Camil Demetrescu
Irene Finocchi

Algorithm Engineering

Course Notes

Sapienza University of Rome

(Version: April 24, 2011)

A designer knows he has achieved perfection not when there
is nothing left to add, but when there is nothing left to take away.

(Antoine de Saint-Exupéry, “Wind, Sand and Stars”, Trans. Lewis Galantiere)

Contents

1

Program Optimization

More computing sins are committed in the name of efficiency
(without necessarily achieving it) than for any other single reason
- including blind stupidity.

(William Allan Wulf)

In this chapter, we addess some fundamental program optimization techniques, discussing capabilities and limitations of modern optimizing compilers. We provide examples of optimizations performed automatically by compilers, and examples of optimizations that must be done manually by programmers in their source code. In our discussion, we use the C language and the `gcc` compiler, starting with an overview of programming basics in x86-64 platforms.

1.1 x86-64 Programming Basics

In this section, we review some key features of machine-level programming in x86-64 platforms based on the System V AMD64 Application Binary Interface (ABI), such as Mac OS X and Linux. Although an exhaustive discussion is beyond the scope of this section, we provide a minimal set of notions that will be needed throughout this book. Our discussion assumes that C programs are written in ISO C90 and compiled with `gcc` 4.2.1. By default, we assume that programs are executed in 64-bit mode.

1.1.1 Address Space and Relevant Sections

Programs can access a linear 64-bit logical address space with addresses ranging in the interval $[0, 2^{64} - 1]$. The address space is partitioned into 4KB pages, some of which are mapped onto physical frames by the operating system, and includes the following relevant sections:

- `TEXT`: machine code of all user functions linked statically;
- `DATA`: string literals and variables with internal and external linkage, explicitly initialized by the program;
- `BSS`: variables with internal and external linkage, not explicitly initialized by the program (set by default to zero);
- `HEAP`: blocks allocated dynamically with `malloc`, `calloc`, etc.;
- `STACK`: call frames for activated functions, containing actual parameters, local variables, etc. The stack grows downward from high addresses to low addresses.

access level	size (bits)	reg 0	reg 1	reg 2	reg 3	reg 4	reg 5	reg 6	reg 7
quadword	64	rax	rbx	rcx	rdx	rdi	rsi	rbp	rsp
doubleword	32	eax	ebx	ecx	edx	edi	esi	esp	ebp
word	16	ax	bx	cx	dx	di	si	bp	sp
byte	8	al	bl	cl	dl	dil	sil	bpl	spl

access level	size (bits)	reg 8	reg 9	reg 10	reg 11	reg 12	reg 13	reg 14	reg 15
quadword	64	r8	r9	r10	r11	r12	r13	r14	r15
doubleword	32	r8d	r9d	r10d	r11d	r12d	r13d	r14d	r15d
word	16	r8w	r9w	r10w	r11w	r12w	r13w	r14w	r15w
byte	8	r8l	r9l	r10l	r11l	r12l	r13l	r14l	r15l

Table 1.1: x86-64 general-purpose registers (GPR).

Some sections may read-only, such as the `TEXT` section, and the pages containing string literals. Attempting to access an unmapped page, or writing to a read-only page of the address space results in an access violation exception (segmentation fault).

1.1.2 General-purpose Registers

x86-64 CPUs are equipped with the 16 general-purpose 64-bit integer registers (GPR) shown in Table 1.1. All of these registers can be accessed at byte, word, doubleword, and quadword level:

- *byte* level: access to the 8 least significant bits of the register, using names `al`, `bpl`, `r8l`, etc. Writing a register at byte level leaves the upper 56 bits of the register unchanged.
- *word* level: access to the 16 least significant bits of the register, using names `ax`, `bp`, `r8w`, etc. Writing a register at word level leaves the upper 48 bits of the register unchanged.
- *doubleword* level: access to the 32 least significant bits of the register, using names `eax`, `ebp`, `r8d`, etc. Writing a register at doubleword level sets to zero the upper 32 bits of the register.
- *quadword* level: access to all 64 bits of the register, using names `rax`, `rbp`, `r8`, etc.

1.1.3 Instruction Operands and Addressing Modes

Most x86-64 instructions are binary operations that take two operands: a *source* operand and a *destination* operand. In our discussion, we use the AT&T syntax, where the source operand is followed by the destination operand. The source operand specifies the first argument. The destination operand specifies the second argument (if any) and the location of the result. For instance, the instruction `addq` $source$ $destination$ computes the sum $source + destination$ and writes the result to a 64-bit $destination$.

Instructions operands can be of three main types listed below:

1. Immediate. A constant value (for source operands only).

Syntax: `$value`, where `value` is a literal (e.g., in decimal or hexadecimal notation).

Examples:

- `$0xF`: the integer constant 15
- `$7`: the integer constant 7
- `$-7`: the integer constant -7

2. Register. A register.

Syntax: `%reg`, where `reg` is the name of a register.

