CSE4201
Computer Architecture

Chapter 5

Caches

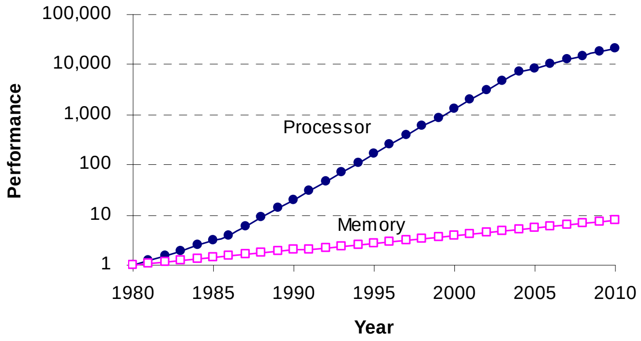
Slides are based on slides by Prof. Shaaban (RIT) and Prof. Paterson (UCB)

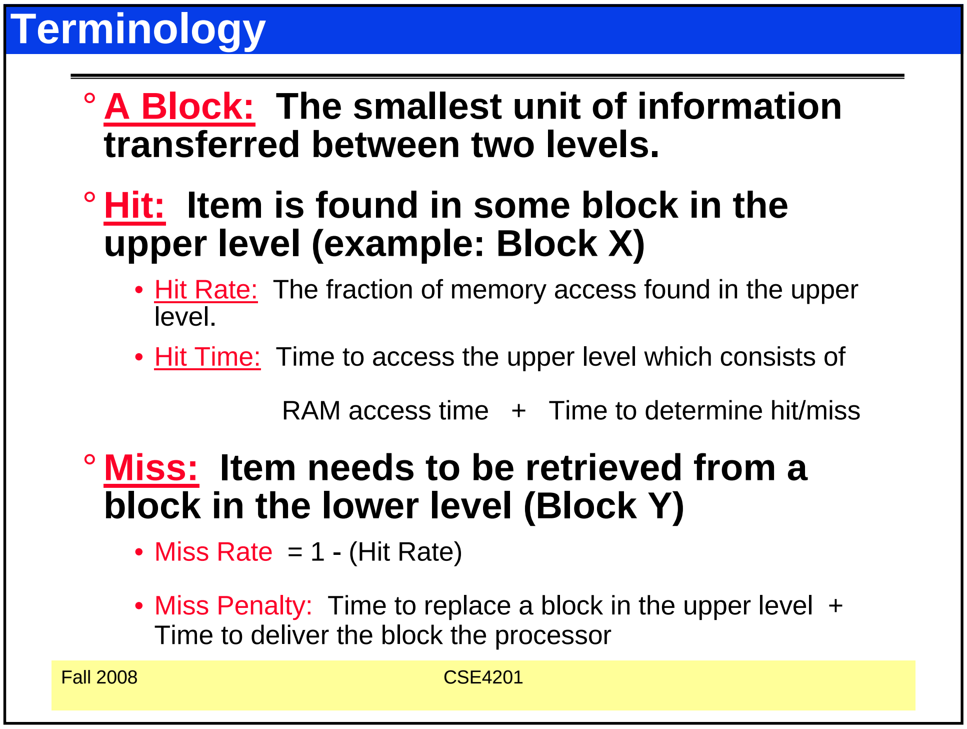
Introduction

The gap between the processor speed and memory speed

Locality: Spatial and Temporal

caches

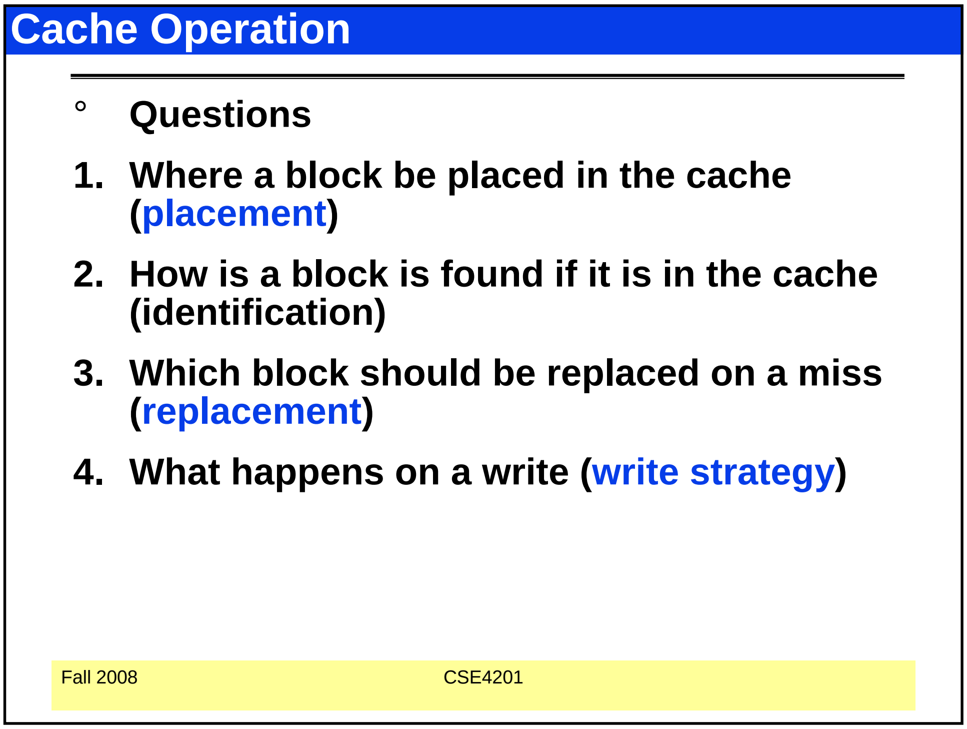




Terminology

- **A Block:** **The smallest unit of information transferred between two levels.**
- **Hit:** **Item is found in some block in the upper level (example: Block X)**
  - Hit Rate: The fraction of memory access found in the upper level.
  - Hit Time: Time to access the upper level which consists of

    RAM access time + Time to determine hit/miss
- **Miss:** **Item needs to be retrieved from a block in the lower level (Block Y)**
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor



Cache Operation

- **Questions**

