Source Code Optimization

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

April 2008

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;

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

We expect no memory references and no additions in the generated code.

Constants: Testing - gcc 4.3

```
foo:
        subq    $8, %rsp
        movl    $26, %edi
        call    a
        movl    $27, %edi
        call    a
        movl    $28, %edi
        addq    $8, %rsp
        jmp     a
```

Constants: Testing - Intel C Compiler 10.1.015

```
foo:
        pushq   %rsi
        movl    $26, %edi
        call    a
        movl    $27, %edi
        call    a
        movl    $28, %edi
        call    a
        popq    %rcx
        ret
```

Constants: Testing - Sun C 5.9

```
foo:
        pushq   %rbp
        movq    %rsp,%rbp
        movl    $26, %edi
        call    a
        movl    $27, %edi
        call    a
        movl    $28, %edi
        call    a
        leave
        ret
```

Constants: Testing - MSVC 2008

```
foo proc near
  sub rsp, 28h
  mov ecx, 1Ah
  call a
  mov ecx, 1Bh
  call a
  mov ecx, 1Ch
  add esp, 28h
  jmp a
foo endp
```

Constants: Testing gcc / icc

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

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

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

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

Note: memory is reserved for a (in case it is referenced externally).

Note: foo does not actually access the memory.

Constants: Testing - MSVC 2008

```
const int a=23;              a dd 17h
static const int b=42;       b dd 2Ah

int foo() { return a+b; }    foo proc near
                               mov eax, 41h
                               ret
                             foo endp
```

Sun C, like MSVC, also generates a local scope object for "b".

I expect future versions of those compilers to get smarter about static.

#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 for inline function

macros vs inline: Testing - gcc / icc

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

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

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

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

```
foo:    # very smart branchless code!
        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
```

macros vs inline: Testing - Sun C

Sun C generates code like gcc, but using r8 instead of rdx.

It also emits abs2 and outputs this bar:

```
bar:
  push   %rbp
  mov    %rsp,%rbp
  leaveq
  jmp    abs2
```

macros vs inline: Testing - MSVC 2008

```
#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 proc near
  test ecx, ecx
  jg short loc_16
  neg ecx
loc_16: mov eax, ecx
  ret
foo endp
bar proc near
  test ecx, ecx
  jns short loc_26
  neg ecx
loc_26: mov eax, ecx
  ret
bar endp
```

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 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: 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 (gcc 4.2)

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

Note: Instead of initializing sum to 0 and adding the first element, this code initializes sum to the first element and then adds the others.

Range Checks: Code With Range Checks (gcc 4.2)

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

Note: Same code! All range checks optimized away!

Range Checks

- gcc 4.3 -O3 removes first loop and vectorizes second with SSE
- 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`

Range Checks - MSVC 2008

MSVC converts first loop to call to memset and leaves range checks in.

```
  xor         r11d,r11d
  mov         rax,r11
loop:
  test        rax,rax
  js          skip
  cmp         r11d,100000
  jae         skip
  mov         byte ptr [rax+rbp],0FFh
skip:
  inc         rax
  inc         r11d
  cmp         rax,100000
  jl          loop
```

Range Checks - Cleverness

```
int regular(int i) {
  if (i>5 && i<100)
    return 1;
  exit(0);
}

int clever(int i) {
  return (((unsigned)i) - 6 > 93);
}
```

Note: Casting to unsigned makes negative values wrap to be very large values, which are then greater than 93. Thus we can save one comparison.

Range Checks - Cleverness - gcc

```
int foo(int i) {
  if (i>5 && i<100)
    return 1;
  exit(0);
}
```

```
foo:
        subl    $6, %edi
        subq    $8, %rsp
        cmpl    $93, %edi
        ja      .L2
        movl    $1, %eax
        addq    $8, %rsp
        ret
.L2:
        xorl    %edi, %edi
        call    exit
```

Note: gcc knows the trick, too!

Range Checks - Cleverness - icc

```
int foo(int i) {
  if (i>5 && i<100)
    return 1;
  exit(0);
}
```

```
foo:
        pushq   %rsi
        cmpl    $6, %edi
        jl      ..B1.4
        cmpl    $99, %edi
        jg      ..B1.4
        movl    $1, %eax
        popq    %rcx
        ret
..B1.4:
        xorl    %edi, %edi
        call    exit
```

Note: Intel does not do the trick, but it knows the exit case is rare; forward conditional jumps are predicted as "not taken".

Range Checks - Cleverness - suncc

```c
int foo(int i) {
  if (i>5 && i<100)
    return 1;
  exit(0);
}
```

```
foo:
        push    %rbp
        movq    %rsp,%rbp
        addl    $-6,%edi
        cmpl    $94,%edi
        jae     .CG2.14
.CG3.15:
        movl    $1,%eax
        leave
        ret
.CG2.14:
        xorl    %edi,%edi
        call    exit
        jmp     .CG3.15
```

Range Checks - Cleverness - msvc

```
int foo(int i) {
  if (i>5 && i<100)
    return 1;
  exit(0);
}
```

```
foo:
  lea           eax,[rcx-6]
  cmp           eax,5Dh
  ja            skip
  mov           eax,1
  ret
skip:
  xor           ecx,ecx
  jmp           exit
```

Note: msvc knows the trick, too, but uses lea instead of add.

