The Memory Hierarchy

Sept. 18, 2012

Topics
- The memory hierarchy
- Cache design
- Cache performance
  - blocking
  - prefetching

The CPU-Memory Performance Gap

The Memory Hierarchy

(from Hennessy & Patterson, CA:AQQ, 5th Edition)
Why is bigger slower?

- Physics slows us down
- Racing the speed of light: \(3.0 \times 10^8\) meters/sec
  - clock = 3GHz
  - how far can I go in a clock cycle?
  - \((3.0 \times 10^8\) m/s) / (3x10^9 cycles/s) = 10cm/cycle
  - For comparison: Core i7 is roughly 1.5cm across
- Capacitance:
  - longer wires have more capacitance
  - either more powerful (bigger) transistors required, or slower
  - signal propagation speed proportional to capacitance
  - going “off chip” has an order of magnitude more capacitance

Questions

Why split the L1 caches into instruction and data?

Why is the L2 cache unified (and not split)?

Why are there 3 levels of on-chip cache?

Why doesn’t the Core i7 have an off-chip cache?

Locality of Reference

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently.
  - Temporal locality: recently referenced items are likely to be referenced in the near future.
  - Spatial locality: items with nearby addresses tend to be referenced close together in time.

Locality in Example:

- Data
  - Reference array elements in succession (spatial)
- Instructions
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)
How important are caches?

Caches take up nearly half of the die area

Caching: The Basic Idea

Main Memory
- Stores words
  A-Z in example

Cache
- Stores subset of the words
  4 in example
- Organized in lines
  - Multiple words
  - To exploit spatial locality

Access
- Word must be in cache for processor to access

Design Issues for Caches

Key Questions:
- Where should a line be placed in the cache? (line placement)
- How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

Constraints:
- Design must be very simple
  - Hardware realization
  - All decision making within nanosecond time scale
- Want to optimize performance for "typical" programs
  - Do extensive benchmarking and simulations
  - Many subtle engineering tradeoffs

Accessing Data in Memory Hierarchy

- Between any two levels, memory is divided into lines (aka "blocks")
- Data moves between levels on demand, in line-sized chunks
- Invisible to application programmer
  - Hardware responsible for cache operation
- Upper-level lines a subset of lower-level lines

Access word \( w \) in line \( a \) (hit)
Access word \( v \) in line \( b \) (miss)
Direct-Mapped Caches

**Simplest Design**
- Each memory line has a unique cache location

**Parameters**
- Line (aka block) size $B = 2^b$
  - Number of bytes in each line
  - Typically 2X-8X word size
- Number of Sets $S = 2^s$
  - Number of lines cache can hold
- Total Cache Size $= B \times S = 2^b \times 2^s$

**Physical Address**
- Address used to reference main memory
- $n$ bits to reference $N = 2^n$ total bytes
- Partition into fields
  - Offset: Lower $b$ bits indicate which byte within line
  - Set: Next $s$ bits indicate how to locate line within cache
  - Tag: Identifies this line when in cache

---

Indexing into Direct-Mapped Cache

- Use set index bits to select cache set

Set 0:
- Tag [Valid] 0 1 · · · $B$-

Set 1:
- · · ·

Set $S-1$:
- Tag [Valid] 0 1 · · · $B$-

---

Direct-Mapped Cache Tag Matching

**Identifying Line**
- Must have tag match high order bits of address
- Must have Valid = 1

- Lower bits of address select byte or word within cache line

---

Properties of Direct Mapped Caches

**Advantages**
- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

**Disadvantages**
- Vulnerable to thrashing
- Two heavily used lines have same cache index
- Repeatedly evict one to make room for other
Vector Product Example

```c
float dot_prod(float x[1024], y[1024])
{
    float sum = 0.0;
    int i;
    for (i = 0; i < 1024; i++)
        sum += x[i]*y[i];
    return sum;
}
```

