15-213
“The course that gives CMU its Zip!”

Code Optimization
Sept. 25, 2003

Topics
- Machine-Independent Optimizations
- Machine Dependent Optimizations
- Code Profiling
Harsh Reality

*There's more to performance than asymptotic complexity*

Constant factors matter too!
- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
  - algorithm, data representations, procedures, and loops

Must understand system to optimize performance
- How programs are compiled and executed
- How to measure program performance and identify bottlenecks
- How to improve performance without destroying code modularity and generality
Limitations of Optimizing Compilers

Operate under fundamental constraint

- Must not cause any change in program behavior under any possible condition
- Often prevents it from making optimizations when would only affect behavior under pathological conditions.

Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles

- e.g., Data ranges may be more limited than variable types suggest

Most analysis is performed only within procedures

- Whole-program analysis is too expensive in most cases

Most analysis is based only on static information

- Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative
Machine-Independent Optimizations

Optimizations that you or compiler should do regardless of processor / compiler

**Code Motion**

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop

```c
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```
Compiler-Generated Code Motion

Most compilers do a good job with array code + simple loop structures

Code Generated by GCC

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + j] = b[j];
```

```assembly
imull %ebx, %eax          # i*n
movl 8(%ebp), %edi       # a
leal (%edi, %eax, 4), %edx # p = a+i*n (scaled by 4)
# Inner Loop
.L40:
    movl 12(%ebp), %edi    # b
    movl (%edi, %ecx, 4), %eax # b+j (scaled by 4)
    movl %eax, (%edx)      # *p = b[j]
    addl $4, %edx          # p++ (scaled by 4)
    incl %ecx              # j++
    jl .L40                # loop if j<n
```
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \rightarrow x \ll 4 \]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles

- Recognize sequence of products

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i, j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;

int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i-1)*n, (i+1)*n
1 multiplication: i*n

```
leal -1(%edx),%ecx # i-1
imull %ebx,%ecx     # (i-1)*n
leal 1(%edx),%eax   # i+1
imull %ebx,%eax     # (i+1)*n
imull %ebx,%edx     # i*n
```
Time Scales

Absolute Time
- Typically use nanoseconds
  - $10^{-9}$ seconds
- Time scale of computer instructions

Clock Cycles
- Most computers controlled by high frequency clock signal
- Typical Range
  - 100 MHz
    - $10^8$ cycles per second
    - Clock period = 10ns
  - 2 GHz
    - $2 \times 10^9$ cycles per second
    - Clock period = 0.5ns
- Fish machines: 550 MHz (1.8 ns clock period)
Cycles Per Element

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- \( T = \text{CPE} \times n + \text{Overhead} \)
Vector Abstract Data Type (ADT)

Procedures

vec_ptr new_vec(int len)

- Create vector of specified length

int get_vec_element(vec_ptr v, int index, int *dest)

- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

int *get_vec_start(vec_ptr v)

- Return pointer to start of vector data

- Similar to array implementations in Pascal, ML, Java
  - E.g., always do bounds checking
 Optimization Example

Procedure

- Compute sum of all elements of integer vector
- Store result at destination location
- Vector data structure and operations defined via abstract data type

Pentium II/III Performance: Clock Cycles / Element

- 42.06 (Compiled -g) 31.25 (Compiled -O2)

```
void combine1(vec_ptr v, int *dest)
{
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))
        goto done;

    loop:
        get_vec_element(v, i, &val);
        *dest += val;
        i++;
        if (i < vec_length(v))
            goto loop
    done:
}

Inefficiency

- Procedure vec_length called every iteration
- Even though result always the same
Move `vec_length` Call Out of Loop

```c
void combine2(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

**Optimization**

- Move call to `vec_length` out of inner loop
  - Value does not change from one iteration to next
  - Code motion
- CPE: 20.66 (Compiled -O2)
  - `vec_length` requires only constant time, but significant overhead
Optimization Blocker: Procedure Calls

*Why couldn’t compiler move `vec_len` out of inner loop?*

- Procedure may have side effects
  - Alters global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure lower could interact with `strlen`

*Why doesn’t compiler look at code for `vec_len`?*

- Interprocedural optimization is not used extensively due to cost

**Warning:**

- Compiler treats procedure call as a black box
- Weak optimizations in and around them
Reduction in Strength

```c
void combine3(vec_ptr v, int *dest) {
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}
```

Optimization

- Avoid procedure call to retrieve each vector element
  - Get pointer to start of array before loop
  - Within loop just do pointer reference
  - Not as clean in terms of data abstraction

- CPE: 6.00 (Compiled -O2)
  - Procedure calls are expensive!
  - Bounds checking is expensive
Eliminate Unneeded MemoryRefs

**Optimization**

- Don’t need to store in destination until end
- Local variable `sum` held in register
- Avoids 1 memory read, 1 memory write per cycle
- CPE: 2.00 (Compiled -O2)
  - Memory references are expensive!

```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;
}
```
## Detecting Unneeded MemoryRefs.