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

Cache Organization: Placement

1. **Direct mapped cache:** **A block can be placed in only one location (cache block frame), given by the mapping function:**

   index= (Block address) MOD (Number of blocks in cache)

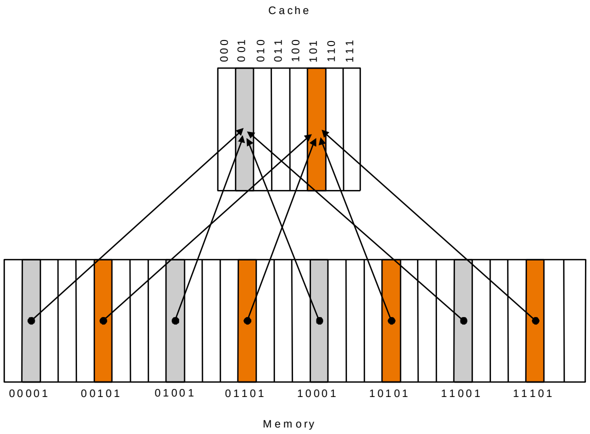
2. **Fully associative cache:** **A block can be placed anywhere in cache. (no mapping function).**

3. **Set associative cache:** **A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:**

   index = (Block address) MOD (Number of sets in cache)

   If there are *n* blocks in a set the cache placement is called *n*-way set-associative.

Direct Mapped Cache



Direct Mapped Cache

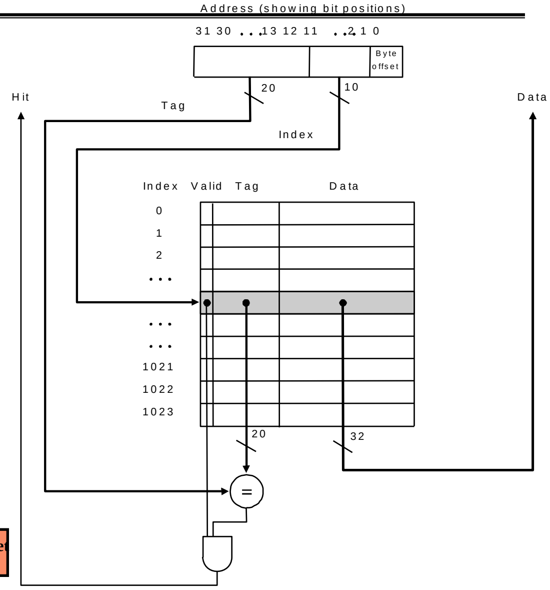
1K = 1024 Blocks
Each block = one word

Can cache up to
2^{32} bytes = 4 GB
of memory

Mapping function:

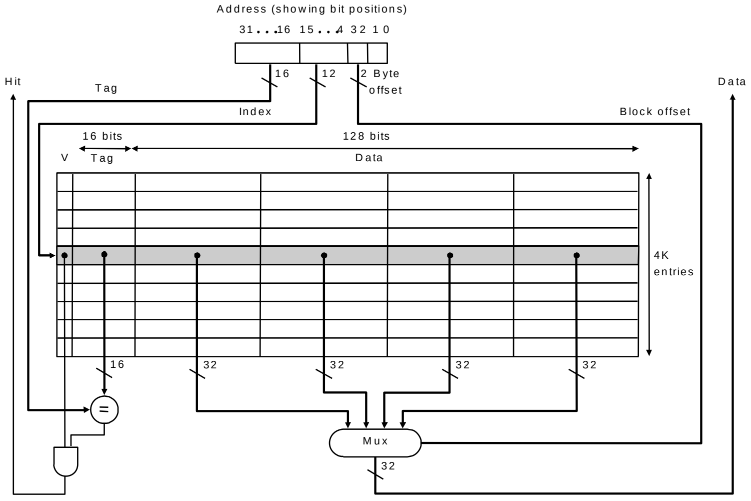
Cache Block frame number =
(Block address) MOD (1024)

