Computer Architecture and Performance Engineering

Saman Amarasinghe
Fall 2009
int fib(int n) { if (n == 0) return 1; if (n == 1) return 1;
return (fib(n-1) + fib(n-2));}

pushq %rbp
pushq %rbx
pushq %rsi
movl %edi, %ebp
testl %ebp, %ebp
je ..B2.4
cmpl $1, %ebp
je ..B2.4:
lea -1(%rbp), %edi
call fib
movl %eax, %ebx
lea -2(%rbp), %edi
call fib
addl %eax, %ebx
movl %ebx, %eax
popq %rcx
movl %edi, %ebp
testl %ebp, %ebp
je ..B2.4
ret
lea -2(%rbp), %edi
call fib
movl $1, %eax
popq %rcx
ret
lea -1(%rbp), %edi
call fib
movl %eax, %ebx
ret
Computer Architecture Overview

Instructions
Memory System
Processor Bus and IO Subsystem
Disk System
GPU and Graphics System
Network

courtesy of Intel Corp.

© Saman Amarasinghe 2008
Intel® Core™ Microarchitecture – Computer Architecture Overview

Instructions
Memory System

courtesy of Intel Corp.
courtesy of Ziff Davis Internet
© Saman Amarasinghe 2008
Pipelining Execution

IF: Instruction fetch
EX: Execution
MEM: Memory access
WB: Write back

Cycles

Instruction # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
---|---|---|---|---|---|---|---|---|---|---
Instruction i | IF | ID | EX | MEM | WB | | | | | |
Instruction i+1 | IF | ID | EX | MEM | WB | | | | | |
Instruction i+2 | IF | ID | EX | MEM | WB | | | | | |
Instruction i+3 | IF | ID | EX | MEM | WB | | | | | |
Instruction i+4 | IF | ID | EX | MEM | WB | | | | | |
Limits to pipelining

**Hazards** prevent next instruction from executing during its designated clock cycle

- **Structural hazards**: attempt to use the same hardware to do two different things at once
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
Data Hazards: True Dependence

Instr_j is **data dependent (aka true dependence)** on Instr_i:

\[
\begin{align*}
&\text{addl rbx, rax} \\
&\text{subl rax, rcx}
\end{align*}
\]

If two instructions are data dependent, they cannot execute simultaneously, be completely overlapped or execute in out-of-order.

If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**
Benefits of Unrolling

```c
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i<1000000; i++)
}

xorl %edi, %edi
..B1.2:
    movl B(%rdx), %eax
    addl %eax, A(%rdx)
    addq $4, %rdx
    cmpq $4000000, %rdx
    jl ..B1.2
..B1.3:
    ret
```

For(i=0; i<N; i += 4) {
    A[i+1] = A[i+1] + 1
}

xorl %edi, %edi
..B1.2:
    movl B(%rdx), %eax
    movl 4+B(%rdi), %edx
    movl 8+B(%rdi), %ecx
    movl 12+B(%rdi), %esi
    addl %eax, A(%rdi)
    addl %edx, 4+A(%rdi)
    addl %ecx, 8+A(%rdi)
    addl %esi, 12+A(%rdi)
    addq $16, %rdi
    cmpq $4000000, %rdi
    jl ..B1.2
..B1.3:
    ret
Name Dependence #1: Anti-dependence

**Name dependence:** when 2 instructions use same register or memory location, called a *name*, but no flow of data between the instructions associated with that name; 2 versions of name dependence

Instr\textsubscript{j} writes operand *before* Instr\textsubscript{i} reads it

\[
\text{subl rax,rbx} \quad \text{addl rcx,rax}
\]

Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”

If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard
Name Dependence #2: Output dependence

Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} writes it.

\begin{align*}
\text{I:} & \quad \text{sub } r1, r4, r3 \\
\text{J:} & \quad \text{add } r1, r2, r3 \\
\text{K:} & \quad \text{mul } r6, r1, r7
\end{align*}

Called an “\textit{output dependence}” by compiler writers. This also results from the reuse of name “r1”

If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Instructions involved in a name dependence can execute simultaneously \textbf{if name used in instructions is changed} so instructions do not conflict

- Register renaming resolves name dependence for registers
- Renaming can be done either by compiler or by HW
Control Hazards

Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.

```plaintext
if p1 {
    S1;
};
if p2 {
    S2;
}
```

*S1* is control dependent on *p1*, and *S2* is control dependent on *p2* but not on *p1*.

