Topic 17 – Optimization

Key Ideas

- Common Subexpression Elimination
- Register Allocation
- Constant Folding
- Constant Propagation
- Dead-code Elimination
- Strength Reduction
- Inlining Procedures
- Tail Recursion

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 11.1 – 11.7 for more detailed explanation

Optimization

Overview

- Recall: for any WLP4 program there are an infinite number of equivalent MIPS assembly language programs.
- What *criteria* to we use to decide if one compiled version of a WLP4 program is better than another?
  - Answer: the time it takes for the program to run
- Finding the equivalent program with the minimum runtime is uncomputable, so we must…
- Use *heuristics*: i.e. *recognize* a pattern of instructions and *replace* them with an equivalent set that
  - runs quicker or
  - (as an approximation) uses a smaller number of instructions

Optimization

Overview

- *Key Point:* These patterns do not necessarily appear in the WLP4 source code
  - They may appear because code is generated by looking at one single node in the parse tree at a time
- *Observation:* for the code x = x+1;
  - in the subtree on the *left hand side* of the '=' sign the parser will generate code that gets the address of 'x'
  - on the subtree on the *right hand side* of the '=' sign the parser will generate code that gets the address of 'x'
  - the parser created code to calculate the same value twice
  - this observation leads to one form of optimization...

Optimization: Common Subexpression

Common Subexpression Elimination

- *Idea:* store the results of *common subexpressions* (often generated by the compiler not the programmer) in registers
- For example: (a+b) * (a+b),
  - calculate the answer to a+b and store in $3 then
    mult $3, $3
    mflo $3
- For "x = x+1" calculate the address of x once, use it twice
- *Caution:* it may not work with functions, e.g. f(1)+f(1) since the functions may have side effects, such as print output
- *Note:* It takes resources to find these common subexpressions
- the "g++" command runs much quicker than "g++ -O3"

Optimization: Register Allocation

Register Allocation

- *Observation:* accessing a register is much quicker than accessing the stack or RAM in general
- using registers also eliminates the code that pushes and pops from the stack, or lw and sw instructions for accessing RAM
- our code generator does not use registers $14-$28
- *Challenge:* must decide how to allocate them if there are more than 15 variables, typically "most used", or "most recently used"
- allocating these registers wisely is a key optimization strategy
- *Caution:* you cannot use the address-of operator on a register location, only a RAM location, so push these values into RAM

Optimization: Register Allocation

Register Allocation

- *Idea:* keep track of the *live ranges* of each variable: from where it is assigned a value to thelocation where it is used with that value.
- If the live ranges of two variables intersect, then you must use two different registers.
- If the live ranges do not intersect, you can reuse the register.

```
int x = 0;
int y = 0;
int z = 0;
x = 3;
y = 4;
x = x + 1;
println(x);
println(y);
z = 7;
println(z);
```

The live ranges of **x** and **y** intersect. The live range of **z** does not intersect with **x** or **y**.

Optimization: Register Allocation

Register Allocation

- *Idea:* **`code()`** specifies available registers in avail and returns where the result is located, e.g. for $expr_1 \rightarrow expr_2 + term$
- after generating the code for $expr_2$, the result is in **`s`**
- when generating the code for *term*, the set avail minus the register s is available for use
- *Enhancement:* provide the ability to specify where you want the result stored

```
// old way
code(expr1)=
code(expr2)
push $3
code(term)
pop $5
add $3, $5, $3

// new way
code(expr1, avail)=
s = code(expr2, avail)
t = code(term, avail\{s})
add $s, $s, $t
return s
```

Optimization: Constant Folding

Example: Code for 2+3

- reduce the number of instructions by calculating answers involving constants at compile time

