffer
  - Memory controller slowly "drains" buffer to memory
- Write Buffer
  - First In First Out (FIFO)
  - Typically holds a small number of writes
  - Works fine if the rate of writes to memory is less than 1 / DRAM write cycle time

Reading vs. Writing With A Cache

- We have talked about **reading** from a cache
  - Access cache to see if data is there
  - If it hits, return the data
  - Else fetch it from memory
- **Writing** a cache can take more time
  - If it is a direct mapped, write through cache
    - Fetch the tag, and write the data
    - Data has to go into the cache, and destroying the old data can't hurt
  - If it is a write-back or not DM write-through cache
    - Need to check the tag before the write (access 1)
    - Then you need to write the data (access 2)
  - In either case, do you fetch the other words in the block?

Write Miss/Fetch Policies

- Write through cache
  - Use fetch-on-write (also known as write allocate)
    - To fetch the rest of the block
  - Use no-fetch-on-write
    - Mark the fact that the other parts of the block are not valid
  - Use no-allocate-on-write (also known as write-around)
    - Write data to memory without placing in the cache at all
- Write back caches
  - Generally use fetch-on-write
    - Fetch the rest of the block, hoping that future writes will be to the block, though no-fetch-on-write and no-allocate-on-write would also work

Write Miss Actions

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

Splitting Caches

- Careful about looking only at miss rate
  - It is important, but not the whole story
- Most processors have a separate instruction and data cache
  - Why split the cache, rather than have a larger unified cache?
    - Split cache often has slightly higher miss rate
  - The split cache doubles the access rate
    - Allows an instruction *and* data reference in a cycle

⇒ Miss rate alone is not a sufficient measurement of cache performance

Measuring Performance

- Memory system
  - Stalls include both cache miss stalls and write buffer stalls
  - Cache access time often determines the overall system clock cycle time since it is often the slowest functional unit in the pipeline
- Memory stalls hurt processor performance
  - CPU Time = (CPU Cycles + Memory Stall Cycles) • Cycle Time
- Memory stalls caused by both reading and writing
  - Mem Stall Cycles = Read Stall Cycles + Write Stalls Cycles

Memory Performance

- Read stalls are fairly easy to understand
  - **`Read Cycles = Reads/prog • Read Miss Rate • Read Miss Penalty`**
- Write stalls depend upon the write policy
  - Write-through

    `Write Stall = (Writes/Prog • Write Miss Rate • Write Miss Penalty) + Write Buffer Stalls`
  - Write-back

    `Write Stall = (Writes/Prog • Write Miss Rate • Write Miss Penalty)`
- "Write miss penalty" can be complex:
  - Can be partially hidden if processor can continue executing
  - Can include extra time to writeback a value we are evicting

Worst-Case Simplicity

- Assume that write and read misses cause the same delay

$$Memory\ Stall\ Cycles = \frac{Memory\ accesses}{\mathrm{Pr}\,ogram} \times Miss\ rate \times Miss\ penalty$$

$$Memory\ Stall\ Cycles = \frac{Instructions}{\mathrm{Pr}\,ogram} \times \frac{Misses}{Instruction} \times Miss\ penalty$$

- In a multi-level cache be careful about *local* miss rates
  - Miss rate for 2nd level cache is often "high" if you look at the local miss rate (misses per reference into the cache)
  - But for misses per instruction, it is often much better

Cache Performance Example

- Consider the following
  - Miss rate for instruction access is 5%
  - Miss rate for data access is 8%
  - Data references per instruction are 0.4
  - CPI with perfect cache is 2
  - Read and write miss penalty is 20 cycles
    - Including possible write buffer stalls
- What is the performance of this machine relative to one without misses?

Performance Solution

- Find the CPI for the base system without misses
  - $CPI_{no\ misses} = CPI_{perfect} = 2$
- Find the CPI for the system with misses
  - Misses/instr = I Cache Misses + D Cache Misses
  - = 0.05 + (0.08 • 0.4) = 0.082
  - Memory Stall Cycles = Misses/instr • Miss Penalty
  - = 0.082 • 20 = 1.64
  - $CPI_{with\ misses} = CPI_{perfect}$ + Memory Stall Cycles
  - = 2 + 1.64 = 3.64
- Compare the performance

$$n = \frac{Performance_{no\ misses}}{Performance_{with\ misses}} = \frac{CPI_{with\ misses}}{CPI_{no\ misses}} = \frac{3.64}{2} = 1.82$$

Another Cache Problem

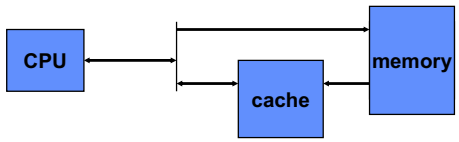
- Given the following data
  - 1 instruction reference per instruction
  - 0.27 loads/instruction
  - 0.13 stores/instruction
  - Memory access time = 4 cycles + # words/block
  - A 64KB S.A. cache with 4 W block size has a miss rate of 1.7%
  - Base CPI of 1.5
- Suppose the cache uses a write through, write-around write strategy without a write buffer. How much faster would the machine be with a perfect write buffer?