Examples:

- `%rax`: 64-bit register `rax`
- `%eax`: 32-bit register `eax`
- `%r8`: 64-bit register `r8`
- `%r8d`: 32-bit register `r8d`
- `addl $7,%eax`: instruction that computes the operation $\texttt{eax} \leftarrow \texttt{eax} + 7$

3. Memory. An object located within the logical address space.

Syntax: there are various forms of memory operands, listed below.

- `(%reg)`: object at address `reg`, where `reg` is a register.

  Example: `(%rax)`: object at address `rax`.

- `d(%reg)`: object at address `reg+d`, where `reg` is a register and `d` is a constant displacement (positive or negative).

  Example: `-6(%rax)`: object at address $\texttt{rax} - 6$

- `d(%base,%index,scale)`: object at address $\texttt{base} + \texttt{index} \cdot \texttt{scale} + \texttt{d}$, where `base` and `index` are registers, `d` is a constant displacement (positive or negative), and `scale` is a constant in $\{1, 2, 4, 8\}$.

  Example: `4(%rax, %rbx, 8)`: object at address $\texttt{rax} + \texttt{rbx} \cdot 8 + 4$.

There are further mixed forms, listed below:

- `(%base,%index)`: object at address $\texttt{base} + \texttt{index}$, where `base` and `index` are a registers.

  Example: `(%rax,%rbx)`: object at address $\texttt{rax} + \texttt{rbx}$.

- `d(%base,%index)`: object at address $\texttt{base} + \texttt{index} + d$, where `d` is a constant displacement (positive or negative) and `base` and `index` are a registers.

  Example: `-24(%rax,%rbx)`: object at address $\texttt{rax} + \texttt{rbx} - 24$.

- `(%base,%index,scale)`: object at address $\texttt{base} + \texttt{index} \cdot \texttt{scale}$, where `base` and `index` are registers, and `scale` is a constant in $\{1, 2, 4, 8\}$.

  Example: `(%rax, %rbx, 8)`: object at address $\texttt{rax} + \texttt{rbx} \cdot 8$.

1.1.4 Instruction Suffixes and Operand Sizes

Some instruction names include a single-letter suffix that specifies the size of the operands. Table 1.2 lists suffixes and their meaning. For instance, instruction `movb $7,(%rax)` writes the value 7 into the 1-byte (`char`) object at the address contained in register `%rax`. Similarly, instruction `movl $7,(%rax)` writes 7 into the 4-bytes (`int`) object at address `%rax`.

suffix	size (bits)	meaning	C analog	example
`b`	8	byte	`char`	`movb`
`w`	16	word	`short`	`movw`
`l`	32	doubleword	`int`	`movl`
`q`	64	quadword	`long`, `void*`	`movq`

Table 1.2: x86-64 instruction suffixes and the corresponding operand sizes.

1.1.5 Stack Frames

Each executing function has a *stack frame*, which is allocated on the runtime stack. The stack frame is a local storage that includes room for local variables, parameters to be passed to called functions, and other information. Stack frames must be of size multiple of 16 and must be aligned on 16-bytes boundaries.

Local objects on the current stack frame are accessed using the *base pointer* register `rbp` or the *stack pointer* register `rsp`.

Base pointer register `rbp`. By convention, the `rbp` register always points 16 bytes below the top of the current frame (see Figure 1.1). We will discuss the 16-bytes area above address `rbp` in Section 1.1.7. The area below address `rbp` typically contains local variables, parameters passed to functions, and more. For instance, a local `int` variable may be stored at address `rbp` -4.

Stack pointer register `rsp`. In general, the `rsp` register points to the lowest address in use on the stack (the "stack top", since the stack grows upside down). However, programs are allowed to store and retrieve data even below the current value of `rsp`, but no lower than 128 bytes from it.

Stack frame chain. To allow debuggers reconstruct the trace of pending calls, stack frames are linked by keeping in each frame a pointer to the previous frame. This chain may be omitted for performance reasons by compiling with `gcc -fomit-frame-pointer`, but this would make debugging impossible on some machines.

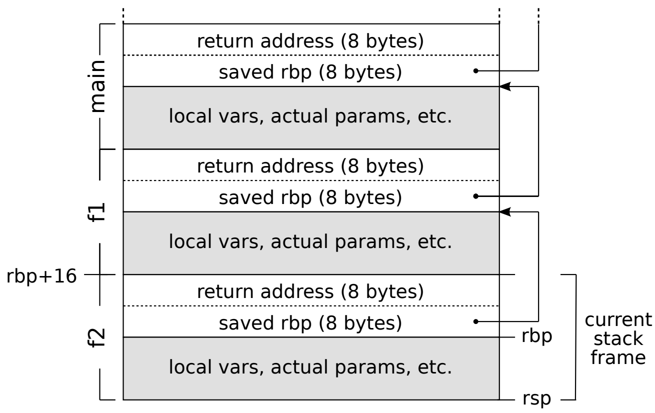

Figure 1.1: System V AMD64 calling conventions: stack frame layout.

