Source Code Optimization

Felix von Leitner
CCC Berlin
felix-optimize@fefe.de

August 2007

Abstract

People often write less readable code because they think it will produce faster code. Unfortunately, in most cases, the code will not be faster. Warning: advanced topic, contains assembly language code.

Introduction

- Optimizing == important.
- But often: Readable code == more important.
- Learn what your compiler does
  Then let the compiler do it.

#define for numeric constants

Not just about readable code, also about debugging.

```
#define CONSTANT 23
const int constant=23;
enum { constant=23 };
```

1. Alternative: `const int constant=23;`
   Pro: symbol visible in debugger.
   Con: uses up memory, unless we use `static`.

2. Alternative: `enum { constant=23 };`
   Pro: symbol visible in debugger, uses no memory.
   Con: integers only

Constants: Testing

```
enum { constant=23 };
#define CONSTANT 23
static const int Constant=23;

int foo(void) {
  a(constant+3);
  a(CONSTANT+4);
  a(Constant+5);
}
```

```
foo:
        movl    $26, %edi
        call    a
        movl    $27, %edi
        call    a
        movl    $28, %edi
        call    a
```

Constants: Testing

```
const int a=23;
static const int b=42;

int foo() { return a+b; }
```

```
foo:
        movl    $65, %eax
        ret

        .section        .rodata
a:
        .long   23
```

#define vs inline

- preprocessor resolved before compiler sees code
- again, no symbols in debugger
- can’t compile without inlining to set breakpoints
- use `static` or `extern` to prevent useless copy

macros vs inline: Testing

```
#define abs(x) ((x)>0?(x):-(x))

static long abs2(long x) {
  return x>=0?x:-x;
}

long foo(long a) {
  return abs(a);
}

long bar(long a) {
  return abs2(a);
}
```

```
foo:
        movq    %rdi, %rdx
        sarq    $63, %rdx
        movq    %rdx, %rax
        xorq    %rdi, %rax
        subq    %rdx, %rax
        ret
bar:
        movq    %rdi, %rdx
        sarq    $63, %rdx
        movq    %rdx, %rax
        xorq    %rdi, %rax
        subq    %rdx, %rax
        ret
```

inline in General

- No need to use ”inline”
- Compiler will inline anyway
- In particular: will inline large static function that’s called exactly once
- Make helper functions `static`!
- Inlining destroys code locality
- Subtle differences between inline in gcc and in C99

inline: gcc vs C99

- Subtle differences in whether symbol is exported and what happens if you do `&inline-function`.
- `static inline`: handled the same; no symbol exported, function code emitted if address taken.
- In gcc, `inline` will generate a copy and export a symbol for it.
- In gcc, `extern inline` will generate neither copy nor symbol, and if you attempt to take the address, it will reference the symbol externally.
- In C99, the meanings of `inline` and `extern inline` are the opposite.

Inline vs modern CPUs

- Modern CPUs have a built-in call stack
- Return addresses still on the stack
- … but also in CPU-internal pseudo-stack
- If stack value changes, discard internal cache, take big performance hit

In-CPU call stack: -fPIC on i486

```
int foo() {
  static int val;
  return ++val;
}

foo:
        call    .L3
.L3:
        popl    %ecx
        addl    $_GLOBAL_OFFSET_TABLE_+[.-.L3], %ecx
        movl    val.1540@GOTOFF(%ecx), %eax
        incl    %eax
        movl    %eax, val.1540@GOTOFF(%ecx)
        ret
```

In-CPU call stack: -fPIC on i686

```
int foo() {
  static int val;
  return ++val;
}

foo:
        call    __i686.get_pc_thunk.cx
        addl    $_GLOBAL_OFFSET_TABLE_, %ecx
        movl    val.1728@GOTOFF(%ecx), %eax
        incl    %eax
        movl    %eax, val.1728@GOTOFF(%ecx)
        ret
__i686.get_pc_thunk.cx:
        movl    (%esp), %ecx
        ret
```

In-CPU call stack: how efficient is it?

```
extern int bar(int x);

int foo() {
  static int val;
  return bar(++val);
}

int main() {
  long c; int d;
  for (c=0; c<100000; ++c) d=foo();
}
```

```
int bar(int x) {
  return x;
}
```

Core 2: 18 vs 14.2, 22%, 4 cycles per iteration. MD5: 16 cycles / byte.

