How to Improve Cache Performance?

\[ AMAT = HitTime + MissRate \times MissPenalty \]

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

Where to 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.
  - 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)
- 4th “C”:
  - Coherence — Misses caused by cache coherence.

Reducing Misses by Hardware Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer”
- Works with data blocks too:
  - Joppi [1990] 1 data stream buffer got 25% misses from 4KB cache, 4 streams get 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 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.
  - 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
Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 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 overlapping
  - Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

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

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

Reducing conflicts between val & key; improve spatial locality
```

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
```

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)
    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 spatial locality
```

Blocking Example

```c
/* 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 N*N elements of y[i]
  - Read N elements of 1 row of z[k] repeatedly
  - Write N elements of 1 row of x[i]
• Capacity Misses a function of N & Cache Size:
  - 2N^2 in (assuming no conflict; otherwise ...)
• Idea: compute on BxB submatrix that fits
```

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

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

- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

<table>
<thead>
<tr>
<th>Blocking Factor</th>
<th>Fully Associative Cache</th>
<th>Direct Mapped Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1</td>
<td>0.05</td>
</tr>
<tr>
<td>50</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>150</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Summary: Miss Rate Reduction

- 3 Cs: Compulsory, Capacity, Conflict
  0. Larger cache
  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 Data
  6. Reducing misses by SW Prefetching Data
  7. Reducing misses by compiler optimizations

- 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. Frees HW/SW to guess!

Review: Improving Cache Performance

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

Write Policy: Write-Through vs Write-Back

- Write-through: all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a valid bit
- Write-back: all writes simply update cache
  - Can’t just discard cached data – may have to write it back to memory
  - Cache control bits: both valid and dirty bits
- Other Advantages:
  - Write-through:
    - More memory (or other processors) always have latest data
  - Write-back:
    - Much lower bandwidth, since data often overwritten multiple times
    - Better tolerance to long-latency memory?

Write Policy 2: Write Allocate vs Non-Allocate (What happens on write-miss)

- Write allocate: allocate new cache line in cache
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- Write non-allocate (or “write-around”):
  - Simply send write data through to underlying memory/cache - don’t allocate new cache line!
1. Reducing Miss Penalty: Read Priority over Write on Miss

- Write-through w/ write buffers => RAW conflicts with main memory reads on cache misses
  - 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 want buffer to hold displaced 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

2. Reduce Miss Penalty: 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
- Spatial locality => tend to want next sequential word, so not clear if benefit by early restart

3. Reduce Miss 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

4: Add a second-level cache

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

- Definitions:
  - Local miss rate — misses in this cache divided by the total number of memory accesses to this cache (Miss rate L1)
  - Global miss rate — misses in this cache divided by the total number of memory accesses generated by the CPU

Value of Hit Under Miss for SPEC

- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss
Partner Discussion

What's different in L2 vs L1 Caches?

Comparing Local and Global Miss Rates

- 32 KByte 1st level cache: Increasing 2nd level cache
- Global miss rate close to single level cache rate provided L2 >> L1
- Don't use local miss rate
- L2 not tied to CPU clock cycle!
- Cost & A.M.A.T.
- Generally Fast Hit Times and fewer misses
- Since hits are few, target miss reduction

Reducing Misses: Which apply to L2 Cache?

- Reducing Miss Rate
  1. Reduce Misses via Larger Block Size
  2. Reduce Conflict Misses via Higher Associativity
  3. Reducing Conflict Misses via Victim Cache
  4. Reducing Conflict Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr., Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Capacity/Conf. Misses by Compiler Optimizations

Reducing Miss Penalty Summary

CPU time = \( IC \times CPI + Memory accesses \times Miss rate \times Miss penalty \times Clock cycle time \)

- Four techniques
  - Read priority over write on miss
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache
- Can be applied recursively to Multilevel Caches
