Optimization III: Cache Memories

15-213/18-243: Introduction to Computer Systems
12th Lecture, 22 February 2010

Instructors:
Bill Nace and Gregory Kesden
The Exam

- Tuesday, 2 March
- Closed book, closed notes, closed friend, open mind
  - We will provide reference material
- Quite unlike past exams
- Material from Lectures 1 - 9, Labs 1 & 2
  - Representation of Integers, Floats
  - Machine code for control structures, procedures
    - Stack discipline
  - Layout of Arrays, Structs, Unions in memory
  - Floating point operations
Last Time

- Program optimization
  - Optimization blocker: Memory aliasing
  - One solution: Scalar replacement of array accesses that are reused

```c
for (i = 0; i < n; i++) {
    double val = 0;
    for (j = 0; j < n; j++)
        val += a[i*n + j];
    b[i] = val;
}
```
Last Time

- Instruction level parallelism
- Latency versus throughput

Latency versus cycles/issue

Integer Multiply

<table>
<thead>
<tr>
<th>Step 1</th>
<th>1 cycle</th>
<th>Step 2</th>
<th>1 cycle</th>
<th>.............</th>
<th>Step 10</th>
<th>1 cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>10</td>
<td>cycles/issue</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Functional Units:
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store
Last Time

- Consequence

Twice as fast
Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations
Problem: Processor-Memory Bottleneck

Processor performance doubles about every 18 months

Bus bandwidth evolves much slower

Core 2 Duo: Can process at least 256 Bytes/cycle (1 SSE two operand add and mult)

Core 2 Duo: Bandwidth 2 Bytes/cycle Latency 100 cycles

Solution: Caches
Cache

- **Definition**: Computer memory with short access time used for the storage of frequently or recently used instructions or data
  - From the French *cacher* ‘to hide’
  - Memory contents *hidden* close to their point of use, in hopes of using them again in the future
  - As opposed to buffers/variables/etc, which are not hidden and require programmer management
General Cache Mechanics

Cache

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

Data is copied in block-sized transfer units

Memory

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

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
- **Replacement policy:** determines which block gets evicted (victim)
Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite 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
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2

- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)
Let us think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
  - Average access time:
    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

- **This is why “miss rate” is used instead of “hit rate”**
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache size
  - Must evict a victim to make space for replacement block

- **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 $(i \mod 4)$
  - 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
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal 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.

- **Fundamental Law (almost):** Programs with good locality run faster than programs with poor locality.
Example: Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- Being able to assess the locality of code is a crucial skill for a programmer
Locality Example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```
Locality Example #2

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```
Locality Example #3

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

- How can it be fixed?
Memory Hierarchies

- **Fundamental and enduring properties of hardware:**
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True of registers $\leftrightarrow$ DRAM, DRAM $\leftrightarrow$ disk, etc.

- **Fundamental and enduring property of software:**
  - Well-written programs tend to exhibit good locality

- These properties complement each other beautifully

- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**
An Example Memory Hierarchy

L0: registers
  - CPU registers hold words retrieved from L1 cache

L1: on-chip L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache

L2: off-chip L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory

L3: main memory (DRAM)
  - Main memory holds disk blocks retrieved from local disks

L4: local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers

L5: remote secondary storage (tapes, distributed file systems, Web servers)
  - Larger, slower, cheaper per byte
  - Smaller, faster, costlier per byte
### Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Throughput:
- L1 I-cache: 16 B/cycle
- L1 D-cache: 8 B/cycle
- L2 unified cache: 2 B/cycle
- Main Memory: 1 B/30 cycles

Latency:
- L1 I-cache: 3 cycles
- L1 D-cache: 14 cycles
- L2 unified cache: 100 cycles
- Main Memory: millions

Disk: ~500 GB

Not drawn to scale
Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations
General Cache Organization \((S, E, B)\)

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

Cache size:
\(S \times E \times B\) data bytes
Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

**Address of word:**
- **t bits**
- **s bits**
- **b bits**
  - *tag*
  - *set index*
  - *block offset*

**E =** \(2^e\) lines per set

**S =** \(2^s\) sets

**B =** \(2^b\) bytes per cache block (the data)
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

```
| t bits | 0...01 | 100 |
```

S = \(2^5\) sets

```
<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes
Example: Direct Mapped Cache \((E = 1)\)

Direct mapped: One line per set  
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

find set
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

valid? + match: yes = hit

compare both

block offset
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:
0...01 100

t bits

match both

valid? + match: yes = hit

v tag 0 1 2 3 4 5 6 7 v tag 0 1 2 3 4 5 6 7

short int (2 Bytes) is here

block offset

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...
Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles
What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk

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

- What to do when a write-miss occurs?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)

- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
Software Caches are More Flexible

- **Examples**
  - File system buffer caches, web browser caches, etc.

- **Some design differences**
  - Almost always fully associative
    - so, no placement restrictions
    - index structures like hash tables are common
  - Often use complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single “block” transfers
    - may fetch or write-back in larger units, opportunistically
Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations
Optimizations for the Memory Hierarchy

- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- **How to achieve?**
  - Proper choice of algorithm
  - Loop transformations

- **Cache versus register level optimization:**
  - In both cases locality desirable
  - Register space much smaller + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)
Example: Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* 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++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)

First iteration:
- $n/8 + n = 9n/8$ misses
- Afterwards in cache: (schematic)
### Cache Miss Analysis

**Assume:**
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

**Second iteration:**
- Again: \( \frac{n}{8} + n = \frac{9n}{8} \) misses

**Total misses:**
- \( \frac{9n}{8} \times n^2 = \frac{9}{8} \times n^3 \)
**Blocked Matrix Multiplication**

```c
int i, j, k;
for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
        for (k = 0; k < n; k+=B)
            c[i*n + j] += a[i*n + k] * b[k*n + j];
```

![Diagram of blocked matrix multiplication](image)
Cache Miss Analysis

Assume:
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)
- Three blocks fit into cache: $3B^2 < C$

First (block) iteration:
- $B^2/8$ misses for each block
- $2n/B \times B^2/8 = nB/4$
  (omitting matrix $c$)
- Afterwards in cache (schematic)
Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size \( C << n \) (much smaller than \( n \))
  - Three blocks fit into cache: \( 3B^2 < C \)

- **Second (block) iteration:**
  - Same as first iteration
  - \( 2n/B \times B^2/8 = nB/4 \)

- **Total misses:**
  - \( nB/4 \times (n/B)^2 = n^3/(4B) \)
Matrix Multiplication: The Bottom Line

- No blocking: \((9/8) \times n^3\)
- Blocking: \(1/(4B) \times n^3\)

- Suggest largest possible block size \(B\), but limit \(3B^2 < C\)!
  (can possibly be relaxed a bit, but there is a limit for \(B\))

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: \(3n^2\), computation \(2n^3\)
    - Every array elements used \(O(n)\) times!
  - But program has to be written properly
Summary

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations

- Next Time: Exceptions
  - Traps, Faults, Aborts, Interrupts
  - Processes
  - Context Switch
  - fork, exit