### Combine 3

```
.L18:
    movl (%ecx,%edx,4),%eax
    addl %eax, (%edi)
    incl %edx
    cmpl %esi,%edx
    jl .L18
```

### Combine 4

```
.L24:
    addl (%eax,%edx,4),%ecx
    incl %edx
    cmpl %esi,%edx
    jl .L24
```

## Performance

- **Combine 3**
  - 5 instructions in 6 clock cycles
  - `addl` must read and write memory

- **Combine 4**
  - 4 instructions in 2 clock cycles
Optimization Blocker: Memory Aliasing

Aliasing
- Two different memory references specify single location

Example
- v: [3, 2, 17]
- combine3(v, get_vec_start(v)+2) --> ?
- combine4(v, get_vec_start(v)+2) --> ?

Observations
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing
**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
Machine-Independent Opt. Summary

**Code Motion**
- Compilers are good at this for simple loop/array structures
- Don’t do well in presence of procedure calls and memory aliasing

**Reduction in Strength**
- Shift, add instead of multiply or divide
  - *compilers are (generally) good at this*
  - *Exact trade-offs machine-dependent*
- Keep data in registers rather than memory
  - *compilers are not good at this, since concerned with aliasing*

**Share Common Subexpressions**
- *compilers have limited algebraic reasoning capabilities*
Modern CPU Design

Instruction Control

Retirement Unit
Register File

Fetch Control
Instruction Decode

Instruction Cache

Execution

Register Updates

Prediction OK?

Operation Results

Instruction Decoded Operations

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

Functional Units

Data Cache

Addr.
Data
Addr.
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 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
  - Based on current PC + predicted targets for predicted branches
  - 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
  - Abstract identifier linking destination of one operation with sources of later operations
Translation Example

Version of Combine4

- Integer data, multiply operation

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

Translation of First Iteration

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

load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0 ➔ %ecx.1
incl %edx.0 ➔ %edx.1
cmpl %esi, %edx.1 ➔ cc.1
jl-taken cc.1
```
Translation Example #1

Split into two operations
- `load` reads from memory to generate temporary result `t.1`
- Multiply operation just operates on registers

Operands
- Register `%eax` does not change in loop. Values will be retrieved from register file during decoding
- Register `%ecx` changes on every iteration. Uniquely identify different versions as `%ecx.0`, `%ecx.1`, `%ecx.2`, ...
  - Register *renaming*
  - Values passed directly from producer to consumers
Translation Example #2

- Register `%edx` changes on each iteration. Rename as `%edx.0`, `%edx.1`, `%edx.2`, ...

```
incl %edx
```

```
incl %edx.0 ➜ %edx.1
```
Translation Example #3

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer
Instruction control unit determines destination of jump
Predicts whether will be taken and target
Starts fetching instruction at predicted destination
Execution unit simply checks whether or not prediction was OK
If not, it signals instruction control
  • Instruction control then “invalidates” any operations generated from misfetched instructions
  • Begins fetching and decoding instructions at correct target
Visualizing Operations

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

Operations

- Arrows show only for operands that are passed within execution unit

Operands

- 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

- 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

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

\[
\begin{align*}
\text{load} &\quad (%eax,%edx.0,4) \rightarrow t.1a \\
\text{iadd} &\quad t.1a, %ecx.0 \rightarrow %ecx.1a \\
\text{load} &\quad 4(%eax,%edx.0,4) \rightarrow t.1b \\
\text{iadd} &\quad t.1b, %ecx.1a \rightarrow %ecx.1b \\
\text{load} &\quad 8(%eax,%edx.0,4) \rightarrow t.1c \\
\text{iadd} &\quad t.1c, %ecx.1b \rightarrow %ecx.1c \\
\text{iadd} &\quad $3,%edx.0 \rightarrow %edx.1 \\
\text{cmpl} &\quad %esi, %edx.1 \rightarrow cc.1 \\
\text{jl} \quad &\text{taken cc.1}
\end{align*}
\]
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</td>
<td>Sum</td>
<td>2.00</td>
<td>1.50</td>
<td>1.33</td>
<td>1.50</td>
<td>1.25</td>
</tr>
<tr>
<td>Integer Product</td>
<td></td>
<td>4.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td>Sum</td>
<td></td>
<td>3.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td>Product</td>
<td></td>
<td>5.00</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

Code Version
- Integer product

Optimization
- Accumulate in two different products
  - Can be performed simultaneously
- Combine at end
- 2-way parallelism

Performance
- CPE = 2.0
- 2X performance

