Action Items:
• Programming Assignment 4 is out
• Cache Problem Set is out (no turn-in)
Announcement

- Programming Assignment 4 is out
  - Main assignment due on 11:59pm, Monday, April 2.
Announcement

- Programming Assignment 4 is out
  - Main assignment due on 11:59pm, **Monday, April 2.**
- I have put up a Problem Set to help you understand cache better
  - No turn-in required. Solutions to be released soon
  - Get a preview of final exam problems!!!
Announcement
Announcement

• Mid-term:
  • Mid-term grades will be updated after the class
  • You can get your exams from the TAs after the class
  • Solutions will be posted very soon
Announcement

• Mid-term:
  • Mid-term grades will be updated after the class
  • You can get your exams from the TAs after the class
  • Solutions will be posted very soon

• Statistics:
  • Average: 50.7; Standard-deviation: 10.0
  • Random guess: $1/\infty = 0$
  • Just writing down “I don’t know”: 15
Announcement

• Mid-term:
  • Mid-term grades will be updated after the class
  • You can get your exams from the TAs after the class
  • Solutions will be posted very soon

• Statistics:
  • Average: 50.7; Standard-deviation: 10.0
  • Random guess: $1/\infty = 0$
  • Just writing down “I don’t know”: 15

• So overall encouraging results
Announcement
Today

• Review: Cache memory organization and operation
• Performance impact of caches
  • Analytical Model
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal locality
General Cache Organization (S, E, B)

- E = $2^e$ lines per set
- S = $2^s$ sets
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)
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)
- valid bit
- tag
- 0 1 2 \ldots \ B-1
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)

- Dirty bit (if write-back)
- Valid bit
- Tag

Diagram:
- Sets
- Lines
- Bytes

Legend:
- $d$: dirty bit
- $v$: valid bit
- $tag$: tag
- $0$, $1$, $2$, etc.: blocks
- $B-1$: end of block

Carnegie Mellon
General Cache Organization (S, E, B)

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

**Cache size:**
\[ C = S \times E \times B \] data bytes

**Overhead:**
- Tag, valid bit, dirty bit.
- Plus bits for implementing replacement policy (not shown).

**Diagram:**
- Sets
- Lines
- Valid bit
- Dirty bit (if write-back)
- Tags
- B-1
Cache Access

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

Address of word:

- \( t \) bits
- \( s \) bits
- \( b \) bits

- tag
- set index
- block offset
Cache Access

$E = 2^e \text{ lines per set}$

$S = 2^s \text{ sets}$

• Locate set

Address of word:
- $t$ bits: tag
- $s$ bits: set index
- $b$ bits: block offset
Cache Access

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

E = $2^e$ lines per set

S = $2^s$ sets

Address of word:
- t bits
- s bits
- b bits

B = $2^b$ bytes per cache block (the data)
Cache Access

E = $2^e$ lines per set

S = $2^s$ sets

$S_0$ $S_1$ $S_2$

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

data begins at this offset

valid bit

$B = 2^b$ bytes per cache block (the data)
Example: Direct Mapped Cache

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

\[ S = 2^s \text{ sets} \]

Address of char:

\[
\begin{array}{c|c|c}
\text{t bits} & 0...01 & 100 \\
\end{array}
\]
Example: Direct Mapped Cache

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

\[ S = 2^s \text{ sets} \]

Address of char:
\[ \begin{array}{c|c|c}
\text{t bits} & 0...01 & 100 \\
\end{array} \]

find set
Example: Direct Mapped Cache

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

V | tag | 0 1 2 3 4 5 6 7

Address of char: t bits 0...01 100
Example: Direct Mapped Cache

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

valid? + match: assume yes = hit

Address of char:

```
0...01 100
```

```
v
tag 0 1 2 3 4 5 6 7
```
Example: Direct Mapped Cache

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

valid? + match: assume yes = hit

Address of char:

