Code optimization & linking

Shuai Mu

Slides adapted from Jinyang Li, Bryant and O’Hallaron
What we’ve learnt so far

- **Hardware**
  - Logical Circuits, Flip-Flops, ...
  - CPU
  - Memory
  - I/O

- **Software**
  - System Software
    - Operating System
  - User Applications
    - User App

- **User Applications**
  - e.g. your C programs rkgrep
  - the x86 ISA (e.g. %rax, %rsp, ..., mov, add, jmp, ret, call)

- **System Software**
  - Operating System
  - javac
  - JVM
  - gcc

- **Compiler**
  - gcc
  - javac
  - e.g. your C programs rake

- **Compiler and Language**
  - e.g. C programs
  - Java programs

- **Operating System**
  - The x86 ISA
  - e.g. %rax, %rsp, ...
  - mov, add, jmp, ret, call

- **User Applications**
  - User App
  - e.g. your C programs rake

- **Compiler**
  - gcc
  - javac
  - e.g. your C programs rake

- **Compiler and Language**
  - C programs
  - Java programs
Today’s plan

- Code optimization (done by the compiler)
  - common optimization techniques
  - what prevents optimization
- C linker
Optimizing Compilers

• Goal: generate efficient, correct machine code
  – allocate registers, choose instructions, ...

• Optimization limitation: must be conservative \(\rightarrow\) do not change program behavior under any scenario
  – analysis is based on static information (no runtime information)
  – most analysis done within a procedure
void set_row(long *matrix, long i, long n) {
    for (long j = 0; j < n; j++)
        matrix[n*i+j] = 0;
}

Optimization: code motion
• Reduce frequency with which computation performed
  – If it will always produce same result

set_row:
  testq %rcx, %rcx            # Test n
  jle .L1                     # If 0, goto done
  imulq %rcx, %rdx            # ni = n*i
  leaq (%rdi,%rdx,8), %rdx    # rowp = A + ni*8
  movq $0, %rax
.L3:
  movq $0, (%rdx,%rax,8)      # M[rowp+8*j] = 0
  addq $1, %rax               # j++
  cmpq %rcx, %rax             # j:n
  jne .L3                    # if !=, goto loop .L3
.L1:
  ret
Optimization: use simpler instructions

- Replace costly operation with simpler one
  - Shift, add instead of multiply or divide
    
    \[ 16 \times x \rightarrow x \ll 4 \]
  - Recognize sequence of products

```c
for (long i = 0; i < n; i++) {
    long ni = 0;
    for (long j = 0; j < n; j++) {
        matrix[n*i+j] = 0;
    }
    ni += n;
}
```

assembly not shown
this is equivalent C code
Optimization: reuse common subexpressions

// 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;

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-1)*n, (i+1)*n, i*n

1 multiplication:
i*n
assembly not shown
this is equivalent C code
What prevents optimization?
// convert uppercase letters in string to lowercase
void lower(char *s) {
    for (size_t i=0; i<strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}

Question: What’s the big-O runtime of lower, O(n)?
Lower Case Conversion Performance

– Quadratic performance!
// convert uppercase letters in string to lowercase
void lower(char *s) {

    for (size_t i=0; i<strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
// convert uppercase letters in string to lowercase
void lower(char *s) {
    size_t len = strlen(s);
    for (size_t i=0; i<len; i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}

Lower Case Conversion Performance

– Now performance is linear w/ length, as expected
Optimization Blocker: Procedure Calls

• Why can’t compiler move `strlen` out of inner loop?
  – Procedure may have side effects
    • May alter global state
  – Procedure may not return same value given same arguments
    • May depend on global state

• Compiler optimization is conservative:
  – Treat procedure call as a black box
  – Weak optimizations near them

• Remedies:
  – Do your own code motion
Optimization Blocker 2: Memory aliasing

// Sum rows of n X n matrix and store in vector a
void sum_rows(long *matrix, long *a, long n) {
    for (long i = 0; i < n; i++) {
        a[i] = 0;
        for (long j = 0; j < n; j++) {
            a[i] += matrix[i*n + j];
        }
    }
}

# inner loop
.L4:
    movq (%rsi,%rax,8), %r9  # %r9 = a[i]
    addq (%rdi), %r9         # %r9 += matrix[i*n+j]
    movq %r9, (%rsi,%rax,8)  # a[i] = r9
    addq $8, %rdi
    cmpq %rcx, %rdi
    jne .L4

