Obfuscated Code


C code to print all 12 verses of The 12 Days of Christmas:

```c
#include <stdio.h> main(t,_,a){return!0<t?t<3?main(-79,-13,a +main(-87,1-_ , main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):
3,main(-94,-27+t,a)&&t==2?<_13? main(2,+_1,"%s %d %d\n"):9:16:t<0?t<-72?
main(_,t, "@n'++,#'*/{}w+/_cdnr/+,{r/*de}+,*{++_/w{++,/w#q#n+,/#{1,+,/n{n+
,/+n+,#/ ;#q#n+,/+k#;++,'r :'d'3,}{wK w'K:'+}e#';dq#'l \ q'#+d'K!/
+k#;q#'r}eKK#{w'r}eKK{nl}#/;#q'#n'{}#w}{nl}'/+#n';d}rw' i;# \ ){nl}!/n{n#'; r{w'r nc{nl}'/#{1,+}K {rw' iK{[nl}/w#q#n'wk nw' \ iwk{KK{nl}!/w
{%l##w# i; :{nl}'/{*q#'ld;r'}{nlwb!/*de}'c \ ;{nl}'-{rw}'+,}{"*' 
#nc,',#nw}'+k'd'+e}+;#'rdq#w! nr'/ '})+}{rl#''{n' '}# \ }'+}##(!!="/") :t<-50?
_==*a?putchar(31[a]):main(-65,_,a+1):main(({a=''/}t,_,a+1) :0<t?
main(2,2,"%s"):a='/' ||main(0,main(-61,*a, '!ek;dc i@bK'(q)-[w]*%n+r3#1,{}: 
\nuwloca-0;m .vpbks,fxntdCeghiry"),a+1);}
```
• Today: Finish (?) chapter 5
• Thursday: Review session (come with questions --- last year’s final up on moodle)
Limitations of Optimizing Compilers

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

```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];
}
```
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;
```

```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;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

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

1 multiplication: i*n

```assembly
imulq  %rcx, %rsi       # i*n
addq  %rdx, %rsi        # i*n+j
movq   %rsi, %rax       # i*n+j
subq  %rcx, %rax        # i*n+j-n
leaq  (%rsi,%rcx), %rcx # i*n+j+n
```
Optimization Blocker #1: Procedure Calls

- 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');
}
```
Lower Case Conversion Performance

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

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

• 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(N²) performance
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
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 –O2
    - See web aside ASM:OPT
  - Do your own code motion
1 int select(int x, int y, int i) {
2   int data[2] = { x, y };   
3   if (i >= 0 && i < 2) return data[i];
4   else return 0;
5 }

1 int test_select() {
2   return select(5, 6, 1);
3 }

Function test_select, optimized -O1
1 test_select:
2   pushl %ebp
3   movl %esp, %ebp
4   subl $12, %esp  Allocate 12 bytes on stack
5   movl $1, 8(%esp)  Set 1 as 3rd argument
6   movl $6, 4(%esp)  Set 6 as 2nd argument
7   movl $5, (%esp)  Set 5 as 1st argument
8   call select  Call select(5,6,1)
9   leave
10  ret
int select(int x, int y, int i) {
    int data[2] = { x, y };  
    if (i >= 0 && i < 2)     
        return data[i];     
    else
        return 0;
}

Function select, optimized -O3
x at %ebp+8, y at %ebp+12, i at %ebp+16
int test_select():
    pushl %ebp
    movl $6, %eax       Set result = 6
    movl %esp, %ebp
    popl %ebp
    ret
Inlining

• **Good**
  – Can be much faster

• **Bad**
  – Harder to debug
  – Code size can increase
  – Hurts modularity
Electric Kitteh Bed 2000

Plz to press CTRL+P for instant warmth & gentle massage
Benchmark Example: Data Type for Vectors

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

/* retrieve vector element and store at val */
double get_vec_element(vec *v, int idx, double *val) {
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```
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
    – float
    – double
  
• Operations
  – Use different definitions of OP and IDENT
    – + / 0
    – * / 1
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: \( \text{CPE} = \text{cycles per OP} \)
- \( T = \text{CPE} \times n + \text{Overhead} \)
  
  - CPE is slope of line

![Graph showing vsum1 and vsum2 with slopes](image)

\( vsum1: \text{Slope} = 4.0 \)

\( vsum2: \text{Slope} = 3.5 \)
```c
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></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>
Basic Optimizations

void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t \text{ OP } d[i];
    *dest = t;
}

