15-213
"The course that gives CMU its Zip!"

Code Optimization:
Machine Independent Optimizations
Feb 12, 2004

Topics
- Machine-Independent Optimizations
- Machine-Dependent_opts
- Understanding Processor Operation
- Branches and Branch Prediction
- Tuning

Great Reality #4
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

Optimizing Compilers
Provide efficient mapping of program to machine
- register allocation
- code selection and ordering
- eliminating minor inefficiencies

Don't (usually) improve asymptotic efficiency
- up to programmer to select best overall algorithm
- big-O savings are (often) more important than constant factors
  - but constant factors also matter

Have difficulty overcoming "optimization blockers"
- potential memory aliasing
- potential procedure side-effects

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

The Bottom Line:
When in doubt, do nothing
i.e., The compiler must be conservative

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
Machine-Independent Optimizations

- Optimizations that should be done 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

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

Compiler-Generated Code Motion

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

**Code Generated by GCC**

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

Strength Reduction†

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x → x << 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 (induction var analysis)

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

Make Use of Registers

- Reading and writing registers much faster than reading/writing memory

**Limitation**

- Limited number of registers
- Compiler cannot always determine whether variable can be held in register
- Possibility of **Aliasing**
- See example later

---

†As a result of Induction Variable Elimination
Machine-Independent Opt. (Cont.)

Share Common Subexpressions†

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

/* 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 multiplies: i*n, (i-1)*n, (i+1)*n
1 multiply: i*n

†AKA: Common Subexpression Elimination (CSE)

Measuring Performance: 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
  - 2 GHz
  - 10⁹ cycles per second
  - 2 X 10⁹ cycles per second
  - Clock period = 10ns
  - Clock period = 0.5ns

- Fish machines: 550 MHz (1.8 ns clock period)

Measuring Performance

For many programs, cycles per element (CPE)

- Especially true of programs that work on lists/vectors
- Total time = fixed overhead + CPE * length-of-list

```c
void vsuml(int n) {
    int i;
    for (i = 0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

```c
void vsum2(int n) {
    int i;
    for (i = 0; i<n; i+=2)
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
}
```

- vsum2 only works on even n.
- vsum2 is an example of loop unrolling.

Cycles Per Element

- Convenient way to express performance of a program
  that operates on vectors or lists
- Length = n
- \( T = \text{CPE} \cdot n + \text{Overhead} \)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
</tr>
</tbody>
</table>

- Slope = 4.0
- Slope = 3.5
Vector ADT

<table>
<thead>
<tr>
<th>length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>length-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- `int vec_length(vec_ptr v)`
  - Return length of vector
- Similar to array implementations in Pascal, ML, Java
- E.g., always do bounds checking

Optimization Example

```c
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;
  }
}
```

Procedure

- Compute sum of all elements of vector
- Store result at destination location

Understanding Loop

```c
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

Pentium II/III Perf: Clock Cycles / Element
- 42.06 (Compiled -g) 31.25 (Compiled -O2)
**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

**Code Motion Example #2**

**Procedure to Convert String to Lower Case**

```c
void lower(char *s) {
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Extracted from 213 lab submissions, Fall, 1998

**Lower Case Conversion Performance**

- Time quadruples when double string length
- Quadratic performance of lower

**Convert Loop To Goto Form**

```c
void lower(char *s) {
    int i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
        i++;
        if (i < strlen(s))
            goto loop;
    done:
}
```

- `strlen` executed every iteration
- `strlen` linear in length of string
  - Must scan string until finds '\0'
- Overall performance is quadratic
### Improving Performance

```c
void lower(char *s) {
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

* Move call to `strlen` outside of loop
* Since result does not change from one iteration to another
* Form of code motion

### Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of `lower2`

![Graph showing performance comparison between lower1 and lower2](image)

### Optimization Blocker: Procedure Calls

**Why doesn’t the compiler move `vec_len` or `strlen` out of the inner loop?**

- Procedure may have side effects
  - Can alter global state each time called
- Function may return diff value for same arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`
- GCC has an extension for this:
  - `int square (int) __attribute__ ((const));`
  - Check out info.

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

**Optimization Blocker: Procedure Calls**

Why doesn't the compiler move `vec_len` or `strlen` out of the inner loop?
- Procedure may have side effects
- Function may return different values for the same arguments

Why doesn't the compiler look at code for `vec_len` or `strlen`?
- Linker may overload with different version
- Usually declared static
- 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**

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

**Aside: Rational for Classes**
- 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

---

**What next?**

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

---

**Eliminate Unneeded Memory Refs**

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

**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!
Detecting Unneeded Memory Refs.

Combine3
\[.L18: \]
movl (%ecx,%edx,4),%eax
addl %eax,(%edi)
incl %edx
cmpl %esi,%edx
jl .L18

Combine4
\[.L24: \]
addl (%eax,%edx,4),%ecx
incl %edx
cmpl %esi,%edx
jl .L24

Performance
- Combine3
  - 5 instructions in 6 clock cycles
  - addl must read and write memory
- Combine4
  - 4 instructions in 2 clock cycles

