Outline

- Memory Hierarchy
- Four Questions for Memory Hierarchy
- Cache Performance
  - 1. Reduce the miss rate,
  - 2. Reduce the miss penalty, or
  - 3. Reduce the time to hit in the cache.

Solution: The Memory Hierarchy (MH)

User sees as much memory as is available in cheapest technology and access it at the speed offered by the fastest technology.

Levels in Memory Hierarchy:
- Upper
- Lower

Processor

Control

Datapath

Fastest

Slowest

Smallest

Largest

Levels
Generations of Microprocessors

- Time of a full cache miss in instructions executed:
  - 1st Alpha: 340 ns/5.0 ns = 68 clks x 2 or 136
  - 2nd Alpha: 266 ns/3.3 ns = 80 clks x 4 or 320
  - 3rd Alpha: 180 ns/1.7 ns = 108 clks x 6 or 648
  - 1/2X latency x 3X clock rate x 3X Instr/clock = -5X

Why hierarchy works?

- Principle of locality
  - Temporal locality: recently accessed items are likely to be accessed in the near future
    ⇒ Keep them close to the processor
  - Spatial locality: items whose addresses are near one another tend to be referenced close together in time
    ⇒ Move blocks consisted of contiguous words to the upper level

Cache Measures

- Hit: data appears in some block in the upper level (Bl. X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: time to access the upper level (RAM access time + Time to determine hit/miss)
- Miss: data needs to be retrieved from the lower level (Bl. Y)
  - Miss rate: 1 - Hit Rate
  - Miss penalty: time to replace a block in the upper level + time to retrieve the block from the lower level
- Average memory-access time:
  = Hit time + Miss rate x Miss penalty (ns or clocks)

Levels of the Memory Hierarchy

- Rule of thumb: Programs spend 90% of their execution time in only 10% of code

Four Questions for Memory Heir.

Q#1: Where can a block be placed in the upper level?
⇒ Block placement
- direct-mapped, fully associative, set-associative

Q#2: How is a block found if it is in the upper level?
⇒ Block identification

Q#3: Which block should be replaced on a miss?
⇒ Block replacement
- Random, LRU (Least Recently Used)

Q#4: What happens on a write?
⇒ Write strategy
- Write-through vs. write-back
- Write allocate vs. No-write allocate

In a direct-mapped cache, each memory address is associated with one possible block within the cache.
- Therefore, we only need to look in a single location in the cache for the data if it exists in the cache.
- Block is the unit of transfer between cache and memory.
Direct-Mapped Cache (cont’d)

- Since multiple memory addresses map to the same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Result: divide memory address into three fields:
  - **TAG**: to check if have the correct block
  - **INDEX**: to select block
  - **OFFSET**: to select byte within the block

```
Block Address

TTTTTTTTTTTTTTTTTT TTTTTTTTTTTTTTTTTTT

TAG: to check if have the correct block
INDEX: to select block
OFFSET: to select byte within the block
```

Direct-Mapped Cache Terminology

- **INDEX**: specifies the cache index (which “row” of the cache we should look in)
- **OFFSET**: once we have found correct block, specifies which byte within the block we want
- **TAG**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location
- **BLOCK ADDRESS**: TAG + INDEX

Direct-Mapped Cache Example

- Conditions
  - 32-bit architecture (word=32 bits), address unit is byte
  - 8KB direct-mapped cache with 4 words blocks
- Determine the size of the Tag, Index, and Offset fields
  - **OFFSET** (specifies correct byte within block):
    - cache block contains 4 words = 16 (2^4) bytes ⇒ 4 bits
  - **INDEX** (specifies correct row in the cache):
    - cache size is 8KB=2^13 bytes, cache block is 2^4 bytes
    - #Rows in cache (1 block = 1 row): 2^13/2^4 = 2^9 = 512 ⇒ 9 bits
  - **TAG**: Memory address length - offset - index = 32 - 4 - 9 = 19 ⇒ tag is leftmost 19 bits

1 KB Direct Mapped Cache, 32B blocks

- For a 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

Q2: How is a block found if it is in the upper level?

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

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Assoc.</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Rand</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Q4: What happens on a write?

