Readings

- **Caches**

- **Required**
  - H&H Chapters 8.1-8.3
  - Refresh: P&P Chapter 3.5

- **Recommended**
  - An early cache paper by Maurice Wilkes
Recall: Cache Structure

Address

Tag Store
(is the address in the cache? + bookkeeping)

Hit/miss?

Data Store
(stores memory blocks)

Data
Cache Performance
Recall: Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy
Recall: How to Improve Cache Performance

- Three fundamental goals
  - Reducing miss rate
    - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
  - Reducing miss latency or miss cost
  - Reducing hit latency or hit cost
  - The above three together affect performance
Recall: Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches
Recall: Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout
- Loop interchange
- Data structure separation/merging
- Blocking
- ...
Recall: Restructuring Data Access Patterns (I)

- **Idea:** Restructure data layout or data access patterns
- **Example:** If column-major
  - `x[i+1,j]` follows `x[i,j]` in memory
  - `x[i,j+1]` is far away from `x[i,j]`

- This is called **loop interchange**
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, ...
Recall: Restructuring Data Access Patterns (II)

- **Blocking**
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- Also called **Tiling**
Restructuring Data Layout (I)

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys

- Why does the code on the left have poor cache hit rate?
  - “Other fields” occupy most of the cache line even though rarely accessed!

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node->key == input-key) {
        // access other fields of node
    }
    node = node->next;
}
```
Restructuring Data Layout (II)

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure

- Who should do this?
  - Programmer
  - Compiler
  - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

```c
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
};

struct Node-data {
    char [256] name;
    char [256] school;
};

while (node) {
    if (node->key == input-key) {
        // access node->node-data
    }
    node = node->next;
}
Multi-Core Issues in Caching
Caches in a Multi-Core System
Caches in Multi-Core Systems

- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?
Private vs. Shared Caches

- **Private** cache: Cache belongs to one core (a shared block can be in multiple caches)
- **Shared** cache: Cache is shared by multiple cores
Resource Sharing Concept and Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory

- Why?
  - Resource sharing improves utilization/efficiency $\rightarrow$ throughput
    - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
  - Reduces communication latency
    - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors
  - Compatible with the shared memory programming model
Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread’s performance
  - Thread performance can be worse than when it is run alone

- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads

- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources
Private vs. Shared Caches

- **Private** cache: Cache belongs to one core (a shared block can be in multiple caches)
- **Shared** cache: Cache is shared by multiple cores
Shared Caches Between Cores

- **Advantages:**
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

- **Disadvantages**
  - Slower access (cache not tightly coupled with the core)
  - Cores incur conflict misses due to other cores’ accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rates of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)
Example: Problem with Shared Caches

Example: Problem with Shared Caches

Example: Problem with Shared Caches

Processor Core 1 ← t1  t2→ Processor Core 2

L1 $  L1 $

L2 $  L2 $

......

t2’s throughput can be significantly reduced due to unfair cache sharing.

Memory System: A *Shared Resource* View

Most of the system is a shared resource, storing and moving data.
Cache Coherence
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

ld r2, x

Interconnection Network

P1

P2

Main Memory

x 1000
The Cache Coherence Problem

ld r2, x

1000

P1

P2

ld r2, x

1000

Interconnection Network

Main Memory

x 1000
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

Interconnection Network

Main Memory

P1

2000

P2

1000
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
1000

Should NOT
load 1000
ld r5, x

ld r2, x
2000

Interconnection Network

x
1000

Main Memory
Cache Coherence: Whose Responsibility?

- **Software**
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided a cache flush instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor’s local cache.
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors’ caches.
    - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

- **Hardware**
  - Simplifies software’s job
  - One idea: Invalidate all other copies of block A when a processor writes to it
A Very Simple Coherence Scheme (VI)

- Caches “snoop” (observe) each other’s write/read operations via a shared bus. If a processor writes to a block, all others invalidate the block.

