CS 61C: Great Ideas in Computer Architecture

Direct-Mapped Caches, Set Associative Caches, Cache Performance

Instructor: Alan Christopher
Great Idea #3: Principle of Locality/ Memory Hierarchy
Extended Review of Last Lecture

• Why have caches?
  – Intermediate level between CPU and memory
  – In-between in size, cost, and speed
• Memory (hierarchy, organization, structures) set up to exploit temporal and spatial locality
  – Temporal: If accessed, will access again soon
  – Spatial: If accessed, will access others around it
• Caches hold a subset of memory (in blocks)
  – We are studying how they are designed for fast and efficient operation (lookup, access, storage)
Extended Review of Last Lecture

• Fully Associative Caches:
  – Every block can go in any slot
    • Use random or LRU replacement policy when cache full
  – Memory address breakdown (on request)
    • Tag field is identifier (which block is currently in slot)
    • Offset field indexes into block
  – Each cache slot holds block data, tag, valid bit, and dirty bit (dirty bit is only for \textit{write-back})
    • The whole cache maintains LRU bits
Extended Review of Last Lecture

• Cache read and write policies:
  – Affect consistency of data between cache and memory
  – Write-back vs. write-through
  – Write allocate vs. no-write allocate

• On memory access (read or write):
  – Look at ALL cache slots in parallel
  – If Valid bit is 0, then ignore
  – If Valid bit is 1 and Tag matches, then use that data
• write, set Dirty bit if write-back
### Extended Review of Last Lecture

- **Fully associative cache layout**
  - 8-bit address space, 32-byte cache with 8-byte blocks
  - LRU replacement (5 bits), write-back and write allocate
  - Offset – 3 bits, Tag – 5 bits

- Each slot has 71 bits; cache has $4 \times 71 + 5 = 289$ bits

---

<table>
<thead>
<tr>
<th>Slot</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0x??</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>0x??</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>0x??</td>
</tr>
</tbody>
</table>

256 B address space  
block size (K)  
cache size (C)  
Need dirty bit  
LRU XXXXX
Agenda

• Direct-Mapped Caches
• Administrivia
• Set Associative Caches
• Cache Performance
Direct-Mapped Caches (1/3)

• Each memory block is mapped to exactly one slot in the cache (direct-mapped)
  – Every block has only one “home”
  – Use hash function to determine which slot

• Comparison with fully associative
  – Check just one slot for a block (faster!)
  – No replacement policy necessary
  – Access pattern may leave empty slots in cache
Direct-Mapped Caches (2/3)

• **Offset field** remains the same as before
• **Recall:** blocks consist of adjacent bytes
  - Do we want adjacent blocks to map to same slot?
  - **Index field:** Apply hash function to block address to determine *which slot* the block goes in
    • *(block address) modulo (# of *blocks* in the cache)*
• **Tag field** maintains same function (identifier), but is now shorter
TIO Address Breakdown

• Memory address fields:
  
  T bits
  I bits
  O bits

• Meaning of the field sizes:
  
  - **O** bits $\leftrightarrow 2^O$ bytes/block $= 2^{O-2}$ words/block
  - **I** bits $\leftrightarrow 2^I$ slots in cache = cache size / block size
  - **T** bits $= A - I - O$, where $A = \#$ of address bits ($A = 32$ here)
Direct-Mapped Caches (3/3)

• What’s actually in the cache?
  – Block of data \(8 \times K = 8 \times 2^O\) bits
  – Tag field of address as identifier (\(T\) bits)
  – Valid bit (1 bit)
  – Dirty bit (1 bit if write-back)
  – No replacement management bits!