- Write through—The information is written to both the block in the cache and to the block in the lower-level memory.
- Write back—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Is block clean or dirty?
  - Pros and Cons of each?
- WT: read misses cannot result in writes
- WB: no repeated writes to same location
- WT always combined with write buffers so that don’t wait for lower level memory

Write stall in write through caches

- When the CPU must wait for writes to complete during write through, the CPU is said to write stall
- Common optimization
  => Write buffer which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with memory updating
- However, write stalls can occur even with write buffer (when buffer is full)

Write Buffer for Write Through

A Write Buffer is needed between the Cache and Memory
- Processor: writes data into the cache and the write buffer
- Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
  - Memory system designer’s nightmare:
    - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
    - Write buffer saturation

What to do on a write-miss?

- Write allocate (or fetch on write)
  The block is loaded on a write-miss, followed by the write-hit actions
- No-write allocate (or write around)
  The block is modified in the memory and not loaded into the cache
- Although either write-miss policy can be used with write through or write back, write back caches generally use write allocate and write through often use no-write allocate
An Example: The Alpha 21264 Data Cache (64KB, 64-byte blocks, 2w)

Cache Performance

- Hit Time = time to find and retrieve data from current level cache
- Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- Hit Rate = % of requests that are found in current level cache
- Miss Rate = 1 - Hit Rate

Cache Performance (cont’d)

- Average memory access time (AMAT)
  \[ \text{AMAT} = \text{Hit time} + \text{Miss Rate} \times \text{Miss Penalty} \]
  \[ = \% \text{ instructions} \times (\text{Hit time}_{\text{Inst}} + \text{Miss Rate}_{\text{Inst}} \times \text{Miss Penalty}_{\text{Inst}}) \]
  \[ + \% \text{ data} \times (\text{Hit time}_{\text{Data}} + \text{Miss Rate}_{\text{Data}} \times \text{Miss Penalty}_{\text{Data}}) \]

An Example: Unified vs. Separate I&D

- Compare 2 design alternatives (ignore L2 caches)?
  - 16KB I&D: Inst misses=3.82 /1K, Data miss rate=40.9 /1K
  - 32KB unified: Unified misses = 43.3 misses/1K

Assumptions:
- I/D frequency is 36% ⇒ 74% accesses from instructions (1.0/1.36)
- Hit time = 1 clock cycle, miss penalty = 100 clock cycles
- Data hit has 1 stall for unified cache (only one port)
Unified vs. Separate I&D (cont'd)

- Miss rate (L1I) = (# L1I misses) / (IC)
- #L1I misses = (L1I misses per 1k) * (IC / 1000)
- Miss rate (L1I) = 3.82/1000 = 0.0038

- Miss rate (L1D) = (# L1D misses) / (# Mem. Refs)
- #L1D misses = (L1D misses per 1k) * (IC / 1000)
- Miss rate (L1D) = 40.9*(IC / 1000) / (.36*IC) = 0.1136

- Miss rate (L1U) = (# L1U misses) / (IC + Mem. Refs)
- #L1U misses = (L1U misses per 1k) * (IC / 1000)
- Miss rate (L1U) = 43.3*(IC / 1000) / (.36 + IC) = 0.0318

AMAT and Processor Performance

- Miss-oriented Approach to Memory Access
- CPIExec includes ALU and Memory instructions

\[
\text{CPU time} = \frac{IC \times CPI_{\text{Exec}} + \frac{\text{MemAccess}}{\text{Inst}} \times \text{MissRate} \times \text{MissPenalty}}{\text{Clock rate}}
\]

\[
\text{CPU time} = \frac{IC \times CPI_{\text{Exec}} + \frac{\text{MemMisses}}{\text{Inst}} \times \text{MissPenalty}}{\text{Clock rate}}
\]

Unified vs. Separate I&D (cont'd)

- AMAT (split) = (% instr.) * (hit time + L1I miss rate * Miss Pen.) + (% data) * (hit time + L1D miss rate * Miss Pen.) = .74(1 + .0038*100) + .26(1 + .1136*100) = 4.2348 clock cycles
- AMAT (unif.) = (% instr.) * (hit time + L1Umiss rate * Miss Pen.) + (% data) * (hit time + L1U miss rate * Miss Pen.)
  = .74(1 + .0318*100) + .26(1 + .0318*100)
  = 4.44 clock cycles