- A simple protocol:
  - Write-through, no-write-allocate cache
  - Actions of the local processor on the cache block: PrRd, PrWr,
  - Actions that are broadcast on the bus for the block: BusRd, BusWr
(Non-)Solutions to Cache Coherence

- No hardware based coherence
  - Keeping caches coherent is software’s responsibility
    + Makes microarchitect’s life easier
  -- Makes average programmer’s life much harder
    ■ need to worry about hardware caches to maintain program correctness?
  -- Overhead in ensuring coherence in software (e.g., page protection and page-based software coherence)

- All caches are shared between all processors
  + No need for coherence
  -- Shared cache becomes the bandwidth bottleneck
  -- Very hard to design a scalable system with low-latency cache access this way
Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location.

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order.

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate.
  - **Write serialization**: provide a consistent order seen by all processors for the same memory location.

- Need a global point of serialization for this store ordering.
Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy

- Two major approaches
  - Snoopy bus (all operations are broadcast on a shared bus)
  - Directory based (a mediator gives permission to each request)

- To learn more, take the Graduate Comp Arch class
Cache Examples:
For You to Study
Cache Terminology

- **Capacity** ($C$):
  - the number of data bytes a cache stores

- **Block size** ($b$):
  - bytes of data brought into cache at once

- **Number of blocks** ($B = C/b$):
  - number of blocks in cache: $B = C/b$

- **Degree of associativity** ($N$):
  - number of blocks in a set

- **Number of sets** ($S = B/N$):
  - each memory address maps to exactly one cache set
How is data found?

- Cache organized into $S$ sets

- Each memory address maps to exactly one set

- Caches categorized by number of blocks in a set:
  - Direct mapped: 1 block per set
  - N-way set associative: N blocks per set
  - Fully associative: all cache blocks are in a single set

- Examine each organization for a cache with:
  - Capacity ($C = 8$ words)
  - Block size ($b = 1$ word)
  - So, number of blocks ($B = 8$)
Direct Mapped Cache

Address
11...11111100
11...11111000
11...11110100
11...11110000
11...11101100
11...11101000
11...11100100
11...11100000
00...01001000
00...01000000
00...00111000
00...00111000
00...00011000
00...00011000
00...00010100
00...00010100
00...00010000
00...00010000
00...00001100
00...00001100
00...00001000
00...00001000
00...00000100
00...00000100
00...00000000
00...00000000

mem[0xFF...FC]
mem[0xFF...F8]
mem[0xFF...F4]
mem[0xFF...F0]
mem[0xFF...EC]
mem[0xFF...E8]
mem[0xFF...E4]
mem[0xFF...E0]
mem[0x00...24]
mem[0x00...20]
mem[0x00...1C]
mem[0x00...18]
mem[0x00...14]
mem[0x00...10]
mem[0x00...0C]
mem[0x00...08]
mem[0x00...04]
mem[0x00...00]
Direct Mapped Cache Hardware

Memory Address

Tag
Set
Byte Offset

00

V
Tag
Data

8-entry x
(1+27+32)-bit
SRAM

Hit

Data

Memory Address

Data

Tag
Set
Byte Offset

=
Direct Mapped Cache Performance

# MIPS assembly code

```mips
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw  $t1, 0x4($0)
lw  $t2, 0xC($0)
lw  $t3, 0x8($0)
addi $t0, $t0, -1
j    loop
done:
```

Miss Rate $= 42$
Direct Mapped Cache Performance

```
# MIPS assembly code
addi $t0, $0, 5

loop:
    beq $t0, $0, done
    lw $t1, 0x4($0)
    lw $t2, 0xC($0)
    lw $t3, 0x8($0)
    addi $t0, $t0, -1
    j loop

done:
```

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...08]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...0C]</td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...0C]</td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...08]</td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td></td>
</tr>
</tbody>
</table>

Miss Rate $= \frac{3}{15} = \frac{20\%}{1}$

Temporal Locality
Compulsory Misses
Direct Mapped Cache: Conflict

![Memory Address Table]

# MIPS assembly code

```
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw  $t1, 0x4($0)
lw  $t2, 0x24($0)
addi $t0, $t0, -1
j    loop
done:
```

![Data Tag Table]

