Source Code Optimization

Felix von Leitner
Code Blau GmbH
leitner@codeblau.de

October 2009

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.
Target audience check

How many of you know what out-of-order superscalar execution means?

How many know what register renaming is?

How knows what cache associativity means?

This talk is for people who write C code. In particular those who optimize their C code so that it runs fast.

This talk contains assembly language. Please do not let that scare you away.
#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.
Source Code Optimization

**Constants: Testing - gcc 4.3**

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

pushq   %rsi
movl    $26, %edi
call    a
movl    $27, %edi
call    a
movl    $28, %edi
call    a
popq    %rcx
ret
foo:
  pushq   %rbp
  movq    %rsp,%rbp
  movl    $26, %edi
  call    a
  movl    $27, %edi
  call    a
  movl    $28, %edi
  call    a
  leave
  ret
foo:
    pushq  %rbp
    movq   %rsp, %rbp
    movl   $26, %edi
    call   a
    movl   $27, %edi
    call   a
    movl   $28, %edi
    call   a
    popq   %rbp
    ret
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
Source Code Optimization

**Constants: Testing gcc / icc / llvm**

```c
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

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

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
Source Code Optimization

macros vs inline: Testing - gcc / icc

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

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

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);
}
About That Branchless Code...

```c
foo:
    mov rdx, rdi  # if input>=0: rdx=0, then xor, sub=NOOP
    sar rdx, 63   # if input<0: rdx=-1
    mov rax, rdx  # xor rdx : NOT
    xor rax, rdi  # sub rdx : +=1
    sub rax, rdx  # note: -x == (~x)+1
    ret

long baz(long a) {
    long tmp = a>>(sizeof(a)*8-1);
    return (tmp ^ a) - tmp;
}
```
macros vs inline: Testing - Sun C

Sun C 5.9 generates code like gcc, but using r8 instead of rdx. Using r8 uses one more byte compared to rax-rbp. Sun C 5.10 uses rax and rdi instead.

It also emits abs2 and outputs this bar:

```
bar:
push %rbp
mov %rsp,%rbp
leaveq
jmp abs2
```
macros vs inline: Testing - LLVM 2.6 SVN

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

foo:  # not quite as smart
    movq  %rdi, %rax
    negq  %rax
    testq %rdi, %rdi
    cmovg %rdi, %rax
    ret

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

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

bar:  # branchless variant
    movq  %rdi, %rcx
    sarq  $63, %rcx
    addq  %rcx, %rdi
    movq  %rdi, %rax
    xorq  %rcx, %rax
    ret

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

Source Code Optimization
Source Code Optimization

macro vs inline: Testing - MSVC 2008

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

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

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

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

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

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

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

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

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
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
ejne .L2
```
**Range Checks: Code With Range Checks (gcc 4.2)**

```assembly
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)
- 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
- llvm inlines but leaves checks in, does not vectorize
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], OFFh
skip:
    inc   rax
    inc   r11d
    cmp   rax, 100000
    jl    loop
```
int zero(char* array) {
    unsigned long i;
    for (i=0; i<1024; ++i)
        array[i]=23;
}

Expected result: write 256 * 0x23232323 on 32-bit, 128 * 0x2323232323232323 on 64-bit, or 64 * 128-bit using SSE.
Vectorization - Results: gcc 4.4

• gcc -02 generates a loop that writes one byte at a time

• gcc -03 vectorizes, writes 32-bit (x86) or 128-bit (x86 with SSE or x64) at a time

• impressive: the vectorized code checks and fixes the alignment first
Vectorization - Results

- icc generates a call to _intel_fast_memset (part of Intel runtime)
- llvm generates a loop that writes one byte at a time
- the Sun compiler generates a loop with 16 movb
- MSVC generates a call to memset
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.
Source Code Optimization

Range Checks - Cleverness - gcc

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

Note: gcc knows the trick, too! gcc knows that exit() does not return and thus considers the return more likely.
int foo(int i) {
    if (i>5 && i<100)
        return 1;
    exit(0);
}