```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;
}
```
Dual Product Computation

Computation

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

Performance

- N elements, D cycles/operation
- \((N/2 + 1) \times D\) 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
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
```
Executing with Parallel Loop

- Predicted Performance
  - Can keep 4-cycle multiplier busy performing two simultaneous multiplications
  - Gives CPE of 2.0
### 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>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 temporaries onto stack
  - Wipes out any performance gains
- Not helped by renaming
  - Cannot reference more operands than instruction set allows
  - Major drawback of IA32 instruction set
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:

```assembly
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
```
### 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>47.14</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>25.08</td>
<td>36.05</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>19.19</td>
<td>32.18</td>
</tr>
<tr>
<td>data access</td>
<td>6.26</td>
<td>12.52</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>1.76</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.51</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.25</td>
<td>9.01</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.19</td>
<td>4.69</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.15</td>
<td>4.12</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.11</td>
<td>4.24</td>
</tr>
<tr>
<td>Worst : Best</td>
<td>36.2</td>
<td>11.4</td>
</tr>
</tbody>
</table>

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler.
### Results for Pentium 4 Processor

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>35.25</td>
<td>35.34</td>
<td>35.85</td>
<td>38.00</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>26.52</td>
<td>30.26</td>
<td>31.55</td>
<td>32.00</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>18.00</td>
<td>25.71</td>
<td>23.36</td>
<td>24.25</td>
</tr>
<tr>
<td>data access</td>
<td>3.39</td>
<td>31.56</td>
<td>27.50</td>
<td>28.35</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.01</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.00</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.02</td>
<td>7.00</td>
<td>2.63</td>
<td>3.50</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.01</td>
<td>3.98</td>
<td>1.82</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.63</td>
<td>4.50</td>
<td>2.42</td>
<td>2.31</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td>35.2</td>
<td><strong>8.9</strong></td>
<td><strong>19.7</strong></td>
<td><strong>19.0</strong></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
Machine-Dependent Opt. Summary

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

Exposing Instruction-Level Parallelism
- Generally helps, but extent of improvement is 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
Important Tools

Observation

- Generating assembly code
  - Lets you see what optimizations compiler can make
  - Understand capabilities/limitations of particular compiler

Measurement

- Accurately compute time taken by code
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
    » Chapter 9 of the CS:APP textbook
- Profile procedure calling frequencies
  - Unix tool gprof
Code Profiling Example

Task
- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

Steps
- Convert strings to lowercase
- Apply hash function
- Read words and insert into hash table
  - Mostly list operations
  - Maintain counter for each unique word
- Sort results

Data Set
- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

Shakespeare’s most frequent words

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>29,801</td>
</tr>
<tr>
<td>and</td>
<td>27,529</td>
</tr>
<tr>
<td>I</td>
<td>21,029</td>
</tr>
<tr>
<td>to</td>
<td>20,957</td>
</tr>
<tr>
<td>of</td>
<td>18,514</td>
</tr>
<tr>
<td>a</td>
<td>15,370</td>
</tr>
<tr>
<td>you</td>
<td>14010</td>
</tr>
<tr>
<td>my</td>
<td>12,936</td>
</tr>
<tr>
<td>in</td>
<td>11,722</td>
</tr>
<tr>
<td>that</td>
<td>11,519</td>
</tr>
</tbody>
</table>
Code Profiling

Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Time computation method
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment its timer by interval (e.g., 10ms)
- Also maintains counter for each function indicating number of times called

Using

```
gcc -O2 -pg prog. -o prog
./prog
  - Executes in normal fashion, but also generates file gmon.out

gprof prog
  - Generates profile information based on gmon.out
```
Profiling Results

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>1</td>
<td>8210.00</td>
<td>8210.00</td>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>4.75</td>
<td>9.21</td>
<td>0.45</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Call Statistics

- Number of calls and cumulative time for each function

Performance Limiter

- Using inefficient sorting algorithm
- Single call uses 87% of CPU time
First step: Use more efficient sorting function
Library function `qsort`
Further Optimizations

- **Iter first**: Use iterative function to insert elements into linked list
  - Causes code to slow down
- **Iter last**: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- **Big table**: Increase number of hash buckets
- **Better hash**: Use more sophisticated hash function
- **Linear lower**: Move `strlen` out of loop
Profiling Observations

Benefits
- Helps identify performance bottlenecks
- Especially useful when have complex system with many components

Limitations
- Only shows performance for data tested
- E.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds
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