1 1.5 1

CPU_{time} = Instruction Count • (CPI_{base} + CPI_{memory}) • Clock Cycle Time

Performance is proportional to CPI = 1.5 + CPI_{memory}

No Write Buffer



CPI_{memory} = reads/instr. • miss rate • read miss penalty
+ writes/instr. • write penalty

write penalty = 4 cycles + 1 word • 1 word/cycle = 5 cycles

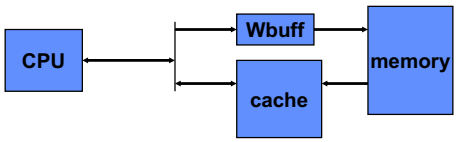
read miss penalty = 4 cycles + 4 words • 1 word/cycle = 8 cycles

CPI_{memory} = (1 if + 0.27 ld) (1/inst.) • (0.017) • 8 cycles + (0.13 st)(1/inst.) • 5 cycles

CPI_{memory} = 0.17 cycles/inst. + 0.65 cycles/inst. = 0.82 cycles/inst.

$CPI_{overall}$ = 1.5 cycles/inst. + 0.82 cycles/inst. = 2.32 cycles/inst.

Perfect Write Buffer



CPI_{memory} = reads/instr. • miss rate • 8 cycle read miss penalty
+ writes/instr. • (1 – miss rate) • 1 cycle hit penalty

• A hit penalty is required because on hits we must:

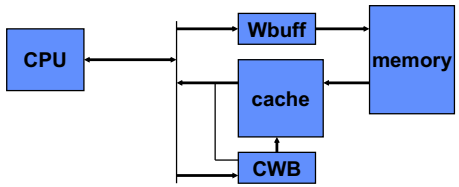
1) *Access the cache tags during the MEM cycle to determine a hit*
2) *Stall the processor for a cycle to update a hit cache block*

CPI_{memory} = 0.17 cycles/inst. + (0.13 st)(1/inst.) • (1 – 0.017) • 1 cycle

CPI_{memory} = 0.17 cycles/inst. + 0.13 cycles/inst. = 0.30 cycles/inst.

$CPI_{overall}$ = 1.5 cycles/inst. + 0.30 cycles/inst. = 1.80 cycles/inst.

Perfect Write Buffer + Cache Write Buffer



CPI_{memory} = reads/instr. • miss rate • 8 cycle read miss penalty

• Avoid a hit penalty on writes by:

- ✓ ***Add a one-entry write buffer to the cache itself***
- ✓ ***Write the last store hit to the data array during next store's MEM***
- ✓ ***Hazard: On loads, must check CWB along with cache!***

CPI_{memory} = 0.17 cycles/inst.

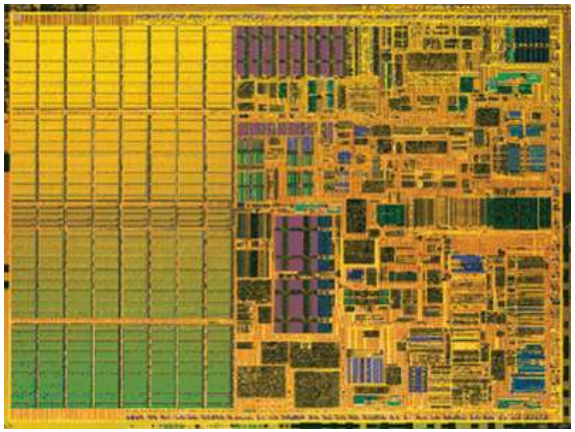
$CPI_{overall}$ = 1.5 cycles/inst. + 0.17 cycles/inst. = 1.67 cycles/inst.

Review: Cache Design Summary

- Three major categories of cache misses
  - Compulsory – First time access
  - Capacity – Insufficient cache size
  - Conflict – Two addresses map to the same cache block
- AMAT = hit time + miss rate • miss penalty

	Hit time	Miss rate	Miss penalty
Larger cache	–	+	
Larger block size		+/–	–
Higher associativity	–	+	
Second level cache			+

Intel Pentium M Memory Hierarchy



- 32 K I-cache
  - 4 way S.A., 32 B block
- 32 KB D-cache
  - 4 way S.A., 32 B block
- 1 MB L2 cache
  - 8 way S.A., 32 B block
- 1 GB main memory
  - 3.2 GB/s bus to memory interface chip
- Half of the die is cache
  - 10.6 x 7.8 = 83 mm^2
  - 77 M transistors
  - 50 M in cache

Fallacy: Caches are Transparent to Programmers

- Example
  - cold cache, 4-byte words, 4-word cache blocks
  - Assume N >> 4
  - Assume M > number of blocks in the cache

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate ≈ 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate ≈ 100%

Matrix multiplication example

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
- Writing cache friendly
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```


Pentium blocked matrix multiply performance