Currently 9 Instructions vs. *Only 2 Instructions*

code(2+3) =

```
lis $3              ; load 2
.word 2
sw $3, -4(30)       ; push 2 on stack
sub $30, $30, $4
lis $3              ; load 3
.word 3
lw $5, 0($30)       ; pop 2 off stack
add $30, $30, $4
add $3, $5, $3      ; answer
```

code(2+3) =

```
lis $3
.word 5
```

Optimization: Constant Propagation

Constant Propagation

WLP4 Code:
```
int x = 2;
// value of x does not change
return x + x;
```

- *Approach*: Recognize that the value doesn't change and return 4.
- If it is the only place that **x** is used, it does not need a stack entry.
- What our compiler currently does:
  - load the value 2 into \$3 (2 instructions): lis and .word
  - store result in **x** (1 instruction): sw
  - push **x** on stack (3 instructions): lw, sw and sub
  - load **x** in \$3 (1 instruction): lw
  - move **x** from stack to \$5 (2 instructions) : lw and sub
  - then add \$5 and \$3 (1 instruction): add

Optimization: Constant Propagation

Constant Propagation

WLP4 Code:
```
int x = 2;
// value of x does not change
return x + x;
```

- Since x is always 2, the compiler could do the following

```
lis $3              ; load 2 into x
.word 2
sw $3, -12($29)     ; where the offset to x is -12
;; do other stuff
lis $3              ; return value is 4
.word 4
jr $31              ; return from function
```

Optimization: Constant Propagation

Constant Propagation

- *Challenge:* need a way to detect and propagate constants
- *Solution:* The function code() could return an order pair (encoding, value) e.g.
  - (register, 3) would say the result is in \$3 (this has been the only option so far)
  - (const, 2) would say the result is the constant 2
- E.g. if the rule $expr_1 \rightarrow expr_2 + term$ had $expr_2$ and *term* both evaluate to constants, e.g. (const, 2) and (const, 3), then $expr_1$ would evaluate to (const, 5)
- (const, 5) would result in two lines of code

```
lis $3
.word 5
```

Optimization: Dead-code Elimination

Dead code

- Sometimes when code is generated, dead code is created.
- *Dead code* is
  - code that is never executed, e.g.
  - because a logical test is always false
  - because it occurs after a return statement
  - code that is executed but whose results are never used
- *Idea:* detect and do not output dead code.

Optimization: Strength Reduction

Strength Reduction

- *Approach:* some operations can be replaced by faster ones
- *Observation:* generally (on a real processor) addition is quicker than multiplication so for small values replace it by addition.

Currently 8 Instructions vs. *Only 1 Instruction*

```
code(n*2;) =
    sw $3, 0(30)        ; push n on stack
    sub $30, $30, $4
    lis $3              ; load 2 into $3
    .word 2
    lw $5, 0($30)       ; pop n off stack
    add $30, $30, $4
    mult $3, $5         ; multiply 2 * n
    mflo $3             ; load answer in $3
```

```
code(n*2;) =
    add $3, $3, $3
```

Optimization: Inlining Procedures

Inlining Procedures

Inlining replace a function call with the body of the function, i.e.

- Replace

```
int f(int x) { return x+x; }
int wain(int a, int b) { return f(a); }
```

  with

```
int wain(int a, int b) { return a+a; }
```

- Pros:
  - if all calls to **`f`** are in-lined, no need to generate code for **`f`** at all
  - save overhead of creating a stack frame for **`f`**
- Con:
  - if **`f`** is big or used often, then we generate a lot of extra code
  - difficult to do for recursive functions

Optimization: Tail Recursion in Procedures

Tail Recursion

```
int fact(int n, int a){
if(n == 0) return a;
else return fact(n-1,n*a);
}
```

- *Note :* the very last instruction the function does is a recursive call, i.e. **`else return fact(…);`**
- *Optimization:* The content of the current stack frame (local variables etc.) will not be used again in the call of the function, therefore ⇒ reuse the stack frame for the next recursive call

Optimization

Intermediate Code

- *Challenge:* one of the challenges that many of these approaches have is that it is difficult to find patterns such as common subexpressions
- Approach: generally, but beyond the scope of this course, after the lexical, syntactic, and semantic analysis stages, an *intermediate code* (rather than assembly language) is generated with the idea that this code is easier to optimize than the final assembly language
- After optimization the intermediate code is converted to assembly language for a particular processor.
- Would only need to change the final step to create code for different processors (x86-64 vs. ARM-8 vs. MIPS)