Warm-Up Problem

Please do your Teaching Evaluations! Go to https://perceptions.uwaterloo.ca

CS 241 Lecture 21

Optimization

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Optimization

How do we optimize our code?

- This problem is very difficult.
- In practice, we want to minimize the runtime of our compiled code.
- We try to minimize line numbers (since this is an easy-to-measure metric)
  - Note: This is 100% the wrong metric! Many optimizations increase code size!

Optimization

- Could easily spend an entire course on this (and that course is called CS444).
- We discuss just a few ideas

Overall Strategy

- Optimization is an *extra step*, after the code generation we did before.
- To make this extra step work, we introduce an *intermediate language*.
- That intermediate language is typically similar to assembly, but with infinite registers
  - But, there are as many IRs as there are compilers, so don't read too much into this

Overall Strategy

- Infinite registers?
- The IR is not intended to be a runnable language; it's meant to restrict you to assembly-like behavior, without having assembly's more troublesome restrictions (such as limited registers)

Overall Strategy

- Why an IR at all?
- Intuitively, optimizing a (parse) tree sounds like a better approach. A tree is a data structure!
- The problem: The parse tree sort of represents the *control flow* of your program, but most optimization is about the *data flow*.

Overall Strategy

- We'll talk about optimizations rather than IR design, but the IR design is actually a big part of this.
- To demonstrate optimizations, I'll use MIPS + infinite registers.
  - But, if you want my (Gregor's) opinion: Static Single Assignment (SSA) forever. Always SSA. SSA is the best. Look up "LLVM IR" for an example of an SSA IR designed for optimization.

Constant Folding

- Consider the following code:

```
int wain(int a, int b) {
  return 2 + 3;
}
```

- Naively, we might do something like this:

Constant Folding

```
int wain(int a, int b) { return 2 + 3; }

; Insert preamble here
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 3
lw $5, 0($30)
add $30, $30, $4
add $3, $3, $5
; Insert epilogue here
```

Issues

- In the previous code, we use the stack, but we really didn't have to. We could have just loaded 3 into a different register.
- One of the things new compiler authors get most confused with: it is not normally the compiler's job to *run* the code, just *translate* the code. But, if all the information is available, the compiler *can* run some of the code early!
- In this case, we know that we're adding 2 and 3; the compiler could do that, instead of making MIPS do it.

Solution

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

- This code is equivalent and correct.
- This is called *constant folding*.
- Its generalized form (running code in the compiler if we have the information) is *partial evaluation*.

Demonstration

- Let's see what this would look like in our "MIPS with infinite registers" IR and talk about data flow.

Constant Propagation

- What if a variable never changes during a function?
- You should be able to replace the variable with the constant first, and then use constant folding!
- For example:

```
int wain(int a, int b) {
  int x = 5;
  return x + x;
}
```

Naive Translation (ignore preamble)

```
int wain(int a, int b){int x = 5; return x + x;}

;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lw $3, -8($29) ;offset for x
sw $3, -4($30) ;push $3
sub $30, $30, $4
lw $3, -8($29)
lw $5, 0($30)  ;pop($5)
add $30, $30, $4
add $3, $3, $5
```

Improvements

```
int wain(int a, int b){int x = 5; return x + x;}
```

- One solution is to note that *x* = 5 and then do:

```
;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 10
```

- Perhaps even easier: if you only use *x* here, then you don't need a stack entry!

```
lis $3
.word 10
```

Improvements

```
int wain(int a, int b){int x = 5; return x + x;}
```

- Another trick is to note that you are using the same expression, so you could do:

```
;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lw $3, -8($29) ;offset for x
add $3, $3, $3
```

  This last trick is called **common subexpression elimination** (see next slide)
- As another example, if you see say $(a - b * c) * (a - b * c)$, then you do not need to compute this value twice and instead just compute the first term and then perform the multiplication with itself.

Dead Code Elimination

- Dead code is code that cannot be reached no matter what input is given.
- This is often a result of code following tests that always simplify to false, for example:

```
int wain(int a, int b){
  if (a < b){
    if (b < a){
      //Dead Code
    }
  }
}
```

- Sometimes, tests simplify to false thanks to constant folding!
- Another dead code situation is to compute values that are never used.
- Detecting this code and removing it could improve runtime.

Register Allocation

- Accessing variables in RAM is expensive.
- It would be nice to put variables in registers instead.
- But, there are a lot of difficulties and caveats...

Register Allocation

- For our scheme, the registers from $14 to $28 are unused. Variables can go there.
- Caveat: When you take the address-of operator, doing so of a register won't make sense: you must give a RAM address. So, certain variables are constrained.

Register Allocation

- Which (unconstrained) variables should we store in registers?
  - Most used?
  - Most recently used?
- Try to do it so that variables are in registers when in a *live range* and then remove them outside of this range.
  - Note: That means a variable can be in multiple places during its lifetime!

