Code Optimization II
October 2, 2001

Topics
- Machine-Dependent Optimization
  - Pointer code
  - Unrolling
  - Enabling instruction level parallelism
- Understanding Processor Operation
  - Translation of instructions into operations
  - Out-of-order execution of operations
- Branches and Branch Prediction
- Advice

General Forms of Combining

```c
void abstract_combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
    { t = t OP data[i];
        *dest = t;
    }
}
```

Data Types
- Use different declarations for data_t
- int
- float
- double

Operations
- Use different definitions of OP and IDENT
- */
- +/

Previous Best Combining Code

```c
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
    { sum += data[i];
        *dest = sum;
    }
}
```

Task
- Compute sum of all elements in vector
- Vector represented by C-style abstract data type

Machine Independent Opt. Results

Optimizations
- Reduce function calls and memory references within loop

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract-g</td>
<td>42.06</td>
<td>41.86</td>
<td>41.44</td>
<td>160.00</td>
</tr>
<tr>
<td>Abstract-O2</td>
<td>31.25</td>
<td>33.25</td>
<td>31.25</td>
<td>143.00</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
<td>21.15</td>
<td>135.00</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
<td>8.00</td>
<td>117.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
</tbody>
</table>

Performance Anomaly
- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary
- Caused by quirk of IA32 floating point
  - Memory uses 64-bit format, register use 80
  - Benchmark data caused overflow of 64 bits, but not 80
**Pointer Code**

```c
void combine4p(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data + length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```

**Optimization**
- Use pointers rather than array references
- CPE: 3.00 (Compiled -O2)
  - *Oops! We're not making progress here!*
  - *Warning: Some compilers do better job optimizing array code*

---

**Pointer vs. Array Code Inner Loops**

**Array Code**

```c
.L24: # Loop:
    addl (%eax,%edx,4),%ecx # sum += data[i]
    incl %edx # i++
    cmpl %esi,%edx # i:length
    jl .L24 # if < goto Loop
```

**Pointer Code**

```c
.L30: # Loop:
    addl (%eax),%ecx # sum += *data
    addl $4,%eax # data ++
    cmpl %edx,%eax # data:dend
    jb .L30 # if < goto Loop
```

**Performance**
- Array Code: 4 instructions in 2 clock cycles
- Pointer Code: Almost same 4 instructions in 3 clock cycles

---

**Modern CPU Design**

- **Instruction Control**
  - Fetch Control
  - Instruction Cache
  - Operations
  - Retirement Unit

- **Addressing**
  - Data Cache

- **Prediction**
  - OK?

- **Register File**
  - Branch Integer
  - Integer
  - FP Add
  - FP MulDiv

- **Operation Results**
  - Add
  - Data

---

**CPU Capabilities of Pentium III**

**Multiple Instructions Can Execute in Parallel**
- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

**Some Instructions Take > 1 Cycle, but Can be Pipelined**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>38</td>
<td>38</td>
</tr>
</tbody>
</table>
Translation Example

Version of Combine

- Integer data, multiply operation

Translation of First Iteration

1. L24: inl ( vex, edx, 4 ), %eax
   - Load data into %eax
2. L24: inl %esi, %edx
   - Load %esi into %edx
3. cmp %esi, %edx
   - Compare %esi with %edx
4. jne -10
   - Jump if not equal

Understanding Translation Ex. (cont.)

- Instruction control
- Jump to new destination
- Assemble instructions
- Branches taken

Instruction Control

- Reads instructions
- Creates operations
- Translates instructions into operations
- Takes branches to correct target
- Predicts whether branches will be taken and target
- Returns code after instruction
- Checks whether no prediction was made
- Begins fetching and decoding instructions at correct target
Visualizing Operations

Operations
- Vertical position denotes time at which executed
  - Cannot begin operation until operands available
- Height denotes latency

Operands
- Arcs shown only for operands that are passed within execution unit

3 Iterations of Combining Product

Unlimited Resource Analysis
- Assume operation can start as soon as operands available
- Operations for multiple iterations overlap in time

Performance
- Limiting factor becomes latency of integer multiplier
- Gives CPE of 4.0

4 Iterations of Combining Sum

Unlimited Resource Analysis
- Can begin a new iteration on each clock cycle
- Should give CPE of 1.0
- Would require executing 4 integer operations in parallel
Combining Sum: Resource Constraints

- Only have two integer functional units
- Some operations delayed even though operands available
- Set priority based on program order

Performance
- Sustain CPE of 2.0

Loop Unrolling

```c
void combine5(vec_ptr v, int *dest) {
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    int i;
    // Combine 3 elements at a time */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+2] + data[i+1];
    }
    // Finish any remaining elements */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
```

Visualizing Unrolled Loop

- Loads can pipeline, since don’t have dependencies
- Only one set of loop control operations

Executing with Loop Unrolling

- Predicted Performance
  - Can complete iteration in 3 cycles
  - Should give CPE of 1.0
- Measured Performance
  - CPE of 1.33
  - One iteration every 4 cycles
**Effect of Unrolling**

<table>
<thead>
<tr>
<th>Unrolling Degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Sum</td>
<td>2.0</td>
<td>1.50</td>
<td>1.33</td>
<td>1.50</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>Integer Product</td>
<td>4.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Sum</td>
<td>3.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Product</td>
<td>5.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

---

**Parallel Loop Unrolling**

