Optimizing Program Performance

CMPU 224 – Computer Organization
Jason Waterman
Performance Realities

• There’s more to performance than asymptotic complexity (big-O notation)
  
• Constant factors matter too!
  • Easily see 10x 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 modern processors + memory systems operate
  • 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 (scheduling)
  • Dead code elimination
  • Eliminating minor inefficiencies

• Have difficulty overcoming “optimization blockers”
  • Potential memory aliasing
  • Potential procedure side-effects

• 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
Limitations of Optimizing Compilers

• Operate under fundamental constraint
  • Must not cause any change in program behavior

• 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
  • Newer versions of GCC do inter-procedural analysis within individual files
    • But, not between code in different files

• Most analysis is based only on static information
  • Compiler has difficulty anticipating run-time inputs

• When in doubt, the compiler must be conservative
Memory Matters

- Both functions seem to have identical behavior
- Both add twice the value at the location yp to the location xp
- sum2 has less memory accesses
- However, the compiler won’t replace sum2 for sum1
- What if xp and yp point to the same memory location?
  - sum1 and sum2 will give different answers

```c
void sum1(long *xp, long *yp) {
    *xp += *yp;
    *xp += *yp;
}

void sum2(long *xp, long *yp) {
    *xp += 2 * *yp;
}
```

```assembly
sum1:
    movq (%rdi), %rax
    addq (%rsi), %rax
    movq %rax, (%rdi)
    addq (%rsi), %rax
    movq %rax, (%rdi)
    ret

sum2:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret
```
When \( xp == yp \)

- \( \text{sum1} \) will increase value at \( xp \) by a factor of 4
- \( \text{sum2} \) will increase value at \( xp \) by a factor of 3

- Compiler must assume that different pointers may point to the same memory location

- Limits the set of possible optimizations

```c
void sum1(long *xp, long *xp) {
    *xp += *xp;
    *xp += *xp;
}

void sum2(long *xp, long *xp) {
    *xp += 2 * *xp;
}
```
Optimization Blocker #1: Memory Aliasing

• Aliasing
  • Two different memory references specify single location
  • Easy to have happen in C
    • Since allowed to do address arithmetic
    • Direct access to storage structures

• How to work around:
  • Get in habit of introducing local variables
    • Accumulating within loops
    • Your way of telling compiler not to check for aliasing

```c
void sum2(long *xp, long *yp)
    *xp += 2 * *yp;
}

void sum3(long *xp, long *yp) {
    long sum;
    sum += *yp;
    sum += *yp;
    *xp += sum;
}
```

```assembly
sum2:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret

sum3:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret
```
Optimization Blocker #2: Procedure Calls

- Procedure to convert string to lower case

```c
#include <string.h>
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time quadruples when string length doubles
- $O(N^2)$
Convert Loop To Goto Form

- `strlen` executed every iteration

```c
#include <string.h>

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

```c
#include <string.h>

void lower(char *s)
{
    size_t 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:
}
```
Calling strlen

- **strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character

- **Overall performance for string of length N:**
  - N calls to `strlen`
  - Each call to `strlen` iterates over all N characters in the string
  - Overall O(N^2) performance

```c
/* My version of strlen */
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```
Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another

```c
#include <string.h>

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

- Time doubles when double string length
- Linear performance of lower2
Optimization Blocker: Procedure Calls

• Why couldn’t compiler move `strlen` out of inner loop?
  • Procedure may have side effects
    • May alter some global state each time it is called
  • Function may not return same value for the same given arguments
    • Depends on other parts of global state
    • Procedure `lower` could interact with `strlen`

• Warning:
  • Compiler treats a procedure call as a black box

• Remedies:
  • Move code if it is safe to do
  • Use of inline functions
    • GCC does this with `–O1`
      • Within single file
Generally Useful Optimizations

• Optimizations that you or the compiler should do regardless of processor or compiler

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

```c
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```
Reduction in Strength

• Replace costly operation with simpler one
• Shift, add instead of multiply or divide
  • $16 \times x \rightarrow x \ll 4$
• Machine dependent
• Depends on cost of multiply or divide instruction
  • On modern Intel processors, integer multiply requires 3 CPU cycles
• Recognize sequence of products