i.e. index field or
10 low bit of block address



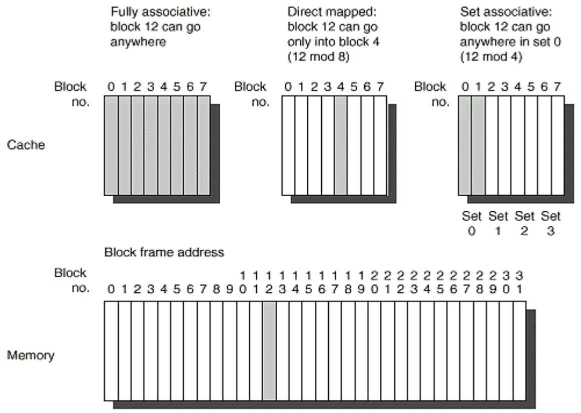
Block Address = 30 bits		Block offset = 2 bits
Tag = 20 bits	Index = 10 bits	

Direct mapped Cache



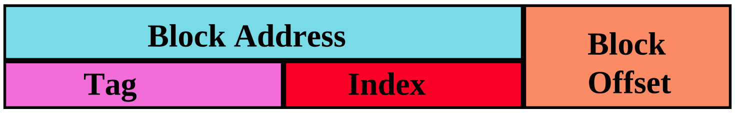
Block Address = 28 bits		Block offset = 4 bits
Tag = 16 bits	Index = 12 bits	

Cache Organization

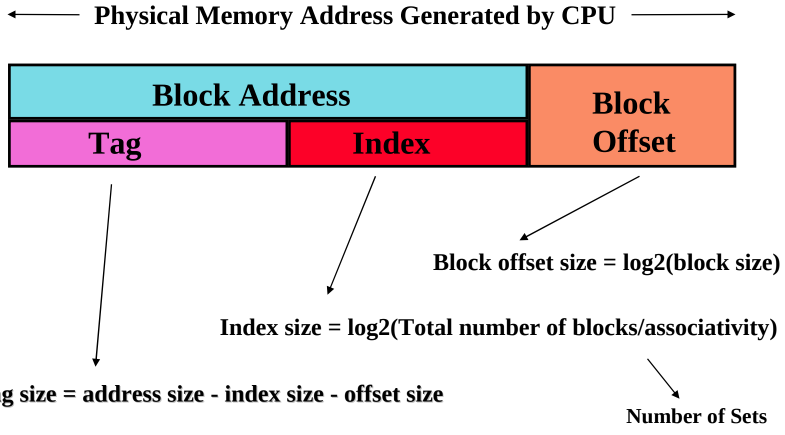


Cache Organization

- **Each block frame in cache has an address tag.**
- **The tags of every cache block that might contain the required data are checked in parallel.**
- **A valid bit is added to the tag to indicate whether this entry contains a valid address.**
- **The address from the CPU to cache is divided into:**
  - A block address, further divided into:
    - An index field to choose a block set in cache.
    - (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block**.**



Cache Organization



Physical Memory Address Generated by CPU

Block Address

Tag

Index

Block Offset

Block offset size = log2(block size)

Index size = log2(Total number of blocks/associativity)

Tag size = address size - index size - offset size

Number of Sets

Mapping function:

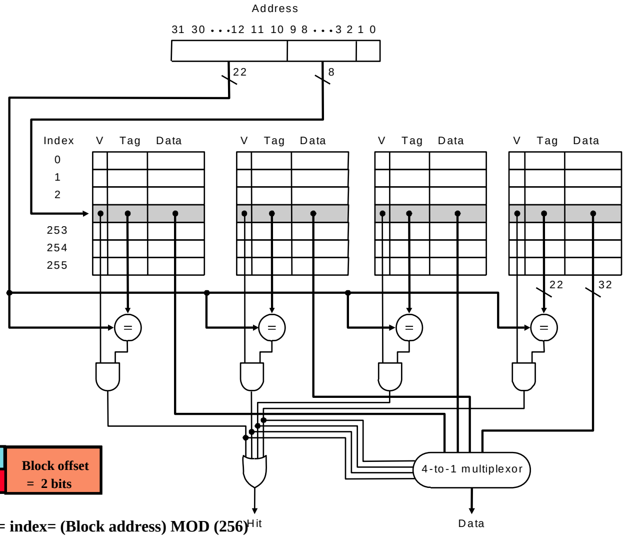
Cache set or block frame number = Index =

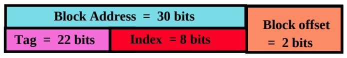
= (Block Address) MOD (Number of Sets)

Set Associative: 4K 4Way

1024 block frames
Each block = one word
4-way set associative
1024 / 4= 256 sets

Can cache up to
2^{32} bytes = 4 GB
of memory





Block Address = 30 bits

Tag = 22 bits

Index = 8 bits

Block offset = 2 bits

Mapping Function: Cache Set Number = index= (Block address) MOD (256)

Cache Replacement Policy

- **If a miss, we might have to choose a block ot be replaced**
  - Random:
    - Any block is randomly selected for replacement providing uniform allocation.
    - Simple to build in hardware.
    - The most widely used cache replacement strategy.
  - Least-recently used (LRU):
    - Accesses to blocks are recorded and the block replaced is the one that was not used for the longest period of time.
    - Full LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated by block usage bits that are cleared at regular time intervals.

