CS 33

Architecture and Optimization (3)
What About Branches?

• Challenge
  – instruction control unit must work well ahead of execution unit to generate enough operations to keep EU busy

  \[
  \begin{align*}
  80489f3: & \quad \text{movl} \quad \$0x1,\%ecx \\
  80489f8: & \quad \text{xorq} \quad %rdx,%rdx \\
  80489fa: & \quad \text{cmpq} \quad %rsi,%rdx \\
  80489fc: & \quad \text{jnl} \quad 8048a25 \\
  80489fe: & \quad \text{movl} \quad %esi,%edi \\
  8048a00: & \quad \text{imull} \quad (%rax,%rdx,4),%ecx
  \end{align*}
  \]

  – when it encounters conditional branch, cannot reliably determine where to continue fetching
Modern CPU Design

Instruction Control

Fetch Control

Instruction Decode

Instruction Cache

Register File

Retirement Unit

Operations

Address

Instructions

Prediction OK?

Register Updates

Instruction Cache

Data Cache

General Integer

FP Add

FP Mult/Div

Load

Store

Functional Units

Load

Store

Operation Results

Addr.

Data

Data

Addr.

Data

Operation Results

Integer/Branch

Execution
Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - branch taken: transfer control to branch target
  - branch not-taken: continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
80489f3:  movl  $0x1,%ecx
80489f8:  xorq  %rdx,%rdx
80489fa:  cmpq  %rsi,%rdx
80489fc:  jnl   8048a25
80489fe:  movl  %esi,%esi
8048a00:  imull (%rax,%rdx,4),%ecx
8048a25:  cmpq  %rdi,%rdx
8048a27:  jl    8048a20
8048a29:  movl  0xc(%rbp),%eax
8048a2c:  leal  0xfffffffff8(%rbp),%esp
8048a2f:  movl  %ecx,(%rax)
```
Branch Prediction

• Idea
  – guess which way branch will go
  – begin executing instructions at predicted position
    » but don’t actually modify register or memory data

80489f3:  movl  $0x1,%ecx
80489f8:  xorq  %edx,%edx
80489fa:  cmpq  %rsi,%rdx
80489fc:  jnl  8048a25

. . .

Predict taken

8048a25:  cmpq  %rdi,%rdx
8048a27:  jl   8048a20
8048a29:  movl  0xc(%rbp),%eax
8048a2c:  leal  0xfffffffff8(%rbp),%esp
8048a2f:  movl  %ecx,(%rax)

Begin execution
Branch Prediction Through Loop

```
80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

i = 98
Predict taken (OK)

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

i = 99
Predict taken (oops)

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

i = 100
Read invalid location

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

i = 101

Assume vector length = 100

Executed

Fetched
```
Branch Misprediction Invalidation

Assume

vector length = 100

Predict taken (OK)

Predict taken (oops)

Invalidate

```assembly
80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)           i = 98
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)           i = 99
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)           i = 100
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)           i = 101
80488b6:  incl  %edx
```
Branch Misprediction Recovery

80488b1: movl (%rcx,%rdx,4),%eax
80488b4: addl %eax,(%rdi)
80488b6: incl %edx
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1
80488bb: leal 0xfffffffe8(%rbp),%esp
80488be: popl %ebx
80488bf: popl %esi
80488c0: popl %edi

\[ i = 99 \]

Definitely not taken

- **Performance Cost**
  - multiple clock cycles on modern processor
  - can be a major performance limiter
Latency of Loads

typedef struct ELE {
    struct ELE *next;
    long data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
    long len = 0;
    while (ls) {
        len++;
        ls = ls->next;
    }
    return len;
}
Clearing an Array ...