Control dependence need not be preserved

- willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program

Speculative Execution
Intel® Core™ Microarchitecture – Pipelining

12 stage Pipeline

courtesy of Intel Corp.

© Saman Amarasinghe 2008
### Superscalar Execution

<table>
<thead>
<tr>
<th>Instruction type</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>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-issue super-scalar machine
ILP and Data Hazards

Finds Instruction Level Parallelism

- Multiple instructions issued in parallel

HW/SW must preserve program order:
order instructions would execute in if executed sequentially as determined by original source program

- Dependences are a property of programs

Importance of the data dependencies

- 1) indicates the possibility of a hazard
- 2) determines order in which results must be calculated
- 3) sets an upper bound on how much parallelism can possibly be exploited

Goal: exploit parallelism by preserving program order only where it affects the outcome of the program
**SIMD:**

- In computing, **SIMD (Single Instruction, Multiple Data)** is a technique employed to achieve data level parallelism, as in a vector or array processor.

- Intel calls the latest version **SSE**
Multimedia Instructions

Packed data type
➢ Separate register file

Single Instruction on Multiple Data (SIMD)

Figure 4: Packed Operation
Multimedia Instructions

Vectorization Converts Loops

for (i=0; i<=MAX; i++)
c[i]=a[i]+b[i];
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i < 1000000; i++)
    {
    }

    xorl   %edx, %edx
    ..B1.2:
    movdqa A(%rax), %xmm0
    padd    B(%rax), %xmm0
    movdqa 16+A(%rax), %xmm1
    padd    16+B(%rax), %xmm1
    movdqa 32+A(%rax), %xmm2
    padd    32+B(%rax), %xmm2
    movdqa 48+A(%rax), %xmm3
    padd    48+B(%rax), %xmm3
    movdqa 64+A(%rax), %xmm4
    padd    64+B(%rax), %xmm4
    movdqa 80+A(%rax), %xmm5
    padd    80+B(%rax), %xmm5
    movdqa 96+A(%rax), %xmm6
    padd    96+B(%rax), %xmm6
    movdqa 112+A(%rax), %xmm7
    padd    112+B(%rax), %xmm7
    movdqa %xmm0, A(%rax)
    movdqa %xmm1, 16+A(%rax)
    movdqa %xmm2, 32+A(%rax)
    movdqa %xmm3, 48+A(%rax)
    movdqa %xmm4, 64+A(%rax)
    movdqa %xmm5, 80+A(%rax)
    movdqa %xmm6, 96+A(%rax)
    movdqa %xmm7, 112+A(%rax)
    addq    $128, %rax
    cmpq    $4000000, %rax
    jl      ..B1.2
    ..B1.3:
    ret
    ..B1.2:
    ret
    ..B1.3:
Intel® Core™ Microarchitecture – Superscalar Execution

3 Load/Store
1 Branch
3 Integer
2 Floating Point
5 SSE

courtesy of ars Technica
Out of Order Execution

Issue varying numbers of instructions per clock

- dynamically scheduled
  - Extracting ILP by examining 100’s of instructions
  - Scheduling them in parallel as operands become available
  - Rename registers to eliminate anti and dependences
  - out-of-order execution
  - Speculative execution
Speculation

Different predictors
- Branch Prediction
- Value Prediction
- Prefetching (memory access pattern prediction)

Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
- Speculation $\Rightarrow$ fetch, issue, and execute instructions as if branch predictions were always correct
- Dynamic scheduling $\Rightarrow$ only fetches and issues instructions

Essentially a data flow execution model: Operations execute as soon as their operands are available
Intel® Core™ Microarchitecture - Out of Order Execution

12 stage Pipeline
14 execution units

18 instructions ready to execute
128 instructions in flight

courtesy of Intel Corp.