1.1.6 Generating Assembly Code with `gcc`

A C translation unit can be compiled into x86-64 code in AT&T syntax using option `-S` of `gcc`, as shown in the following examples. Table 1.3 lists some of the most common x86-64 instructions encountered in C programs compiled with `gcc`.

prefix	description	example	C analog
`add`	add source to destination	`addl $5,%ecx`	`ecx += 5`
`call`	procedure call	`call _foo`	`foo()`
`cltq`	sign-extend `eax` to `rax`	–	–
`dec`	decrement destination	`decq %rcx`	`rcx--`
`imul`	multiply destination with source	`imull %esi,%eax`	`eax *= esi`
`inc`	increment destination	`incl %ecx`	`ecx++`
`ja`	jump if above *(unsigned comparison)*	`cmpl %eax,%ebx` `ja L2`	`if ((unsigned)eax > (unsigned)ebx) goto L2`
`jae`	jump if above or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jae L2`	`if ((unsigned)eax >= (unsigned)ebx) goto L2`
`jb`	jump if below *(unsigned comparison)*	`cmpl %eax,%ebx` `jb L2`	`if ((unsigned)eax < (unsigned)ebx) goto L2`
`jbe`	jump if below or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jbe L2`	`if ((unsigned)eax <= (unsigned)ebx) goto L2`
`je`	jump if equal	`cmpq %rax,%rbx` `je L2`	`if (rax == rbx) goto L2`
`jg`	jump if greater *(signed comparison)*	`cmpq %rax,%rbx` `jg L2`	`if (rax > rbx) goto L2`
`jge`	jump if greater or equal *(signed comparison)*	`cmpq %rax,%rbx` `jge L2`	`if (rax >= rbx) goto L2`
`jl`	jump if less *(signed comparison)*	`cmpq %rax,%rbx` `jl L2`	`if (rax < rbx) goto L2`
`jle`	jump if less or equal *(signed comparison)*	`cmpq %rax,%rbx` `jle L2`	`if (rax <= rbx) goto L2`
`jmp`	unconditional jump	`jmp L2`	`goto L2`
`jne`	jump if not equal	`cmpq %rax,%rbx` `jne L2`	`if (rax != rbx) goto L2`
`jnz`	identical to `jne`	–	–
`jz`	identical to `je`	–	–
`lea`	copy address to destination (load effective address)	`leaq -12(%rax),%rcx`	`rcx=rax-12`
`leave`	pop the current stack frame, and restore the caller's frame	–	–
`mov`	copy data from source to destination	`movq $7,(%rax)`	`*(long*)rax=7`
`movabs`	copy 64-bit immediate to destination register	`movabsq $-7,%rax`	`rax=-7`
`movsl`	copy sign-extended word to destination register	`movslq %bx,%rax`	`rax=bx`
`movzl`	copy zero-extended word to destination register	`movzlw %bx,%eax`	`eax=(unsigned)bx`
`movsb`	copy sign-extended byte to destination register	`movsbq %bl,%rax`	`rax=bl`
`movzb`	copy zero-extended byte to destination register	`movzbq %bl,%rax`	`rax=(unsigned)bl`
`pop`	pop value from stack and write it to destination	`popq %rbx`	–
`push`	push value on stack	`pushq %rbx`	–
`ret`	return from procedure call	`ret`	`return`
`sub`	subtract source from destination	`subl $5,%ecx`	`ecx -= 5`

Table 1.3: Common x86-64 instructions.

Example 1. Consider the following C source code in a file `first.c`:

Listing 1.1: `first.c`

```
int main() {
    int a, b;
    a = 7;
    b = 5;
    a++;
    a = b - 5;
    return 0;
}
```

The program can be assembled using the following command line:

```
$ gcc -S first.c
```

The command generates a text file `first.s` containing the following lines:

Listing 1.2: `first.s`

```
    .text
.globl _main
_main:
LFB2:
    pushq   %rbp
LCFI0:
    movq    %rsp, %rbp
LCFI1:
    movl    $7, -4(%rbp)
    movl    $5, -8(%rbp)
    incl    -4(%rbp)
    movl    -8(%rbp), %eax
    subl    $5, %eax
    movl    %eax, -4(%rbp)
    movl    $0, %eax
    leave
    ret
LFE2:
    .section __TEXT,__eh_frame,coalesced,no_toc+strip_static_syms+live_support
EH_frame1:
    .set L$set$0,LECIE1-LSCIE1
    .long L$set$0
LSCIE1:
    .long   0x0
    .byte   0x1
    .ascii
    .byte   0x1
    .byte   0x78
    .byte   0x10
    .byte   0x1
    .byte   0x10
    .byte   0xc
    .byte   0x7
    .byte   0x8
    .byte   0x90
    .byte   0x1
```

```
    .align 3
LECIE1:
.globl _main.eh
_main.eh:
LSFDE1:
    .set L$set$1,LEFDE1-LASFDE1
    .long L$set$1
LASFDE1:
    .long   LASFDE1-EH_frame1
    .quad   LFB2-.
    .set L$set$2,LFE2-LFB2
    .quad L$set$2
    .byte   0x0
    .byte   0x4
    .set L$set$3,LCFI0-LFB2
    .long L$set$3
    .byte   0xe
    .byte   0x10
    .byte   0x86
    .byte   0x2
    .byte   0x4
    .set L$set$4,LCFI1-LCFI0
    .long L$set$4
    .byte   0xd
    .byte   0x6
    .align 3
LEFDE1:
    .subsections_via_symbols
```

