Cache Memory: Instruction Cache, HW/SW Interaction

Computer Science 104

Admin

- Project Due April 20
- Homework #5 Due April 11, in class

What’s Ahead
- Finish Caches
- Virtual Memory
- Input/Output (1 homework)
- Advanced Topics
Review: Cache Memory

• Cost effective memory system
  ➢ big cheap slow + small fast expensive
• For a \(2^{10}\) byte cache with 32-byte blocks:
  ➢ The uppermost \(22 = (32 - 10)\) address bits are the Cache Tag
  ➢ The lowest 5 address bits are the Block Offset (Byte Select) (Block Size = \(2^5\))
  ➢ The next 5 address bits (bit5 - bit9) are the Cache Index

1KB Direct Mapped Cache with 32B Blocks

© Alvin R. Lebeck
A N-way Set Associative Cache

- **N-way set associative**: N entries for each Cache Index
  - N direct mapped caches operating in parallel
- **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

And yet Another Extreme Example: Fully Associative cache

- **Fully Associative Cache** -- push the set associative idea to its limit!
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32B blocks, we need N 27-bit comparators
Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Invalidation**: other process (e.g., I/O) updates memory

Sources of Cache Misses

<table>
<thead>
<tr>
<th></th>
<th>Direct Mapped</th>
<th>N-way Set Associative</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>Big</td>
<td>Medium</td>
<td>Small</td>
</tr>
<tr>
<td>Compulsory Miss</td>
<td>Same</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>Conflict Miss</td>
<td>High</td>
<td>Medium</td>
<td>Zero</td>
</tr>
<tr>
<td>Capacity Miss</td>
<td>Low(er)</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Invalidation Miss</td>
<td>Same</td>
<td>Same</td>
<td>Same</td>
</tr>
</tbody>
</table>

Note:
If you are going to run “billions” of instruction, Compulsory Misses are insignificant.
The Need to Make a Decision!

- **Direct Mapped Cache:**
  - Each memory location can only map to 1 cache location
  - No need to make any decision 😊
    - Current item replaces the previous item in that cache location

- **N-way Set Associative Cache:**
  - For each memory location have a choice of N cache locations

- **Fully Associative Cache:**
  - Each memory location can be placed in ANY cache location

- **Cache miss in a N-way Set Associative or Fully Associative Cache:**
  - Bring in new block from memory
  - Throw out a cache block to make room for the new block
  - We need to make a decision on which block to throw out!

---

Cache Block Replacement Policy

- **Random Replacement:**
  - Hardware randomly selects a cache item and throw it out

- **Least Recently Used:**
  - Hardware keeps track of the access history
  - Replace the entry that has not been used for the longest time.
  - For two way set associative cache one needs one bit for LRU replacement.

- **Example of a Simple “Pseudo” Least Recently Used Implementation:**
  - Assume 64 Fully Associative Entries
  - Hardware replacement pointer points to one cache entry
  - Whenever an access is made to the entry the pointer points to:
    - Move the pointer to the next entry
    - Otherwise: do not move the pointer

<table>
<thead>
<tr>
<th>Entry 0</th>
<th>Entry 1</th>
<th>...</th>
<th>Entry 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement Pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck
Cache Write Policy: Write Through versus Write Back

- Cache read is much easier to handle than cache write:
  - Instruction cache is much easier to design than data cache
- Cache write:
  - How do we keep data in the cache and memory consistent?
- Two options (decision time again :-)
  - Write Back: write to cache only. Write the cache block to memory when that cache block is being replaced on a cache miss.
    » Need a "dirty bit" for each cache block
    » Greatly reduce the memory bandwidth requirement
    » Control can be complex
  - Write Through: write to cache and memory at the same time.
    » What!!! How can this be? Isn't memory too slow for this?

Instruction Cache

- Separate Inst & Data Caches
  - Harvard Architecture
- Can access both at same time
- Combined L2
  - L2 >> L1

© Alvin R. Lebeck
Four Questions for Memory Hierarchy Designers

- Q1: Where can a block be placed in the upper level? *(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag/Block
- Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Random, LRU
- Q4: What happens on a write? *(Write strategy)*
  - Write Back or Write Through (with Write Buffer)

Cache Performance

\[ \text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time} \]

\[ \text{Memory stall clock cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty} \]

**Example**

- Assume every instruction takes 1 cycle
- Miss penalty = 20 cycles
- Miss rate = 10%
- 1000 total instructions, 300 memory accesses
- Memory stall cycles? CPU clocks?
Cache Performance

- Memory Stall cycles = $300 \times 0.10 \times 20 = 600$
- CPU clocks = $1000 + 600 = 1600$

- 60% slower because of cache misses!

- Change miss penalty to 100 cycles
- CPU clocks = $1000 + 3000 = 4000$ cycles

Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
Reducing Misses (The 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. *(Misses in 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 Size X Cache)*
- **Conflict**—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses. *(Misses in N-way Associative, Size X Cache)*

Cache Performance

- Your program and caches
- Can you affect performance?
- Think about 3Cs
Mapping Arrays to Memory

<table>
<thead>
<tr>
<th>Row-major</th>
<th>Column major</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4</td>
<td>0 5 10 15 20</td>
</tr>
<tr>
<td>5 6 7 8 9</td>
<td>1 6 11 16 21</td>
</tr>
<tr>
<td>10 11 12 13 14</td>
<td>2 7 12 17 22</td>
</tr>
<tr>
<td>15 16 17 18 19</td>
<td>3 8 13 18 23</td>
</tr>
<tr>
<td>20 21 22 23 24</td>
<td>4 9 14 19 24</td>
</tr>
</tbody>
</table>

Part of the Row maps into cache

Array Mapping and Cache Behavior

- Elements spread out in memory because of column-major mapping
- Fixed mapping into cache
- Self-interference in cache

© Alvin R. Lebeck
Data Cache Performance

- **Instruction Sequencing**
  - *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 entire columns or rows

- **Data Layout**
  - *Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - *Nonlinear Array Layout*: Mapping 2 dimensional arrays to the linear address space
  - *Pointer-based Data Structures*: node-allocation

---

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
```
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:
  ➢ 3 NxN => no capacity misses; otherwise ...

• Idea: compute on BxB submatrix that fits
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 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
Data Layout Optimizations

• So far program control
• Changes the order in which memory is accessed

• We can also change the way our data structures map to memory
• 2-dimensional array
• Pointer-based data structures

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
Layout and Cache Behavior

• Tile elements spread out in memory because of column-major mapping
• Fixed mapping into cache
• Self-interference in cache

Making Tiles Contiguous

• Elements of a quadrant are contiguous
• Recursive layout
• Elements of a tile are contiguous
• No self-interference in cache
Pointer-based Data Structures

- Linked List, Binary Tree
- Basic idea is to group linked elements close together in memory
- Need relatively static traversal pattern
- Or could do it during garbage collection/compaction

Summary of Compiler Optimizations to Reduce Cache Misses
Reducing I-Cache Misses by Compiler Optimizations

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

Summary

• Cost Effective Memory Hierarchy
• Work by exploiting locality (temporal & spatial)
• Associativity, Blocksize, Capacity (ABCs of caches)
• Know how a cache works
  ➢ Break address into tag, index, block offset
• Know how to draw a block diagram of a cache
• CPU cycles/time, Memory Stall Cycles
• Your programs and cache performance

Next Time
• Exceptions and Interrupts
• Reading Chapter 5.6, Appendix A.7