© Saman Amarasinghe 2008
Branch Prediction and Speculative Execution

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i (branch)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Branch target decided
Branch Prediction and Speculative Execution

Instruction # 1  2  3  4  5  6  7  8  9  10

Instruction i (branch)  IF  ID  EX  MEM  WB
stall
stall
stall
stall
Instruction i+1

Cycles

Branch target decided

IF  ID  EX  MEM  WB
Branch Prediction and Speculative Execution

Build a predictor to figure out which direction branch is going
- Today we have complex predictors with 99+% accuracy
- Even predict the address in indirect branches / returns

Fetch and speculatively execute from the predicted address
- No pipeline stalls

When the branch is finally decided, the speculative execution is confirmed or squashed
Intel® Core™ Microarchitecture – Branch Prediction

Complex predictor
Multiple predictors
➢ Use branch history
➢ Different algorithms
➢ Vote at the end
Indirect address predictor
Return address predictor

© Saman Amarasinghe 2008 courtesy of Intel Corp.
Memory System

The Principle of Locality:
- Program access a relatively small portion of the address space at any instant of time.

Two Different Types of Locality:
- Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)

Last 30 years, HW relied on locality for memory perf.
Levels of the Memory Hierarchy

**Capacity**
- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)
- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte
- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte
- **Disk**
  - 10s T Bytes, 10 ms
  - (10,000,000 ns)
  - ~ $1 / GByte
- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Access Time**
- **Registers**
  - Instr. Operands
  - Blocks
- **L1 Cache**
  - Blocks
  - Memory
  - L2 Cache
  - Memory
  - Disk
  - Tape

**Cost**
- **Registers**
  - Prog./Compiler
  - 1-8 bytes
- **L1 and L2 Cache**
  - Cache cntl
  - 32-64 bytes
- **Main Memory**
  - Cache cntl
  - 64-128 bytes
- **Disk**
  - OS
  - 4K-8K bytes
- **Files**
  - User/operator
  - Mbytes

**Upper Level**
- Faster

**Lower Level**
- Larger

8/28 CS252-Fall’07
Cache Issues

**Cold Miss**
- The first time the data is available
- Prefetching may be able to reduce the cost

**Capacity Miss**
- The previous access has been evicted because too much data touched in between
- “Working Set” too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

**Conflict Miss**
- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

**True Sharing Miss**
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

**False Sharing Miss**
- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don’t get into the same cache line
Intel Core™ Microarchitecture – Memory Sub-system

Intel Core 2 Quad Processor

**L1 Data Cache**
- Size: 32 KB
- Line Size: 64 bytes
- Latency: 3 cycles
- Associativity: 8-way

**L1 Instruction Cache**
- Size: 32 KB
- Line Size: 64 bytes
- Latency: 3 cycles
- Associativity: 8-way

**L2 Cache**
- Size: 6 MB
- Line Size: 64 bytes
- Latency: 14 cycles
- Associativity: 24-way
Outline

Overview of Computer Architecture

Profiling a Program

Set of Example Programs
Performance Analyzer

Helps you identify and characterize performance issues by:

- Collecting performance data from the system running your application.
- Organizing and displaying the data in a variety of interactive views, from system-wide down to source code or processor instruction perspective.
- Identifying potential performance issues and suggesting improvements.

Example: Intel Vtune, gprof
What Is a Hotspot?

**Where** in an application or system there is a **significant amount of activity**

- **Where** = address in memory
- => OS process
- => OS thread
- => executable file or *module*
- => user function (requires symbols)
- => line of source code (requires symbols with line numbers) or assembly instruction

- **Significant** = activity that occurs infrequently probably does not have much impact on system performance

- **Activity** = time spent or other internal processor event
  - Examples of other events: Cache misses, branch mispredictions, floating-point instructions retired, partial register stalls, and so on.
Two Ways to Track Location

**Problem:** I need to know where you spend most of your time.

**Statistical Solution:** I call you on your cellular phone every 30 minutes and ask you to report your location. Then I plot the data as a histogram.

**Instrumentation Solution:** I install a special phone booth at the entrance of every site you plan to visit. As you enter or exit every site, you first go into the booth, call the operator to get the exact time, and then call me and tell me where you are and when you got there.
Sampling Collector

Periodically interrupt the processor to obtain the execution context

- Time-based sampling (TBS) is triggered by:
  - Operating system timer services.
  - Every $n$ processor clockticks.

- Event-based sampling (EBS) is triggered by processor event counter overflow.
  - These events are processor-specific, like L2 cache misses, branch mispredictions, floating-point instructions retired, and so on.
The Statistical Solution: Advantages

No Installation Required
- No need to install a phone everywhere you want a man in the field to make a report.

Wide Coverage
- Assuming all his territory has cellular coverage, you can track him wherever he goes.

