Advanced cache optimizations - overview

Why More on Memory Hierarchy?

Adapted from Patterson and Hennessey
(Morgan Kaufman Pubs)
Review: 6 Basic Cache Optimizations

- 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
- Multilevel Caches

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

11 Advanced Cache Optimizations

- Reducing hit time
  1. Small and simple caches
  2. Way prediction
  3. Trace caches
  4. Increasing cache bandwidth
  5. Pipelined caches
  6. Multibanked caches
  7. Critical word first
  8. Merging write buffers

- Reducing Miss Penalty
  9. Compiler optimizations

- Reducing Miss Rate
  10. Hardware prefetching
  11. Compiler prefetching
1. Fast Hit times via Small and Simple Caches

- Index tag memory and then compare takes time
- ⇒ Small cache can help hit time since smaller memory takes less time to index
  - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple ⇒ direct mapping

![Graph showing cache size vs. access time](image)

2. Fast Hit times via Way Prediction

- The idea: 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.
  - Miss ⇒ 1st check other blocks for matches in next clock cycle

• Accuracy can be as high as 85%
3. Fast Hit times via Trace Cache (Pentium 4)

- P4 translated X86 to “RISC” micro-ops
  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

- Problems:
  - complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
  - instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

4: Increasing Cache Bandwidth by Pipelining

- 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
- ⇒ more clock cycles between the issue of the load and the use of the data
5. Increasing Cache Bandwidth: Non-Blocking Caches

- For Out-of-order execution, the processor need not stall on a cache miss. “hit under miss” reduces the effective miss penalty by working during miss vs. ignoring CPU requests.
- Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a miss.
  - requires multi-bank memories

6: Increasing Cache Bandwidth via Multiple Banks

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses.
- 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; ...
7. Reduce Miss Penalty: Early Restart 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
- *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

8. Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- When a new entry is loaded in the buffer, its address is checked against the other blocks in the buffer
- If there’s a match, blocks are combined
9. 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 overlap
  – Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

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];

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

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; 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];
         };
        };

• Two Inner Loops:
  – Read all N x N 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 B x B submatrix that fits
```

### 10. Reducing Misses by Hardware

#### Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- Instruction Prefetching
  - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
  - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
11. 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

Adapted from Patterson and Hennessey
(Morgan Kaufman Pubs)