CS 201

Code Optimization, Part 2

Gerson Robboy
Portland State University

Topics

■ Machine-Dependent Optimizations
  ● 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
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
- Achieved CPE of 2.00
  - Cycles per element
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`
  - `+ / 0`
  - `* / 1`
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>
</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>
</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**

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

.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

.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

Execution

Instruction Control

Fetch Control
Instruction Decode
Instruction Cache

Instrs.

Fetch
Address

Operations

Predictions OK?

Register Updates

Retirement Unit
Register File

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

Functional Units

Operation Results

Addr.
Data

Data
Cache
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 Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>38</td>
<td>38</td>
</tr>
</tbody>
</table>
Instruction Control

Grabs Instruction Bytes From Memory
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into *Operations*
- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*
- Internal registers, not visible to us
- Keeps track of destination register for retiring the instruction
Combine4

.L24:             # Loop:
imull (%eax,%edx,4),%ecx  # t *= data[i]
incl %edx               # i++
cmpl %esi,%edx          # i:length
jl .L24                # if < goto Loop

.L24:
imull (%eax,%edx,4),%ecx
incl %edx
cmp %esi,%edx
jl .L24

Why can’t the result of “incl %edx.0” be kept in %edx.0? Why %edx.1?
Visualizing Operations

- Why can’t the `load` and the `imull` execute in parallel?
- Why can we do the `load` and the `incl` in parallel?
- Do we accomplish anything by doing the `jl` so soon before the `imull` is finished?

```
load (%eax, %edx, 4) → t.1
imull t.1, %ecx.0 → %ecx.1
incl %edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jl-taken cc.1
```
Visualizing Operations (cont.)

Operations

- Same as before, except that add has latency of 1

load (%eax,%edx,4) \(\rightarrow\) t.1
iaddl t.1, %ecx.0 \(\rightarrow\) %ecx.1
incl %edx.0 \(\rightarrow\) %edx.1
cmpl %esi, %edx.1 \(\rightarrow\) cc.1
jl-taken cc.1
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

Performance

- We can pipeline the loads 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

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;
}

Optimization

- Amortizes loop overhead across multiple iterations
- Finish extras at end
- Measured CPE = 1.33
Visualizing Unrolled Loop

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

```plaintext
load (%eax,%edx.0,4) → t.1a
iaddl t.1a, %ecx.0c → %ecx.1a
load 4(%eax,%edx.0,4) → t.1b
iaddl t.1b, %ecx.1a → %ecx.1b
load 8(%eax,%edx.0,4) → t.1c
iaddl t.1c, %ecx.1b → %ecx.1c
iaddl $3,%edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jl -taken cc.1
```
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

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is not monotonic with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

<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>2.00</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></td>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>FP Sum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>FP Product</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.00</td>
<td></td>
</tr>
</tbody>
</table>
Serial Computation

Computation

$(((1 \times x_0) \times x_1) \times x_2) \times x_3) \times x_4) \times x_5) \times x_6) \times x_7) \times x_8) \times x_9) \times x_{10}) \times x_{11})$

Performance

- $N$ elements, $D$ cycles/operation
- $N \times D$ cycles
Parallel Loop Unrolling

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 products
  - Can be performed simultaneously
- Combine at end

Performance
- CPE = 2.0
- 2X performance
Visualizing Parallel Loop

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

load (%eax,%edx.0,4) ➔ t.1a
imull t.1a, %ecx.0 ➔ %ecx.1
load 4(%eax,%edx.0,4) ➔ t.1b
imull t.1b, %ebx.0 ➔ %ebx.1
iaddl $2,%edx.0 ➔ %edx.1
cmpl %esi, %edx.1 ➔ cc.1
jl-taken cc.1
Dual Product Computation

Computation

\[
\left( \left( \left( \left( \left( 1 \times x_0 \right) \times x_2 \right) \times x_4 \right) \times x_6 \right) \times x_8 \right) \times x_{10} \right) \times \\
\left( \left( \left( \left( \left( 1 \times x_1 \right) \times x_3 \right) \times x_5 \right) \times x_7 \right) \times x_9 \right) \times x_{11} \right)
\]

Performance

- N elements, D cycles/operation
- (N/2+1)*D cycles
- ~2X performance improvement
Requirements for Parallel Computation

Mathematical

- Combining operation must be associative & commutative
  - OK for integer multiplication
  - Not strictly true for floating point
    » OK for most applications

Hardware

- Pipelined functional units
- Ability to dynamically extract parallelism from code
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 Integer Floating Point Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>42.06</td>
<td>41.86</td>
<td>41.44</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>31.25</td>
<td>33.25</td>
<td>31.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
<td>21.15</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
<td>8.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Pointer</td>
<td>3.00</td>
<td>4.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>4.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.06</td>
<td>4.00</td>
<td>3.00</td>
</tr>
<tr>
<td>2 X 2</td>
<td>1.50</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>4 X 4</td>
<td>1.50</td>
<td>2.00</td>
<td>1.50</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.25</td>
<td>1.25</td>
<td>1.50</td>
</tr>
<tr>
<td>Theoretical Opt.</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td>39.7</td>
<td>33.5</td>
<td>27.6</td>
</tr>
</tbody>
</table>
Parallel Unrolling: Method #2

void combine6aa(vec_ptr v, int *dest) {
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x *= (data[i] * data[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x *= data[i];
    }
    *dest = x;
}