```c
#define ITERS 100000000

void clear_array() {
    long dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<100; i++)
            dest[i] = 0;
    }
}
```

- 1 CPE
Store/Load Interaction

```c
void write_read(long *src, long *dest, long n) {
    long cnt = n;
    long val = 0;

    while(cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```
Store/Load Interaction

\[
\text{long } a[] = \{-10, 17\};
\]

Example A: `write_read(&a[0], &a[1], 3)`

<table>
<thead>
<tr>
<th>Iter.</th>
<th>cnt</th>
<th>a</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>3</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>Iter. 1</td>
<td>2</td>
<td>-10</td>
<td>-9</td>
</tr>
<tr>
<td>Iter. 2</td>
<td>1</td>
<td>-10</td>
<td>-9</td>
</tr>
<tr>
<td>Iter. 3</td>
<td>0</td>
<td>-10</td>
<td>-9</td>
</tr>
</tbody>
</table>

Example B: `write_read(&a[0], &a[0], 3)`

<table>
<thead>
<tr>
<th>Iter.</th>
<th>cnt</th>
<th>a</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>3</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>Iter. 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Iter. 2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Iter. 3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

`void write_read(long *src, long *dest, long n) {
    long cnt = n;
    long val = 0;
    while (cnt--) {
        *dest = val;
        val = (*src) + 1;
    }
}

- CPE 1.3
- CPE 7.3
Getting High Performance

• Good compiler and flags
• Don’t do anything stupid
  – watch out for hidden algorithmic inefficiencies
  – write compiler-friendly code
    » watch out for optimization blockers: procedure calls & memory references
  – look carefully at innermost loops (where most work is done)

• Tune code for machine
  – exploit instruction-level parallelism
  – avoid unpredictable branches
  – make code cache friendly (covered soon)
Hyper Threading
Multiple Cores

Chip

Instruction Control

Retirement Unit
Register File

Fetch Control
Instruction Decode

Instruction Cache

Address
Instructions
Operations

Functional Units

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store

Operation Results

Data Cache

Instruction Control

Retirement Unit
Register File

Fetch Control
Instruction Decode

Instruction Cache

Address
Instructions
Operations

Functional Units

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store

Operation Results

Data Cache

Other Stuff

More Cache

Other Stuff
CS 33

Memory Hierarchy I
Random-Access Memory (RAM)

- **Key features**
  - RAM is traditionally packaged as a chip
  - basic storage unit is normally a cell (one bit per cell)
  - multiple RAM chips form a memory
- **Static RAM (SRAM)**
  - each cell stores a bit with a four- or six-transistor circuit
  - retains value indefinitely, as long as it is kept powered
  - relatively insensitive to electrical noise (EMI), radiation, etc.
  - faster and more expensive than DRAM
- **Dynamic RAM (DRAM)**
  - each cell stores bit with a capacitor; transistor is used for access
  - value must be refreshed every 10-100 ms
  - more sensitive to disturbances (EMI, radiation,...) than SRAM
  - slower and cheaper than SRAM
### SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</td>
<td>100x</td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>

- **EDC** = error detection and correction
  - to cope with noise, etc.
Conventional DRAM Organization

- $d \times w$ DRAM:
  - $dw$ total bits organized as $d$ supercells of size $w$ bits
Reading DRAM Supercell (2,1)

Step 1(a): row access strobe (RAS) selects row 2
Step 1(b): row 2 copied from DRAM array to row buffer
Reading DRAM Supercell (2,1)

Step 2(a): column access strobe (CAS) selects column 1

Step 2(b): supercell (2,1) copied from buffer to data lines, and eventually back to the CPU
Memory Modules

addr (row = i, col = j)

64 MB memory module consisting of eight 8Mx8 DRAMs

Memory controller

64-bit doubleword at main memory address A

64-bit doubleword
Enhanced DRAMs

• Basic DRAM cell has not changed since its invention in 1966
  – commercialized by Intel in 1970

• DRAMs with better interface logic and faster I/O:
  – synchronous DRAM (SDRAM)
    » uses a conventional clock signal instead of asynchronous control
    » allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
  – double data-rate synchronous DRAM (DDR SDRAM)
    » DDR1
      • twice as fast
    » DDR2
      • four times as fast
    » DDR3
      • eight times as fast
Enhanced DRAMs