Strength Reduction

```
unsigned foo(unsigned a) {    gcc/icc/sunc:  shrl    $2, %edi
  return a/4;                         msvc:  shr     ecx,2
}
```

```
unsigned bar(unsigned a) {    gcc/icc/sunc:  leal    17(%rdi,%rdi,8), %eax
  return a*9+17;                      msvc:  lea     eax,[rcx+rcx*8+11h]
}
```

Note: No need to write `a>>2` when you mean `a/4`!

Note: compilers express `a*9+17` better than most people would have.

Strength Reduction - readable version

```
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
```

Note: "rep ; ret" works around a shortcoming in the Opteron branch prediction logic, saving a few cycles. Very few humans know this.

Strength Reduction - unreadable version

```
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
# Note: code is actually worse!
```

Strength Reduction

- gcc 4.3 -O3 vectorizes a but not b
- icc -O2 completely unrolls a, but not b
- sunc completely unrolls a, tries 16x unrolling b with prefetching, produces ridiculously bad code for b
- MSVC 2008 2x unrolls both, generates smaller, faster and cleaner code for a

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
```

Note: iterative code generated, no recursion!

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

Outsmarting the Compiler - simd-shift

```
unsigned int foo(unsigned char i) {  // gcc,sun: 3*shl, 3*or (scheduled)
  return i | (i<<8) | (i<<16) | (i<<24);
} /* found in a video codec */       // icc,msvc: 3*shl, 3*or

unsigned int bar(unsigned char i) {  // all: 2*shl, 2*or
  unsigned int j=i | (i << 8);
  return j | (j<<16);
} /* my attempt to improve foo */

unsigned int baz(unsigned char i) {  // gcc: 1*imul (2*shl+2*add for p4)
  return i*0x01010101;               // msvc/icc,sunc: 1*imul
} /* "let the compiler do it" */
```

Note: gcc is smarter than the video codec programmer on all platforms.

Outsmarting the Compiler - for vs while

```
for (i=1; i<a; i++)
  array[i]=array[i-1]+1;
```

```
i=1;
while (i<a) {
  array[i]=array[i-1]+1;
  i++;
}
```

- gcc: identical code, vectorized with -O3
- icc: identical code, not vectorized
- sunc: identical code, unrolled
- msvc: identical code

Outsmarting the Compiler - shifty code

```
int foo(int i) {
  return ((i+1)>>1)<<1;
}
```

Same code for all compilers: one add/lea, one and.

Limits of the Optimizer: 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");   /* what they did */

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");   /* my improvement patch */
```

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;
}   /* how you can do the same thing in C */

[intel compiler:]
   movl      8(%esp), %eax
   movl      4(%esp), %edx
   shll      $20, %eax    # note: just like my improvement patch
   shrl      $12, %edx
   orl       %edx, %eax
   ret
```

Rotating

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

```
gcc: ror $3, %edi
icc: rol $29, %edi
sunc: rol $29, %edi
msvc: ror ecx,3
```

Integer Overflow

```
size_t add(size_t a,size_t b) {
  if (a+b<a) exit(0);
  return a+b;
}
```

```
gcc:
  mov %rsi,%rax
  add %rdi,%rax
  jb  .L1 # no cmp needed!
  ret
```

```
icc:
  add %rdi,%rsi
  cmp %rsi,%rdi # superfluous
  ja  .L1       # but not expensive
  mov %rsi,%rax
  ret
```

Sun does lea+cmp+jb. MSVC does lea+cmp and a forward jae over the exit (bad, because forward jumps are predicted as not taken).

Integer Overflow - Not There Yet

```
unsigned int mul(unsigned int a,unsigned int b) {
  if ((unsigned long long)a*b>0xffffffff)
    exit(0);
  return a*b;
}

fefe:    # this is how I'd do it
  mov    %esi,%eax
  mul    %edi
  jo     .L1
  ret
```

compilers: imul+cmp+ja+imul (+1 imul, +1 cmp)

Integer Overflow - Not There Yet

So let's rephrase the overflow check:

```
unsigned int mul(unsigned int a,unsigned int b) {
  unsigned long long c=a;
  c*=b;
  if ((unsigned int)c != c)
    exit(0);
  return c;
}
```

compilers: imul+cmp+jne (still +1 cmp, but we can live with that).

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++.

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.

Branchless Code

```
int foo(int a) {         int bar(int a) {
  if (a<0) a=0;            int x=a>>31;
  if (a>255) a=255;        int y=(255-a)>>31;
  return a;                return (unsigned char)(y | (a & ~x));
}                        }

  for (i=0; i<100; ++i) {    /* maximize branch mispredictions! */
    foo(-100); foo(100); foo(1000);
  }
  for (i=0; i<100; ++i) {
    bar(-100); bar(100); bar(1000);
  }
```

foo: 72 cycles. bar: 72 cycles. (Core 2)

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?