Code optimization & linking

Jinyang Li

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

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

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

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

- **User Applications**
  - User App
What we’ve learnt so far

• C program $\rightarrow$ x86 instructions
  – data storage
  – control flows: sequential, jumps, call/ret

• Buffer overflow
  – overwrite a code pointer (return address)
  – execute code intended by the attacker
Today’s lesson 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, ...

Generated code must have the same behavior as the original C program under all scenarios

gcc’s optimization levels: -O1, -O2, -O3, -Og
Common optimization: code motion

- Move computation outside of loop if result remains the same

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

```assembly
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      # j = 0
.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
```

done inside the loop
done outside the loop
Common Optimization: use simpler instructions

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

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

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

assembly not shown
this is equivalent C code
Common 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!
Calling strlen in loop

// 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');
        }
    }
}

• Strlen takes O(n) to finish
• Strlen is called n times
Calling strlen in loop

// 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 obstacle: 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:
  – Typically treat procedure call as a black box
  – Weak optimizations near them

• Remedy:
  – Do your own code motion
Optimization obstacle 2: Memory aliasing

//sum all elements of the array “a”
void sum(long *a, long n, long *result) {
  *result = 0;
  for (long i = 0; i < n; i++) {
    (*result) += a[i];
  }
}

• Code updates *result on every iteration
• Why not keep sum in a register and write once at the end?
Memory aliasing: different pointers may point to the same location

```c
int main() {
  long a[3] = {1, 1, 1};
  long *result;
  long r;

  result = &r;
  sum(a, 3, result);

  result = &a[2];
  sum(a, 3, result);
}
```

Value of a:
- before loop: {1, 1, 0}
- after i = 0: {1, 1, 1}
- after i = 1: {1, 1, 2}
- after i = 2: {1, 1, 4}

*result may alias to some location in array a
→ updates to *result may change a
Optimization obstacle: memory aliasing

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

```c
void sum(long *a, long n, long *result) {
    long sum = 0;
    for (long i = 0; i < n; i++) {
        sum += a[i];
    }
    *result = sum;
}
```
Getting High Performance

• Use compiler optimization flags
• Watch out for:
  – hidden algorithmic inefficiencies
  – Optimization obstacles:
    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

main.c

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

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

sum.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.h

```c
int sum(int *a, int n);
```
Linking

Compile:

- `gcc -c main.c`
- `gcc -c sum.c`

Link:

- `gcc main.o sum.o`

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

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

**Executable file**
- `a.out`

contains `sum` function
contains `main` function
contains both `main` and `sum` and other library functions
Why a separate link phase?

• Modular 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 count;     // define global variable (symbol) count`

  – 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
  
  – With “gcc –c ...”, compiler gives a defined symbol a temporary address. When referencing an unknown symbol, compiler uses a temporary placeholder.
  
  – Re-locates symbols in the .o files to their final memory locations in the executable. Replace placeholders with actual addresses.

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

```
+--------------------------------+-----------+
| ELF header                     | 0         |
|                               |           |
|                               |           |
|                               |           |
| .text section                 |           |
|                               |           |
| .rodata section               |           |
|                               |           |
| .data section                 |           |
|                               |           |
| .bss section                  |           |
|                               |           |
| .symtab section               |           |
|                               |           |
| .rel.txt section              |           |
|                               |           |
| .rel.data section             |           |
|                               |           |
| .debug section                |           |
|                               |           |
| ...                           |           |
```


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.
Step 1: Symbol Resolution

```c
#include "sum.h"

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

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

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

Defining a global
Linker knows nothing of `val`

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?

```
int x=7;
ip1() {}
```

```
double x;  
p2() {}
```

Writes to x in p2 will overwrite y! Nasty!
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**

#### Executable Object File
- **Headers**
  - System code
  - main()
  - swap()
  - More system code
- **System data**
  - int array[2]={1,2}
- **.symtab**
- **.debug**
- **.text**
- **.data**

---

<table>
<thead>
<tr>
<th>Source File</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.o</td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td>int array[2]={1,2}</td>
</tr>
<tr>
<td>sum.o</td>
<td>sum()</td>
</tr>
<tr>
<td></td>
<td>int array[2]={1,2}</td>
</tr>
</tbody>
</table>
Relocation Entries

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

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

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

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

00000000004004e8 <sum>:
00000000004004e8:       b8 00 00 00 00 mov $0x0,%eax
00000000004004ed:       ba 00 00 00 00 mov $0x0,%edx
00000000004004f2:       eb 09 jmp 4004fd <sum+0x15>
00000000004004f4:       48 63 ca movslq %edx,%rcx
00000000004004f7:       03 04 8f add (%rdi,%rcx,4),%eax
00000000004004fa:       83 c2 01 add $0x1,%edx
00000000004004fd:       39 f2 cmp %esi,%edx
00000000004004ff:       7c f3 jl 4004f4 <sum+0xc>
0000000000400501:       c3 retq

objdump -d a.out
Loading Executable Object Files

Executable Object File

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td>0</td>
</tr>
<tr>
<td>Program header table</td>
<td></td>
</tr>
<tr>
<td>(required for executables)</td>
<td></td>
</tr>
<tr>
<td>.text section</td>
<td></td>
</tr>
<tr>
<td>.rodata section</td>
<td></td>
</tr>
<tr>
<td>.data section</td>
<td></td>
</tr>
<tr>
<td>.bss section</td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>.line</td>
<td></td>
</tr>
<tr>
<td>.strtab</td>
<td></td>
</tr>
<tr>
<td>Section header table</td>
<td></td>
</tr>
<tr>
<td>(required for relocatables)</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- User stack (created at runtime)
- Run-time heap (created by malloc)
- Shared libraries
- Read/write data segment (.data, .bss)
- Read-only code segment (.text, .rodata)
- Unused

Symbols:

- \%rsp (stack pointer)
- brk

Sections:

- .text
- .data
- .bss
- .rodata
- .symtab
- .debug
- .line
- .strtab

Shared libraries and run-time heap are created at runtime, while the other sections are loaded from the executable file.
Dynamic linking: Shared Libraries

• Dynamic linking can occur at program load-time
  – Handled automatically by the dynamic linker (`ld-linux.so`).
  – Standard C library (`libc.so`) usually dynamically linked.

• Dynamic linking can also occur at run-time.
  – In Linux, this is done by `dlopen`.

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

1. **Compile**
   - `main.c`
   - `sum.h`
   - `gcc -shared -o libmysum.so sum.c myotherfunctions.c`

2. **Link**
   - Relocatable object file
   - `main.o`
   - `ld -o a.out`

3. **Load**
   - Partially linked executable object file
   - `a.out`
   - `execve`

4. **Execute**
   - Fully linked executable in memory
   - `Dynamic linker (ld-linux.so)`
   - `Code and data`