Miss Rate = 44
Direct Mapped Cache: Conflict

```
# MIPS assembly code
addi $t0, $0, 5
loop:   beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addi $t0, $t0, -1
j loop
done:
```

Miss Rate = 10/10 = 100%

Conflict Misses
N-Way Set Associative Cache
N-way Set Associative Performance

# MIPS assembly code

```
addi $t0, $0, 5
loop:
    beq $t0, $0, done
    lw $t1, 0x4($0)
    lw $t2, 0x24($0)
    addi $t0, $t0, -1
    j loop
done:
```

Miss Rate =

<table>
<thead>
<tr>
<th>Way 1</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>mem[0x00...24]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3
Set 2
Set 1
Set 0
N-way Set Associative Performance

Miss Rate = 2/10 = 20%

Associativity reduces conflict misses

# MIPS assembly code

```mips
addi $t0, $0, 5
loop:
    beq $t0, $0, done
    lw $t1, 0x4($0)
    lw $t2, 0x24($0)
    addi $t0, $t0, -1
    j loop
done:
```

Way 1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 0

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 0
Set 1
Set 2
Set 3
Fully Associative Cache

- No conflict misses
- Expensive to build
**Spatial Locality?**

- Increase block size:
  - Block size, $b = 4$ words
  - $C = 8$ words
  - Direct mapped (1 block per set)
  - Number of blocks, $B = C/b = 8/4 = 2$

![Diagram of memory address and block structure]
Direct Mapped Cache Performance

```
addi $t0, $0, 5
loop:  beq  $t0, $0, done
lw   $t1, 0x4($0)
lw   $t2, 0xC($0)
lw   $t3, 0x8($0)
addi $t0, $t0, -1
j    loop
done:
```

Miss Rate =
Direct Mapped Cache Performance

```
addi $t0, $0, 5
loop:
  beq $t0, $0, done
  lw $t1, 0x4($0)
  lw $t2, 0xC($0)
  lw $t3, 0x8($0)
  addi $t0, $t0, -1
  j loop
done:
```

Miss Rate = $1/15 = 6.67$

Larger blocks reduce compulsory misses through spatial locality
Cache Organization Recap

- Main Parameters
  - Capacity: $C$
  - Block size: $b$
  - Number of blocks in cache: $B = C/b$
  - Number of blocks in a set: $N$
  - Number of Sets: $S = B/N$

<table>
<thead>
<tr>
<th>Organization</th>
<th>Number of Ways (N)</th>
<th>Number of Sets ($S = B/N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>1</td>
<td>$B$</td>
</tr>
<tr>
<td>N-Way Set Associative</td>
<td>$1 &lt; N &lt; B$</td>
<td>$B/N$</td>
</tr>
<tr>
<td>Fully Associative</td>
<td>$B$</td>
<td>1</td>
</tr>
</tbody>
</table>
Capacity Misses

- Cache is too small to hold all data of interest at one time
  - If the cache is full and program tries to access data X that is not in cache, cache must evict data Y to make room for X
  - **Capacity miss** occurs if program then tries to access Y again
  - X will be placed in a particular set based on its address

- In a **direct mapped** cache, there is only one place to put X

- In an **associative cache**, there are multiple ways where X could go in the set.

- How to choose Y to minimize chance of needing it again?
  - Least recently used (LRU) replacement: the least recently used block in a set is evicted when the cache is full.
Types of Misses

- **Compulsory**: first time data is accessed

- **Capacity**: cache too small to hold all data of interest

- **Conflict**: data of interest maps to same location in cache

- **Miss penalty**: time it takes to retrieve a block from lower level of hierarchy
# MIPS assembly

```mips
lw $t0, 0x04($0)  
lw $t1, 0x24($0)  
lw $t2, 0x54($0)
```

<table>
<thead>
<tr>
<th>Set Number</th>
<th>3 (11)</th>
<th>2 (10)</th>
<th>1 (01)</th>
<th>0 (00)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Table of data and tags with set numbers:

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</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>
</tr>
</tbody>
</table>

(b) Table of data and tags with set numbers:

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LRU Replacement

# MIPS assembly

```mips
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

(a) Way 1

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Way 1

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 0

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...000</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)