Although file `first.s` contains several sections, most of them can be ignored. The only portion of interest for our discussion, excerpted from `first.s`, is shown below. The fragment contains the x86-64 assembly instructions corresponding to the `main` function of Listing 1.1:

Listing 1.3: x86-64 code for the `main` function of the program in Listing 1.1

```
_main:
LFB2:
    pushq   %rbp
LCFI0:
    movq    %rsp, %rbp
LCFI1:
    movl    $7, -4(%rbp)
    movl    $5, -8(%rbp)
    incl    -4(%rbp)
    movl    -8(%rbp), %eax
    subl    $5, %eax
    movl    %eax, -4(%rbp)
    movl    $0, %eax
    leave
    ret
```

The following instructions correspond to the lines 3–6 of the program in Listing 1.1:

```
    movl    $7, -4(%rbp)        // a = 7;
    movl    $5, -8(%rbp)        // b = 5;
    incl    -4(%rbp)            // a++;
```

```
        movl    -8(%rbp), %eax      // int temp = b;
        subl    $5, %eax            // temp = temp - 5;
        movl    %eax, -4(%rbp)      // a = temp;
```

We notice that:

- the local `int` variable `a` of `main` is referred to by memory operand `-4(%rbp)` (see Section 1.1.5)
- the local `int` variable `b` of `main` is referred to by memory operand `-8(%rbp)` (see Section 1.1.5)
- the `incl -4(%rbp)` instruction increments by 1 local variable `a`
- the assignment `a=b-5` is done in three steps:
  - `movl -8(%rbp),%eax`: loads the value of `int` variable `a` in a temporary register `%eax`;
  - `subl $5,%eax`: subtracts 5 from register `%eax`;
  - `movl %eax,-4(%rbp)`: writes `%eax` to variable `b`.

We will discuss the remaining instructions of Listing 1.3 (lines 2–6 and 13–15) in Section 1.1.7.

Example 2. In this second example, we show the effect of changing from `int` to `char` the type of variables `a` and `b` of Listing 1.1 on the x86-64 code generated by `gcc`:

Listing 1.4: `first2.c`

```
1 int main() {
2     char a, b;   // now variables are char rather than int
3     a = 7;
4     b = 5;
5     a++;
6     a = b - 5;
7     return 0;
8 }
```

The code generated by `gcc -S first2.c` for lines 3–6 of Listing 1.4 is:

```
        movb    $7, -1(%rbp)        // a = 7;
        movb    $5, -2(%rbp)        // b = 5;
        incb    -1(%rbp)            // a++;
        movzbl  -2(%rbp), %eax      // int temp = b;
        subl    $5, %eax            // temp = temp - 5;
        movb    %al, -1(%rbp)       // a = temp;
```

Observe that:

- instructions that modify `char` variables end with `b` (`movb` and `incb`) as dicussed in Section 1.1.4;
- `movzbl` is used to read data from variable `b`.

1.1.7 Procedure Calls

In this section we describe how procedure calls are implemented in System V AMD64 platforms. If a currently executing function y is called by a function x, we refer to x as the *caller* and to y as the *callee*. The invocation of a callee by a caller involves the following steps:

1. *Parameter passing*: the caller sets up the arguments to be passed to the callee;

arguments to pass/retrieve	**where to pass/retrieve them**
return value	`rax`
1^{st} argument	`rdi`
2^{nd} argument	`rsi`
3^{rd} argument	`rdx`
4^{th} argument	`rcx`
5^{th} argument	`r8`
6^{th} argument	`r9`
7^{th} argument	`rbp` $+ 16$
8^{th} argument	`rbp` $+ 24$
$\vdots$	$\vdots$
i^{th} argument	`rbp` $+ 16 + 8 \cdot (i - 7)$

Table 1.4: System V AMD64 calling conventions: parameter passing.

2. *Procedure call*: the caller invokes the callee using the `call` instruction;
3. *Callee setup*: the callee creates a new *stack frame* for the computation;
4. *Callee execution*: the callee executes the instructions in its body;
5. *Callee cleanup*: the callee destroys the current stack frame and restores the caller's stack frame using the `leave` instruction;
6. *Return from procedure*: the `ret` instruction terminates the execution of the callee, returning the control back to the caller.