AMAT and Processor Performance (cont'd)

- Separating out Memory component entirely
- AMAT = Average Memory Access Time
- CPIALUOps does not include memory instructions

\[
\text{CPU time} = \frac{IC \times \frac{\text{ALUops}}{\text{Inst}} + CPI_{\text{ALUops}} + \frac{\text{MemAccess}}{\text{Inst}} \times \text{AMAT}}{\text{Clock rate}}
\]

\[
\text{AMAT} = \text{Hit time} + \text{Miss Rate} \times \text{Miss Penalty}
\]

\[
= \% \text{ instructions} \times (\text{Hit time} + \text{Miss Rate}_{\text{inst}} \times \text{Miss Penalty}_{\text{inst}}) + \% \text{ data} \times (\text{Hit time}_{\text{data}} + \text{Miss Rate}_{\text{data}} \times \text{Miss Penalty}_{\text{data}})
\]
Summary: Caches

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity
- Write Policy:
  - Write Through: needs a write buffer.
  - Write Back: control can be complex
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?

Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (l-cache, D-cache, TLB)
  - depends on technology / cost
  - Simplicity often wins

Where Misses Come From?

- Classifying Misses: 3 Cs
  - Compulsory — The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses. (Misses in even an Infinite Cache)
  - Capacity — If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. (Misses in Fully Associative Size X Cache)
  - Conflict — If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (Misses in N-way Associative, Size X Cache)
- More recent, 4th “C”:
  - Coherence — Misses caused by cache coherence.

How to Improve Cache Performance?

- Cache optimizations
  1. Reduce the miss rate
  2. Reduce the miss penalty
  3. Reduce the time to hit in the cache

\[ AMAT = \text{HitTime} + \text{MissRate} \times \text{MissPenalty} \]
Cache Organization?

- Assume total cache size not changed
- What happens if:
  - Change Block Size
  - Change Cache Size
  - Change Cache Internal Organization
  - Change Associativity
  - Change Compiler
- Which of 3Cs is obviously affected?
1st Miss Rate Reduction Technique: Larger Block Size (cont’d)

- Example:
  - Memory system takes 40 clock cycles of overhead, and then delivers 16 bytes every 2 clock cycles
  - Miss rate vs. block size (see table): hit time is 1 cc
  - AMAT? AMAT = Hit Time + Miss Rate \times Miss Penalty

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>BS MP</th>
<th>1K 4K 16K 64K 256K</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>15.05</td>
<td>8.87 3.94 2.04 1.00</td>
</tr>
<tr>
<td>32</td>
<td>13.54</td>
<td>7.24 2.87 1.35 0.70</td>
</tr>
<tr>
<td>64</td>
<td>12.16</td>
<td>5.00 2.66 1.36 0.87</td>
</tr>
<tr>
<td>128</td>
<td>10.64</td>
<td>3.78 2.77 1.02 0.49</td>
</tr>
<tr>
<td>256</td>
<td>22.01</td>
<td>8.61 3.26 1.76 0.49</td>
</tr>
</tbody>
</table>

- Block size depends on both latency and bandwidth of lower level memory
- low latency and bandwidth \(\Rightarrow\) decrease block size
- high latency and bandwidth \(\Rightarrow\) increase block size

2nd Miss Rate Reduction Technique: Larger Caches

- Reduce Capacity misses
- Drawbacks: Higher cost, Longer hit time

3rd Miss Rate Reduction Technique: Higher Associativity

- Miss rates improve with higher associativity
- Two rules of thumb
  - 8-way set-associative is almost as effective in reducing misses as fully-associative cache of the same size
  - 2:1 Cache Rule: Miss Rate DM cache size \(N\) = Miss Rate 2-way cache size \(N/2\)
- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%
3rd Miss Rate Reduction Technique: Higher Associativity (cont’d)

- Example
  - \( \text{CCT2-way} = 1.10 \times \text{CCT1-way}, \)
  - \( \text{CCT4-way} = 1.12 \times \text{CCT1-way}, \) \( \text{CCT8-way} = 1.14 \times \text{CCT1-way} \)
  - Hit time = 1 cc, Miss penalty = 50 cc
  - Find AMAT using miss rates from Fig 5.9 (old textbook)