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

block offset
Example: Direct Mapped Cache

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

valid? + match: assume yes = hit

Address of char:

v tag 0 1 2 3 4 5 6 7

Byte 4 is here

block offset
Example: Direct Mapped Cache

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

If tag doesn’t match: old line is evicted and replaced
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

| 0   | [0000₂], |
| 1   | [0001₂], |
| 7   | [0111₂], |
| 8   | [1000₂], |
| 0   | [0000₂], |

```
<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

- 0: [0000]₂, miss
- 1: [0001]₂
- 7: [0111]₂
- 8: [1000]₂
- 0: [0000]₂

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0    [0000₂],     miss
1    [0001₂],
7    [0111₂],
8    [1000₂],
0    [0000₂]

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 \rightarrow [0000_2], \text{miss}
1 \rightarrow [0001_2], \text{hit}
7 \rightarrow [0111_2], \text{miss}
8 \rightarrow [1000_2],
0 \rightarrow [0000_2]

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

<table>
<thead>
<tr>
<th>Address space, i.e., Memory = 16 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>B=2 bytes/line, S=4 sets, E=1 line/set</td>
</tr>
</tbody>
</table>

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

Set 0
- | 1 | 1 | M[8-9] |
Set 1
- |   |   |        |
Set 2
- |   |   |        |
Set 3
- | 1 | 0 | M[6-7] |
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂], miss</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[0001₂], hit</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111₂], miss</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂], miss</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂], miss</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
E-way Set Associative Cache (Here: $E = 2$)

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

Address of `short int`:

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

```
<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>v</td>
<td>tag</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>v</td>
<td>tag</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>v</td>
<td>tag</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>v</td>
<td>tag</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>v</td>
<td>tag</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>v</td>
<td>tag</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>v</td>
<td>tag</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>
```
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

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:

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

```
v  tag  0 1 2 3 4 5 6 7  v  tag  0 1 2 3 4 5 6 7
```
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

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

```
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

compare both
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

match: yes = hit

valid? +

compare both
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

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

compare both

valid? + match: yes = hit

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:

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

Compare both
valid? + match: yes = hit

block offset
short int (2 Bytes) is here
E-way Set Associative Cache (Here: E = 2)

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

![Diagram of E-way Set Associative Cache]

Address of short int: t bits 0...01 100

valid? + match: yes = hit

block offset

short int (2 Bytes) is here

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000\textsubscript{2}],</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[0001\textsubscript{2}],</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111\textsubscript{2}],</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000\textsubscript{2}],</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000\textsubscript{2}]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):
0 [0000₂], miss
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

Set 0

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 1

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

| t=2 s=1 b=1 | 4-bit address space, i.e., Memory = 16 bytes |
| xx | x | x |

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

Set 0

<table>
<thead>
<tr>
<th>1</th>
<th>00</th>
<th>M[0-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 1

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=2</th>
<th>s=1</th>
<th>b=1</th>
<th>XX</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Decoded</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

Set 0:
- v: 01
- Tag: 01
- Block: M[6-7]

Set 1:
- v: 01
- Tag: 01
- Block: M[6-7]
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>
4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Today

• Review: Cache memory organization and operation
• Performance impact of caches
  • Analytical Model
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal locality
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.
Cache Performance Metrics

• 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:
    • 1~4 clock cycle for L1
    • 5~10 clock cycles for L2
Cache Performance Metrics

• Miss Penalty
  • Additional time required because of a miss
    • Typically 50-200 cycles for main memory
    • Trend: increasing!
Let’s think about those numbers

• Huge difference between a hit and a miss
  • Could be 100x, if just L1 and main memory
Let’s think about those numbers

• Huge difference between a hit and a miss
  • Could be 100x, if just L1 and main memory
• Compare 97% hit rate with 99% hit rate
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  • Average access time:
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  • Average access time:
    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  • Average access time:
    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  • Average access time:
    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles

• Think of it as reducing the miss rate from 3% to 1% (3X improvement) rather than improving hit rate
Let’s think about those numbers

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

• Compare 97% hit rate with 99% hit rate
  • Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  • Average access time:
    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles

• Think of it as reducing the miss rate from 3% to 1% (3X improvement) rather than improving hit rate