Machine
- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

Performance
- Good case: 24 cycles / element
- Bad case: 66 cycles / element

Thrashing Example

Access Sequence
- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - Hit
- Read y[1]
  - Hit
- ...
- 2 misses / 8 reads

Analysis
- x[i] and y[i] map to different cache lines
- Miss rate = 25%
- Two memory accesses / iteration
- On every 4th iteration have two misses

Timing
- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = 10 + 0.25 * 2 * 28 = 24 cycles

Thrashing Example: Good Case

Access Pattern
- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - x[0], x[1], x[2], x[3] loaded
- Read y[1]
  - y[0], y[1], y[2], y[3] loaded
- ...
- 2 misses / 8 reads

Analysis
- x[i] and y[i] map to different cache lines
- Miss rate = 25%
- Two memory accesses / iteration
- On every 4th iteration have two misses

Timing
- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = 10 + 0.25 * 2 * 28 = 24 cycles

Thrashing Example: Bad Case

Access Pattern
- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - x[0], x[1], x[2], x[3] loaded
- Read y[1]
  - y[0], y[1], y[2], y[3] loaded
- ...
- 8 misses / 8 reads

Analysis
- x[i] and y[i] map to same cache lines
- Miss rate = 100%
- Two memory accesses / iteration
- On every iteration have two misses

Timing
- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = 10 + 1.0 * 2 * 28 = 66 cycles
Set Associative Cache

Mapping of Memory Lines
- Each set can hold \( E \) lines (usually \( E = 2 - 16 \))
- Given memory line can map to any entry within its given set

Eviction Policy
- Which line gets kicked out when bring new line in
- Commonly either “Least Recently Used” (LRU) or pseudo-random
  - LRU: least-recently accessed (read or written) line gets evicted

Indexing into 2-Way Associative Cache
- Use middle \( s \) bits to select from among \( S = 2^s \) sets

Set Associative Cache Tag Matching

Two-Way Set Associative Cache Implementation
- Set 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
Fully Associative Cache

Mapping of Memory Lines
- Cache consists of single set holding E lines
- Given memory line can map to any line in set
- Only practical for small caches

| Entire Cache
| Line 0: | Tag | Valid |
| Line 1: | Tag | Valid |
| Line E-1: | Tag | Valid |

LRU State

Replacement Algorithms
- When a block is fetched, which block in the target set should be replaced?
  **Optimal algorithm:**
  - replace the block that will not be used for the longest period of time
  - must know the future
  **Common Algorithms:**
  - Least recently used (LRU)
    - replace the block that has been referenced least recently
    - tracking this information requires some effort
  - Random (RAND)
    - replace a random block in the set
    - trivial to implement

Fully Associative Cache Tag Matching

Identifying Line
- Must check all of the tags for match
- Must have Valid = 1 for this line

Implementing LRU and RAND
- LRU
  - Need state machine for each set
  - Encodes usage ordering of each element in set
  - E! possibilities => ~ E log E bits of state
- RAND:
  - maintain a single modulo E counter
  - counter points to next block for replacement in any set
  - increment counter according to some schedule:
    - each clock cycle,
    - each memory reference, or
    - each replacement anywhere in the cache
Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

Write Through:
- Store by processor updates cache and memory
- (typically ~2X more loads than stores)
- Memory always consistent with cache
- Never need to store from cache to memory

Write Policy (Cont.)

Write Back:
- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires "dirty bit" for each line
    - Set when line in cache is modified
    - Indicates that line in memory is stale
- Memory not always consistent with cache

Write Buffering

Write Buffer
- Common optimization for write-through caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address

Multi-Level Caches

Options: separate data and instruction caches, or a unified cache

How does this affect self modifying code?
Cache Performance Metrics

**Miss Rate**
- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 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:
  - 1-3 clock cycles for L1
  - 10-14 clock cycles for L2

**Miss Penalty**
- additional time required because of a miss
  - Typically 100-300 cycles for main memory

Impact of Cache and Block Size

**Cache Size**
- Effect on miss rate?
- Effect on hit time?

**Block Size**
- Effect on miss rate?
- Effect on miss penalty?

Impact of Associativity

- Direct-mapped, set associative, or fully associative?
- Total Cache Size (tags+data)?
- Miss rate?
- Hit time?
- Miss Penalty?

