CS 201
Computer Systems Programming II – Program Optimization

Notes based on instructors notes from B&O’s web site and Robboy’s notes.
http://csapp.cs.cmu.edu

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

Loop Unrolling

- 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

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>2.00</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>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 not monotonic with degree of unrolling
  - Many subtle effects determine exact scheduling of 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

Serial Computation

- Computation
  \[
  (((((x_0 \times x_1) \times x_2) \times x_3) \times x_4) \times x_5) \times x_6) \times x_7) \times x_8) \times x_9)
  \]

- Performance
  - N elements, D cycles/operation
  - N*D cycles
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 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

Dual Product Computation

- Computation
  - \(((1 \times x_1) \times x_2) \times x_4 \times x_6 \times x_8) \times ((1 \times x_1) \times x_3 \times x_5 \times x_7 \times x_9)\)
- 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

Parallel Unrolling: Method #2

- 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
  $$(((x_0 \times x_1) \times (x_2 \times x_3)) \times (x_4 \times x_5)) \times (x_6 \times x_7)) \times (x_8 \times x_9))$$
- Performance
  - N elements, D cycles/operation
  - Should be (N/2+1)*D cycles
  - CPE = 2.0
  - Measured CPE worse

Unrolling | CPE (measured) | CPE (theoretical)
--- | --- | ---
2 | 2.50 | 2.00
3 | 1.67 | 1.33
4 | 1.50 | 1.00
6 | 1.78 | 1.00

Optimization Results for Combining

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract -g</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>data access</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Pointer</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>2 X 2</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>4 X 4</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>8 X 4</td>
<td>+</td>
<td>*</td>
</tr>
</tbody>
</table>

Worst:Best
39.7 33.5 27.6 80.0
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 they 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)
imull 8(%eax),%edi
movl %edi,-8(%ebp)
imull 12(%eax),%edi
movl %edi,-12(%ebp)
imull 16(%eax),%edi
movl %edi,-16(%ebp)
...
addl $32,%eax
addl $8,%edx
cmpl -32(%ebp),%edx
jl .L165
```

Summary: Results for Pentium III

- Biggest gain doing basic optimizations
- But, last little bit helps

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract -g</td>
<td>42.06</td>
<td>41.44</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>31.25</td>
<td>31.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.15</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>8.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.06</td>
<td>3.00</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.50</td>
<td>2.50</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.25</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.88</td>
<td>2.00</td>
</tr>
</tbody>
</table>

Worst : Best 39.7  27.6  80.0
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

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

Idea

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

Branch Prediction

Idea

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

On conditional branch, cannot reliably determine where to continue fetching
Branch Prediction Through Loop

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

Assume vector length = 100

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

Predict Taken (OK)

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

Predict Taken (Oops)

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

Read invalid location

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

Fetched

```
i = 98
```

```
i = 99
```

```
i = 100
```

Predict Taken (Oops)

```
i = 101
```

Branch Misprediction Invalidation

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

Assume vector length = 100

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

Predict Taken (OK)

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

Predict Taken (Oops)

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

Invalidate

```
i = 98
```

```
i = 99
```

```
i = 100
```

Branch Misprediction Recovery

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

Assume vector length = 100

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

Predict Taken (OK)

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

Definitely not taken

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

```
i = 98
```

```
i = 99
```

```
i = 101
```

```
i = 100
```

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

- **Performance Cost**
  - Misprediction on Pentium III wastes ~14 clock cycles
  - That's a lot of time on a high performance processor
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