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{CSize [KB]} & \text{1-way} & \text{2-way} & \text{4-way} & \text{8-way} \\
\hline
1 & 7.85 & 6.60 & 6.22 & 5.44 \\
4 & 5.00 & 4.90 & 4.84 & 4.60 \\
8 & 3.30 & 3.00 & 2.87 & 2.59 \\
32 & 2.00 & 1.80 & 1.77 & 1.78 \\
64 & 1.70 & 1.60 & 1.57 & 1.59 \\
128 & 1.50 & 1.45 & 1.42 & 1.44 \\
\hline
\end{array}
\]

4th Miss Rate Reduction Technique: Way Prediction, “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way Prediction: extra bits are kept to predict the way or block within a set
  - Mux is set early to select the desired block
  - Only a single tag comparison is performed
  - What if miss?
    - => check the other blocks in the set
- Used in Alpha 21264 (1 bit per block in IC$)
  - 1 cc if predictor is correct, 3 cc if not
  - Effectiveness: prediction accuracy is 85%
- Used in MIPS 4300 embedded proc. to lower power

Example: Pseudo-Associativity

- Pseudo-Associative Cache
  - Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit. (slow hit)
  - Accesses proceed just as in the DM cache for a hit
  - On a miss, check the second entry
    - Simple way is to invert the MSB bit of the INDEX field to find the other block in the “pseudo set”

\[
\begin{array}{|c|c|c|}
\hline
\text{Hit Time} & \text{Pseudo Hit Time} & \text{Miss Penalty} \\
\hline
\text{Time} & \text{CSize [KB]} & \text{1-way} & \text{2-way} & \text{Pseudo} \\
\hline
2 & 5.90 & 4.90 & 4.84 \\
128 & 1.50 & 1.45 & 1.35 \\
\hline
\end{array}
\]

- What if too many hits in the slow part?
  - swap contents of the blocks
5th Miss Rate Reduction Technique: Compiler Optimizations

- Reduction comes from software (no Hw ch.)
- McFarling [1989] reduced caches misses by 75% (6KB, DM, 4 byte blocks) in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Loop Interchange

- Motivation: some programs have nested loops that access data in nonsequential order
- Solution: Simply exchanging the nesting of the loops can make the code access the data in the order it is stored => reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded

Loop Interchange Example

```c
/* 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];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality.

Reduces misses if the arrays do not fit in the cache.

Blocking

- Motivation: multiple arrays, some accessed by rows and some by columns
- Storing the arrays row by row (row major order) or column by column (column major order) does not help: both rows and columns are used in every iteration of the loop (Loop Interchange cannot help)
- Solution: instead of operating on entire rows and columns of an array, blocked algorithms operate on submatrices or blocks => maximize accesses to the data loaded into the cache before the data is replaced
Blocking Example

/* 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;
;
- 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 (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

/* 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;
    }

Blocking Example (cont’d)

- B called Blocking Factor
- Capacity Misses from 2N^3 + N^2 to N^3/B+2N^2
- Conflict Misses Too?

Merging Arrays

- Motivation: some programs reference multiple arrays in the same dimension with the same indices at the same time => these accesses can interfere with each other, leading to conflict misses
- Solution: combine these independent matrices into a single compound array, so that a single cache block can contain the desired elements

Merging Arrays Example

/* 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];
Loop Fusion

- Some programs have separate sections of code that access with the same loops, performing different computations on the common data.

- Solution:
  "Fuse" the code into a single loop => the data that are fetched into the cache can be used repeatedly before being swapped out => reducing misses via improved temporal locality.

Loop Fusion Example