```c
void combine6(vec_ptr v, int *dest) {
    int length = vec_length(v);
    int limit = length - 1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

**Code Version**

- Integer product

**Optimization**

- Accumulate in two different sums
  - Can be performed simultaneously
- Combine at end
- Exploits property that integer addition & multiplication are associative & commutative
- FP addition & multiplication not associative, but transformation usually acceptable

---

**Visualizing Parallel Loop**

- Two multiplies within loop no longer have data depency
- Allows them to pipeline

```
load (%eax, %edx.0, 4) \rightarrow t.1a
imull t.1a, %ecx.0 \rightarrow %ecx.1
load 4(%eax, %edx.0, 4) \rightarrow t.1b
imull t.1b, %ebx.0 \rightarrow %ebx.1
iaddl $2, %edx.0 \rightarrow %edx.1
cmpl %esi, %edx.1 \rightarrow cc.1
jl-taken cc.1
```

---

**Executing with Parallel Loop**

- **Predicted Performance**
  - Can keep 4-cycle multiplier busy performing two simultaneous multiplications
  - Gives CPE of 2.0
**Optimization Results for Combining**

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract-g</td>
<td>42.06</td>
<td>41.86</td>
</tr>
<tr>
<td>Abstract-O2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Pointer</td>
<td>3.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.06</td>
<td>4.00</td>
</tr>
<tr>
<td>2 X 2</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>4 X 4</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>Theoretical Opt:</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Worst:Best</td>
<td>39.7</td>
<td>33.5</td>
</tr>
</tbody>
</table>

**Limitations of Parallel Execution**

**Need Lots of Registers**
- To hold sums/products
- Only 6 usable integer registers
  - Also needed for pointers, loop conditions
- 8 FP registers
  - When not enough registers, must spill temporary onto stack
  - Wipes out any performance gains
- Not helped by renaming
  - Cannot reference more operands than instruction set allows

**Example**
- 8 X 8 integer product
- 7 local variables share 1 register

---

**What About Branches?**

**Challenge**
- Instruction Control Unit must work well ahead of Exec. Unit
  - To generate enough operations to keep EU busy

- When encounters conditional branch, cannot reliably determine where to continue fetching

```
80489f3:  movl $0x1,%ecx
80489f8:  xorl %edx,%edx
80489fa:  cmpl %esi,%edx
80489fc:  jnl 8048a25
80489fe:  movl %esi,%esi
8048a00:  imull (%eax,%edx,4),%ecx
```

- Executing
- Fetching & Decoding

**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:  xorl %edx,%edx
80489fa:  cmpl %esi,%edx
80489fc:  jnl 8048a25
80489fe:  movl %esi,%esi
8048a00:  imull (%eax,%edx,4),%ecx
```

Branch Not-Taken

```
8048a25:  cmpl %edi,%edx
8048a27:  jl 8048a20
8048a29:  movl 0xc(%ebp),%eax
8048a2c:  leal 0xffffffff(%ebp),%esp
8048a2f:  movl %ecx,(%eax)
```

Branch Taken
Branch Prediction

Idea
- Guess which way branch will go
- Begin executing instructions at predicted position
  - But don’t actually modify register or memory data

Execute

Predict Taken

Branch Misprediction Invalidation

Assume vector length = 100

Predict Taken (OK)

Invalidate

Predict Taken (Oops)

Branch Misprediction Recovery

Assume vector length = 100

Predict Taken (OK)

Definitely not taken

Performance Cost
- Misprediction on Pentium III wastes ~14 clock cycles
- That’s a lot of time on a high performance processor
Results for Alpha Processor

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>40.14</td>
<td>52.07</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>25.08</td>
<td>37.37</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>19.19</td>
<td>28.73</td>
</tr>
<tr>
<td>data access</td>
<td>6.26</td>
<td>13.26</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>1.76</td>
<td>8.08</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.51</td>
<td>6.32</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.25</td>
<td>6.33</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.19</td>
<td>4.44</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.15</td>
<td>2.34</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.11</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Worst : Best  
36.2  
11.4  
22.3  
26.7

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler

Machine-Dependent Opt. Summary

Pointer Code
- Look carefully at generated code to see whether helpful

Loop Unrolling
- Some compilers do this automatically
- Generally not as clever as what can be achieved by hand

Exposing Instruction-Level Parallelism
- Very machine-dependent

Warning:
- Benefits depend heavily on particular machine
- Best if performed by compiler
  - But GCC on IA32/Linux is not very good
- Do only for performance-critical parts of code

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

Don’t: Smash Code into Oblivion
- Hard to read, maintain, & assure correctness

Do:
- Select best algorithm
- Write code that’s readable & maintainable
  - Procedures, recursion, without built-in constant limits
  - Even though these factors can slow down code
- Eliminate optimization blockers
  - Allows compiler to do its job

Focus on Inner Loops
- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here