Cache Operation

Associativity:	**2-way**		**4-way**		**8-way**	
Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	**5.18%**	**5.69%**	**4.67%**	**5.29%**	**4.39%**	**4.96%**
64 KB	**1.88%**	**2.01%**	**1.54%**	**1.66%**	**1.39%**	**1.53%**
256 KB	**1.15%**	**1.17%**	**1.13%**	**1.13%**	**1.12%**	**1.12%**

Cache Performance

- CPUtime = Instruction count x CPI x Clock cycle time
- CPIexecution = CPI with ideal memory
- CPI = CPIexecution + Mem Stall cycles per instruction
- Mem Stall cycles per instruction =

  Mem accesses per instruction x Miss rate x Miss penalty
- CPUtime = Instruction Count x (CPIexecution +

  Mem Stall cycles per instruction) x Clock cycle time
- CPUtime = IC x (CPIexecution + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time
- Misses per instruction = Memory accesses per instruction x Miss rate
- CPUtime = IC x (CPIexecution + Misses per instruction x Miss penalty) x

  Clock cycle time

Cache Performance

- **Assuming the following execution and cache parameters:**
  - Cache miss penalty = 50 cycles
  - Normal instruction execution CPI ignoring memory stalls = 2.0 cycles
  - Miss rate = 2%
  - Average memory references/instruction = 1.33
- **CPU time = IC x [CPI execution + Memory accesses/instruction x Miss rate x Miss penalty] x Clock cycle time**
- **CPUtime with cache = IC x (2.0 + (1.33 x 2% x 50)) x clock cycle time**
- **= IC x 3.33 x Clock cycle time**
- ***Lower CPI execution increases the impact of cache miss clock cycles***

Cache Performance

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- CPIexecution = 1.1
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.
- CPI = $\mathrm{CPI}_{\mathrm{execution}}$ + mem stalls per instruction
- Mem Stalls per instruction =
- Mem accesses per instruction x Miss rate x Miss penalty
- Mem accesses per instruction = 1 + .3 = 1.3
- Mem Stalls per instruction = 1.3 x .015 x 50 = 0.975
- CPI = 1.1 + .975 = 2.075
- The ideal memory CPU with no misses is 2.075/1.1 = 1.88 times faster

Cache Performance

- Suppose for the previous example we double the clock rate to 400 MHZ, how much faster is this machine, assuming similar miss rate, instruction mix?
- Since memory speed is not changed, the miss penalty takes more CPU cycles:
- Miss penalty = 50 x 2 = 100 cycles.
- CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05
- Speedup = (CPIold x Cold)/ (CPInew x Cnew)
- = 2.075 x 2 / 3.05 = 1.36
- The new machine is only 1.36 times faster rather than 2

times faster due to the increased effect of cache misses.

- *CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.*

Cache Performance

° **Suppose a CPU uses separate level one (L1) caches for instructions and data (Harvard memory architecture) with different miss rates for instruction and data access:**

- $CPI_{execution}$ = 1.1
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 0.5% for instruction fetch and a cache data miss rate of 6%.
- A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes. Find the resulting CPI using this cache? How much faster is the CPU with ideal memory?

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stall cycles per instruction = Instruction Fetch Miss rate x Miss Penalty + Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

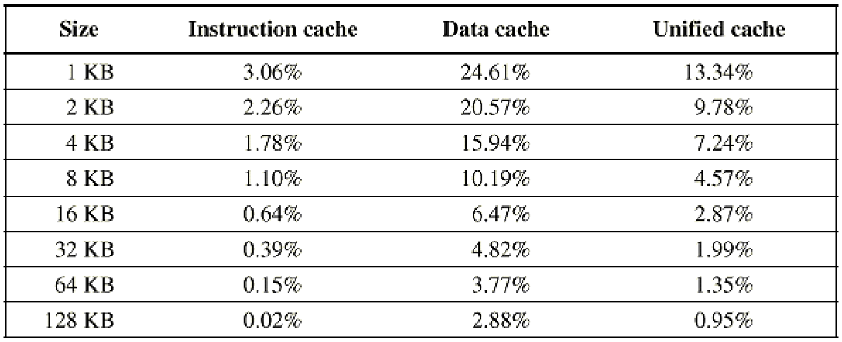
Mem Stall cycles per instruction = 1 x 0.5/100 x 200 + 0.3 x 6/100 x 200 = 1 + 3.6 = 4.6

CPI = $CPI_{execution}$ + mem stalls per instruction = 1.1 + 4.6 = 5.7

The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster

With no cache the CPI would have been = 1.1 + 1.3 X 200 = 261.1

Cache Performance



Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Cache Read/Write

- Statistical data suggest that reads (*including instruction fetches)* dominate processor cache accesses (writes account for 25% of data cache traffic).
- In cache reads, a block is read at the same time while the tag is being compared with the block address. If the read is a hit the data is passed to the CPU, if a miss it ignores it.
- In cache writes, modifying the block cannot begin until the tag is checked to see if the address is a hit.
- Thus for cache writes, tag checking cannot take place in parallel, and only the specific data (between 1 and 8 bytes) requested by the CPU can be modified.
- Cache can be classified according to the write and memory update strategy in place: write through, or write back.