```c
/* 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];}
```

2 misses per access to a & c vs. one miss per access; improve temporal locality.

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)

<table>
<thead>
<tr>
<th>Performance Improvement</th>
<th>Array Merging</th>
<th>Loop Interchange</th>
<th>Loop Fusion</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta (nasa7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gatsby (nasa7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomscaty</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brix (nasa7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mwim (nasa7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spvec</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cholesky</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary: Miss Rate Reduction

\[
\text{CPU time} = \frac{1}{\text{Clock rate}} \times (\text{CPI} + \text{MemAccess CPI}) \times (\text{Inst} \times \text{MissRate} \times \text{MissPenalty})
\]

- 3 Cs: Compulsory, Capacity, Conflict
  1. Larger Cache => Reduce Capacity
  2. Larger Block Size => Reduce Compulsory
  3. Higher Associativity => Reduce Conflicts
  4. Way Prediction & Pseudo-Associativity
  5. Compiler Optimizations
Reducing Miss Penalty

- **Motivation**
  - AMAT = Hit Time + Miss Rate x Miss Penalty
  - Technology trends => relative cost of miss penalties increases over time
- **Techniques that address miss penalties**
  1. Multilevel Caches
  2. Critical Word First and Early Restart
  3. Giving Priority to Read Misses over Writes
  4. Merging Write Buffer
  5. Victim Caches

1st Miss Penalty Reduction Technique: Multilevel Caches

- **Architect’s dilemma**
  - Should I make the cache faster to keep pace with the speed of CPUs
  - Should I make the cache larger to overcome the widening gap between CPU and main memory
- **L2 Equations**
  - \( \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1} \)
  - \( \text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2} \)
  - \( \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} + \text{Miss Penalty}_{L2}) \)
- **Definitions**:
  - Local miss rate—misses in this cache divided by the total number of memory accesses to this cache (Miss rateL2)
  - Global miss rate—misses in this cache divided by the total number of memory accesses generated by the CPU (Miss RateL1 x Miss RateL2)

Reducing Misses: Which apply to L2 Cache?

- **Reducing Miss Rate**
  1. Reduce Capacity Misses via Larger Cache
  2. Reduce Compulsory Misses via Larger Block Size
  3. Reduce Conflict Misses via Higher Associativity
  4. Reduce Conflict Misses via Way Prediction & Pseudo-Associativity
  5. Reduce Conflict/Capac. Misses via Compiler Optimizations
**L2 cache block size & A.M.A.T.**

- 32KB L1, 8 byte path to memory

<table>
<thead>
<tr>
<th>Block Size</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative CPU Time</td>
<td>1.36</td>
<td>1.28</td>
<td>1.37</td>
<td>1.24</td>
<td>1.54</td>
<td>1.95</td>
</tr>
</tbody>
</table>

**Multilevel Inclusion: Yes or No?**

- Inclusion property:
  - L1 data are always present in L2
  - Good for I/O & caches consistency (L1 is usually WT, so valid data are in L2)

- Drawback: What if measurements suggest smaller cache blocks for smaller L1 caches and larger blocks for larger L2 caches?
  - E.g., Pentium4: 64B L1 blocks, 128B L2 blocks
  - Add complexity: when replace a block in L2 should discard 2 blocks in the L1 cache => increase L1 miss rate

- What if the budget for a L2 cache is slightly bigger than the L1 cache => L2 keeps redundant copy of L1
  - Multilevel Exclusion: L1 data is never found in a L2 cache
    - E.g., AMD Athlon uses this:
      - 64KB L1I$ + 64KB L1D$ vs. 256KB L2U$

**2nd Miss Penalty Reduction Technique:** Early Restart and Critical Word First

- Don’t wait for full block to be loaded 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
- 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.
  - Also called wrapped fetch and requested word first
- Generally useful only in large blocks
- Problem of spatial locality: tend to want next sequential word, so not clear if benefit by early restart and CWF

**3rd Miss Penalty Reduction Technique:** Giving Read Misses Priority over Writes

- Delayed Write Buffer
  - CPU writes to the Delayed Write Buffer
  - If read, data is transferred to the CPU
  - If write, data is transferred to the cache
- Lower level memory

- Write buffer
  - Data is transferred to the Delayed Write Buffer
  - Address is sent to the cache

- Tag
  - Check if tag matches
  - If match, data is transferred to the CPU
  - If no match, data is transferred to the Delayed Write Buffer

- 2 MB
  - Data is transferred to the Delayed Write Buffer
  - If read, data is transferred to the CPU
  - If write, data is transferred to the cache
3rd Miss Penalty Reduction Technique: Read Priority over Write on Miss (2)

- Write-through with write buffers offer RAW conflicts with main memory reads on cache misses.
  