Optimization Blocker: Memory Aliasing

Aliasing
- Two different memory references specify one location

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

Observations
- Can easily 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

Machine-Independent Opt. Summary

Code Motion/Loop Invariant Code Motion
- Compilers good if for simple loop/array structures
- Bad in presence of procedure calls and memory aliasing

Strength Reduction/Induction Var Elimination
- 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/CSE
- compilers have limited algebraic reasoning capabilities

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
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>41.44</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>
</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;
}

Array Code

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

Pointer Code

.L30: # Loop:
    addl (%eax),%ecx # sum += *data
    addl $4,%eax # data +=
    cmpl %edx,%eax # data:dend
    jbe .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

Translation Example

CPU Capabilities of Pentium III

Translation of First Iteration

Instruction Control

Version of Combine4

Grabs Instruction Bytes From Memory

Translation Example

- 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

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

- Instruction

- Latency

- Cycles/Issue

- Load / Store

- 3

- 1

- Integer Multiply

- 4

- 1

- Integer Divide

- 36

- 36

- Double/Single FP Multiply

- 5

- 2

- Double/Single FP Add

- 3

- 1

- Double/Single FP Divide

- 38

- 38

Translation of First Iteration

Load (%eax,%edx,4) ➔ t.1

imull (%eax,%edx,4),%ecx ➔ data[i] incl %edx ➔ i++

cmpl %esi,%edx ➔ c.1

cc.1

jl .L24 ➔ if < goto Loop

Translation Example

Version of Combine4

- Integer data, multiply operation

.L24:

imull (%eax,%edx,4),%ecx ➔ t *= data[i] incl %edx ➔ i++

cmpl %esi,%edx ➔ c.1

cc.1

jl .L24 ➔ if < goto Loop
Translation Example #1

- Split into two operations
  - *load* reads from memory to generate temporary result \( t.1 \)
  - Multiply operation just operates on registers
- Operands
  - Registers \( \%eax \) 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 \( \%ecx.0, \%ecx.1, \%ecx.2, \ldots \)
    - Register *renaming*
    - Values passed directly from producer to consumers

```
imull (%eax, %edx, 4), %ecx
load (%eax, %edx.0, 4) → t.1
imull t.1, %ecx.0 → %ecx.1
```

Translation Example #2

- Register \( \%edx \) changes on each iteration. Rename as \( \%edx.0, \%edx.1, \%edx.2, \ldots \)

```
imul %edx
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

```
cmpl %esi, %edx
```

Translation Example #4

- 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

```
jl .L24
jltaken cc.1
```
Visualizing Operations

Operations

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

Operands

- Arrows 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;
}
```

Optimization
- Combine multiple iterations into single loop body
- 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

```
load (%eax,%edx.0,4)  t.1a
iaddi t.1.a, %ecx.0c  vecx.1a
load 4(%eax,%edx.0,4)  t.1b
iaddi t.1.b, %ecx.1a  vecx.1b
load 8(%eax,%edx.0,4)  t.1c
iaddi t.1.c, %ecx.1b  vecx.1c
iaddi $3,%edx.0      vecx.4
cmp4 %esi, %edx.0    vecx.5
jlt-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

<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 nonlinear with degree of unrolling
- Many subtle effects determine exact scheduling of operations

Serial Computation

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

Performance
- \( N \) elements, \( D \) cycles/operation
- \( N \times D \) cycles

Parallel Loop Unrolling

```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];
        x1 *= data[i+1];
    }
    /* Combine at end */
    *dest = x0 * x1;
}
```

Code Version
- Integer product

Optimization
- Accumulate in two different products
- Can be performed simultaneously

Performance
- CPE = 2.0
- 2X performance

Dual Product Computation

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

Performance
- \( N \) elements, \( D \) cycles/operation
- \((N/2+1)*D\) 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

Visualizing Parallel Loop

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

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>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>Pointer</td>
<td>3.00</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>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>1.50</td>
</tr>
<tr>
<td>Theoretical Opt.</td>
<td>1.00</td>
<td>2.00</td>
</tr>
<tr>
<td>Worst : Best</td>
<td>39.7</td>
<td>27.6</td>
</tr>
</tbody>
</table>

Worst : Best 39.7 27.6 20.0
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

\[
(((1 \cdot (x0 \cdot x1)) \cdot (x2 \cdot x3)) \cdot (x4 \cdot x5)) \cdot (x6 \cdot x7)) \cdot (x8 \cdot x9)) \cdot (x10 \cdot x11))
\]

Performance
- N elements, D cycles/operation
- Should be \((N/2+1)\cdot D\) 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 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
• Notice: locals are stored on the stack
• E.g., at −8 (ebp)

.l165:
   imull (%eax),%ecx
   movl %edi,-4(%ebp)
   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>25.06</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

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>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>Worst : Best</td>
<td>39.7</td>
<td>33.5</td>
</tr>
</tbody>
</table>

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

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>25.00</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>Worst : Best</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

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

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

Branch Prediction

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

Branch Prediction Through Loop

Assume vector length = 100

Predict Taken (OK)

Read invalid location

Predict Taken (Oops)

Executed

Fetched
Branch Misprediction Invalidation

Assume vector length = 100

Predict Taken (OK)

i = 98

Predict Taken (Oops)

I nvalidate

i = 99

i = 100

% 73 -

Branch Misprediction Recovery

Assume vector length = 100

Predict Taken (OK)

Definitely not taken

Invalidate

Assume vector length = 100

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

Avoiding Branches

On Modern Processor, Branches Very Expensive
- Unless prediction can be reliable
- When possible, best to avoid altogether

Example
- Compute maximum of two values
  - 14 cycles when prediction correct
  - 29 cycles when incorrect