We now address each step in detail using the example given in Listing 1.5:

Listing 1.5: `call.c`

```
1 long f(long p1, long p2, long p3, long p4, long p5, long p6, long p7, long p8) {
2     return p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8;
3 }
4
5 int main() {
6     long x = f(1, 2, 3, 4, 5, 6, 7, 8);
7     return x;
8 }
```

1. Parameter passing. Parameters are passed by the caller to the callee in accordance with the conventions listed in Table 1.4 (left). Up to 6 arguments (each of size up to 8 bytes) can be passed using registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. Further parameters are pushed on stack in reverse order so that the 7th argument is pointed to by register `rsp` prior to the procedure call.

For instance, the eight parameters of the call `f(1, 2, 3, 4, 5, 6, 7, 8)` at line 6 of Listing 1.5 can be passed as follows:

```
movq    $8, 8(%rsp)   // passing 8th argument on stack
movq    $7, (%rsp)    // passing 7th argument on stack
movl    $6, %r9d      // passing 6th argument in r9
movl    $5, %r8d      // passing 5th argument in r8
movl    $4, %ecx      // passing 4th argument in rcx
movl    $3, %edx      // passing 3rd argument in rdx
```

registers to be saved by callee
rbx
rbp
r12
r13
r14
r15

Table 1.5: System V AMD64 calling conventions: callee save registers.

```
movl    $2, %esi      // passing 2nd argument in rsi
movl    $1, %edi      // passing 1st argument in rdi
```

Notice that the non-negative constant actual parameters 1, 2, 3, etc. are written in the lowest 32 bits of 64-bit destination registers, while the upper 32 bits are automatically filled with zero (see word-level register write access in Section 1.1.2).

2. Procedure call. The callee is invoked by the caller using the `call` instruction, which pushes on stack the return address, i.e., the address of the instruction that follows immediately the `call` instruction in the caller's code.

In our example, the call of function `f` at line 6 of Listing 1.5 is done as follows:

```
call    _f
```

3. Callee setup. If the callee needs local storage, e.g, for local variables or parameter passing, it has to allocate a new frame on stack. Also, if callee's body needs to modify any of the registers `rbx` or `r12`–`r15`(see Table 1.5), they must be saved on stack during the setup.

In our example, the callee setup of function `main` of Listing 1.5 is done as follows:

```
pushq   %rbp         // save caller's base pointer
movq    %rsp, %rbp   // set callee's base pointer
subq    $16, %rsp    // make room for parameters to be passed
```

The setup consists of saving the current value of `rbp` on stack, letting `rbp` point to the new stack frame, and making room for parameters to be passed to `f` (see Figure 1.2 on the left). The callee setup of function `f` of Listing 1.5 is:

```
pushq   %rbp         // save caller's base pointer
movq    %rsp, %rbp   // set callee's base pointer
```

The setup is the same as in the case of the `main` function, except that no room below `rbp` is allocated (see Figure 1.2 on the right).

4. Callee execution. The callee executes the instructions in its body. In our example, the body returns in `rax` the sum of the arguments:

```
addq    %rdi, %rsi
addq    %rdx, %rsi
addq    %rcx, %rsi
addq    %r8, %rsi
```

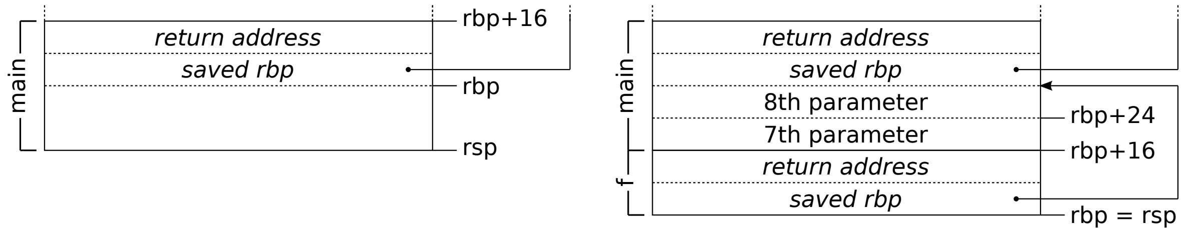

Figure 1.2: Stack frame of function `main` before calling `f` (left) and stack frame of function `f` called by `main` (right), as specified in Listing 1.5.

```
        leaq    (%rsi,%r9), %rax
        addq    16(%rbp), %rax
        addq    24(%rbp), %rax
```

5. Callee cleanup. The cleanup operation consists of deallocating the current stack frame. This is done by the `leave` instruction, which is equivalent to performing the following steps:

- `movq %rbp,%rsp`: shrink the stack to the current value of `rbp`;
- `popq %rbp`: restore caller's base pointer.