Athlon 64: 10 vs 7, 30%, 3 cycles per iteration.

Range Checks

- Compilers can optimize away superfluous range checks for you
- Common Subexpression Elimination eliminates duplicate checks
- Invariant Hoisting moves loop-invariant checks out of the loop
- Inlining lets the compiler do variable value range analysis

Range Checks: Testing

```
static char array[100000];
static int write_to(int ofs,char val) {
  if (ofs>=0 && ofs<100000)
    array[ofs]=val;
}
int main() {
  int i;
  for (i=0; i<100000; ++i) array[i]=0;
  for (i=0; i<100000; ++i) write_to(i,-1);
}
```

Range Checks: Code Without Range Checks

```
        movb    $0, array(%rip)
        movl    $1, %eax
.L2:
        movb    $0, array(%rax)
        addq    $1, %rax
        cmpq    $100000, %rax
        jne     .L2
```

Range Checks: Code With Range Checks

```
        movb    $-1, array(%rip)
        movl    $1, %eax
.L4:
        movb    $-1, array(%rax)
        addq    $1, %rax
        cmpq    $100000, %rax
        jne     .L4
```

Range Checks

- gcc cannot inline code from other .o file (yet)
- Cannot optimize away checks without inlining
- icc -O2 vectorizes the first loop using SSE (only the first one)
- icc -fast completely removes the first loop
- sunc99 unrolls the first loop 16x and does software pipelining, but fails to inline `write_to`

Strength Reduction

```
unsigned foo(unsigned a) {        foo:
  return a/4;                             shrl    $2, %edi
}

unsigned bar(unsigned a) {        bar:
  return a*9+17;                          leal    17(%rdi,%rdi,8), %eax
}
```

Strength Reduction

```
extern unsigned int array[];

unsigned a() {
  unsigned i,sum;
  for (i=sum=0; i<10; ++i)
    sum+=array[i+2];
  return sum;
}
```

```
        movl    array+8(%rip), %eax
        movl    $1, %edx
.L2:
        addl    array+8(,%rdx,4), %eax
        addq    $1, %rdx
        cmpq    $10, %rdx
        jne     .L2
        rep ; ret
```

Strength Reduction

```
extern unsigned int array[];

unsigned b() {
  unsigned sum;
  unsigned* temp=array+3;
  unsigned* max=array+12;
  sum=array[2];
  while (temp<max) {
    sum+=*temp;
    ++temp;
  }
  return sum;
}
```

```
  movl    array+8(%rip), %eax
  addl    array+12(%rip), %eax
  movl    $1, %edx
.L9:
  addl    array+12(,%rdx,4), %eax
  addq    $1, %rdx
  cmpq    $9, %rdx
  jne     .L9
  rep ; ret
```

Constant Folding

```
#define MYNAME "myprog"

char* foo(const char* s) {
  char* x=malloc(strlen(s)+18);
  sprintf(x,"%s: error: %s!\n",
          MYNAME,s);
  return x;
}
```

```
#define MYNAME "myprog"
char* foo(const char* s) {
  char* x=malloc(strlen(s)+
          (sizeof(MYNAME)-1)+
          sizeof(": error: !\n"));
  sprintf(x,"%s: error: %s!\n",
          MYNAME,s);
  return x;
}
```

Where does the 18 come from? Don't waste your auditor's time.

The generated code is the same.