1 **Write Though: Data is written to both the cache block and to a block of main memory.**

- The lower level always has the most updated data; an important feature for I/O and multiprocessing.
- Easier to implement than write back.
- A write buffer is often used to reduce CPU write stall while data is written to memory.

2 **Write back: Data is written or updated only to the cache block. The modified or dirty cache block is written to main memory when it’s being replaced from cache.**

- Writes occur at the speed of cache
- A status bit called a dirty or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory when replaced.
- Uses less memory bandwidth than write through.

Write Policy

Write Allocate:

The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:

The block is modified in the lower level (lower cache level, or main

memory) and not loaded into cache.

Example

- **Which has a lower miss rate 16KB cache for both instruction or data, or a combined 32KB cache? (0.64%, 6.47%, 1.99%).**
- **Assume hit=1cycle and miss =50 cycles. 75% of memory references are instruction fetch.**
- Miss rate of split cache=0.75*0.64%+0.25*6.47%=2.1%
- Slightly worse than 1.99% for combined cache. But, what about average memory access time?
- Split cache: 75%(1+0.64%*50)+25%(1+6.47%*50) = 2.05 cycles.
- Combined cache: 75%(1+1.99*50)+25%(1+1+1.99%*50) = 2.24

Extra cycle for load/store

example

- A CPU with $CPI_{execution}$ = 1.1 Mem accesses per instruction = 1.3
- Uses a unified L1 Write Through, No Write Allocate, with:
  - No write buffer.
  - Perfect Write buffer
  - A realistic write buffer that eliminates 85% of write stalls
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

$$CPI = CPI_{execution} + \text{mem stalls per instruction}$$

% reads = 1.15/1.3 = 88.5%   % writes = .15/1.3 = 11.5%

Example

- A CPU with $CPI_{execution}$ = 1.1 uses a unified L1 with with write back, with write allocate, and the probability a cache block is dirty = 10%
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

1.3 mem refs / inst

$$\frac{1.5}{100}\left(1.3 \times 50 \times 0.9 + 1.3 \times 0.1 \times 100\right)$$

Example

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHz with a miss rate of 5%
- L_2 cache operates at 250 MHz with local miss rate 40%, (T_2 = 2 cycles)
- Memory access penalty, M = 100 cycles. Find CPI.

$$1.3\left(\frac{5}{100}\times 0.6\times 2+\frac{5}{100}\times 0.4\times 100\right)$$

Example

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- For L_1 :
  - Cache operates at 500 MHz with a miss rate of 1-H1 = 5%
  - Write though to L_2 with perfect write buffer with write allocate
- For L_2:
  - Cache operates at 250 MHz with local miss rate 1- H2 = 40%, (T_2 = 2 cycles)
  - Write back to main memory with write allocate
  - Probability a cache block is dirty = 10%
- Memory access penalty, M = 100 cycles. Find CPI.

$$0.05\left(0.6\times 2+0.4\times 0.9\times 100 + 0.4\times 0.1\times 200\right)$$

Example

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHz with a miss rate of 5%
- L_2 cache operates at 250 MHz with a local miss rate 40%, (T_2 = 2 cycles)
- L_3 cache operates at 100 MHz with a local miss rate 50%, (T_3 = 5 cycles)
- Memory access penalty, M= 100 cycles. Find CPI.

HW

Caches -- Performance

- Classification of misses
  - Compulsory
  - Capacity
  - Conflict
- How to Improve Cache Performance
  - Reduce miss rate
  - Reduce miss penalty
  - Reduce hit time

1**Compulsory:** On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.

2**Capacity:** Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).

3**Conflict:** In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.

Improving Cache Performance

- **Reducing hit time**
  1. **Giving Reads Priority over Writes**

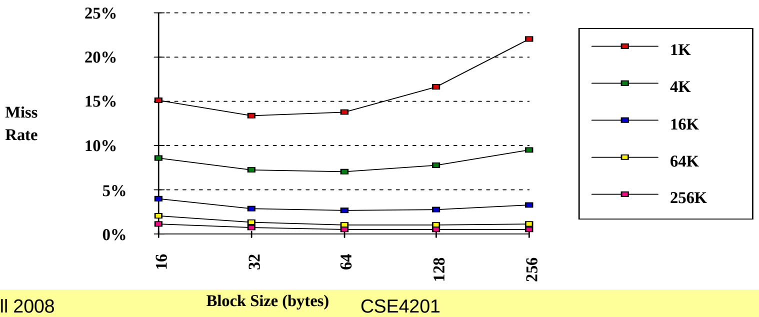
     E.g., Read complete before earlier writes in write buffer
  2. **Avoiding Address Translation during Cache Indexing**