6. Return from procedure. The call is terminated by the `ret` instruction, which pops the return value from stack, completing the current stack frame deallocation, and resumes the caller's execution.

Listing 1.6 summarizes the full x86-64 code generated by `gcc -S -O2`[1] for functions `f` and `main` of Listing 1.5.

Listing 1.6: x86-64 code for functions `f` and `main` of Listing 1.5

```
_f:
    pushq   %rbp
    movq    %rsp, %rbp
    addq    %rdi, %rsi
    addq    %rdx, %rsi
    addq    %rcx, %rsi
    addq    %r8, %rsi
    leaq    (%rsi,%r9), %rax
    addq    16(%rbp), %rax
    addq    24(%rbp), %rax
    leave
    ret

_main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movq    $8, 8(%rsp)
    movq    $7, (%rsp)
    movl    $6, %r9d
    movl    $5, %r8d
```

[1] `-O2` means optimization level 2 and will be discussed later in this chapter.

```
    movl    $4, %ecx
    movl    $3, %edx
    movl    $2, %esi
    movl    $1, %edi
    call    _f
    leave
    ret
```

1.2 Optimization Techniques

In this section we survey some basic optimization techniques commonly supported by compilers, discussing limitations that must be taken into account explicitly by the programmers to write efficient code. Experiments in this section have been performed on a MacBook Pro Intel Core 2 Duo @ 2.8 GHz with 4 GB RAM running Mac OS X 10.6.6 and `gcc` 4.2.1.

1.2.1 Register Allocation

Registers provide the fastest access to data objects, much faster than accessing cache or memory. Keeping frequently used objects of a program cached in registers is one of the most effective optimization techniques, called *register allocation*. This allows programs to reduce accesses to memory, resulting in substantial speedups. Compilers use fast algorithms for register allocation: even if they do not always produce an optimal usage of registers, in general they work very well, relieving programmers from the burden of getting into low-level assembly programming to allocate registers manually.

We show an example of register allocation done by `gcc` discussing the following function that swaps the content of two `int` objects:

Listing 1.7: `swap.c`

```c
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

By default, `gcc` does not do any optimization (optimization level 0). The command `gcc -S swap.c -o swap-O0.s` (which is equivalent to `gcc -S -O0 swap.c -o swap-O0.s`) yields:

Listing 1.8: `swap-O0.s` **without register allocation (**`gcc -O0`**)**

```
_swap:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -24(%rbp)
    movq    %rsi, -32(%rbp)
    movq    -24(%rbp), %rax
    movl    (%rax), %eax
    movl    %eax, -4(%rbp)
    movq    -32(%rbp), %rax
    movl    (%rax), %edx
    movq    -24(%rbp), %rax
    movl    %edx, (%rax)
    movq    -32(%rbp), %rdx
```

```
    movl    -4(%rbp), %eax
    movl    %eax, (%rdx)
    leave
    ret
```

The function keeps parameter `a` at address `rbp` -24, parameter `b` at address `rbp` -32, and local variable `temp` at address `rbp` -4 in the stack frame[2]. Notice that each read or write operation on `a`, `b` and `temp` causes a memory access.

Compiling with `gcc` with optimization level 1 turns on register allocation. The following code is produced by command `gcc -S -O1 swap.c -o swap-O1.s`:

Listing 1.9: `swap-O1.s` with register allocation (`gcc -O1`)

```
_swap:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    leave
    ret
```

The code makes the minimum number of memory accesses required to swap the objects, without placing any temporary value on stack.

Experimental Analysis

To assess the performance boost given by register allocation on the `swap` function, we measured the time required to execute 1 billion swaps of two `int` variables:

Listing 1.10: Test program for the swap function

```
void swap(int* a, int* b);

int main() {
    int i, x = 7, y = 5;
    for (i=0; i<1000000000; i++) swap(&x, &y);
    return 0;
}
```

The `main` function was compiled with `-O1` and linked separately with the two versions of `swap` to produce two executable files `swap-O0` and `swap-O1`. We measured performance with `time ./swap-O0` and `time ./swap-O1`. The total times required by the two versions of the program to perform 1 billion swaps were:

`swap-O0` (**no register allocation**)	`swap-O1` (**register allocation**)
4.2 seconds	2.5 seconds

[2]Notice that the code accesses portions of the stack below `rsp`, and therefore not explicitly allocated. We recall that this is allowed by the ABI conventions up to 128 bytes below `rsp` (see Section 1.1.5).

Notice that the reported figures include both the time spent in the `swap` function and the time spent in `main` for function calls and for the loop. Even considering total times, register allocation in the `swap` function yielded a 40% time reduction for the whole program compared to using the unoptimized `swap`. Analyzing `swap` alone would yield a much higher performance boost.

1.2.2 Function Inlining

Calling a function incurs some overhead due to stack, registers, and control flow operations. If a function is frequently called and its body is reasonably short, it may be convenient to expand each call with the instructions in the body itself, saving the time required for function activations at the price of increasing the code size. This technique is called *function inlining* and can be done by programmers at source code level using C macros. Code inlining has the additional benefits of allowing further local optimizations that would not be applied by the compiler across function calls, such as register allocation. As we will see, under some circumstances, compilers can do inlining of regular functions automatically.

We illustrate these ideas by considering again the example of Section 1.2.1 and replacing the `swap` function with a C macro that performs the same task:

Listing 1.11: Test program for `swap` defined as a macro