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse

L2 cache block size & A.M.A.T.

- 32KB L1, 8 byte path to memory

What is the Impact of What You've Learned About Caches?

- 1960-1985: Speed = \( f(\text{no. operations}) \)
- 1990
  - Pipelined Execution & Fast Clock Rate
  - Out-of-Order execution
  - Superscalar Instruction Issue
- 1998: Speed = \( f(\text{non-cached memory accesses}) \)
  - Superscalar, Out-of-Order machines hide L1 data cache miss (-5 clocks) but not L2 cache miss (-50 clocks)?
1. Fast Hit times via Small and Simple Caches

- Why Alpha 21164 has 8KB Instruction and 8KB data cache + 96KB second level cache?
  - Small data cache and clock rate
- Direct Mapped, on chip

Address Translation

- Page table is a large data structure in memory
- Two memory accesses for every load, store, or instruction fetch!!!
- Virtually addressed cache?
- Synonym problem
- Cache the address translations?

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is Translation Lookaside Buffer or TLB

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Only</th>
<th>Hit</th>
<th>Valid</th>
<th>Access</th>
</tr>
</thead>
</table>

Really just a cache on the page table mappings

TLB access time comparable to cache access time (much less than main memory access time)

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

2. Fast hits by Avoiding Address Translation

- If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag

Overlap $ index access with VA translation: requires 6 index to remain invariant across translation

- Limits cache to page size: what if want bigger caches and uses same trick?
  - Higher associativity moves barrier to right
  - Page coloring
2. Fast hits by Avoiding Address Translation

- 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 hit
- Cast is time to flush “compulsory” misses from empty cache
- Add process identifier tag that identifies process as well as address within process; can’t get a hit if wrong process

- Dealing with aliases (sometimes called synonyms):
  Two different virtual addresses map to same physical address
  - solve by fast: realizing What are the implications?
  - HW anti-aliasing guarantees every cache block has unique address
    - verify on miss (rather than on every hit)
  - what if it gets larger?
  - Every time process is switched logically must flush the cache; otherwise get false hit
  - How can SW simplify the problem? (called page coloring)

- I/O must interact with cache, so need virtual address

Case Study: MIPS R4000

<table>
<thead>
<tr>
<th>TWO Cycle</th>
<th>IF IS RF</th>
<th>EX DF</th>
<th>TC WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Latency</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>THREE Cycle</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>Branch Latency</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>(conditions evaluated during EX phase)</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>Delay slot plus two stalls</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>Branch likely cancels delay slot if not taken</td>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
</tr>
<tr>
<td>IF IS RF</td>
<td>EX DF</td>
<td>TC WB</td>
<td></td>
</tr>
</tbody>
</table>

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles = unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)

3. Fast Hits by pipelining Cache Case Study: MIPS R4000

- 8 Stage Pipeline:
  - IF-first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS-second half of access to instruction cache.
  - RF-instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX-execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF-data fetch, first half of access to data cache.
  - DS-second half of access to data cache.
  - TC-tag check, determine whether the data cache access hit.
  - WB-write back for loads and register-register operations.

- What is impact on Load delay?
  - Need 2 instructions between a load and its use!

What is the Impact of What You’ve Learned About Caches?

- 1960-1985: Speed = f(no. operations)
- 1990
  - Pipelined Execution & Fast Clock Rate
  - Out-of-Order execution
  - Superscalar Instruction Issue
- 1998: Speed = f(non-cached memory accesses)
Alpha Memory Performance:
Miss Rates of SPEC92

- AlphaSort
- TPC-B (db1)
- Li
- Sc
- Compress
- Ora
- Ear
- Doduc
- Tomcatv
- Mdljp2
- Spice
- Su2cor

Alpha Memory Performance:
Miss Rates of SPEC92

- AlphaSort
- TPC-B (db1)
- Li
- Sc
- Compress
- Ora
- Ear
- Doduc
- Tomcatv
- Mdljp2
- Spice
- Su2cor

Alpha CPI Components
- Instruction stall: branch mispredict (green);
- Data cache (blue): Instruction cache (yellow): L2$ (pink)
- Other: compute + reg conflicts, structural conflicts

Pitfall: Predicting Cache Performance from Different Prog. (ISA, compiler, ...)

- 4KB Data cache miss rate 8%, 12%, or 28%?
- 1KB Instr cache miss rate 0%, 3%, or 10%?
- Alpha vs. MIPS for 8KB Data: 17% vs. 10%?
- Why 2X Alpha v. MIPS7?

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>-</td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Early restart &amp; Critical Word 1st</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>*</td>
<td>3</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Better-memory-system</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>0</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>