SENG440 Embedded Systems

– Lesson 5: Software Optimization Techniques III –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 5: Software optimizations for embedded systems – III

Lesson 5: Course Progress

- **Software optimization techniques**
  - What is wrong with plain software
  - Profile driven compilation
  - Efficient C programming
- **Standard peripherals for embedded systems**
  - **Timers, counters**, watchdog timers, real-time clocks
  - **Digital-to-Analog (D/A) and Analog-to-Digital (A/D) converters**
  - Pulse-Width Modulation (PWM) peripherals
  - Universal Asynchronous Receiver/Transmitters (UART)
- **Hardware – software – firmware.**
  - A taxonomy of processors

Constant folding and constant propagation

- **Constant folding** = simplify constant expressions at compile time
  - Example: `a = 2 * 7 * 9;`
  - A decent compiler would not generate two multiply instructions
  - The computed value is substituted at compile time
- **Constant propagation** = substitute the values of known constants in expressions at compile time

Original code:

```
int x = 14;
int y = 7 - x / 2;
y *= (28 / x + 2);
```

Apply constant propagation once:

```
int x = 14;
int y = 7 - 14 / 2;
y *= (28 / 14 + 2);
```

Common subexpression elimination

- Replaces the occurrences of targets of direct assignments with their values

```
y = x;
z = 3 + y;
```

can be replaced by:

```
z = 3 + x;
```

- Any decent compiler should be able to do that automatically.

Common subexpression elimination

- Replaces instances of identical expressions (i.e., they all evaluate to the same value) with a single variable holding the computed value.

```
a = b * c + g;
d = b * c * d;
```

can be replaced by:

```
tmp = b * c;
a = tmp + g;
d = tmp * d;
```

- You have to determine whether the cost of the store to `tmp` is less than the cost of the multiplication.

Partial redundancy elimination

- Eliminates expressions that are redundant on some but not necessarily all paths through a program
- It is a form of common subexpression elimination
- `z` is computed twice if `i < 10`

Original code:

```
if ( i < 10) {
  z = x + 4;
  ...
}
else {
  ...
}
z = x + 4;
```

Optimized code:

```
if ( i < 10) {
  y = x + 4;
  t = y;
  ...
}
else {
  t = x + 4;
  ...
}
z = t;
```

Peephole optimization

- It is performed on a very small set of assembly instructions.
- This set is called a **peephole** or a **window**.
- It is usually performed after machine code has been generated.
- This optimization technique examines a few adjacent instructions to see whether they can be replaced by a single instruction or a shorter sequence of instructions.
  - Collapse redundant move operations
  - Replace multiplication by a power of 2 with a shift operation (this is also an instance of operator strength reduction)
- For project: look at the assembly code and try to improve it yourself.

Dead code elimination

- It reduces the program size by removing code which does not affect the program.
- Dead code includes
  - code that can never be executed (unreachable code)
  - code that only affects dead variables (variables that are irrelevant to the program)

```
int example( void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a << 2;
  return c;
  b = 24; /* Unreachable code */
}
```

Dead code elimination (cont'd)

- Consider the following code:

```
int example( void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a << 2;
  return c;
  b = 24; /* Unreachable code */
}
```

- Code after the return statement cannot possibly be executed
- Variable b might be eliminated entirely from the generated code
- The function returns a static value (96) – it may be simplified to the value it returns

Loop fusion: Merges multiple loops into a single one

Original code:

```
int i, a[100], b[100];
for ( i = 0; i < 100; i++) {
  a[i] = 1;
}
for ( i = 0; i < 100; i++) {
  b[i] = 2;
}
```

Optimized code:

```
int i, a[100], b[100];
for ( i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

It does not always improve the run-time performance – why?

Loop fission: Break a loop into multiple loops

Reverse action to loop fusion

Original code:

```
int i, a[100], b[100];
for ( i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

Optimized code:

```
int i, a[100], b[100];
for ( i = 0; i < 100; i++) {
  a[i] = 1;
}
for ( i = 0; i < 100; i++) {
  b[i] = 2;
}
```

Further reading

- Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
- Y.N. Srikant and Priti Shankar (editors), *The Compiler Design Handbook: Optimizations and Machine Code Generation*
- Mark Larson, *Assembly Optimization Tips*: **http://mark.masmcode.com/**

Questions, feedback

Notes I

Notes II

Notes III

Notes IV