LLVM knows the trick but considers the return statement more likely.
int foo(int i) {
    if (i>5 && i<100)
        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

    return 1;
    exit(0);
}

Note: Intel does not do the trick, but it knows the exit case is rare; forward conditional jumps are predicted as “not taken”.
int foo(int i) {
    if (i>5 && i<100) {
        push %rbp
        movq %rsp,%rbp
        addl $-6,%edi
        cmpl $94,%edi
        jae .CG2.14
    }
    xorl %edi,%edi
    call exit
    jmp .CG3.15
}

.CG2.14:
movl $1,%eax
leave
ret
.CG3.15:
Range Checks - Cleverness - msvc

int foo(int i) {
    if (i>5 && i<100) {
        lea eax,[rcx-6]
        cmp eax,5Dh
        ja skip
        mov eax,1
        ret
    }
    xor ecx,ecx
    jmp exit
}

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

```c
unsigned foo(unsigned a) {
    return a/4;
}
```

```c
unsigned bar(unsigned a) {
    return a*9+17;
}
```

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

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

**Strength Reduction - readable version**

```c
extern unsigned int array[];

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

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

- gcc 4.3 -O3 vectorizes a but not b
- icc -O2 completely unrolls a, but not b
- suncc 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
- LLVM completely unrolls a, but not b
long fact(long x) {
    if (x<=0) return 1;
    return x*fact(x-1);
}

Note: iterative code generated, no recursion!

gcc has removed tail recursion for years. icc, suncc and msvc don’t.
unsigned int foo(unsigned char i) { // all: 3*shl, 3*or
    return i | (i<<8) | (i<<16) | (i<<24);
} /* found in a video codec */

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,llvm: 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

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

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

- gcc: identical code, vectorized with -O3
- icc, llvm, msvc: identical code, not vectorized
- sunc: identical code, unrolled
int foo(int i) {
    return ((i+1)>>1)<<1;
}

Same code for all compilers: one add/lea, one and.
Outsmarting the Compiler - boolean operations

```c
int foo(unsigned int a, unsigned int b) {
    return ((a & 0x80000000) ^ (b & 0x80000000)) == 0;
}
```

```
icc 10:
    xor %esi, %edi  # smart: first do XOR
    xor %eax, %eax
    and $0x80000000, %edi  # then AND result
    mov $1, %edx
    cmove %edx, %eax
    ret
```
int foo(unsigned int a, unsigned int b) {
    return ((a & 0x80000000) ^ (b & 0x80000000)) == 0;
}

sunc:
xor %edi,%esi     # smart: first do XOR
test %esi,%esi    # smarter: use test and sign bit
setns %al         # save sign bit to al
movzbl %al,%eax   # and zero extend
ret
int foo(unsigned int a,unsigned int b) {
    return ((a & 0x80000000) ^ (b & 0x80000000)) == 0;
}

llvm:
xor  %esi,%edi          # smart: first do XOR
shrl $31, %edi          # shift sign bit into bit 0
movl %edi, %eax         # copy to eax for returning result
xorl $1, %eax           # not
ret                      # holy crap, no flags dependency at all
Source Code Optimization

Outsmarting the Compiler - boolean operations

```c
int foo(unsigned int a, unsigned int b) {
    return ((a & 0x80000000) ^ (b & 0x80000000)) == 0;
}
```

gcc / msvc:
- xor %edi,%esi  # smart: first do XOR
- not %esi       # invert sign bit
- shr $31,%esi   # shift sign bit to lowest bit
- mov %esi,%eax  # holy crap, no flags dependency at all
- ret            # just as smart as llvm
Source Code Optimization

Outsmarting the Compiler - boolean operations

```
int foo(unsigned int a, unsigned int b) {
    return ((a & 0x80000000) ^ (b & 0x80000000)) == 0;
}
```

```c
icc 11:
xor %esi,%edi # smart: first do XOR
not %edi
and $0x80000000,%edi # superfluous!
shr $31,%edi
mov %edi,%eax
ret
```

Version 11 of the Intel compiler has a regression.
Source Code Optimization

**Outsmarting the Compiler - boolean operations**

```c
int bar(int a, int b) { /* what we really wanted */
    return (a<0) == (b<0);
}
```

gcc:    # same code!!
not %edi
xor %edi,%esi
shr $31,%esi
mov %esi,%eax
retq

msvc:
xor eax,eax
test ecx,ecx
mov r8d,eax
mov ecx,eax
sets r8b
test edx,edx
sets cl
cmp r8d,ecx
sete al
ret

Source Code Optimization
**Outsmarting the Compiler - boolean operations**