```c
int max(int x, int y) {
    return (x < y) ? y : x;
}
```

Avoiding Branches with Bit Tricks

- In style of Lab #1
- Use masking rather than conditionals

```c
int bmax(int x, int y) {
    int mask = -(x>y);
    return (mask & x) | (~mask & y);
}
```

- Compiler still uses conditional
  - 16 cycles when predict correctly
  - 32 cycles when mispredict

```c
xorl %edx,%edx # mask = 0
movl 8(%ebp),%eax
movl 12(%ebp),%ecx
cmp1 %ecx,%eax
jle L13 # skip if x<=y
movl $-1,%edx # mask = -1
L13:
```
Avoiding Branches with Bit Tricks

- Force compiler to generate desired code

```c
int bvmax(int x, int y) {
    volatile int t = (x>y);
    int mask = ~t;
    return (mask & x) | (~mask & y);
}
```

- volatile declaration forces value to be written to memory
  - Compiler must therefore generate code to compute t
  - Simplest way is setg/movzbl combination
- Not very elegant!
  - A hack to get control over compiler
  - 22 clock cycles on all data
  - Better than misprediction

Conditional Move

- Added with P6 microarchitecture (PentiumPro onward)
- `cmovXXl %edx, %eax`
  - If condition XX holds, copy %edx to %eax
  - Doesn’t involve any branching
  - Handled as operation within Execution Unit

```assembly
movl 8(%ebp),%ecx # Get x
movl 12(%ebp),%edx # Get y
cmpl %edx,%ecx # x:y
setg %al # (x>y)
movb %al,%eax # Zero extend
movl %eax,-4(%ebp) # Save as t
movl -4(%ebp),%eax # Retrieve t
```

- Current version of GCC won’t use this instruction
  - Thinks it’s compiling for a 386
  - Performance
  - 14 cycles on all data

Machine-Dependent Opt. Summary

- Look carefully at generated code to see whether helpful
- Look Unrolling
  - Some compilers do this automatically
  - Generally not as clever as what can achieve 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

Important Tools

Measurement

- Accurately compute time taken by code
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
- Profile procedure calling frequencies
  - Unix tool `gprof`

Observation

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

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

**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

---

**Profiling Results**

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>time</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</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>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</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>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</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

---

**Code Profiling**

**Add information gathering to executable**
- Computes (approximate) 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 collect number of times each function is called

**Using**

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

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

---

**Code Optimizations**

What should we do?
**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

**Further Optimizations**
- Iter first: Use iterative func to insert elmts into linked list
- Iter last: Iterative func, places new entry at end of list
- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function
- Linear lower: Move `strlen` out of loop
How Much Effort Should we Expends?

Amdahl's Law:
Overall performance improvement is a combination
- How much we spend up a piece of the system
- How important that piece is!
Example, suppose Chose to optimize "rest" & you succeed!
It goes to ZERO seconds!
Amdahl's Law
- Total time = (1−α)T + αT
- Component optimizing takes αT time.
- Improvement is factor of k, then:
- \(T_{\text{new}} = T_{\text{old}}(1-\alpha) + \alpha/k\)
- Speedup = \(T_{\text{old}}/T_{\text{new}} = 1/[(1-\alpha) + \alpha/k]\)
- Maximum Achievable Speedup (k = \(\infty\)) = 1/(1−α)

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 (AKA: Profile first!)
  - Do detailed optimizations where code will be executed repeatedly
  - Will get most performance gain here

A Stack Based Optimization

```
_fib:
pushl %ebp
movl %esp,%ebp  L3:  movl $1,%eax
subl $16,%esp  L5:  leal -24(%ebp),%esp
pushl %esi
pushl %ebx
movl $1(%ebp),%eax
cmpi $1,%ebx
jle L3
addl $-12,%esp
leal -1(%ebx),%eax
pushl %eax
leal -1(%ebp),%eax
pushl %eax
call _fib
movl %eax,%esi
int fib(int n)
addl $-12,%esp
leal -2(%ebx),%eax
if (n <= 1) return 1;
return fib(n-1)+fib(n-2);
call _fib
addl %esi,%eax
jmp L5
```

```
.int fib(int n) {
if (n <= 1) return 1;
return fib(n-1)+fib(n-2);
}
```