- **Reducing Miss Penalty**
  3. **Multilevel Caches**

     Faster RAM

- **Reducing Miss Rate**
  4. **Larger Block size (Compulsory misses)**
  5. **Larger Cache size (Capacity misses)**
  6. **Higher Associativity (Conflict misses)**

- **Larger Block Size**
- **Larger block size reduces misses up to a point, then start to increase it (larger block size takes advantage of spatial locality, but it decrease the number of different block in the cache and increases the miss penalty since it will take longer to load the block from the memory to the cache)**



Larger Block Size

- **Larger block size**
- **From the last graph, if the memory takes 40 cycles overhead and then deliver 16 bytes every 2 cycle. Compare between 16,128 block size for 64K cache.**
- **16 byte block** access time = 1+(1.06%*42)=1.5088
- **128 bytes block** access time = 1+(1.02*56)=1.5712

Reducing Cache Miss

Higher Associativity

- Practically 8-way set associative is as god as a fully associative
- The 2:1 cache rule of thumb states that a direct-mapped cache of size N has the same miss rate as a 2-way set associative of size N/2
- One problem with associative caches is that we need to increase the clock cycle (at least, we need a MUX to choose which set)
- Practically 10% increase in clock time for TTL, or ECL, and 2% for custom CMOS (when we go from direct mapped to 2-way set associative).

Reducing Cache Miss

Victim Caches

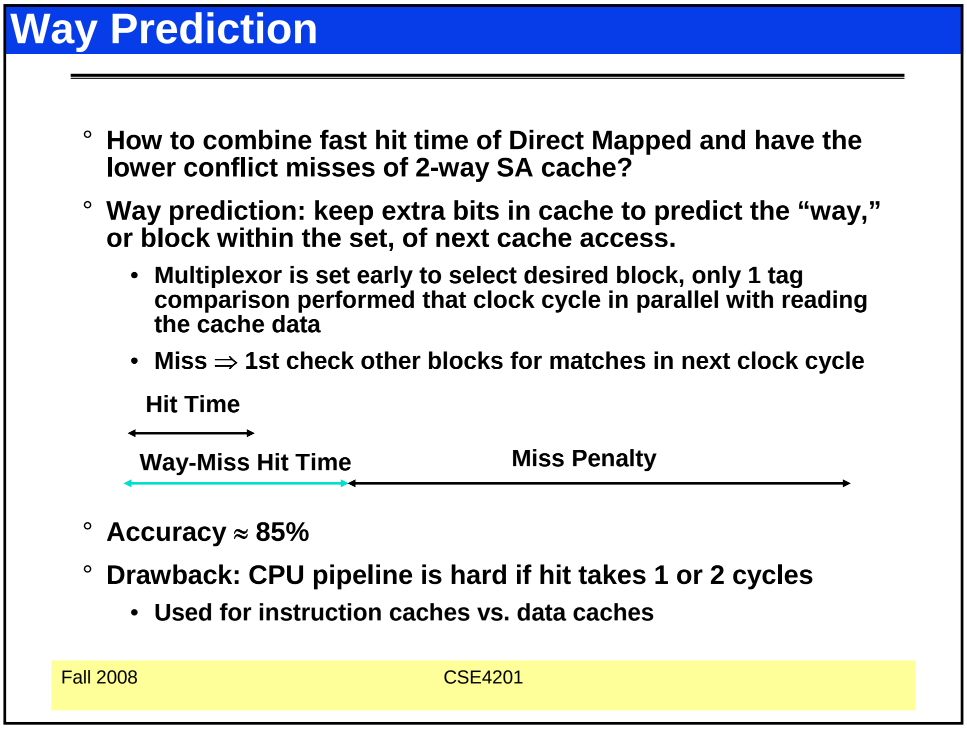
- Add a small fully associative cache between the cache and the memory.
- This *victim* cache contain only blocks that were discarded from the original cache.
- The victim cache is checked on miss, if the data is found there, it will be swapped with the data in the cache.
- A small victim cache of 1-5 blocks is sufficient (4-block victim cache removed 20% to 95% of conflict misses.

Pseudo Associative Caches

- Pseudo-Associative Cache
- This cache behaves like direct-mapped
- On a miss, before going to the main memory, check another cache entry
- 2 hit times, one slow and one fast
- Example
- Compare between direct-mapped, 2-way set associative and pseudo associative (2 extra cycles fro pseudo hit)
- Direct = 1+9.8%*50 = 5.9
- 2-way = 1.1 +7.6%*50 = 4.9 (10% increase in Tc)
- Pseudo = 1 + (9.8%-7.6%)*2 + 7.6%*50 = 4.844

Advanced Optimization Techniques

- Way prediction
- Trace caches
- Pipelined cache access
- Nonblocking caches
- Multibanked caches
- Early start and critical word first
- Merging Write Buffers
- Compiler optimizations to reduce miss rate
- Prefetching



