ECE 454
Computer Systems Programming

Memory Hierarchy

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
Content

• Cache basics and organization

• Understanding/Profiling Memory

• Optimizing for caches (later)
  • Loop reordering
  • Tiling/blocking
double a[4][4];
double b[4][4];
double c[4][4]; // assume already set to zero

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        // actual work
        c[i][j] += a[i][k] * b[k][j];
}

How much performance improvement can we get by optimizing this code?
MMM Performance

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz
Gflop/s (giga floating point operations per second)

- Standard desktop computer
- Both versions compiled using optimization flags
- Both implementations have exactly the same # of operations ($2n^3$)
- What is going on?
Problem: Processor-Memory Bottleneck

- L1 cache reference time = 1-4 ns
  - However, L1 cache size <= 64 KB

- Main memory reference time = 100 ns, 100X slower!
  - However, memory size >= GBs

- Some data:
  - 1 ns = 1/1,000,000,000 second
  - For a 2.5 GHz CPU (my laptop), 1 cycle = 0.4 ns
Memory Hierarchy

- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers

- remote secondary storage (tapes, distributed file systems, web servers)
- local secondary storage (local disks)
- main memory (DRAM)
- on-chip L2 cache (SRAM)
- on-chip L1 cache (SRAM)
- registers

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
Cache Basics
(Review Hopefully!)
General Cache Mechanics

Data is copied in block-sized transfer units.

Smaller, faster, more expensive memory caches a subset of the blocks.

Larger, slower, cheaper memory viewed as partitioned into fixed size “blocks”.

Cache

Memory

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
General Cache Concepts: Hit

Request: 14
Data in block 14 is needed

Block 14 is in cache: Hit!
Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache:
- **Placement policy:** Determines where b goes in cache
- **Replacement policy:** Determines which block gets evicted (victim)
Cache Performance Metrics

• Miss Rate
  • Fraction of memory references not found in cache
  • \( \text{miss rate} = \frac{\text{misses}}{\text{accesses}} = 1 - \text{hit rate} \)
  • 3-10% for L1, small (e.g., < 1%) for L2, depending on size, etc.

• Hit Time
  • Time to deliver a line in the cache to the processor
    • Includes time to determine whether the line is in the cache
  • 1-4 clock cycles for L1, 5-20 clock cycles for L2

• Miss Penalty
  • Additional time required due to a miss
    • Typically 50-400 cycles for main memory
Let's Think About Those Numbers

- Huge difference between a hit and a miss
  - 100x between L1 and main memory

- Performance with 99% hit rate doubles compared to 97%!
  - Say cache hit time = 1 cycle, miss penalty of 100 cycles
  - Average access time:
    - 97% hits: \(0.97 \times 1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 3.97 \text{ cycles}\)
    - 99% hits: \(0.99 \times 1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 1.99 \text{ cycles}\)

- This is why miss (instead of hit) rate is used to think about cache performance
  - 3% is much worse than 1% miss rate
Types of Cache Misses (1)

- Three types
- Cold (compulsory) miss
  - Occurs on first access to a block
  - Can’t do too much about these (except prefetching---more later)
Types of Cache Misses (2)

• **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
  - e.g., block i must be placed in slot (i mod 8)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
  - Conflict misses are less of a problem today (more later)

• **Capacity miss**
  - Occurs when the set of active cache blocks is larger than the cache
    - Working set is larger than cache size
    - This is the most significant problem today
Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses equal or near to those they have used recently.

- **Temporal locality**: 
  - Recently referenced items are likely to be referenced again in the near future.