Declaring variables ”register”

- Useful in the late 70ies
- Today… Not so much
- Compilers ignore the keyword
- So save yourself the effort

Tail Recursion

```
long fact(long x) {
  if (x<=0) return 1;
  return x*fact(x-1);
}
```

```
fact:
        testq   %rdi, %rdi
        movl    $1, %eax
        jle     .L6
.L5:
        imulq   %rdi, %rax
        subq    $1, %rdi
        jne     .L5
.L6:
        rep ; ret
```

gcc has removed tail recursion for years, icc and suncc don't.

Aliasing

```
struct node {
  struct node* next, *prev;
};

void foo(struct node* n) {
  n->next->prev->next=n;
  n->next->next->prev=n;
}
```

```
movq    (%rdi), %rax
movq    8(%rax), %rax
movq    %rdi, (%rax)
movq    (%rdi), %rax
movq    (%rax), %rax
movq    %rdi, 8(%rax)
```

The compiler reloads `n->next` because `n->next->prev->next` could point to `n`, and then the first statement would overwrite it.

This is called "aliasing".

Dead Code

The compiler and linker can automatically remove:

- Unreachable code inside a function (sometimes)
- A static (!) function that is never referenced.
- Whole .o/.obj files that are not referenced.
  If you write a library, put every function in its own object file.

Note that function pointers count as references, even if noone ever calls them, in particular C++ vtables.

Inline Assembler

- Using the inline assembler is hard
- Most people can't do it
- Of those who can, most don't actually improve performance with it
- Case in point: madplay

If you don't have to: don't.

Inline Assembler: madplay

```
asm ("shrdl %3,%2,%1"  \
     : "=rm" (__result)  \
     : "0" (__lo_), "r" (__hi_), "I" (MAD_F_SCALEBITS)  \
     : "cc");  \

asm ("shrl %3,%1\n\t"  \
     "shll %4,%2\n\t"  \
     "orl %2,%1\n\t"  \
     : "=rm" (__result)  \
     : "0" (__lo_), "r" (__hi_), "I" (MAD_F_SCALEBITS),
       "I" (32-MAD_F_SCALEBITS)  \
     : "cc");  \
```

Speedup: 30% on Athlon, Pentium 3, Via C3. (No asm needed here, btw)

Inline Assembler: madplay

```
enum { MAD_F_SCALEBITS=12 };

uint32_t doit(uint32_t __lo__,uint32_t __hi__) {
  return ((((uint64_t)__hi__) << 32) | __lo__) >> MAD_F_SCALEBITS;
}

[intel compiler:]
   movl     8(%esp), %eax
   movl     4(%esp), %edx
   shll     $20, %eax
   shrl     $12, %edx
   orl      %edx, %eax
   ret
```

Rotating

```
unsigned int foo(unsigned int x) {
  return (x >> 3) | (x << (sizeof(x)*8-3));
}
```

```
foo:
        rorl    $3, %edi
        movl    %edi, %eax
        ret
```

Pre- vs Post-Increment

- a++ returns a temp copy of a
- then increments the real a
- can be expensive to make copy
- … and construct/destruct temp copy
- so, use ++a instead of a++

This advice was good in the 90ies, today it rarely matters, even in C++.

Cache Lines

Array element has same size as CPU cache line.

Code walks through elements in array.

Looks at first byte in each.

CPU evicts same cache line all the time.

Solution: add dummy byte to array elements.

On some super computers, the FORTRAN compiler can do this. gcc can't.

Fancy-Schmancy Algorithms

- If you have 10-100 elements, use a list, not a red-black tree
- Fancy data structures help on paper, but rarely in reality
- More space overhead in the data structure, less L2 cache left for actual data
- If you manage a million elements, use a proper data structure
- Pet Peeve: "Fibonacci Heap".

If the data structure can't be explained on a beer coaster, it's too complex.

Memory Hierarchy

- Only important optimization goal these days
- Use mul instead of shift: 5 cycles penalty.
- Conditional branch mispredicted: 10 cycles.
- Cache Miss to main memory: 250 cycles.

That’s It!

If you do an optimization, test it on real world data.

If it’s not drastically faster but makes the code less readable: undo it.

Questions?