Low Overhead
- Answering his cellular telephone once in a while, reporting his location, and returning to other tasks do not take much of his time.
The Statistical Solution: Disadvantages

**Approximate Precision:**
- A number of factors can influence exactly how long he takes to answer the phone.

**Limited Report:**
- Insufficient time to find out how he got to where he is or where he has been since you last called him.

Statistical Significance: There are some places you might not locate him, if he does not go there often or he does not stay very long. Does that really matter?
The Instrumentation Solution: Advantages

**Perfect Accuracy**

- I know where you were immediately before and after your visit to each customer.
- I can calculate how much time you spent at each customer site.
- I know how many times you visited each customer site.
The Instrumentation Solution:

Disadvantages

**Low Granularity**
- Too coarse; the site is the site.

**High Overhead**
- You spend valuable time going to phone booths, calling operators, and calling me.

**High Touch**
- I have to build all those phone booths, which expands the space in each site you visit.
Events

**Intel provide 100’s of types of events**

- Can be very confusing (ex: “number of bogus branches”)
- Some useful event categories
  - Total instruction count and mix
  - Branch events
  - Load/store events
  - L1/L2 cache events
  - Prefetching events
  - TLB events
  - Multicore events
Use Event Ratios

In isolation, events may not tell you much.

Event ratios are dynamically calculated values based on events that make up the formula.
- Cycles per instruction (CPI) consists of clockticks and instructions retired.

There are a wide variety of predefined event ratios.
Outline

Overview of Computer Architecture
Profiling a Program
Set of Example Programs
```c
#define MAXA 10000
int maxa_half = MAXA/2;
int32_t A[MAXA];
  //          [0, 1, 2, 3, 4, ...]
int32_t incA[MAXA];
  //                  [ 0..MAXA-1 randomly]
int32_t rndA[MAXA];

multiple passes over data
  for(j=0; j<MAXA; j++) {
  }

test div by 4
  for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
  }

test incA[i] < maxa_half
  for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
  }

test j < maxa_half
  for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
  }

test rndA[i] < maxa_half
  for(j=0; j<MAXA; j++) {
    if(rndA[j] < maxa_half)
  }
```

© Saman Amarasinghe 2008
multiple passes over data

```c
for(j=0; j<MAXA; j++)
```

test \( j < \text{maxa}_\text{half} \)

```c
for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
}
```

```assembly
..B3.3:
    incl (%rax)
    addq $4, %rax
    cmpq %rdx, %rax
    jl ..B3.3

..B3.2:
    cmpl %edx, %ecx
    jge ..B3.4

..B3.4:
    incl %ecx
    cmpl $10000, %ecx
    jl ..B3.2
```

© Saman Amarasinghe 2008
Assembly listings

test div by 4

\[
\text{for}(j=0; j<\text{MAXA}; j++) \{ \\
\quad \text{if}((j \& 0x03) == 0) \\
\quad \quad A[j] = A[j]+1;
\}
\]

\[
\text{xorl} \quad %edx, %edx
\]

..B2.2:

\[
\text{testb} \quad $3, %dl \\
\text{jne} \quad ..B2.4
\]

..B2.3:

\[
\text{movslq} \quad %edx, %rax \\
\text{incl} \quad A(,%rax,4)
\]

..B2.4:

\[
\text{incl} \quad %edx \\
\text{cmpl} \quad $10000, %edx \\
\text{jl} \quad ..B2.2
\]

test incA[i] < maxa_half

\[
\text{for}(j=0; j<\text{MAXA}; j++) \{ \\
\quad \text{if}(\text{incA}[j] < \text{maxa}_\text{half}) \\
\quad \quad A[j] = A[j]+1;
\}
\]

\[
\text{movslq} \quad \text{maxa}_\text{half}(\%rip), \%rdx \\
\text{xorl} \quad %ecx, %ecx \\
\text{xorl} \quad %eax, %eax
\]

..B1.2:

\[
\text{cmpq} \quad \text{incA}(,%rcx,8), %rdx \\
\text{jle} \quad ..B1.4
\]

..B1.3:

\[
\text{incl} \quad A(,%rax)
\]

..B1.4:

\[
\text{addq} \quad $4, %rax \\
\text{addq} \quad $1, %rcx \\
\text{cmpq} \quad $10000, %rcx \\
\text{jl} \quad ..B1.2
\]