- **Spatial locality**: 
  - Items with nearby addresses tend to be referenced close together in time.
Example: Locality?

```plaintext
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

• **Data:**
  • Temporal: *sum* referenced in each iteration
  • Spatial: close by elements of array *a* accessed (in stride-1 pattern)

• **Instructions:**
  • Temporal: cycle through loop repeatedly
  • Spatial: reference close by instructions in sequence

• **Important to be able to assess the locality in your code!**
Cache Organization
General Cache Organization
(S, E, B)

- $S = 2^s$ sets
- $E = 2^e$ blocks per set
- $B = 2^b$ bytes per cache block (the data)

Cache size: $S \times E \times B$ data bytes

Valid bit

Set

Block

v

tag

0 1 2 ···· B-1
Direct Mapped Cache (E = 1)

- Direct mapped: one block per set

S = 64 sets

block size = 8 bytes
Direct Mapped Cache

- Incoming memory address divided into tag, index and offset bits
  - Index determines set
  - Tag is used for matching
  - Offset determines starting byte within block

Address (32 bits)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>011…1</td>
<td>0…01</td>
<td>100</td>
</tr>
</tbody>
</table>

- Tag [31:9], Index [8:3], Offset [2:0]
- 23 bits, 6 bits, 3 bits

S = 64 sets

Block size = 8 bytes
Direct Mapped Cache: Index Lookup

S = 64 sets

block size = 8 bytes

index lookup

011…1 0…01 100

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:9]</td>
<td>[8:3]</td>
<td>[2:0]</td>
</tr>
<tr>
<td>23 bits</td>
<td>6 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>
Direct Mapped Cache: Match Tag

- S = 64 sets
- block size = 8 bytes
- check valid, match tag

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:9]</td>
<td>[8:3]</td>
<td>[2:0]</td>
</tr>
<tr>
<td>23 bits</td>
<td>6 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

Table:
- valid (v)
- tag (01234567)
Direct Mapped Cache: Lookup Bytes

- Assume address being looked up is for a short int (2 bytes)
- If the tag doesn’t match, old cache line is evicted and replaced with entire new cache line
Direct Mapped Cache Example

S = 64 sets

block size = 8 bytes

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>01234567</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>01234567</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>01234567</td>
</tr>
</tbody>
</table>

Adrr 0x0   = 0b0 000 = (0, 0, 0) maps to Set 0
Adrr 0x8   = 0b1 000 = (0, 1, 0) maps to Set 1
Adrr 0x10  = 0b10 000 = (0, 2, 0) maps to Set 2
Adrr 0x100 = 0b0 100000 000 = (0, 32, 0) maps to Set 32
Adrr 0x200 = 0b1 000000 000 = (1, 0, 0) maps to Set 0

cache size = 64 * 8 = 512 = 0x200

check valid, match tag

<table>
<thead>
<tr>
<th>011...1</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>index</td>
<td>offset</td>
</tr>
<tr>
<td>[31:9]</td>
<td>[8:3]</td>
<td>[2:0]</td>
</tr>
<tr>
<td>23 bits</td>
<td>6 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

cache size =
64 * 8 = 512 = 0x200
Two-way Set Associative Cache (E = 2)

- 2-way set associative: two blocks per set

<table>
<thead>
<tr>
<th>S = 32 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
</tbody>
</table>

- Total cache size is same as direct mapped cache
- But number of sets is halved

<table>
<thead>
<tr>
<th>011…1</th>
<th>0…01</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>index</td>
<td>offset</td>
</tr>
<tr>
<td>[31:8]</td>
<td>[7:3]</td>
<td>[2:0]</td>
</tr>
<tr>
<td>24 bits</td>
<td>5 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>
Two-way Set Associative Cache

(E = 2)

S = 32 sets

\[ \text{block size} = 8 \text{ bytes} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
v & tag & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
v & tag & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
v & tag & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
v & tag & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
v & tag & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

index lookup

\[ \begin{array}{c|c|c|c|c|c|c|c}
\hline
011...1 & 0...01 & 100 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c}
tag & index & offset \\
24 bits & 5 bits & 3 bits \\
\hline
\end{array} \]
Two-way Set Associative Cache (E = 2)

S = 32 sets

block size = 8 bytes

check valid, compare and match with any one tag

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>[7:3]</td>
<td>[2:0]</td>
</tr>
<tr>
<td>24 bits</td>
<td>5 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

011…1 0…01 100
Two-way Set Associative Cache (E = 2)

- $S = 32$ sets

- Block size = 8 bytes

- If no match then one line in set is selected for eviction and replacement

- Replacement policies: random, least recently used (LRU), …

- Tag: $31:8$, Index: $7:3$, Offset: $2:0$
  - 24 bits
  - 5 bits
  - 3 bits

lookup bytes

011…1 0…01 100
Two-way Cache Example

- **Block size**: 8 bytes
- **Cache size**: $64 \times 8 = 512 = 0x200$

### Address Mapping

- **Addr 0x0** = 0b0 000 = (0, 0, 0) maps to Set 0
- **Addr 0x8** = 0b1 000 = (0, 1, 0) maps to Set 1
- **Addr 0x10** = 0b10 000 = (0, 2, 0) maps to Set 2
- **Addr 0x100** = 0b1 00000 000 = (1, 0, 0) maps to Set 0
- **Addr 0x200** = 0b10 00000 000 = (2, 0, 0) maps to Set 0
Intel Core i7: Cache Associativity

L1/L2 cache have 64 B blocks

- L1 I-cache: 32 KB, 8-way associative!
- L1 D-cache: 32 KB, 8-way associative!
- L2 unified cache: 256 KB, 8-way associative!
- L3 shared cache: 8 MB, 16-way associative!
- Main Memory: 16 GB, 10s of millions of cycles
- Disk: > 500 GB, 10s of millions of cycles

Latency:
- CPU Reg: 4 cycles
- L1 I-cache: 10 cycles
- L2 unified cache: 40-75 cycles
- L3 shared cache: 60-100 cycles
- Main Memory: 10s of millions of cycles
- Disk: 10s of millions of cycles

Conflict misses are not as much issue today, Staying within on-chip cache capacity is key
What About Writes?

- Multiple copies of data exist in L1, L2, main memory, disk
  - Need to ensure consistency

- What to do on a write-hit?
  - Write-through (write to cache and immediately to memory)
  - Write-back (defer write to memory until line is replaced)
    - Need a dirty bit (cache line different from memory or not)

- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more reads and writes to the location follow
  - No-write-allocate (write immediately to memory)
    - For streaming writes (write once and then no reads in the near future)
What About Writes?

- Multiple copies of data exist in L1, L2, main memory, disk

- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (cache line different from memory or not)

- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more reads and writes to the location follow
  - No-write-allocate (write immediately to memory)
    - For streaming writes (write once and then no reads in the near future)

- Typically:
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
Understanding/Profiling Memory
UG Machines

1 CPU – Intel Core i7-4790, 3.6 GHz, with 4 HT cores

Run `lscpu` on UG machine shows:
- 32KB, 8-way L1 data cache
- 32KB, 8-way L1 inst cache
- 256KB, 8-way L2 cache
- 8M, 16-way L3 cache
Get Memory Hierarchy Details: lstopo

- Running lstopo on UG machine shows:

Machine (16GB)

- Package L#0 + L3 L#0 (8192KB)
  - L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
  - L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
  - L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
  - L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)

Shared L3

4 cores per CPU
Get More Cache Details: L1 dcache

- `ls /sys/devices/system/cpu/cpu00/cache/index0`
  - `coherency_line_size: 64`  // 64B cache lines
  - `level: 1`  // L1 cache
  - `number_of_sets: 64`
  - `physical_line_partition`
  - `shared_cpu_list: 0`  // shared by cpu0 only
  - `shared_cpu_map`
  - `size: 32K`
  - `type: data`  // data cache
  - `ways_of_associativity: 8`  // 8-way set associative
Get More Cache Details: L2

- `ls /sys/devices/system/cpu/cpu0/cache/index2`
  - `coherency_line_size`: 64 // 64B cache lines
  - `level`: 2 // L2 cache
  - `number_of_sets`: 512
  - `physical_line_partition`
  - `shared_cpu_list`
  - `shared_cpu_map`
  - `size`: 256K
  - `type`: Unified // unified cache, means instructions and data
  - `ways_of_associativity`: 8 // 8-way set associative
Access Hardware Counters: perf

- The `perf` tool allows you to access performance counters.

- To measure L1 data cache load misses for program `foo`, run:
  ```
  perf stat -e L1-dcache-load-misses foo
  7803  L1-dcache-load-misses    # 0.000 M/sec
  ```

- To see a list of all events you can measure:
  ```
  perf list
  ```

- Note: you can measure multiple events at once.
A binary search tree and a hash table store 100 million items.

How much time will it take to search BST versus hash table?
Analysis

- When file size = $2^{29}$ (536,870,912)
  - Number of unique words is ~100 million, average word len ~ 5
- With large hash table, there is an initial fixed cost for allocating this large table
- While the hash table beats BST, the performance improvement is not what is expected from Big-O analysis
  - With 100 million (~$2^{26}$) unique words:
    - Number of pointer traversals with BST should be $\log(2^{26})$ or ~26
    - Number of pointer traversals with HashTable should be ~2
    - However, hash table performance is only 2.5-2.8 times better
- Why is that the case? Let’s look at it in more detail.
Perf on BST and HashTable

- Initial hypothesis: memory hierarchy is the culprit, so run `perf`
  - E.g., Does hash table has a lot more LLC accesses than BST?