```
#define swap(a, b) do { \
    int temp = *a;      \
    *a = *b;            \
    *b = temp;          \
} while(0)

int main() {
    int i, x = 0, y = 5;
    for (i=0; i<1000000000; i++)
        swap(&x, &y);
    return x;
}
```

Preprocessing the program with `gcc -E` yields the following code in which the occurrence of the macro name is replaced by the token sequence of the macro definition, which is inlined in the code of `main`:

Listing 1.12: Preprocessed test program generated by `gcc -E`

```
int main() {
    int i, x = 0, y = 5;
    for (i=0; i<1000000000; i++)
        do { int temp = *&x; *&x = *&y; *&y = temp; } while(0);
    return x;
}
```

The x86-64 code generated by `gcc -S -O1` for this version is:

Listing 1.13: Assembly code for the program of Listing 1.11 generated by `gcc -S -O1`

```
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $0, %ecx
    movl    $0, %esi
    movl    $5, %eax
```

```
7 L2:
8     incl    %ecx
9     movl    %esi, %edx
10    movl    %eax, %esi
11    cmpl    $1000000000, %ecx
12    je  L3
13    movl    %edx, %eax
14    jmp L2
15 L3:
16    leave
17    ret
```

Notice that code inlining done by macro expansion allows the compiler to perform register allocation so that the loop of lines 7–14 causes no memory accesses!

When optimization level is `-O3` (or the `-finline-functions` option is specified) `gcc` performs automatic code inlining of simple enough functions defined in the same translation unit. Consider for instance the following program where `swap` is defined as a function within the same translation unit as the `main`:

Listing 1.14: Test program for `swap` defined as a function

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int i, x = 0, y = 5;
    for (i=0; i<1000000000; i++) swap(&x, &y);
    return x;
}
```

Compiling the program with `gcc -S -O1` yields the following assembly code for the `main` function:

Listing 1.15: Assembly code for the program of Listing 1.14 generated by `gcc -S -O1`

```
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %r13
    pushq   %r12
    pushq   %rbx
    subq    $24, %rsp
    movl    $0, -36(%rbp)
    movl    $5, -40(%rbp)
    movl    $0, %ebx
    leaq    -40(%rbp), %r13
    leaq    -36(%rbp), %r12
L4:
    movq    %r13, %rsi
    movq    %r12, %rdi
    call    _swap
    incl    %ebx
    cmpl    $1000000000, %ebx
    jne L4
```

```
20      movl    -36(%rbp), %eax
21      addq    $24, %rsp
22      popq    %rbx
23      popq    %r12
24      popq    %r13
25      leave
26      ret
```

Notice that each iteration of the loop of lines 13–19 makes a call of the `swap` function of Listing 1.9. Conversely, compiling the program with `gcc -S -O1 -finline-functions` performs automatic inlining of function `swap`, yielding *exactly the same result* of Listing 1.13, as if `swap` were defined as a macro.

Experimental Analysis

To assess the benefits of code inlining, we compared the performance of the swap operation by measuring the execution times of three test programs with the `time` command:

- Listing 1.11 compiled with `gcc -O1` (macro)
- Listing 1.14 compiled with `gcc -O1` (no function inlining)
- Listing 1.14 compiled with `gcc -O1 -finline-functions` (function inlining)

The result was:

macro	no function inlining	function inlining
1.0 seconds	2.5 seconds	1.0 seconds

We notice that the macro and the inlined function versions of the test program (whose assembly codes are identical) run more than twice as fast as the non-inlined version. This is due to the combination of eliminating the overhead of function calls and a more aggressive register allocation enabled by the fact that the swap operation becomes local to the body of the `for` loop. Compared to the initial version without register allocation of Section 1.2.1 (compiled with `-O0`), inlining plus register allocation (options `-O1 -finline-functions`) provided a speedup of over a factor of 4 for the `swap` test program.

1.2.3 Constant Folding

The *constant folding* technique consists of replacing expressions on constant operands with the result of the expression. This reduces code size and, since the expression evaluation is performed at compile time and not at run time, it produces a faster code. For instance, in the code below, the expression `8+(14/2)*3` can be replaced with the constant `29`:

Listing 1.16: Code example for illustrating constant folding

```
int f() {
    return 8+(14/2)*3;
}
```

We can apply constant folding manually to the source code and write the following equivalent fragment:

Listing 1.17: Code of Listing 1.16 after constant folding

```
int f() {
    return 29;
}
```

We remark that `gcc` performs constant folding automatically even with optimization level `-O0`, without the need for programmers to apply it manually in the source code:

Listing 1.18: Assembly code for Listing 1.16 generated by `gcc -S -O0`

```
_f:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $29, %eax
    leave
    ret
```

Althouth programmers naturally tend to perform constant folding in their programs, the role of the compiler optimization becomes important when expressions are the result of macro expansions that involve constants.