Code Version
- Integer product

Optimization
- Multiply pairs of elements together
- And then update product
- “Tree height reduction”

Performance
- CPE = 2.5
Method #2 Computation

Computation

\[((((1 \times x_0 \times x_1) \times (x_2 \times x_3)) \times (x_4 \times x_5)) \times (x_6 \times x_7)) \times (x_8 \times x_9)) \times (x_{10} \times x_{11}))\]

Performance

- N elements, D cycles/operation
- Should be \((N/2+1)\times D\) cycles
  - CPE = 2.0
- Measured CPE worse

<table>
<thead>
<tr>
<th>Unrolling</th>
<th>CPE (measured)</th>
<th>CPE (theoretical)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.50</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>1.50</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.78</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Understanding Parallelism

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * data[i]) * data[i+1];
}

- CPE = 4.00
- All multiplies performed in sequence

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = x * (data[i] * data[i+1]);
}

- CPE = 2.50
- Multiplies overlap
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 temporaries onto stack
  - Wipes out performance gains
  - Major drawback of IA32 instruction set
- Mitigating factor: local variables are in cache
  - Not as fast as registers, but *much* faster than memory
Register Spilling Example

Example

- 8 X 8 integer product
- 7 local variables share 1 register
- See that are storing locals on stack
- E.g., at -8 (%ebp)

.L165:

```
imull (%eax),%ecx
movl -4(%ebp),%edi
imull 4(%eax),%edi
movl %edi,-4(%ebp)
movl -8(%ebp),%edi
imull 8(%eax),%edi
movl %edi,-8(%ebp)
movl -12(%ebp),%edi
imull 12(%eax),%edi
movl %edi,-12(%ebp)
movl -16(%ebp),%edi
imull 16(%eax),%edi
movl %edi,-16(%ebp)
...```

```
addl $32,%eax
addl $8,%edx
cmpl -32(%ebp),%edx
jl .L165```

- 31 -
A problem for you

Here is the main loop for a variant of `combine`

To what degree is the loop unrolled?
With what degree of parallel unrolling?
Which variable is spilled to the stack?
Why is that a good choice of which variable to spill?
How many execution unit operations does it take to execute these 12 instructions?

Draw a chart showing the scheduling of operations.

How many clock cycles does it take to complete the first iteration of the loop?

When does the second iteration of the loop begin executing?

```assembly
.L165:
  addl (%eax),%ecx
  addl 4(%eax),%esi
  addl 8(%eax),%edi
  addl 12(%eax),%ebx
  addl 16(%eax),%ecx
  addl 20(%eax),%esi
  addl 24(%eax),%edi
  addl 28(%eax),%ebx
  addl $32,%eax
  addl $8,%edx
  cmpl -32(%ebp),%edx
  jl .L165
```
### Summary: Results for Pentium III

<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></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>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>4 X 2</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>8 X 8</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td><strong>39.7</strong></td>
<td><strong>33.5</strong></td>
</tr>
<tr>
<td></td>
<td><strong>27.6</strong></td>
<td><strong>80.0</strong></td>
</tr>
</tbody>
</table>

- Biggest gain doing basic optimizations
- But, last little bit helps
Results for Alpha Processor

- Overall trends very similar for very different architectures
- Lack of registers doesn’t hurt as much as it appears, because of cache
Results for Pentium 4

<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>35.25</td>
<td>35.34</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>26.52</td>
<td>30.26</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>18.00</td>
<td>25.71</td>
</tr>
<tr>
<td>data access</td>
<td>3.39</td>
<td>31.56</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>14.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.01</td>
<td>14.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.00</td>
<td>14.00</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.02</td>
<td>7.00</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.01</td>
<td>3.98</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.63</td>
<td>4.50</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td>35.2</td>
<td>8.9</td>
</tr>
</tbody>
</table>

- Higher latencies (int * = 14, fp + = 5.0, fp * = 7.0)
  - Clock runs at 2.0 GHz
  - Not an improvement over 1.0 GHz P3 for integer *
- Avoids FP multiplication anomaly
What About Branches?

Challenge

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

- On 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 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:  xorl  %edx,%edx
80489fa:  cmpl  %esi,%edx
80489fc:  jnl   8048a25
...
```

Predict Taken

```
8048a25:  cmpl  %edi,%edx
8048a27:  jl    8048a20
8048a29:  movl  0xc(%ebp),%eax
8048a2c:  leal  0xffffffff%e8(%ebp),%esp
8048a2f:  movl  %ecx,(%eax)
```

Execute
Branch Prediction Through Loop

Assume vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Read invalid location

Executed

Fetched
### Branch Misprediction Invalidation

```asm
80488b1:  movl (%ecx,%edx,4),%eax
80488b4:  addl %eax,(%edi)
80488b6:  incl %edx
80488b7:  cmpl %esi,%edx
80488b9:  jl 80488b1

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

Assume vector length = 100

- **i = 98** Predict Taken (OK)

- **i = 99** Predict Taken (Oops)

- **i = 100** Invalidate

- **i = 101**
Branch Misprediction Recovery

- Misprediction on Pentium III wastes ~14 clock cycles
- That’s a lot of time on a high performance processor
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 achieve by hand

Exposing Instruction-Level Parallelism
- Very machine dependent
- 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