- Code updates a[i] on every iteration
- Why not keep sum in register and stores once at the end?
Memory aliasing: different pointers may point to the same location

```c
void sum_rows(long *matrix, long *a, long n) {
    for (long i = 0; i < n; i++) {
        a[i] = 0;
        for (long j = 0; j < n; j++) {
            a[i] += matrix[i*n + j];
        }
    }
}

int main() {
    long matrix[3][3] = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}};

    long *a;
    a = (&matrix[0][0])+3;
    sum_rows(&matrix[0][0],a,3);
}
```

Value of `a`:

- before loop: [1, 1, 1]
- after i = 0: [3, 1, 1]
- after i = 1: [3, 7, 1]
- after i = 2: [3, 7, 3]
Optimization blocker: memory aliasing

- Compiler cannot optimize due to potential aliasing
- Manual “optimization”

```c
void sum_rows(long *matrix, long *a, long n) {
    for (long i = 0; i < n; i++) {
        long sum = 0;
        for (long j = 0; j < n; j++) {
            sum += matrix[i*n + j];
            a[i] = sum;
        }
    }
    // compiler will move a[i] = sum out of inner loop
}
```
Getting High Performance

• Use compiler optimization flags
• Watch out for:
  – hidden algorithmic inefficiencies
  – Watch out for optimization blockers: procedure calls & memory aliasing
• Profile the program’s performance
Today’s lesson plan

• Common code optimization (done by the compiler)
  – common optimization
  – what prevents optimization

• C linker
Example C Program

```c
#include "sum.h"
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}

main.c
```

```c
#include "sum.h"

int sum(int *a, int n);

int sum(int *a, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}

sum.c
```
Linking

**Source files**
- main.c
- sum.c
- sum.h

**Re-locatable object files**
- main.o
- sum.o

**Fully linked executable file**
- a.out

*Fully linked executable file (contains code and data for all functions defined in main.c and sum.c)*
Why a separate link phase?

• Modula code & efficient compilation
  – Better to structure a program as smaller source files
  – Change of a source file requires only re-compile that file, and then relink.

• Support libraries (no source needed)
  – Build libraries of common functions, other files link against libraries
    • e.g., Math library, standard C library
How does linker merge object files?

• Step 1: Symbol resolution

  – Programs define and reference symbols (global variables and functions):
    • `void swap() {...} /* define symbol swap */`
    • `swap(); /* reference symbol swap */`
    • `int *xp = &x; /* define symbol xp, reference x */`

  – Symbol definitions are stored in object file in symbol table.
    • Each symbol table entry contains size, and location of symbol.

  – Linker associates each symbol reference with its symbol definition (i.e. the address of that symbol)
How does linker merge object files?

- **Step 2: Relocation**
  - Merge separate object files into one binary executable file
  - Re-locates symbols in the `.o` files to their final absolute memory locations in the executable.

Let’s look at these two steps in more detail....
Format of the object files

• ELF is Linux’s binary format for object files, including
  – Object files (.o),
  – Executable object files (a.out)
  – Shared object files, i.e. libraries (.so)
** ELF Object File Format **

- **Elf header**
  - file type (.o, exec, .so) ...

- **.text section**
  - Code

- **.rodata section**
  - Read only data

- **.data section**
  - Initialized global variables

- **.bss section**
  - Uninitialized global variables
  - “Better Save Space”
  - Has section header but occupies no space

---

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>.text section</td>
</tr>
<tr>
<td>.rodata section</td>
</tr>
<tr>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
</tr>
<tr>
<td>.symtab section</td>
</tr>
<tr>
<td>.rel.txt section</td>
</tr>
<tr>
<td>.rel.data section</td>
</tr>
<tr>
<td>.debug section</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
ELF Object File Format (cont.)

- **.symtab section**
  - Symbol table (symbol name, type, address)

- **.rel.text section**
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable

- **.rel.data section**
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable

- **.debug section**
  - Info for symbolic debugging (`gcc -g`)
Linker Symbols

• Global symbols
  – Symbols that can be referenced by other object files
  – E.g. non-static functions & global variables.