© Saman Amarasinghe 2008
## Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
### results

<table>
<thead>
<tr>
<th></th>
<th>RUNTIME (ms)</th>
<th>INST RETIRED.ANY EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
</tr>
</tbody>
</table>

**INST RETIRED.ANY**  Instructions retired.

This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

---

© Saman Amarasinghe 2008
## Results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>INST RETIRED.ANY events</th>
<th>INST RETIRED LOADS events</th>
<th>BR INST RETIRED.ANY events</th>
<th>Retired (ANY - LOAD - BR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>0.50</td>
<td>1.99</td>
<td>1.75</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>0.25</td>
<td>1.99</td>
<td>1.62</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>1.50</td>
<td>2.00</td>
<td>1.50</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>1.50</td>
<td>1.99</td>
<td>1.50</td>
</tr>
</tbody>
</table>

**INST RETIRED LOADS**
Instructions retired, contain a load

**INST RETIRED STORE**
Instructions retired, contain a store

**BR INST RETIRED ANY**
Number of branch instructions retired

© Saman Amarasinghe 2008
## Results

<table>
<thead>
<tr>
<th></th>
<th>Run time (ms)</th>
<th>INSTRETIREDF.ANY events</th>
<th>Clocks per Instructions Retired - CPI</th>
<th>CPI*Tot Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>0.84</td>
<td>1.25</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>1.60</td>
<td>2.19</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>0.82</td>
<td>1.33</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>4.17</td>
<td>6.78</td>
</tr>
</tbody>
</table>

### CPI

CPI = \( \frac{CPU\_CLK\_UNHALTED\_CORE}{INST\_RETIRED\_ANY} \)

High CPI indicates that instructions require more cycles to execute than they should. In this case there may be opportunities to modify your code to improve the efficiency with which instructions are executed within the processor. CPI can get as low as 0.25 cycles per instruction.

© Saman Amarasinghe 2008
## Results

<table>
<thead>
<tr>
<th>Multi Pass Over</th>
<th>Runtime (ms)</th>
<th>INST_RETIRED.ANY</th>
<th>CPI*Tot Instructions</th>
<th>BR_INST_RETIRED.MISPRED %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>1.25</td>
<td>2.50</td>
</tr>
<tr>
<td>Test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>2.19</td>
<td>400.00</td>
</tr>
<tr>
<td>Test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>1.33</td>
<td>2.00</td>
</tr>
<tr>
<td>Test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>6.78</td>
<td>2134.00</td>
</tr>
</tbody>
</table>

### BR_INST_RETIRED.MISPRED

This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. ....

© Saman Amarasinghe 2008
### results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>INST RETIRED.ANY events</th>
<th>BR INST RETIRED.MIS PRED %</th>
<th>&quot;Instructions wasted&quot; of mispredicted branches</th>
<th>Total 'Cost'</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>2.50</td>
<td>4.97</td>
<td>1.50</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>400.00</td>
<td>797.51</td>
<td>2.21</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>2.00</td>
<td>3.99</td>
<td>1.63</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>2134.00</td>
<td>4254.69</td>
<td>6.10</td>
</tr>
</tbody>
</table>

**Assume the cost of a mispredicted branch is 21 “instructions wasted”**

- Number 21 got the closest answer

© Saman Amarasinghe 2008
Accessing Memory

**inner accumulate**

```c
for(j=0; j<MAXB; j++)
    for(i=0; i<DRV; i++)
        B[j] = B[j]+1;
```

**Blocked access**

```c
for(j=0; j<MAXB; j += MAXA)
    for(i=0; i<DRV; i++)
        B[j+k] = B[j+k]+1;
```

**multiple passes over array**

```c
for(i=0; i<DRV; i++)
    for(j=0; j<MAXB; j++)
        B[j] = B[j]+1;
```

**stride access**

```c
for(i=0; i<DRV; i++)
    for(k=0; k<MAXA; k++)
        for(j=0; j<MAXB; j += MAXA)
            B[j+k] = B[j+k]+1;
```

© Saman Amarasinghe 2008
Memory access pattern

- Inner accumulate
- Multiple passes over array
- Blocked access
- Stride access
## Results

