Improving Cache Performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

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

Reducing Misses

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. These are also called cold start misses or first reference misses.
- Capacity—If C is the size of the cache (in blocks) and there have been more than C unique cache blocks accessed since this cache was last accessed.
- Conflict—Any miss that is not a compulsory miss or capacity miss must be a byproduct of the cache mapping algorithm. A conflict miss occurs because too many active blocks are mapped to the same cache set.

How To Measure

- Misses in infinite cache
- Non-compulsory misses in size X fully associative cache
- Non-compulsory, non-capacity misses

How To Reduce Misses?

- Compulsory Misses?
- Capacity Misses?
- Conflict Misses?
- What can the compiler do?

Reduce Misses via Larger Block Size

- 16K cache, miss penalty for 16-byte block = 42, 32-byte is 44, 64-byte is 48. Miss rates are 3.94, 2.87, and 2.64%?
Reduce Misses via Higher Associativity

- **2:1 Cache Rule:**
  - MR of DM cache size \( N \) ≅ MR of 2-way cache size \( N/2 \)
- **Beware:** Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time external cache +10%, internal + 2% for 2-way vs. 1-way

---

Example: Avg. Memory Access Time vs. Miss Rate

- Example: assume CT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CT direct mapped

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.65</td>
<td>6.60</td>
<td>6.22</td>
<td>5.44</td>
</tr>
<tr>
<td>2</td>
<td>5.90</td>
<td>4.90</td>
<td>4.62</td>
<td>4.09</td>
</tr>
<tr>
<td>4</td>
<td>4.60</td>
<td>3.95</td>
<td>3.57</td>
<td>3.19</td>
</tr>
<tr>
<td>8</td>
<td>3.30</td>
<td>3.00</td>
<td>2.87</td>
<td>2.59</td>
</tr>
<tr>
<td>16</td>
<td>2.45</td>
<td>2.20</td>
<td>2.12</td>
<td>2.04</td>
</tr>
<tr>
<td>32</td>
<td>2.00</td>
<td>1.80</td>
<td>1.77</td>
<td>1.79</td>
</tr>
<tr>
<td>64</td>
<td>1.70</td>
<td>1.60</td>
<td>1.57</td>
<td>1.59</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.42</td>
<td>1.44</td>
</tr>
</tbody>
</table>

---

Reducing Misses by emulating associativity: Victim Cache

- **HR of associative + access time of direct mapped?**
- **Add buffer to hold data recently discarded from cache.**
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache

---

Reducing Misses by emulating associativity: Pseudo-Associativity

- **Combines fast hit time of Direct Mapped and the lower conflict misses of a 2-way SA cache.**
- **Divide cache:** on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)
  - **Drawback:** CPU pipeline is hard if hit can take 1 or 2 cycles
  - Better for caches not tied directly to processor
  - Similar technique, simpler → way prediction
Pseudo-Associativity

Index = 1031-1024 = 7

“DM” cache

2048 sets

Reducing Misses by HW Prefetching of Instruction & 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 extra memory bandwidth that can be used without penalty

Reducing Misses by SW Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC, IA64, Tera)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC)
  - Special prefetching instructions cannot cause faults; a form of speculative execution

- Issuing Prefetch Instructions (including address calculation) takes time
  - Is cost of prefetch issues < savings in reduced misses?

Reducing Misses by Various Compiler Optimizations

- Instructions
  - Reorder procedures in memory so as to reduce misses
  - Profiling to look at conflicts
  - McFarling [1989] reduced cache misses by 75% on 8KB direct mapped cache with 4 byte blocks

- 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
Merging Arrays Example

/* Before */
int val[SIZE];
int key[SIZE];

/* After */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];

Reducing conflicts between val & key

Loop Interchange 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];

Sequential accesses instead of striding through memory every 100 words

Loop Fusion Example

/* 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

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:
  – worst case => 2N^3 + N^2.

• Idea: compute on BxB submatrix that fits in cache
**Blocking Example**

/* 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;};

- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- B called **Blocking Factor**
- Conflict Misses Are Not As Easy...