Impact of Replacement Strategy

- RAND or LRU?
- Total Cache Size (tags+data)?
- Miss Rate?
- Miss Penalty?
Impact of Write Strategy

• Write-through or write-back?

Advantages of Write Through?

Advantages of Write Back?

Allocation Strategies

• On a write miss, is the block loaded from memory into the cache?

Write Allocate:
• Block is loaded into cache on a write miss.
• Usually used with write-back
• Otherwise, write-back requires read-modify-write to replace word within block

<table>
<thead>
<tr>
<th>write buffer block</th>
<th>read</th>
<th>modify</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

| temporary buffer    | 5 7 11 13 | 5 7 17 13 | 5 7 17 13 |
|                     | 5 7 11 13 | 5 7 17 13 | 5 7 17 13 |

| memory block        | 5 7 11 13 | 5 7 17 13 | 5 7 17 13 |
|                     | 5 7 11 13 | 5 7 17 13 | 5 7 17 13 |

• But if you’ve gone to the trouble of reading the entire block, why not load it in cache?

Allocation Strategies (Cont.)

• On a write miss, is the block loaded from memory into the cache?

No-Write Allocate (Write Around):
• Block is not loaded into cache on a write miss
• Usually used with write-through
  - Memory system directly handles word-level writes

Qualitative Cache Performance Model

Miss Types

• Compulsory ("Cold Start") Misses
  - First access to line not in cache
• Capacity Misses
  - Active portion of memory exceeds cache size
• Conflict Misses
  - Active portion of address space fits in cache, but too many lines map to same cache entry
  - Direct mapped and set associative placement only
• Coherence Misses
  - Block invalidated by multiprocessor cache coherence mechanism

Hit Types

• Temporal locality hit
  - Accessing same word that previously accessed
• Spatial locality hit
  - Accessing word spatially near previously accessed word
### Interactions Between Program & Cache

#### Major Cache Effects to Consider
- **Total cache size**
  - Try to keep heavily used data in highest level cache
- **Block size** (sometimes referred to as "line size")
  - Exploit spatial locality

#### Example Application
- Multiply $n \times n$ matrices
- $O(n^3)$ total operations
- Accesses:
  - $n$ reads per source element
  - $n$ values summed per destination
  - But may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

---

### Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

---

### Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

---

### Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

---

Page 11
Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}

Pentium Matrix Multiply Performance

Array size (n)

Cycles/iteration

Key idea:
Sub-blocks (i.e., $A_{xy}$) can be treated just like scalars.

Example: Blocked matrix multiplication
- "block" (in this context) does not mean "cache block".
- Instead, it means a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
}

for (kk=0; kk<n; kk+=bsize) {
  for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++)
        sum += a[i][k] * b[k][j];
      c[i][j] += sum;
    }
  }
}

Blocked Matrix Multiply (bijk)

for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
}

for (kk=0; kk<n; kk+=bsize) {
  for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++)
        sum += a[i][k] * b[k][j];
      c[i][j] += sum;
    }
  }
}
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a $1 \times \text{bsize}$ sliver of $A$ by a $\text{bsize} \times \text{bsize}$ block of $B$ and accumulates into $1 \times \text{bsize}$ sliver of $C$
- Loop over $i$ steps through $n$ rows slivers of $A$ & $C$, using same $B$

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

Pentium Blocked Matrix Multiply Performance

Blocking ($bijk$ and $bikj$) improves performance by a factor of two over unblocked versions ($ijk$ and $jik$)
- relatively insensitive to array size.