Way Prediction

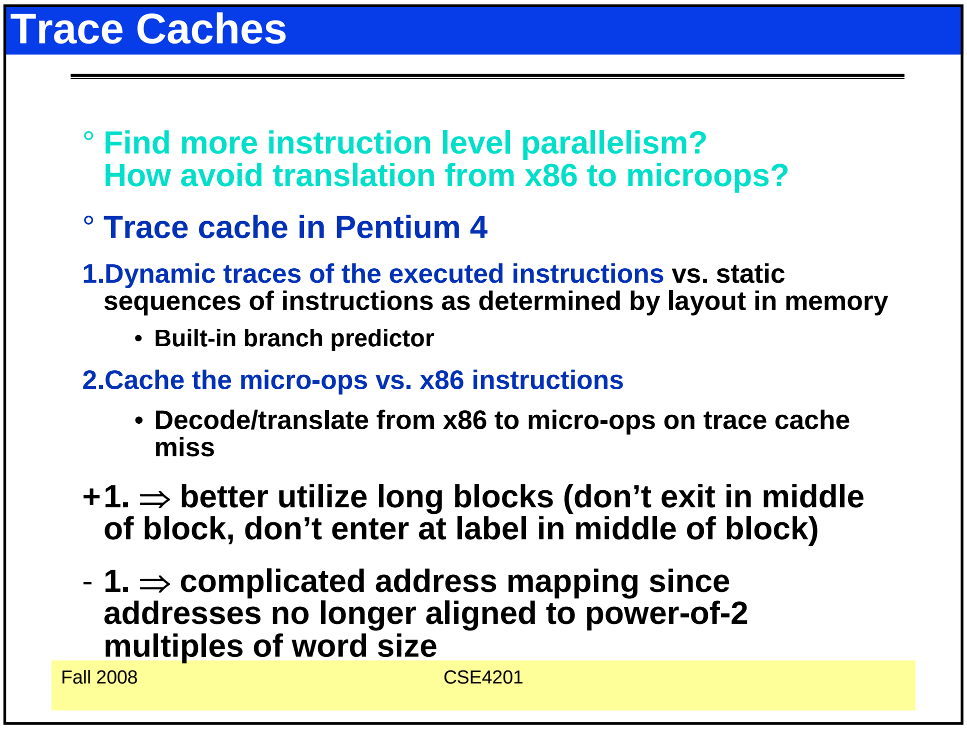
- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way prediction: keep extra bits in cache to predict the “way,” or block within the set, of next cache access.
  - Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
  - Miss ⇒ 1st check other blocks for matches in next clock cycle

Hit Time

Way-Miss Hit Time

Miss Penalty

- Accuracy ≈ 85%
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Used for instruction caches vs. data caches



Trace Caches

- Find more instruction level parallelism?
  How avoid translation from x86 to microops?
- Trace cache in Pentium 4

1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
   - Built-in branch predictor
2. Cache the micro-ops vs. x86 instructions
   - Decode/translate from x86 to micro-ops on trace cache miss

+1. ⇒ better utilize long blocks (don’t exit in middle of block, don’t enter at label in middle of block)

- 1. ⇒ complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size

Pipelined Cache Access

° Pipeline cache access to maintain bandwidth, but higher latency

° Instruction cache access pipeline stages:

1: Pentium

2: Pentium Pro through Pentium III

4: Pentium 4

- ⇒ greater penalty on mispredicted branches (increases the number of pipeline stages)

- ⇒ more clock cycles between the issue of the load and the use of the data

Nonblocking Caches

° *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss

- requires F/E bits on registers or out-of-order execution
- requires multi-bank memories

° "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests

° "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses

- Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
- Requires muliple memory banks (otherwise cannot support)
- Penium Pro allows 4 outstanding memory misses

Multiple Banks

° Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses

  - E.g.,T1 (“Niagara”) L2 has 4 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; …