• Improving hit rate by even a little bit helps overall speed a lot
Writing Cache Friendly Code

• Make the common case go fast
  • Inner loops get executed most often. So focus on those

• Minimize the misses in the inner loops
  • Repeated references to variables are good (temporal locality)
  • Stride-1 reference patterns are good (spatial locality)
Today

• Review: Cache memory organization and operation
• Performance impact of caches
  • Analytical Model
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal locality
Matrix Multiplication Example

\[
C_{ij} = A_{ik} \times B_{kj}
\]
Matrix Multiplication Example

```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;
    }
}
```

Diagram:

- **C**
  - i down
  - j right

- **A**
  - i down
  - k right

- **B**
  - j down
  - k right

C = A \times B
Matrix Multiplication Example

```
/* 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;
    }
}
```

Variable `sum` held in register
Matrix Multiplication Example

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations

```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;
    }
}
```

Variable `sum` held in register
Miss Rate Analysis for Matrix Multiply

• Assume:
  • Block size = 32B (big enough for four doubles)
  • Matrix dimension (N) is very large
    • Approximate 1/N as 0.0
    • Cache is not even big enough to hold multiple rows

• Analysis Method:
  • Look at access pattern of inner loop
Layout of C Arrays in Memory (review)

• C arrays allocated in row-major order
  • each row in contiguous memory locations

• Stepping through columns in one row:
  • for (i = 0; i < N; i++)
    sum += a[0][i];
  • accesses successive elements
  • cache line size (32) > size of an element (8 bytes), exploiting spatial locality!
    • miss rate = 8 / 32 = 25%
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through rows in one column:
  - `for (i = 0; j < n; j++)`
    - `sum += a[i][0];`
    - accesses distant elements
    - no spatial locality!
    - miss rate = 1 (i.e. 100%)
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>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

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

Matmult/mult.c

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</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 (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<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>

Inner loop:

- Column-wise
- Fixed
- Column-wise

matmult/mm.c
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        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>
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
Core i7 Matrix Multiply Performance

Cycles per inner loop iteration vs Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
Today

• Review: Cache memory organization and operation
• Performance impact of caches
  • Analytical Model
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal locality
Example: Matrix Multiplication

```c

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];
}
```

### Example Matrix Multiplication

\[
\begin{pmatrix}
    c
\end{pmatrix}
= \begin{pmatrix}
    a & b
\end{pmatrix}
\begin{pmatrix}
    i
\end{pmatrix}
\]
Cache Miss Analysis

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

• First iteration:
Cache Miss Analysis

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

• First iteration:
  • $n/8 + n = 9n/8$ misses
Cache Miss Analysis

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

• Second iteration:

\[
\begin{array}{c}
\text{8 wide} \\
 n
\end{array}
\]
Cache Miss Analysis

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

• Second iteration:
  • Again:
    $n/8 + n = 9n/8$ misses

\[ n \cdot \text{8 wide} \]
Cache Miss Analysis

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

• Second iteration:
  • Again: $n/8 + n = 9n/8$ misses

• Total misses:
  • $9n/8 \times n^2 = (9/8) \times n^3$
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+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                    for (i1 = i; i1 < i+B; i++)
                        for (j1 = j; j1 < j+B; j++)
                            for (k1 = k; k1 < k+B; k++)
                                c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
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:
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
  • $2n/B \times B^2/8 = nB/4$
Cache Miss Analysis

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

• Second (block) iteration:
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
  • $2n/B * B^2/8 = nB/4$
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:
  • $nB/4 \times (n/B)^2 = n^3/(4B)$
Blocking Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
Blocking Summary

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

• Suggest largest possible block size \(B\), but limit \(3B^2 < C\)!
Blocking Summary

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

• Suggest largest possible block size $B$, but limit $3B^2 < C!$

• 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
Cache Summary

• Cache memories can have significant performance impact

• You can write your programs to exploit this!
  • Focus on the inner loops, where bulk of computations and memory accesses occur.
  • Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  • Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.