Tolerating Cache Miss Latencies

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        sum += A[i][j];
```

What can the hardware do to avoid miss penalties in this case?

Prefetching: Start moving data close to the processor before it is needed

Prefetching allows cache misses to be overlapped with:
- computation, and
- other cache misses.
Prefetches can be issued either by Software or by Hardware

Software-Controlled Prefetching:
• programmer or compiler inserts explicit prefetch instructions
  - e.g., prefetch 8(%eax)

Hardware-Controlled Prefetching:
• the hardware tries to recognize strided memory accesses

Today’s x86 machines typically support both forms

Advantages/disadvantages of each approach?

Prefetches vs. Memory Loads

Similarities:
• both are given a data address as an argument
• if that location is not in the L1 data cache, then a cache miss is triggered, and the data is moved into the cache

Differences:
• prefetches do not have a register destination
  - Hence they are “non-binding”
• prefetches are non-blocking
  - i.e. the processor does not stall: it keeps executing
• prefetches do not trigger memory exceptions
  - it is ok to prefetch invalid memory addresses

Software Pipelining Prefetches

Original Loop:
for (i = 0; i < 1024; i++)
a[i][0] = 0;

Software Pipelined Loop (prefetching 16 iters ahead):
for (i = 0; i < 16; i++) /* Prolog */
prefetch(&a[i][0]);
for (i = 0; i < 1008; i++) /* Steady State */
prefetch(&a[i+16][0]);
a[i][0] = 0;
}

for (i = 1008; i < 1024; i++) /* Epilog */
a[i][0] = 0;

Reducing Software Overhead

Original Loop:
for (i = 0; i < 1024; i++)
a[i] = 0;

Unrolled Steady-State Loop:
for (i = 0; i < 1008; i+=8) {
  /* Prolog */
prefetch(&a[i+16][0]);
a[i][0] = 0;
a[i+1][0] = 0;
a[i+2][0] = 0;
a[i+3][0] = 0;
a[i+4][0] = 0;
a[i+5][0] = 0;
a[i+6][0] = 0;
a[i+7][0] = 0;
}

Spatial Locality (4B elements, 32B lines)
Reducing Software Overhead

Original Loop:
for (i = 0; i < 1024; i++)
a[i] = 0; ← Spatial Locality (48 elements, 32B lines)

"Strip-Mined" Steady-State Loop:
for (i = 0; i < 1008; i+=8) {
prefetch(&a[i+16][0]);
for (i2 = i; i2 < i+8; i2++)
a[i2][0] = 0;
}

More on Software Prefetching

Compiler algorithm for inserting prefetches:

Info on prefetch support in gcc can be found here:

Hardware Prefetching on the Pentium 4

The hardware attempts to detect repeated patterns of misses to consecutive caches lines
- the hardware must see several iterations first before it recognizes this pattern
- if the loop contains too many different memory accesses, it may not recognize the pattern

Once the pattern is recognized, the hardware then attempts to prefetch 2 cache lines ahead
- it will not prefetch beyond 4KB page boundaries
- it will continue prefetching beyond the end of the loop

Word of caution: hardware and software prefetching may interfere with each other

Hardware Prefetching Speedup
on an Intel Pentium 4

(from Hennessy & Patterson, CA-AQQ, 5th Edition)
**Prefetching Summary**

Limitations of Prefetching:
- requires that access patterns be predictable
  - easy for array accesses, more difficult for pointer chasing
- performance is ultimately limited by memory hierarchy bandwidth
  - but it is easier to increase bandwidth than to reduce latency!

Prefetching is generally more applicable than “cache blocking”

Many commercial compilers now insert software prefetches, and many processors support some form of stride-1 hardware prefetching

Prefetching is a valuable technique for hiding large memory access latencies

---

**10 Cache Optimizations (from H&P)**

1. Small and simple caches to reduce hit time
2. Way prediction to reduce hit time
3. Pipelined cache access to increase cache BW
4. Nonblocking caches to increase cache BW
5. Multibank caches to increase cache BW
6. Critical word first and early restart to reduce miss penalty
7. Merging write buffer to reduce miss penalty
8. Compiler optimizations reduce miss rate
9. Hardware prefetching to reduce miss penalty/rate
10. Compiler prefetching to reduce miss penalty/rate