Early Start and Critical Word First

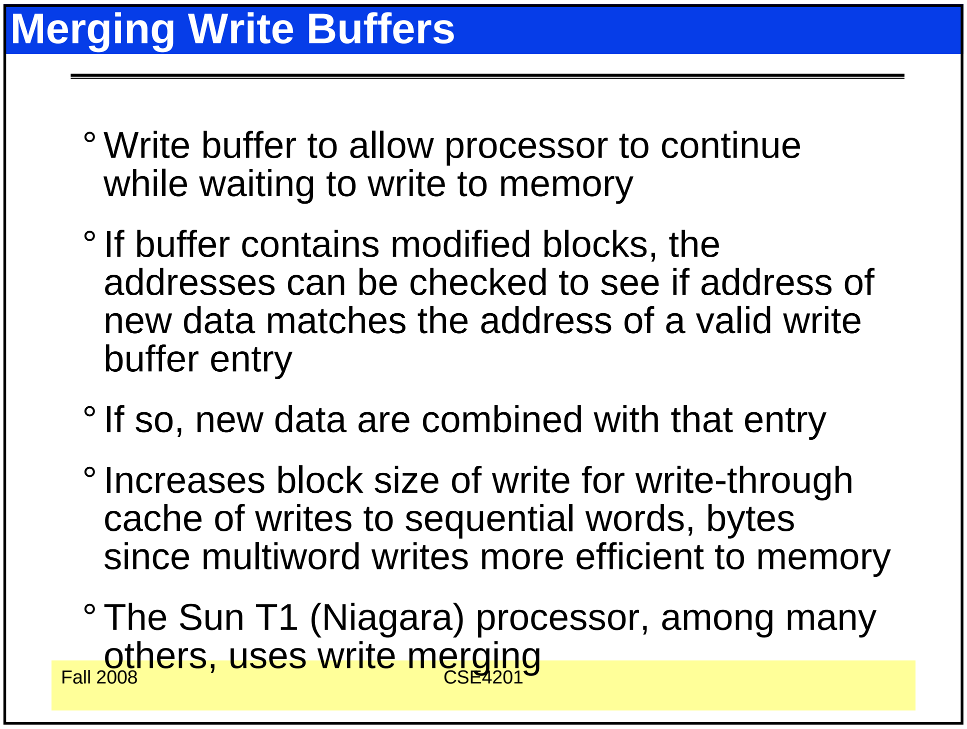
° Don’t wait for full block before restarting CPU

° ***Early restart***—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution

  - Spatial locality ⇒ tend to want next sequential word, so not clear size of benefit of just early restart

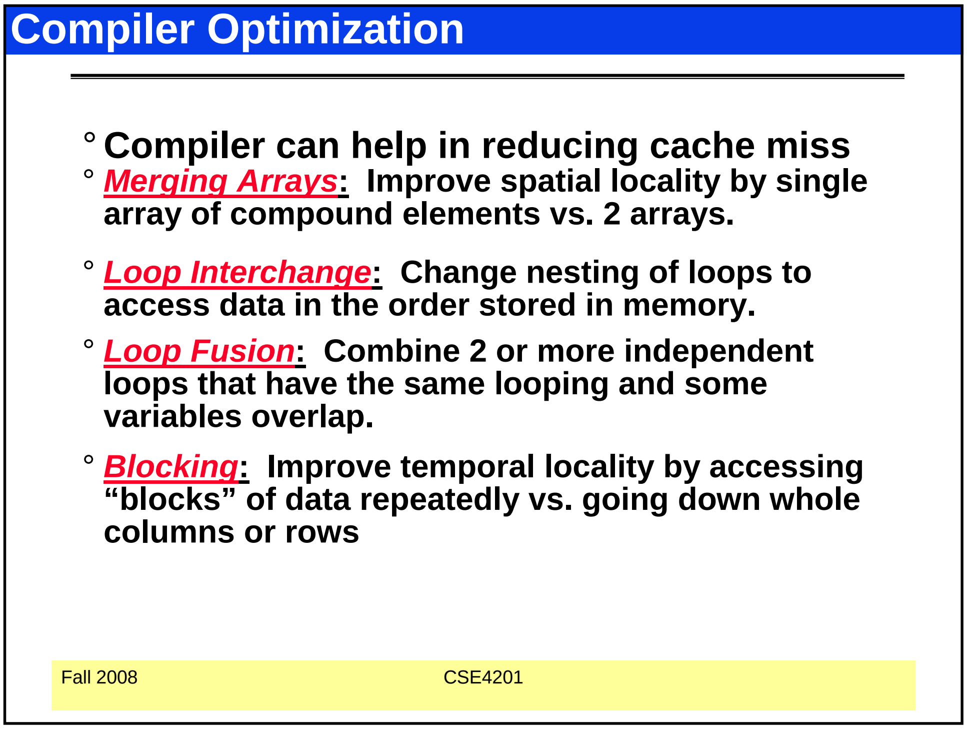
° ***Critical Word First***—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block

  - Long blocks more popular today ⇒ Critical Word 1st Widely used



Merging Write Buffers

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging



Compiler Optimization

- **Compiler can help in reducing cache miss**
- ***Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 arrays.**
- ***Loop Interchange*: Change nesting of loops to access data in the order stored in memory.**
- ***Loop Fusion*: Combine 2 or more independent loops that have the same looping and some variables overlap.**
- ***Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows**

Merging Arrays

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

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

Reduces conflict between val and key and reduces compulsory misses if they are accessed in the same pattern

Interchanging Loops

° Assume row-major matrix allocation

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

Sequential accesses instead of striding through memory every 100 words in this case improves spatial locality (reduces compulsory misses)

Loop Fusion

```
/* 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];}
```

Blocking

```
for (i = 0; i < 12; i = i+1)
 for (j = 0; j < 12; j = j+1)
      {r = 0;
       for (k = 0; k < 12; k = k+1){
             r = r + y[i][k]*z[k][j];};
       x[i][j] = r;
      };
/* After Blocking */
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,N); j = j+1)
      {r = 0;
       for (k = kk; k < min(kk+B,N); k = k+1) {
             r = r + y[i][k]*z[k][j];};
       x[i][j] = x[i][j] + r;
      };
```