Example of Live Ranges

```
1  int wain(int a, int b){
2    int x = 0; int y = 0; int z = 0;
3    x = 3;
4    y = 10;
5    println(x);
6    z = 7;
7    y = y - x;
8    y = y - z;
9    println(z);
10   return z
11 }
```

Live ranges:

x lines 3-7 y lines 4-8 z lines 6-10

Note: It would be easy here to put all three variables into registers, but with more variables, not all would fit ☺

Register Allocation Notes

- Register allocation isn't actually about variables!
- Intermediate values can (and should) be assigned to registers too!
- More generally, variables are just a special case of values. What you're allocating is where to put values computed at runtime.

Register Allocation Notes

- Remember our basic IR concept: assembly with infinite registers.
- If that kind of IR is used, register allocation isn't even an optimization; it's mandatory! We need to reallocate those infinite registers to real registers and the stack.
- Because of this, many compilers don't even consider register allocation to be an optimization.

Register Allocation Notes

- An exception (sort of) is GCC. If you use no optimizations, it still performs register allocation, but puts them all on the stack.

Register Allocation Algorithm

- The algorithm is *graph coloring*: we can produce a graph where edges are values with conflicting live ranges (i.e., values that are alive at the same time)
- If no connected nodes have the same color (color=register), then every value can be assigned to a register.
- If the graph can't be colored, some need to go in the stack.

Register Allocation Algorithm

- Graph coloring is NP-hard, so really we approximate.
- How hard this problem is depends a lot on how many registers are available. MIPS is luxurious (dozens of registers!) compared to, e.g., x86-64 (lucky to have ten free) or x86-32 (lucky to have three free).

Measuring Optimization Redux

- Until this point, all our optimizations reduced code size *and* improved performance
- Remaining optimizations are much harder to gauge, since they won't (usually) reduce code size

Strength Reduction

- In the real world, addition is much faster than multiplication. If you can avoid multiplying, this is best. Consider code(n*2):

```
; load n into $3
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, -4($30)
add $30, $30, $4
mult $3, $5
mflo $3
```

- This is more work than:

```
add $3, $3, $3
```

Inlining Procedures

Consider the following:

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

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

It would be far faster to use the equivalent code:

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

as we wouldn't need to have all of the overhead associated with the function call!

Inlining Procedures

Does the previous slide always give us shorter code?

- If we call f a lot, then we might get many copies of f.
- This can be a win if the body of f is shorter than the code to call f.
- Sometimes this is harder to tell. Can also cause problems with recursive calls.
- If all calls to f are inline, then we never need the code for the procedure f.

Inlining Catalyst

- Ultimately, inlining itself is barely an optimization. Calling a procedure is just not that expensive.
- But inlining is a *catalyst optimization*: it optimizes little itself, but allows other optimizations!
- Let's look at an example where inlining can catalyze well.

Tail Recursion

Consider the following:

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

- Notice that the last operation made in each branch is just returning a value (that is, after the recursive call, there is no more work to do in this stack frame).
- We could reuse the current stack frame to save some operations.
- This is called **tail recursion.**
- Can we do this in our WLP4 grammar? No! return statements are not allowed in if statements and we cannot refactor the code to get this behaviour!

Hypothetically

Let's suppose that we did allow for multiple return statements like this however. What would tail recursive code look like? Let's look at

```
return fact(n-1, acc*n);
```

which is of the form `factor → ID(expr1,...,exprn)`. Then this would be:

```
code(factor) = code(expr1) + push ($3)
             + ...
             + code(exprn) + push($3)
```

Code for Factor

factor → ID(expr1,...,exprn)

```
code(factor) = code(exprn) + push($3)
             + ...
             + code(expr1) + push($3)
             + pop($5)
             + sw $5, 4n($29)
             + pop($5)
             + sw $5, 4(n-1)($29)
             + ...
             + pop($5)
             + sw$5, 4($29)
             + lis $5
             +.word FID
             + add $30, $29, $4 ;reset stack
             + jr $5 ;NOT jalr don't save
                     ;$31, $29 or pop args
```

Code for Procedure

```
code(procedure) = FID: sub $29, $30, $4
                + push(registers)
                + code(statements)
                + code(expr)
                + pop(registers)
                + add $29, $30, $4
                + jr $31
```

More Advanced

- A lot of optimization is simply predicting values so that we compute them at compile time.
- Many optimizations can be improved upon by *profiling* this information.
- JIT compilers for languages like JavaScript are profiling machines! They enable all the previous optimizations by profiling and recompiling code while they run.

Even More Advanced

- If you want more optimizations, well, take CS444 ☺
- However, a lot of optimizations are very rarified, and so aren't really discussed in academic courses. Once you learn the basics of optimization, you can read the documentation of real compilers. In particular, clang (LLVM) documents this very well.