CSCI-564 Advanced Computer Architecture

Lecture 4: Review of Memory Hierarchy

Bo Wu
Colorado School of Mines
From C to MIPS
Compiling: C to bits

Architecture-independent

Your Brain

Brain/Fingers/SWE

Programming Languages (C, C++)

Compiler

Assembly Language

Assembler

Machine code (.o files)

Linker

Executable (.exe files)

Architecture-dependent
Count the number of 1’s in the binary representation of i

```c
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j))
            c++;
    }
    return c;
}
```
In the Compiler

```
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j)) {
            c++;
        }
    }
    return c;
}
```
In the Compiler

Function popcount

Arguments
- int i
- int c
- int j

Body

= for

return c

= <

j 0 j 32 j +

= 1

if

&

j 1 c + i <<

c 1 l j

Abstract Syntax Tree
In the Compiler

Function popcount

Arguments
int i  int c  int j

Body
=  
for  return c

=  
if  

return c

t0 = 0
t1 = 0
t2 = t1 < 32
t2 == 0  t2 != 0

Abstract Syntax Tree

Control Flow Graph
In the Compiler

Control flow graph

Assembly

```
t0 = 0

t1 = 0

t2 = t1 < 32

t4 = 1

t5 = t4 << t1

t6 = t5 & a0

t0 = t0 + 1

t1 = t1 + 1

return t0

popcount:
ori $v0, $zero, 0
ori $t1, $zero, 0

ori $v0, $zero, 0
ori $t1, $zero, 0

slti $t2, $t1, 32
beq $t2, $zero, end
nop
addi $t3, $zero, 1
sllv $t3, $t3, $t1
and $t3, $a0, $t3
beq $t3, $zero, notone
nop
addi $v0, $v0, 1
notone:
beq $zero, $zero, top
addi $t1, $t1, 1
end:
jr $ra
nop
```

```
In the Assembler

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Executable Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>popcount:</code></td>
<td></td>
</tr>
<tr>
<td>ori $v0, $zero, 0</td>
<td>0011010000000010000000000000000000</td>
</tr>
<tr>
<td>ori $t1, $zero, 0</td>
<td>0011010000000010000000000000000000</td>
</tr>
<tr>
<td><code>top:</code></td>
<td></td>
</tr>
<tr>
<td>slti $t2, $t1, 32</td>
<td>00101001001010101000000000000010000</td>
</tr>
<tr>
<td>beq $t2, $zero, end</td>
<td>00010001010000000000000000100000001</td>
</tr>
<tr>
<td>nop</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>addi $t3, $zero, 1</td>
<td>00100000000010110000000000000001</td>
</tr>
<tr>
<td>sllv $t3, $t3, $t1</td>
<td>0000000000000000000000000000001000</td>
</tr>
<tr>
<td>and $t3, $a0, $t3</td>
<td>0000000000000000000000000000000010</td>
</tr>
<tr>
<td>beq $t3, $zero, notone</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>nop</td>
<td>00100000000010110000000000000001</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>00010000000000001111111111110110</td>
</tr>
<tr>
<td><code>notone:</code></td>
<td></td>
</tr>
<tr>
<td>beq $zero, $zero, top</td>
<td>001000000000000000000000000000001</td>
</tr>
<tr>
<td>addi $t1, $t1, 1</td>
<td>0001000000000000000000000000000000</td>
</tr>
<tr>
<td><code>end:</code></td>
<td></td>
</tr>
<tr>
<td>jr $ra</td>
<td>0010000001000010000000000000000001</td>
</tr>
<tr>
<td>nop</td>
<td>0001000000000000000000000000000000</td>
</tr>
</tbody>
</table>
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j))
            c++;
    }
    return c;
}

Take a look: [http://llvm.org/](http://llvm.org/)
Top Reasons to Use Assembly Code
Top Reasons to Use Assembly Code

• You are writing a compiler, so you don’t have a choice
Top Reasons to Use Assembly Code

• You are writing a compiler, so you don’t have a choice
• You want to understand what the machine is doing
Top Reasons to Use Assembly Code

• You are writing a compiler, so you don’t have a choice
• You want to understand what the machine is doing
• Assembly code is probably faster
Top Reasons to Use Assembly Code

• You are writing a compiler, so you don’t have a choice
• You want to understand what the machine is doing
• Assembly code is probably faster
• You want to show other people you are cool
Why do we need memory hierarchy?
Processor vs Memory Performance

CPU-DRAM Gap

1980: no cache in microprocessor;
1995 2-level cache
Really, how bad can it be?
Memory’s impact

\[ M = \% \text{ mem ops} \]
\[ \text{Mlat (cycles)} = \text{average memory latency} \]
\[ \text{BCPI} = \text{base CPI with single-cycle data memory} \]

\[ \text{CPI} = \]
Memory’s impact