• Total bits in cache = \# slots \(\times (8\times K + T + 1 + 1)\)
  \(= 2^I \times (8\times 2^O + T + 1 + 1)\) bits
DM Cache Example (1/5)

• Cache parameters:
  – Direct-mapped, address space of 64B, block size of 1 word, cache size of 4 words, write-through

• TIO Breakdown:
  – 1 word = 4 bytes, so \( O = \log_2(4) = 2 \)
  – Cache size / block size = 4, so \( I = \log_2(4) = 2 \)
  – \( A = \log_2(64) = 6 \) bits, so \( T = 6 - 2 - 2 = 2 \)

• Bits in cache = \( 2^2 \times (8 \times 2^2 + 2 + 1) = 140 \) bits
DM Cache Example (2/5)

• Cache parameters:
  – Direct-mapped, address space of 64B, block size of 1 word, cache size of 4 words, write-through
  – Offset – 2 bits, Index – 2 bits, Tag – 2 bits

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

• 35 bits per index/slot, 140 bits to implement
Which blocks map to each row of the cache? (see colors)

On a memory request:
(let’s say 001011_2)

1) Take Index field (10)
2) Check if Valid bit is true in that row of cache
3) If valid, then check if Tag matches
DM Cache Example (4/5)

- Consider the sequence of memory address accesses

Starting with a cold cache: 0 2 4 8 20 16 0 2

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>20</th>
<th>16</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

**0 miss**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

**2 hit**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

**4 miss**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

**8 miss**
DM Cache Example (5/5)

• Consider the sequence of memory address accesses

Starting with a cold cache: 0 2 4 8 20 16 0 2

- **20 miss**

```
 11 0 00 0x?? 0x?? 0x?? 0x??
```

- **16 miss**

```
 11 0 00 0x?? 0x?? 0x?? 0x??
```

- **0 miss**

```
 11 0 00 0x?? 0x?? 0x?? 0x??
```

- **2 hit**

```
 11 0 00 0x?? 0x?? 0x?? 0x??
```

• 8 requests, 6 misses – last slot was never used!
Worst-Case for Direct-Mapped

• Cold DM $ that holds 4 1-word blocks
• Consider the memory accesses: 0, 16, 0, 16,...

0 Miss

<table>
<thead>
<tr>
<th></th>
<th>M[0-3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

16 Miss

<table>
<thead>
<tr>
<th></th>
<th>M[0-3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

0 Miss

<table>
<thead>
<tr>
<th></th>
<th>M[16-19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

• HR of 0%
  - Ping pong effect: alternating requests that map into the same cache slot

• Does fully associative have this problem?
Comparison So Far

• Fully associative
  - Block can go into any slot
  - Must check ALL cache slots on request (“slow”)
  - TO breakdown (i.e. $l = 0$ bits)
  - “Worst case” still fills cache (more efficient)

• Direct-mapped
  - Block goes into one specific slot (set by Index field)
  - Only check ONE cache slot on request (“fast”)
  - TIO breakdown
  - “Worst case” may only use 1 slot (less efficient)
Agenda

• Direct-Mapped Caches
• Administrivia
• Set Associative Caches
• Cache Performance
• Proj1 still due Sunday
  - My OH tomorrow “go until they finish (sorta)”
    • As long as there's a student with pertinent questions I'll hang around.
    • I won't stay later than 7pm (go to Andrew's office hours at that point)
• HW4 will be released Friday, due next Sunday
Agenda

• Direct-Mapped Caches
• Administrivia
• Set Associative Caches
• Cache Performance
Set Associative Caches

• Compromise!
  – More flexible than DM, more structured than FA

• *N-way set-associative:* Divide $ into sets, each of which consists of N slots
  – Memory block maps to a set determined by Index field and is placed in any of the N slots of that set
  – Call N the *associativity*
  – New hash function:
    (block address) modulo (# sets in the cache)
  – Replacement policy applies to every set
Effect of Associativity on TlO (1/2)

- Here we assume a cache of fixed size (C), fixed block size (K)
- **Offset**: Points to a byte in a block (same as before)
- **Index**: Instead of pointing to a *slot*, now points to a *set*, so \( I = \log_2(C/K/N) \)
  - Fully associative (1 set): 0 Index bits!
  - Direct-mapped (N = 1): max Index bits
  - Set associative: somewhere in-between
- **Tag**: Remaining identifier bits \( T = A - I - O \)
Effect of Associativity on TIO (2/2)

- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e. the number of slots) and halves the number of sets – decreasing the size of the Index by 1 bit and increasing the size of the Tag by 1 bit.
Example: Eight-Block Cache Configs