<table>
<thead>
<tr>
<th></th>
<th>RUNTIME (ms)</th>
<th>INST_RETIRED.ANY events</th>
<th>Clocks per Instructions Retired - CPI</th>
<th>CPI*Tot Instructions</th>
<th>INST_RETIRED LOADS events</th>
<th>L1 Data Cache Miss Rate</th>
<th>L2_LINES_IN.SELF.DEM AND events</th>
<th>RESOURCE_STALLS.ANY %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner Accumulate</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiple Passes</td>
<td>1.5</td>
<td>0.8</td>
<td>1.8</td>
<td>1.5</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>Blocked Access</td>
<td>0.9</td>
<td>0.8</td>
<td>1.0</td>
<td>0.9</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Strided Access</td>
<td>8.8</td>
<td>0.8</td>
<td>10.4</td>
<td>8.8</td>
<td>2</td>
<td>167</td>
<td>202</td>
<td>220</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Indirect random access

for(i=0; i<DRV; i++)
  for(j=0; j<MAXB; j += MAXA)
    for(k=0; k<MAXA; k++)

Indirect stride access

for(i=0; i<DRV; i++)
  for(j=0; j<MAXB; j += MAXA)
    for(k=0; k<MAXA; k++)

Pointer chase

struct list * curr;
for(i=0; i<DRV; i++) {
  curr = start;
  for(j=0; j<MAXB; j++) {
    curr->i = curr->i + 1;
    curr = curr->next;
  }
}
## Results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>INST RETIRED.ANY events</th>
<th>Clocks per Instructions Retired - CPI</th>
<th>CPI*Tot Instructions</th>
<th>INST RETIRED LOADS events</th>
<th>L1 Data Cache Miss Rate</th>
<th>L2_LINES_IN.SELF.DEM AND events</th>
<th>RESOURCESTALLS.ANY %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner Accumulate</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>indirect random access</td>
<td>8.1</td>
<td>2.7</td>
<td>3.0</td>
<td>8.3</td>
<td>4.0</td>
<td>53.3</td>
<td>0.0</td>
<td>180.0</td>
</tr>
<tr>
<td>indirect strided access</td>
<td>1.5</td>
<td>1.1</td>
<td>1.5</td>
<td>1.6</td>
<td>4.0</td>
<td>12.7</td>
<td>6.0</td>
<td>25.0</td>
</tr>
<tr>
<td>pointer chase</td>
<td>12.4</td>
<td>0.8</td>
<td>14.9</td>
<td>12.5</td>
<td>4.0</td>
<td>41.7</td>
<td>43.0</td>
<td>320.0</td>
</tr>
<tr>
<td>pointer chase -- post randomize</td>
<td>146</td>
<td>0.9</td>
<td>174</td>
<td>149</td>
<td>4.0</td>
<td>194</td>
<td>7752</td>
<td>3927</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Compiler Heroics

**What else can be done**

- Deeper optimizations (see lecture 2)
- Vectorization for SSE
- Loop interchange

<table>
<thead>
<tr>
<th></th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>101.78</td>
<td>32.25</td>
<td>3.16</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>127.28</td>
<td>34.78</td>
<td>3.66</td>
</tr>
<tr>
<td>test div by 3</td>
<td>223.43</td>
<td>37.24</td>
<td>6.00</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>134.50</td>
<td>134.11</td>
<td>1.00</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>687.44</td>
<td>658.72</td>
<td>1.04</td>
</tr>
<tr>
<td>Inner Accumulate</td>
<td>158.59</td>
<td>64.02</td>
<td>2.48</td>
</tr>
<tr>
<td>Multiple Passes</td>
<td>242.08</td>
<td>233.43</td>
<td>1.04</td>
</tr>
<tr>
<td>Blocked Access</td>
<td>136.79</td>
<td>82.27</td>
<td>1.66</td>
</tr>
<tr>
<td>Strided Access</td>
<td>1392.56</td>
<td>1400.19</td>
<td>0.99</td>
</tr>
<tr>
<td>indirect random access</td>
<td>1315.34</td>
<td>1308.75</td>
<td>1.01</td>
</tr>
<tr>
<td>indirect strided access</td>
<td>248.37</td>
<td>250.55</td>
<td>0.99</td>
</tr>
<tr>
<td>pointer chase</td>
<td>2005.12</td>
<td>2002.99</td>
<td>1.00</td>
</tr>
<tr>
<td>pointer chase -- post randomize</td>
<td>23581.20</td>
<td>23603.33</td>
<td>1.00</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008