1.2.4 Constant Propagation

If a variable is assigned a constant value, later occurrences of the variable may be replaced by that value, getting a smaller and faster code. This optimization technique, known as *constant propagation*, is illustrated in the example below:

Listing 1.19: Code example for illustrating costant propagation

```
int x;

int f() {
    x = 8;
    return x - 2;
}
```

We can apply constant propagation manually to the source code and write the following equivalent fragment:

Listing 1.20: Code of Listing 1.19 after constant propagation

```
int x;

int f() {
    x = 8;
    return 8 - 2;
}
```

Notice that in the code above we could also apply constant folding, replacing `8 - 2` with `6`. Constant propagation is not done by default by `gcc`:

Listing 1.21: Assembly code of Listing 1.19 generated by `gcc -S -O0`

```
_f:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    _x@GOTPCREL(%rip), %rax   // rax = &x
    movl    $8, (%rax)                // *(int*)rax = 8     (x = 8)
    movq    _x@GOTPCREL(%rip), %rax   // rax = &x
    movl    (%rax), %eax              // eax = *(int*)rax   (eax = x)
    subl    $2, %eax                  // eax = eax - 2      (return eax - 2)
    leave
    ret
```

Notice that the address of global variable `x` is denoted by `_x@GOTPCREL(%rip)`. Compiling at optimization level 1 (`-O1`) enables constant propagation and constant folding in `gcc`:

Listing 1.22: Assembly code of Listing 1.19 generated by `gcc -S -O1`

```
_f:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    _x@GOTPCREL(%rip), %rax   // rax = &x
    movl    $8, (%rax)                // *(int*)rax = 8     (x = 8)
    movl    $6, %eax                  // eax = 6            (return 6)
    leave
    ret
```

Experimental Analysis

To assess the performance boost given by constant propagation and constant folding on the `f` function, we measured the time required to execute 1 billion calls of `f`:

Listing 1.23: Test program for the `f` function of Listing 1.19

```
int f();

int main() {
    int i, j;
    for (i=0; i<1000000000; i++) j = f();
    return j;
}
```

The `main` function was compiled with `-O1` and linked separately with the two versions of `f` to produce two executable files `f-O0` and `f-O1`. We measured performance with `time ./f-O0` and `time ./f-O1`. The total times required by the two versions of the program to perform 1 billion calls to `f` were:

`f-O0` (**no optimization**)	`f-O1` (**constant propagation + folding**)
2.9 seconds	2.5 seconds

Notice that the reported figures include both the time spent in the `f` function and the time spent in `main` for function calls and for the loop. Analyzing `f` alone would yield a much higher performance boost.

1.2.5 Common Subexpression Elimination

Complex expressions that contain repeated subexpressions can be simplified by computing the common subexpressions separately and reusing them. This optimization technique, called *common subexpression elimination*, is illustrated in the example below:

Listing 1.24: Code example for illustrating common subexpression elimination

```
int expr(int x, int y) {
    return (x + y)*(x + y);
}
```

We can apply common subexpression elimination manually to the source code and write the following equivalent fragment where the common subexpression `x + y` is only computed once:

Listing 1.25: Code of Listing 1.24 after common subexpression elimination

```
int expr(int x, int y) {
    int z = x + y;
    return z * z;
}
```

Common subexpression elimination is not performed by default by `gcc`:

Listing 1.26: Assembly code of Listing 1.24 generated by `gcc -S -O0`

```
1  _expr:
2      pushq   %rbp
3      movq    %rsp, %rbp
4      movl    %edi, -4(%rbp)
5      movl    %esi, -8(%rbp)
6      movl    -8(%rbp), %eax
7      movl    -4(%rbp), %edx
8      addl    %eax, %edx
9      movl    -8(%rbp), %eax
10     addl    -4(%rbp), %eax
11     imull   %edx, %eax
12     leave
13     ret
```

Notice that the code computes the expression `(x + y)` twice at lines 6–8 and 9–10. Compiling at optimization level 1 (`-O1`) enables register allocation and common subexpression elimination in `gcc`:

Listing 1.27: Assembly code of Listing 1.24 generated by `gcc -S -O1`

```
1  _expr:
2      pushq   %rbp
3      movq    %rsp, %rbp
4      addl    %edi, %esi
5      movl    %esi, %eax
6      imull   %esi, %eax
7      leave
8      ret
```

In this second version, the expression `(x + y)` is only computed once at line 4.