- Total size of $ = \# \text{sets} \times \text{associativity}$
- For fixed $\text{size}$, associativity ↑ means \# \text{sets} ↓ and slots per set ↑
- With 8 blocks, an 8-way set associative $\text{}$ is same as a fully associative $\text{}$

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**One-way set associative (direct mapped)**

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Two-way set associative**

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Four-way set associative**

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Eight-way set associative (fully associative)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Block Placement Schemes

- Place memory block 12 in a cache that holds 8 blocks

- **Fully associative**: Can go in *any* of the slots (all 1 set)
- **Direct-mapped**: Can only go in slot \((12 \mod 8) = 4\)
- **2-way set associative**: Can go in either slot of set \((12 \mod 4) = 0\)
SA Cache Example (1/5)

• Cache parameters:
  - 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through

• How many sets?
  - $C/K/N = 4/1/2 = 2$ sets

• TIO Breakdown:
  - $O = \log_2(4) = 2$, $I = \log_2(2) = 1$, $T = 6 - 1 - 2 = 3$
SA Cache Example (2/5)

- Cache parameters:
  - 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through
  - Offset – 2 bits, Index – 1 bit, Tag – 3 bits

36 bits per slot, $36 \times 2 + 1 = 73$ bits per set, $2 \times 73 = 146$ bits to implement
SA Cache Example (3/5)

Each block maps into one set (either slot) (see colors)

On a memory request:
(let’s say 001011\textsubscript{two})

1) Take Index field (0)

2) For EACH slot in set, check valid bit, then compare Tag

Main Memory shown in blocks, so offset bits not shown (x’s)

Set numbers exactly match the Index field
SA Cache Example (4/5)

- Consider the sequence of memory address accesses
  Starting with a cold cache: 0 2 4 8 20 16 0 2

| Address | Cache State
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 miss</td>
</tr>
<tr>
<td>2</td>
<td>2 hit</td>
</tr>
<tr>
<td>4</td>
<td>4 miss</td>
</tr>
<tr>
<td>8</td>
<td>8 miss</td>
</tr>
</tbody>
</table>

Starting with the cache state:

```
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
```

For each access:

- **0 miss**:
  - Cache state:
    ```
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    ```

- **2 hit**:
  - Cache state:
    ```
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    ```

- **4 miss**:
  - Cache state:
    ```
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    ```

- **8 miss**:
  - Cache state:
    ```
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    1 0 0 0 0 0x?? 0x?? 0x?? 0x??
    ```
SA Cache Example (5/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:

Starting with a cold cache:

\[0 \quad 2 \quad 4 \quad 8 \quad 20 \quad 16 \quad 0 \quad 2\]

\[M \quad H \quad M \quad M\]

\[\begin{array}{l|c|c|c|c|c|c}
\hline
\end{array}\]

\[\begin{array}{l|c|c|c|c|c|c}
\hline
\end{array}\]

\[\begin{array}{l|c|c|c|c|c|c}
\hline
\end{array}\]

\[\begin{array}{l|c|c|c|c|c|c}
\hline
\end{array}\]

\[\begin{array}{l|c|c|c|c|c|c}
\end{array}\]

- 8 requests, 6 misses
Worst Case for Set Associative

• Worst case for DM was repeating pattern of 2 into same cache slot (HR = 0/n)
  – Set associative for N > 1: HR = (n-2)/n
• Worst case for N-way SA with LRU?
  – Repeating pattern of at least N+1 that maps into same set
  – Back to HR = 0: 0, 8, 16, 0, 8, ...

\[ \begin{array}{ccc}
000 & 010 & 001 \\
M & M & M
\end{array} \]

\[ \begin{array}{cc}
000 & 001 \\
M[8-B1] & M[8-B1]
\end{array} \]
**Question:** What is the TIO breakdown for the following cache?

