Code Optimization II: Machine Independent Optimizations

Topics

- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing

- Tuning
  - Identifying performance bottlenecks
Vector 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

```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 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)
Reduction in Strength

void combine2(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 Memory Refs

```c
void combine3(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;
}
```

### 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.

**Combine2**

.L18:

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

**Combine3**

.L24:

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

**Performance**

- **Combine2**
  - 5 instructions in 6 clock cycles
  - `addl` must read and write memory
- **Combine3**
  - 4 instructions in 2 clock cycles
Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location

Example

- \( v = [3, 2, 17] \)
- \( \text{combine2}(v, \text{get_vec_start}(v)+2) \rightarrow ? \)
- \( \text{combine3}(v, \text{get_vec_start}(v)+2) \rightarrow ? \)

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
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

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

Operations
- Use different definitions of OP and IDENT
  - + / 0
  - * / 1

```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;
}
```
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
Chapter 11: Pointer Code

**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

```c
void combine4p(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data + length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```
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:den
    jb .L30 # if < goto Loop

Performance

- **Array Code**: 4 instructions in 2 clock cycles
- **Pointer Code**: Almost same 4 instructions in 3 clock cycles
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*
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 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.c -o prog
./prog
  ● Executes in normal fashion, but also generates file gmon.out

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

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 Optimizations

- 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
Summary

Today

- Optimization blocker: procedure calls
- Optimization blocker: memory aliasing
- Tools (profiling) for understanding performance

Next time

- Memory system optimization