\[ M = \% \text{ mem ops} \]

\[ M_{\text{lat}} \text{ (cycles)} = \text{average memory latency} \]

\[ \text{TotalCPI} = \text{BaseCPI} + M \times M_{\text{lat}} \]

Example:

\[ \text{BaseCPI} = 1; \ M = 0.2; \ M_{\text{lat}} = 240 \text{ cycles} \]

\[ \text{TotalCPI} = 49 \]

\[ \text{Speedup} = 1/49 = 0.02 \Rightarrow 98\% \text{ drop in performance} \]
Memory’s impact

\[ M = \% \text{ mem ops} \]
\[ \text{Mlat (cycles)} = \text{average memory latency} \]
\[ \text{TotalCPI} = \text{BaseCPI} + M \times \text{Mlat} \]

Example:
\[ \text{BaseCPI} = 1; \ M = 0.2; \ \text{Mlat} = 240 \text{ cycles} \]
\[ \text{TotalCPI} = 49 \]
\[ \text{Speedup} = 1/49 = 0.02 \implies 98\% \text{ drop in performance} \]

Remember!: Amdahl’s law does not bound the slowdown.
Poor memory performance can make your program arbitrarily slow.
More deeply into register file

Naive Register File

Diagram of a naive register file with inputs for write data, read address, and clock signals, as well as outputs for read data and write address.
More deeply into register file
Why is cache fast?

- Registers and cache are built on SRAM (Static Random Access Memory) technology
  - not dense
  - Bandwidth
    - registers — 324 GB/S
    - L1 cache — 128 GB/S

- Main memory is built on DRAM (Dynamic Random Access Memory) technology
  - need refreshing to keep data
  - very dense
  - Bandwidth of DDR3 — 16 GB/S per DIMM (dual in-line memory module)
Why is cache fast?

• Registers and cache are built on SRAM (Static Random Access Memory) technology
  • not dense
  • Bandwidth
    • registers — 324 GB/S
    • L1 cache — 128 GB/S

• Main memory is built on DRAM (Dynamic Random Access Memory) technology
  • need refreshing to keep data
  • very dense
  • Bandwidth of DDR3 — 16 GB/S per DIMM (dual in-line memory module)
What does “dense” mean?

1 Memory cell in 0.5μm processes
a) Gate Array SRAM
b) Embedded SRAM
c) Standard SRAM (6T cell with local interconnect)
d) ASIC DRAM
e) Standard DRAM (stacked cell)

[ From Foss, R.C. “Implementing Application-Specific Memory”, ISSCC 1996 ]
OK, now I understand cache is fast, but why don’t we build SRAM main memory?
OK, now I understand cache is fast, but why don’t we build SRAM main memory?

SRAM is much more expensive than DRAM!
Why is cache fast?

- Signals have further to travel
- Fan out to more locations
Memory hierarchy

- Capacity: Register << SRAM << DRAM
- Latency: Register << SRAM << DRAM
- Bandwidth: on-chip >> off-chip
- On a data access:
  - if data is in fast memory -> low-latency access to SRAM
  - if data is not in fast memory -> long-latency access to DRAM
Memory hierarchy

- Capacity: Register $<<$ SRAM $<<$ DRAM
- Latency: Register $<<$ SRAM $<<$ DRAM
- Bandwidth: on-chip $>>$ off-chip
- On a data access:
  - if data is in fast memory $\rightarrow$ low-latency access to SRAM
  - if data is not in fast memory $\rightarrow$ long-latency access to DRAM
- Memory hierarchies only work if the small, fast memory actually stores data that is reused by the processor
The principle of locality

- "Locality" is the tendency of data access to be predictable. There are two kinds:
  - Spatial locality: The program is likely to access data that is close to data it has accessed recently.
  - Temporal locality: The program is likely to access the same data repeatedly.
Evidence of locality

Spatial Locality

Temporal Locality

Temporal & Spatial Locality

Time (one dot per access to that address at that time)
Locality in action

- Label each access with whether it has temporal or spatial locality or neither
  - 1
  - 2
  - 3
  - 10
  - 4
  - 1800
  - 11
  - 30
  - 1
  - 2
  - 3
  - 4
  - 10
  - 190
  - 11
  - 30
  - 12
  - 13
  - 182
  - 1004
Locality in action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n

- 1 t
- 2 s, t
- 3 s, t
- 4 s, t
- 10 s, t
- 190 n
- 11 s, t
- 30 s
- 12 s
- 13 s
- 182 n?
- 1004 n
Caches exploit both types of locality

- Exploit **temporal locality** by remembering the contents of recently accessed locations
- Exploit **spatial locality** by fetching blocks of data around recently accessed locations
Basic problems in caching