• Move vec\_length out of loop
• Avoid bounds check on each cycle
• Accumulate in temporary
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;
    }
}

void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
Effect of Basic Optimizations

```c
void combine4(vec_ptr v, data_t *dest) {
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

<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>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Combine4 = Serial Computation (OP = *)

- Computation (length=8)
  
  \[ ((((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7]) \]

- Sequential dependence
  - Performance: determined by latency of OP
Loop Unrolling

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

- Perform 2x more useful work per iteration
# Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Function</th>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>combine 4</td>
<td>accum in local</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td>combine5</td>
<td>unroll by x2</td>
<td>2.0</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td>unroll by x3</td>
<td></td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
</tr>
</tbody>
</table>
/* Unroll loop by 2 */
void combine5(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = (acc OP data[i]) OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

Figure 5.16 Unrolling loop by factor $k = 2$. Loop unrolling can reduce the effect of loop overhead.

Practice Problem 5.7
Modify the code for combine5 to unroll the loop by a factor $k = 5$. 
Loop Unrolling with Reassociation

void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

• Can this change the result of the computation?
• Yes, for FP.
Reassociated Computation

\[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]); \]

- What changed:
  - Ops in the next iteration can be started early (no dependency)

- Overall Performance
  - N elements, D cycles latency/op
  - Should be \((N/2+1)*D\) cycles:
    \[ \text{CPE} = \frac{D}{2} \]
  - Measured CPE slightly worse for FP
Loop Unrolling with Separate Accumulators

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}

• Different form of reassociation
Separate Accumulators

What changed:
- Two independent “streams” of operations

Overall Performance
- N elements, D cycles latency/op
- Should be (N/2+1)*D cycles:
  \[ \text{CPE} = \frac{D}{2} \]
- CPE matches prediction!

\[
x_0 = x_0 \text{ OP } d[i];
\]
\[
x_1 = x_1 \text{ OP } d[i+1];
\]
Unrolling & Accumulating

Idea
- Can unroll to any degree L
- Can accumulate K results in parallel
- L should be multiple of K

Limitations
- Diminishing returns
  - Cannot go beyond throughput limitations (execution units)
- Large overhead for short lengths
Calculating Improvement

- $\text{CPE}_{\text{old}} / \text{CPE}_{\text{new}}$
Calculating Improvement

- $\frac{CPE_{old}}{CPE_{new}}$
- $\frac{CPE_{new}}{CPE_{old}}$
- $100 \left( \frac{CPE_{new} - CPE_{old}}{CPE_{new}} \right)$
- $100 \left( \frac{CPE_{new} - CEP_{old}}{CEP_{old}} \right)$
Calculating Improvement

\[ \text{CPE}_{\text{new}} = 1.0 \quad \text{CPE}_{\text{old}} = 2.0 \]

- \[ \frac{\text{CPE}_{\text{old}}}{\text{CPE}_{\text{new}}} = 2.0 \]
- \[ \frac{\text{CPE}_{\text{new}}}{\text{CPE}_{\text{old}}} = 0.5 \]
- \[ 100 \left( \frac{\text{CPE}_{\text{old}} - \text{CPE}_{\text{new}}}{\text{CPE}_{\text{new}}} \right) = 100\% \]
- \[ 100 \left( \frac{\text{CPE}_{\text{old}} - \text{CEP}_{\text{new}}}{\text{CEP}_{\text{old}}} \right) = 50\% \]
Profiling Demo