SDR: \( n \) B/sec

DDR1: \( 2n \) B/sec

DDR2: \( 4n \) B/sec

DDR3: \( 8n \) B/sec
Summary

• Memory transfer speed increased by a factor of 8
  – no increase in DRAM Cell Array speed
  – 8 times more data transferred at once
    » 64 adjacent bytes fetched from DRAM
A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR3 system at a speed that’s $n$ times that of an SDR system, where $n$ is:

a) 1  
b) 2  
c) 4  
d) 8
A Mismatch

- A processor clock cycle is ~0.3 nsecs
  - SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?
Caching to the Rescue

CPU

Cache
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory
- Typical system structure:
General Cache Concepts

Larger, slower, cheaper memory viewed as partitioned into “blocks”

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: **Miss**!

Block b is fetched from memory

Block b is stored in cache

• placement policy: determines where b goes
• replacement policy: determines which block gets evicted (victim)
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - cold misses occur because the cache is empty

- **Conflict miss**
  - most caches limit blocks to a small subset (sometimes a singleton) of the block positions in RAM
    - e.g., block i in RAM must be placed in block (i mod 4) in the cache
  - conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache block
    - e.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - occurs when the set of active cache blocks (working set) is larger than the cache
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)**

**Cache size:**

\[ C = S \times E \times B \text{ data bytes} \]
Cache Read

E = \( 2^e \) lines per set

S = \( 2^s \) sets

B = \( 2^b \) bytes per cache block (the data)

\( E = 2^e \) lines per set

\( S = 2^s \) sets

\( B = 2^b \) bytes per cache block (the data)

Address of word:

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

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

data begins at this offset

valid bit

\( v \) valid bit

0 1 2 \( \cdots \) B-1

Set

Block

Offset

Index

Tag
Example: Direct Mapped Cache (E = 1)

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

Address of int:

\[ \text{t bits} \quad 0\ldots01 \quad 100 \]

find set

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

\[ \begin{array}{cc}
\text{v} & \text{tag} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{cc}
\text{v} & \text{tag} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{cc}
\text{v} & \text{tag} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{cc}
\text{v} & \text{tag} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]
Example: Direct Mapped Cache (E = 1)

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

valid? + match: assume yes = hit

t bits

Address of int:

0...01 100

v

tag

0 1 2 3 4 5 6 7

block offset
Example: Direct Mapped Cache (E = 1)

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

No match: old line is evicted and replaced
Direct-Mapped Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 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>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</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>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>
A Higher-Level Example

```c
int sum_array_rows(double a[16][16]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16]) {
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

32 B = 4 doubles
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];

    return sum;
}
```
Conflict Misses: Aligned

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}
Different Alignments

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}

32 B = 4 doubles
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 |

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...

short int (2 Bytes) is here
Given the address above and the cache contents as shown, what is the value of the `int` at the given address?

a) 1111  
b) 3333  
c) 4444  
d) 7777
2-Way Set-Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

<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>
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; i++)
        for (i = 0; i < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables `sum`, `i`, `j`.

assume: cold (empty) cache,
a[0][0] goes here

32 B = 4 doubles
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j
A Higher-Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

32 B = 4 doubles
Conflict Misses

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}

32 B = 4 doubles
Intel Core i5 and i7 Cache Hierarchy

Processor package

Core 0

Regs

L1 d-cache

L1 i-cache

L2 unified cache

... 

Core 3

Regs

L1 d-cache

L1 i-cache

L2 unified cache

L3 unified cache (shared by all cores)

L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles

L2 unified cache: 256 KB, 8-way, Access: 11 cycles

L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches

Main memory

CS33 Intro to Computer Systems

XVI–53
What About Writes?

• Multiple copies of data exist:
  – L1, L2, main memory, disk

• What to do on a write-hit?
  – write-through (write immediately to memory)
  – write-back (defer write to memory until replacement of line)
    » need a dirty bit (line different from memory or not)

• What to do on a write-miss?
  – write-allocate (load into cache, update line in cache)
    » good if more writes to the location follow
  – no-write-allocate (writes immediately to memory)

• Typical
  – write-through + no-write-allocate
  – write-back + write-allocate