**Key Points**

$$CPU\text{time} = IC \times CPI_{\text{Instr}} + \frac{\text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}}{\text{Instruction}} \times \text{Clock cycle time}$$

- 3 Cs: Compulsory, Capacity, Conflict Misses
- Reducing Miss Rate
  - 1. Reduce Misses via Larger Block Size
  - 2. Reduce Misses via Higher Associativity
  - 3. Reducing Misses via Victim Cache
  - 4. Reducing Misses via Pseudo-Associativity
  - 5. Reducing Misses by HW Prefetching Instr, Data
  - 6. Reducing Misses by SW Prefetching Data
  - 7. Reducing Misses by Compiler Optimizations
- Remember danger of concentrating on just one parameter when evaluating performance
- Next: reducing Miss penalty

**Reducing Miss Penalty: Read Priority over Write on Miss**

- Write buffers may offer RAW conflicts with main memory reads on cache misses
- If simply wait for write buffer to empty might increase read miss penalty by 50%
- Check write buffer contents before read; if no conflicts, let the memory access continue
- Write Back Caches?
  - Read miss may require write of 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 stalls less since it can restart as soon as read completes

**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
- Most useful with large blocks,
- Spatial locality a problem; often we want the next sequential word soon, so not always a benefit (early restart).
Non-blocking Caches to reduce stalls on misses

- *Non-blocking cache* (or *lockup-free cache*) allow the data cache to continue to supply cache hits during a miss
- “hit under miss” reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the CPU
- “hit under multiple miss” or “miss under miss” can 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
- assumes “stall on use” not “stall on miss” which works naturally with dynamic scheduling, but can also work with static.

---

Miss Penalty Reduction: Second Level Cache

- L2 Equations
  \[
  AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}
  \]
  \[
  \text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}
  \]
  \[
  AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})
  \]

- Definitions:
  - *Local miss rate*—misses in this cache divided by the total number of memory accesses to this cache (Miss rate_{L2})
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses generated by the CPU (Miss Rate_{L1} \times Miss Rate_{L2})

---

Multi-level Caches, cont.

- L1 cache local miss rate 10%, L2 local miss rate 40%. What are the global miss rates?
- L1 highest priority is fast hit time. L2 typically low miss rate.
- Design L1 and L2 caches in concert.
- Property of inclusion -- if it is in L1 cache, it is guaranteed to be in the L2 cache -- simplifies design of consistent caches.
- L2 cache can have different associativity (good idea?) or block size (good idea?) than L1 cache.
- These principles can continue to be applied recursively to Multilevel Caches
  - Danger is that time to DRAM will grow with multiple levels in between

---

Reducing Miss Penalty Summary

- Four techniques
  - Read priority over write on miss
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit Under Miss)
  - Multi-level Caches
Fast Hit times via Small and Simple Caches

- This is why Alpha 21164 has 8KB Instruction and 8KB data cache + 96KB second level cache
- I and D caches used to be typically Direct Mapped, on chip

Fast hits by Avoiding Address Translation: Virtual Cache

- 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
    - I/O must interact with cache…
- Solution to aliases
  - HW that guarantees that every cache block has unique physical address
  - SW guarantee: lower n bits must have same address; as long as covers index field &
    direct mapped, they must be unique; called **page coloring**
- 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

Virtual Cache

- Physical Cache
  - Physical address
  - Virtual address
  - TLB
  - Cache
- Virtual Cache
  - Physical address
  - Virtual address
  - TLB
  - Cache

Cache Bandwidth: Trace Caches

- Fetch Bottleneck – Cannot execute instructions faster than you can fetch them into the processor.
- Cannot typically fetch more than about one taken branch per cycle, at best (why? Why one taken branch?)
- Trace cache is an instruction cache that stores instructions in **dynamic execution order** rather than program/address order.
- Implemented on the Pentium 4
Trace Cache

Conventional Cache

A B C beq J: D E F G
H jsr I J K L M N
J: G H jsr W X ret I
W X ret ...

Trace Cache

CSE 240A

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></td>
</tr>
<tr>
<td>Higher Associativity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Victim Caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second Level Caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace Cache?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recent Cache Research at UCSD

- Hardware prefetching of complex data structures (e.g., pointer chasing)
- Fetch Target Buffer
  - Let branch predictor run ahead of fetch engine
- Runtime identification of cache conflict misses
- Speculative Precomputation
  - Spawn threads at runtime to calculate addresses of delinquent (problematic) loads and prefetch → creates prefetcher from application code.
- Code Layout to Reduce Icache Conflict Misses
  - Also, for multithreaded processors