```
perf stat -e instructions -e L1-dcache-loads -e L1-dcache-misses -e LLC-loads -e LLC-misses {bst,hash}-program
```

<table>
<thead>
<tr>
<th></th>
<th>BST</th>
<th>HashTable</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>13.11 s</td>
<td>4.77 s</td>
<td>2.75</td>
</tr>
<tr>
<td>Instructions</td>
<td>50 b</td>
<td>21 b</td>
<td>2.4</td>
</tr>
<tr>
<td>L1 hits (4 cycles)</td>
<td>13000 m</td>
<td>4500 m</td>
<td>2.9</td>
</tr>
<tr>
<td>L1 misses (10 cycles)</td>
<td>1000 m</td>
<td>214 m</td>
<td></td>
</tr>
<tr>
<td>LLC loads (40-75 cycles)</td>
<td>333 m</td>
<td>101 m</td>
<td></td>
</tr>
<tr>
<td>LLC misses (60-100 cycles)</td>
<td>21 m</td>
<td>18 m</td>
<td></td>
</tr>
</tbody>
</table>
Further Analysis

- Number of instructions executed and L1 cache hits is proportional to runtime

- Need to understand what instructions are being executed
  - Need to use `gprof` to see where time is being spent in code
  - Lesson: use code profiling before memory profiling

- Found that hash key calculation takes significant time, reducing the improvements we expect from the hash table!
  - Required disabling the inlining of this function!

- But really, can the hash key calculation slow down expected improvements by so much?
Digging Even Further

• Looking at assembly for each insertion of a word, the number of load/store operations is as follows:
  • BST: 31 (5 initial load/stores + 2 loads per iteration * 13 (roughly the average depth of the tree))
  • Hash Table: 13 (10 initial load/stores (including hash key calculation) + 2 loads per iteration * 1.5 (for linked list traversal))
    • $\frac{13}{1.5} \approx 8$, which is the extra amount of pointer traversals that we measured that the BST code does over the hash table code

• Ratio of load/stores of BST to HashTable = 31/13 = 2.38
  • Roughly the same as observed performance

• So initialization has a significant impact on speedup!