```c
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```
```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
- GCC will do this with –O1

```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;
```

3 multiplications: \(i\cdot n\), \((i-1)\cdot n\), \((i+1)\cdot n\)

```assembly
leaq 1(%rsi), %rax  # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi  # i\cdot n
imulq %rcx, %rax  # (i+1)\cdot n
imulq %rcx, %r8   # (i-1)\cdot n
addq %rdx, %rsi  # i\cdot n+j
addq %rdx, %rax  # (i+1)\cdot n+j
addq %rdx, %r8   # (i-1)\cdot n+j
```

1 multiplication: \(i\cdot n\)

```assembly
imulq %rcx, %rsi  # i\cdot n
addq %rdx, %rsi  # i\cdot n+j
addq %rdx, %rax  # (i+1)\cdot n+j
addq %rdx, %r8   # (i-1)\cdot n+j
```

```assembly
movq  %rsi, %rax  # i\cdot n+j
subq  %rcx, %rax  # i\cdot n+j-n
```

```assembly
leaq (%rsi,%rcx), %rcx # i\cdot n+j+n
```
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;

/* retrieve vector element and store at val */
/* return 1 if successful, 0 otherwise */
int get_vec_element(*vec v, size_t idx, data_t *val) {
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

• Data Types
  • Use different declarations for data_t
    • int
    • long
    • float
    • double
Benchmark Computation

void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}

• Data Types
  • Use different declarations for data_t
  • int
  • long
  • float
  • double

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

Compute sum or product of vector elements
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line

![Graph of Cycles vs. Elements with two lines: psum1 with slope 9.0 and psum2 with slope 6.0.](image)
void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}

Compute sum or product of vector elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
</tbody>
</table>
Eliminating Loop Inefficiencies

- Move call to `vec_length` outside of loop
- Code motion

```c
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine2</td>
<td>7.02</td>
<td>9.03</td>
</tr>
</tbody>
</table>
Reducing Procedure Calls

- `get_vec_start` returns the starting address of the data array
  - This breaks the abstraction barrier!
- Remove call to `get_vec_element`
- We are no longer doing bounds checking
- No effect on performance
  - Other operations are the bottleneck
- Will help us later when we remove these bottlenecks

```c
/* Direct access to vector data*/
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine2</td>
<td>7.02</td>
<td>9.03</td>
</tr>
<tr>
<td>Combine3</td>
<td>7.17</td>
<td>9.02</td>
</tr>
</tbody>
</table>
Eliminating Unneeded Memory References

- Disassembly of inner loop for integer addition
- One memory read and one memory write per cycle
- But *dest doesn’t need to be updated till the end of the loop

```c
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

.L3: # i in %rdx, length in %rbp, data in %rax
    cmpq %rbp, %rdx     # i:length
    jge .L1             # jump to end of loop
    movq (%rax,%rdx,8), %rcx   # read data[i]
    addq %rcx, (%rbx)     # write *dest
    addq $1, %rdx         # i++
    jmp .L3
.L1:
```
Accumulate result in temporary

- Reduce to only one memory read per element using a temporary variable

```c
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

L3:  # i in %rdx, data in %rax
cmpq %rbp, %rdx     # i:length
jge .L1            # jump to end of loop
movq (%rax,%rdx,8), %rcx # read data[i]
addq %rcx, (%rbx)  # write *dest
addq $1, %rdx      # i++
jmp .L3
.L1:
```
Accumulate result in temporary

• Why doesn’t the compiler perform this transformation automatically?
  • Memory aliasing
  • The dest could be one of the elements in the vector (e.g., the last element)
    • combine3 and combine4 could have different results in this case
  • Using a temporary register tells the compiler not to check for memory aliasing

```c
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}

void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```
Effect of Basic Optimizations

- 4x to 18x improvement over original unoptimized code
- To seek better performance, we must consider optimizations that exploit the microarchitecture of the processor
  - Code tuned for a specific processor
- We’ll tackle this next class

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 Unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine3</td>
<td>7.17</td>
<td>9.02</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>