```c
int bar(int a, int b) { /* what we really wanted */
    return (a<0) == (b<0);
}
```

llvm/sunc:
```
shr $31,%esi
shr $31,%edi
cmp %esi,%edi
sete %al
movzbl %al,%eax
ret
```

icc:
```
xor %eax,%eax
mov $1,%edx
shr $31,%edi
shr $31,%esi
cmp %esi,%edi
cmove %edx,%eax
retq
```
Limits of the Optimizer: Aliasing

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

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

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

```c
asm ("shrdl %3,%2,%1"
    : "=rm" (_,result)
    : "0" (_,lo_), "r" (_,hi_), "I" (MAD_F_SCALEBITS)
    : "cc"); /* what they did */
asm ("shrl %3,%1
    "shll %4,%2
    "orl %2,%1"
    : "=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)
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    # gcc 4.4 also does this like this, but only on x64 :-(
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
llvm: rol $29, %eax
msvc: ror ecx,3
Source Code Optimization

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,%rsi
cmp %rsi,%rdi # superfluous
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).
Source Code Optimization

**Integer Overflow**

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

**llvm:**

```asm
movq %rsi, %rbx
addq %rdi, %rbx  # CSE: only one add
cmpq %rdi, %rbx  # but superfluous cmp
jae .LBB1_2      # conditional jump forward
xorl %edi, %edi  # predicts this as taken :-(
call exit
.LBB1_2:
movq %rbx, %rax
ret
```
Source Code Optimization

**Integer Overflow - Not There Yet**

```c
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)**
So let’s rephrase the overflow check:

```c
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).
Conditional Branches

How expensive is a conditional branch that is not taken?

Wrote a small program that does 640 not-taken forward branches in a row, took the cycle counter.

Core 2 Duo: 696

Athlon: 219
### Branchless Code

```c
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: 4116 cycles. bar: 3864 cycles. On Core 2. Branch prediction has context and history buffer these days.
```
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.
## Memory Access Timings, Linux 2.6.31, Core i7

<table>
<thead>
<tr>
<th>Description</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Fault, file on IDE disk</td>
<td>1,000,000,000 cycles</td>
</tr>
<tr>
<td>Page Fault, file in buffer cache</td>
<td>10,000 cycles</td>
</tr>
<tr>
<td>Page Fault, file on ram disk</td>
<td>5,000 cycles</td>
</tr>
<tr>
<td>Page Fault, zero page</td>
<td>3,000 cycles</td>
</tr>
<tr>
<td>Main memory access</td>
<td>200 cycles (Intel says 159)</td>
</tr>
<tr>
<td>L3 cache hit</td>
<td>52 cycles (Intel says 36)</td>
</tr>
<tr>
<td>L1 cache hit</td>
<td>2 cycles</td>
</tr>
</tbody>
</table>

The Core i7 can issue 4 instructions per cycle. So a penalty of 2 cycles for L1 memory access means a missed opportunity for 7 instructions.
Source Code Optimization

memory latency on Core i7 920

CPU cycles vs. data points, sorted by latency
What does it mean?

Test: memchr, iterating through \n in a Firefox http request header (362 bytes).

<table>
<thead>
<tr>
<th>Method</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive byte-by-byte loop</td>
<td>1180</td>
</tr>
<tr>
<td>Clever 128-bit SIMD code</td>
<td>252</td>
</tr>
<tr>
<td>Read 362 bytes, 1 at a time</td>
<td>772</td>
</tr>
<tr>
<td>Read 362 bytes, 8 at a time</td>
<td>116</td>
</tr>
<tr>
<td>Read 362 bytes, 16 at a time</td>
<td>80</td>
</tr>
</tbody>
</table>

It is easier to increase throughput than to decrease latency for cache memory. If you read 16 bytes individually, you get 32 cycles penalty. If you read them as one SSE2 vector, you get 2 cycles penalty.
On x86, there are several ways to write zero to a register.

```assembly
mov $0,%eax
and $0,%eax
sub %eax,%eax
xor %eax,%eax
```

Which one is best?
Bonus Slide

So, sub or xor? Turns out, both produce a false dependency on %eax. But CPUs know to ignore it for xor.

Did you know?

The compiler knew.

I used sub for years.
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?