  **Example:** DM, WT, 512 & 1024 map to the same block:
  
  ```
  SW 512(R0), R3 ; cache index 0
  LW R1, 1024(R0) ; cache index 0
  LW R2, 512(R0) ; cache index 0
  ```

  - If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
  - Check write buffer contents before read; if no conflicts, let the memory access continue
  - Write-back also want buffer to hold misplaced blocks
    - Read miss replacing dirty block
    - Normal: Write dirty block to memory, and then do the read
      Instead copy the dirty block to a write buffer, then do the read, and then do the write
    - CPU stall less since restarts as soon as do read

4th Miss Penalty Reduction Technique: Merging Write Buffer

- Write Through caches relay on write-buffers
  
  - on write, data and full address are written into the buffer; write is finished from the CPU's perspective
  - Problem: WB full stalls
  - Write merging
    - multibit writes are faster than a single word writes => reduce write-buffer stalls
  
Is this applicable to I/O addresses?

5th Miss Penalty Reduction Technique: Victim Caches

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
  - Idea: Add buffer to place data discarded from cache in the case it is needed again
  
  - Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
  - Used in Alpha, HP machines, AMD Athlon (8 entries)

Summary of Miss Penalty Reducing Techniques

- 1. Multilevel Caches
- 2. Critical Word First and Early Restart
- 3. Giving Priority to Read Misses over Writes
- 4. Merging Write Buffer
- 5. Victim Caches
Reducing Cache Miss Penalty or Miss Rate via Parallelism

- **Idea:** overlap the execution of instructions with activity in memory hierarchy
- **Miss Rate/Penalty reduction techniques**
  1. **Nonblocking caches**
     - reduce stalls on cache misses in CPUs with out-of-order completion
  2. **Hardware prefetching of instructions and data**
     - reduce miss penalty
  3. **Compiler controlled prefetching**

Reduce Misses/Penalty: Non-blocking Caches to reduce stalls on misses

- 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 multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Hit Under 1 Miss</th>
<th>Hit Under 2 Misses</th>
<th>Hit Under 4 Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECint92</td>
<td>85%</td>
<td>70%</td>
<td>40%</td>
</tr>
<tr>
<td>SPECint95</td>
<td>90%</td>
<td>75%</td>
<td>45%</td>
</tr>
<tr>
<td>SPECint96</td>
<td>95%</td>
<td>80%</td>
<td>50%</td>
</tr>
<tr>
<td>SPECint2000</td>
<td>97%</td>
<td>85%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Reducing Misses/Penalty by Hardware Prefetching of Instructions & Data

- **E.g., Instruction Prefetching**
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer”
  - On miss check stream buffer
- **Works with data blocks too:**
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
  - Prefetching relies on having extra memory bandwidth that can be used without penalty
Reducing Misses/Penalty by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution

- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - Can be incorrect. Faults?

- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

Review: Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

\[ AMAT = \text{HitTime} + \text{MissRate} \cdot \text{MissPenalty} \]

1st Hit Time Reduction Technique: Small and Simple Caches

- Smaller hardware is faster =>
  - small cache helps the hit time
- Keep the cache small enough to fit on the same chip as the processor (avoid the time penalty of going off-chip)
- Keep the cache simple
  - Use Direct Mapped cache: it overlaps the tag check with the transmission of data

2nd Hit Time Reduction Technique: Avoiding Address Translation

Conventional Organization

Virtually Addressed Cache

Translate only on miss

Synonym Problem

Overlap $ access with VA translation: requires $ index to remain invariant across translation
2nd Hit Time Reduction Technique: Avoiding Address Translation (cont’d)

- Send virtual address to cache? Called Virtually Addressed Cache or just Virtual Cache vs. Physical Cache
  - Every time process is switched logically must flush the cache; otherwise get false hits
  - Cost is time to flush + “compulsory” misses from empty cache
- Dealing with aliases (sometimes called synonyms): Two different virtual addresses map to same physical address => multiple copies of the same data in a virtual cache
- I/O typically uses physical addresses; if I/O must interact with cache, mapping to virtual addresses is needed
- Solution to aliases
  - HW solutions guarantee every cache block a unique physical address
- Solution to cache flush
  - Add process identifier tag that identifies process as well as address within process: can’t get a hit if wrong process

Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>-</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Better memory system</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Pipelining Caches</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>