- A cache holds a small fraction of all the cache lines, yet the cache itself may be quite large (i.e., it might contains 1000s of lines)
- Where do we look for our data?
- How do we tell if we’ve found it and whether it’s any good?
Basic cache organization
Cache geometry calculations

- Addresses break down into: tag, index, and offset.
- How they break down depends on the “cache geometry”

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)

- Index bits = \log_2(L)
- Offset bits = \log_2(B)
- Tag bits = A - (index bits + offset bits)
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits:
- Tag bits:
- off set bits:
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits:
- off set bits:
Practice

• 1024 cache lines. 32 Bytes per line.
• Index bits:  10
• Tag bits: 
• off set bits:  5
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits: 17
- off set bits: 5
Practice

- 32KB cache.
- 64byte lines.

- Index
- Offset
- Tag
Practice

- 32KB cache.
- 64byte lines.

- Index
- Offset
- Tag
Practice

- 32KB cache.
- 64byte lines.

- Index 9
- Offset 6
- Tag
Practice

- 32KB cache.
- 64byte lines.

- Index 9
- Offset 6
- Tag 17
Where to place data in cache?
Where to place data in cache?

- **Fully Associative**: block 12 can be placed anywhere.
- **(2-way) Set Associative**: block 12 can be placed anywhere in set 0 (12 mod 4).
- **Direct Mapped**: block 12 can be placed only into block 4 (12 mod 8).
New cache geometry calculations

- Addresses break down into: tag, index, and offset.
- How they break down depends on the “cache geometry”

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)
- Associativity = W

- Index bits = \( \log_2(L/W) \)
- Offset bits = \( \log_2(B) \)
- Tag bits = A - (index bits + offset bits)
Practice

• 32KB, 2048 Lines, 4-way associative.

• Line size:
• Sets:
• Index bits:
• Tag bits:
• Offset bits:
Practice

• 32KB, 2048 Lines, 4-way associative.

• Line size: 16B
• Sets:
• Index bits:
• Tag bits:
• Offset bits:
Practice

• 32KB, 2048 Lines, 4-way associative.

• Line size: 16B
• Sets: 512
• Index bits:
• Tag bits:
• Offset bits:
Practice

- 32KB, 2048 Lines, 4-way associative.

- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits: 
- Offset bits:
Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits:
- Offset bits: 4
Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits: 19
- Offset bits: 4
How to find block in cache?

- Cache uses index and offset to find potential match, then checks tag
- Tag check only includes higher order bits
- In this example (Direct-mapped, 8B block, 4 line cache)
How to find block in cache?

- Cache checks all potential blocks with parallel tag check
- In this example (2-way associative, 8B block, 4 line cache)
The cost of associativity

- Increased associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
- This limits associativity L1 caches to 2-8.
- Larger, slower caches can be more associative.
- Example: Nehalem
  - 8-way L1
  - 16-way L2 and L3.
- Core 2’s L2 was 24-way
Write through vs. write back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we *do not* forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we *do* forward the write, the cache is “write through.” In this case, a cache line is never dirty.
• Write back advantages

• Write through advantages
Write through vs. write back

- When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
- If we *do not* forward the write, the cache is "Write back", since the data must be written back when it’s evicted (i.e., the line can be dirty)
- If we *do* forward the write, the cache is "write through." In this case, a cache line is never dirty.
- Write back advantages
  Fewer writes farther down the hierarchy. Less bandwidth. Faster writes
- Write through advantages
Write through vs. write back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we *do not* forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we *do* forward the write, the cache is “write through.” In this case, a cache line is never dirty.
• Write back advantages
  Fewer writes farther down the hierarchy. Less bandwidth. Faster writes
• Write through advantages
  No write back required on eviction.