Program Optimization

Fall 2015

Yusung Kim
yskim525@gmail.com
Performance Realities

- *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 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

- 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

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

- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural 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
Performance Metrics

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

- **Clock Cycles**
  - Measurements are independent of actual processors used
  - More focused on codes executed
  - 100 MHz
    - $10^8$ cycles per second, Clock period = 10ns
  - 2 GHz
    - $2 \times 10^9$ cycles per second, Clock period = 0.5ns
Generally Useful Optimizations

- Optimizations that you or the 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
Strength Reduction

- 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 Intel Nehalem, 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*n, (i−1)*n, (i+1)*n

```c
long 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;
```

1 multiplication: i*n
Optimization Blocker #1: Procedure Calls

```c
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');
}
```

Procedure to Convert String to Lower Case

```c
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
Calling strlen

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

- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - 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

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');
}

- 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
Side Effects in Procedure Calls

```c
int f();

int func1() {
    return f() + f() + f() + f();
}

int func2() {
    return 4*f();
}
```

```c
int counter = 0;

int f() {
    return counter++;
}
```
Optimization Blocker: Procedure Calls

Why couldn’t compiler move `strlen` 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`

Warning:

- Compiler treats procedure call as a black box
- Weak optimizations near them

Remedies:

- Use of inline functions
  - GCC does this with `-O1`
  - (Within single file)
- Do your own code motion

```c
size_t lencnt = 0;
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```
Optimization Blockers #2: Memory Aliasing

```c
void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2* *yp;
}
```
Memory 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

- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing
Benchmark Example: Vectors

```c
/* data structure for vectors */
typedef struct{
    size_t len;
    int *data;
} vec;

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

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

```c
void combine1(vec *v, int *sum)
{
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *sum += val;
    }
}
```

- Compute sum of vector elements
- Procedure vec_length() called every iteration
Benchmark Performance

```c
void combine1(vec* v, int* sum)
{
    int i;
    *sum = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *sum += val;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>run times (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No-op</td>
</tr>
<tr>
<td>Combine1</td>
<td>1.282</td>
</tr>
</tbody>
</table>
Eliminating Loop Inefficiencies

- Move vec_length out of loop
- vec_length does not change for all iterations

```c
void combine2(vec* v, int* sum) {
    int i;
    int length = vec_length(v);
    *sum = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *sum += val;
    }
}
```
Reduce procedure calls

- Avoid procedure calls to retrieve each element
  - Get pointer to start of array before loop
  - Within loop just do pointer reference

```c
void combine3(vec* v, int* sum) {
    int i;
    int length = vec_length(v);
    int* data = get_vec_start(v);
    *sum = 0;
    for (i = 0; i < length; i++)
    {
        *sum += data[i];
    }
}
```
Eliminate Unneeded Memory References

- Don’t need to store in final destination until end
  - Local variable local_sum can be held in register
  - Avoids 1 memory read and 1 memory write per iteration

```c
void combine4(vec* v, int* sum)
{
    int i;
    int length = vec_length(v);
    int* data = get_vec_start(v);
    int local_sum = 0;
    for (i = 0; i < length; i++)
    {
        local_sum += data[i];
    }
    *sum = local_sum;
}
```
## Optimization Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>No-op</th>
<th>-O1</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1</td>
<td>1.282</td>
<td>1.01</td>
<td>0.921</td>
</tr>
<tr>
<td>Combine2</td>
<td>1.189</td>
<td>0.764</td>
<td>0.764</td>
</tr>
<tr>
<td>Combine3</td>
<td>0.607</td>
<td>0.448</td>
<td>0.323</td>
</tr>
<tr>
<td>Combine4</td>
<td>0.585</td>
<td>0.287</td>
<td>0.286</td>
</tr>
</tbody>
</table>
Summary

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

- **Strength Reduction**
  - Shift, add instead of multiply or divide

- **Share Common Subexpressions**

- **Eliminating Loop Inefficiencies**

- **Reducing Procedure Calls**

- **Eliminating Unneeded Memory References**
Appendix: Code Profiling

- Computes (approximate) amount of time spent in each function

```bash
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