• Local symbols
  – Symbols that can only be referenced by this object file.
  – E.g. static functions & global variables

• External symbols
  – Symbols referenced by this object file but defined in other object files.

needs to be resolved
Step 1: Symbol Resolution

```c
#include "sum.h"

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```c
int sum(int *a, int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

Referencing a global... that’s defined here

Referencing a global... that’s defined here

Linker knows nothing of `val`

Defining a global

Linker knows nothing of `i` or `s`

...that’s defined here
C linker quirks: it allows symbol name collision!

- Program symbols are either **strong** or **weak**
  - **Strong**: procedures and initialized globals
  - **Weak**: uninitialized globals
Symbol resolution in the face of name collision

- Rule 1: Multiple strong symbols are not allowed
  - Otherwise: Linker error

- Rule 2: If there’s a strong symbol and multiple weak symbols, they all resolve to the strong symbol.

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`
Linker Puzzles

```
int x;
p1() {}  // Link time error: two strong symbols (p1)
```

```
int x;
p1() {}  // References to x will refer to the same uninitialized int. Is this what you really want?
p2() {}
```

```
int x;
int y;
p1() {}  // Writes to x in p2 might overwrite y! Evil!
p2() {}
```

```
int x=7;
int y=5;
p1() {}  // Writes to x in p2 will overwrite y! Nasty!
p2() {}
```

```
int x=7;
p1() {}  // References to x will refer to the same initialized variable.
p2() {}
```
How to avoid symbol resolution confusion

• Avoid global variables if you can
• Otherwise
  – Use `static` if you can
  – Initialize if you define a global variable
  – Use `extern` if you reference an external global variable
Step 2: Relocation

Relocatable Object Files

System code
System data

main.o
main()
int array[2]={1,2}

sum.o
sum()

Executable Object File

Headers
System code

main()

swap()

More system code
System data
int array[2]={1,2}

.data
.symtab
.debug

.text
Relocation Entries

```c
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```assembly
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00  mov    $0x2,%esi
   9:   bf 00 00 00 00  mov    $0x0,%edi  # %edi = &array

   a: R_X86_64_32 array
      # Relocation entry

   e:   e8 00 00 00 00  callq  13 <main+0x13>  # sum()
   f:   R_X86_64_PC32 sum-0x4

   13:   48 83 c4 08  add    $0x8,%rsp
   17:   c3            retq

Source: objdump -r -d main.o
### Relocated .text section

```assembly
00000000004004d0 <main>:
  4004d0:  48 83 ec 08       sub    $0x8,%rsp
  4004d4:  be 02 00 00 00 00 mov    $0x2,%esi
  4004d9:  bf 18 10 60 00 00 mov    $0x601018,%edi  # %edi = &array
  4004de:  e8 05 00 00 00 00 callq  4004e8 <sum>  # sum()
  4004e3:  48 83 c4 08       add    $0x8,%rsp
  4004e7:  c3                      retq

00000000004004e8 <sum>:
  4004e8:  b8 00 00 00 00 00 mov    $0x0,%eax
  4004ed:  ba 00 00 00 00 00 mov    $0x0,%edx
  4004f2:  eb 09 jmp     4004fd <sum+0x15>
  4004f4:  48 63 ca movslq  %edx,%rcx
  4004f7:  03 04 8f add     (%rdi,%rcx,4),%eax
  4004fa:  83 c2 01 add     $0x1,%edx
  4004fd:  39 f2 cmp      %esi,%edx
  4004ff:  7c f3 jl       4004f4 <sum+0xc>
  400501:  c3 retq
```

`objdump -d a.out`
Loading Executable Object Files

Executable Object File

- ELF header
- Program header table (required for executables)
- .init section
- .text section
- .rodata section
- .data section
- .bss section
- .symtab
- .debug
- .line
- .strtab
- Section header table (required for relocatables)

User stack (created at runtime)

Memory-mapped region for shared libraries

Run-time heap (created by malloc)

Read/write data segment (.data, .bss)

Read-only code segment (.init, .text, .rodata)

Unused

%rsp (stack pointer)

brk

Loaded from the executable file
Dynamic linking: Shared Libraries

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.

- Dynamic linking can also occur after program has begun (run-time linking).
  - In Linux, this is done by calls to the `dlopen()` interface.

- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory.
Dynamic Linking at Load-time

```
main.c  sum.h

compile

main.o

Linker (ld)

a.out

Loader (execve)

Dynamic linker (ld-linux.so)

unix> gcc -shared -o libmysum.so \
    sum.c myotherfunctions.c

libc.so
libmysum.so

Relocation and symbol table info

Relocatable object file

Partially linked executable object file

Fully linked executable in memory

Code and data
```