- 32-bit address space
- 32 KiB 4-way set associative cache
- 8 word blocks

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>I</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>21</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>19</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>P</td>
<td>19</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>17</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>
Technology Break
Agenda

• Direct-Mapped Caches
• Administrivia
• Set Associative Caches
• Cache Performance
Cache Performance

- Two things hurt the performance of a cache:
  - Miss rate and miss penalty
  - *Average Memory Access Time* (AMAT): average time to access memory considering both hits and misses

\[
\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
\]

(abbreviated \(\text{AMAT} = \text{HT} + \text{MR} \times \text{MP}\))

- **Goal 1**: Examine how changing the different cache parameters affects our AMAT
- **Goal 2**: Examine how to optimize your code for better cache performance
AMAT Example

• **Processor specs**: 200 ps clock, MP of 50 clock cycles, MR of 0.02 misses/instruction, and HT of 1 clock cycle

  \[
  \text{AMAT} = 1 + 0.02 \times 50 = 2 \text{ clock cycles} = 400 \text{ ps}
  \]

• Which improvement would be best?
  - 190 ps clock
  - MP of 40 clock cycles
  - MR of 0.015 misses/instruction
Cache Parameter Effects

• What is the potential impact of much larger cache on AMAT? (same block size)
  - Increase HR
  - Longer HT: smaller is faster
  - At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance

• Effect on TIO? Bits in cache? Cost?
Effect of Cache Performance on CPI

• **Recall:** CPU Performance
  
  CPU Time = Instructions × CPI × Clock Cycle Time
  
  \[ \text{CPU Time} = \text{IC} \times \text{CPI} \times \text{CC} \]

• Include memory accesses in CPI:

  \[ \text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \text{Average Memory-stall Cycles} \]

  CPU Time = IC × CPI_{stall} × CC

• Simplified model for memory-stall cycles:

  \[ \text{Memory-stall cycles} = \frac{\text{Accesses}}{\text{Instruction}} \times \text{MR} \times \text{MP} \]
CPI Example

- **Processor specs:** $CPI_{\text{base}}$ of 2, a 100 cycle MP, 36% load/store instructions, and 2% I$ and 4% D$ MRs
  - How many times per instruction do we access the I$? The D$?
  - MP is assumed the same for both I$ and D$
  - Memory-stall cycles will be sum of stall cycles for both I$ and D$
CPI Example

- **Processor specs:** CPI_{base} of 2, a 100 cycle MP, 36% load/store instructions, and 2% I$ and 4% D$ MRs
  - Memory-stall cycles
    \[= (100\% \times 2\% + 36\% \times 4\%) \times 100 = 3.44\]
    
    \[\text{I$} \quad \text{D$}\]
  - CPI_{stall} = 2 + 3.44 = 5.44 \text{ (more than 2 x CPI}_{base}!)\]

  What if the CPI_{base} is reduced to 1?

- What if the D$ miss rate went up by 1%?
Impacts of Cache Performance

\[ \text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \text{Memory-stall Cycles} \]

• Relative penalty of cache performance increases as processor performance improves (faster clock rate and/or lower \( \text{CPI}_{\text{base}} \))
  - Relative contribution of \( \text{CPI}_{\text{base}} \) and memory-stall cycles to \( \text{CPI}_{\text{stall}} \)
  - Memory speed unlikely to improve as fast as processor cycle time

• What can we do to improve cache performance?
Sources of Cache Misses: The 3Cs

• **Compulsory**: (cold start or process migration, 1st reference)
  - First access to block impossible to avoid; Effect is small for long running programs

• **Capacity**:
  - Cache cannot contain all blocks accessed by the program

• **Conflict**: (collision)
  - Multiple memory locations mapped to the same cache location
The 3Cs: Design Solutions

• Compulsory:
  – Increase block size (increases MP; too large blocks could increase MR)

• Capacity:
  – Increase cache size (may increase HT)

• Conflict:
  – Increase cache size
  – Increase associativity (may increase HT)
Summary

• Set associativity determines flexibility of block placement
  – Fully associative: blocks can go anywhere
  – Direct-mapped: blocks go in one specific location
  – N-way: cache split into sets, each of which have $n$ slots to place memory blocks

• Cache Performance
  – $AMAT = HT + MR \times MP$
  – CPU time $= IC \times CPI_{\text{stall}} \times CC$
    $= IC \times (CPI_{\text{base}} + \text{Memory-stall cycles}) \times CC$