Compiler Optimizations

Fast! No! Faster! Everyone wants computers to work as fast as possible. But what is possible and what isn't? Computer programs would be much slower if compilers were not already optimizing the code they produce. The word optimization is a misnomer when used for compiler optimizations as it seems to refer to an `optimal` solution. Finding the optimal running time/minimal code is computationally unsolvable for all input programs (CS360 touches upon this). Optimization, as it pertains to compilers, refers to making the program faster, not necessarily optimally fast. This is a very large and interesting space and often bleeds into other fields such as formal verification, checking that a program does/doesn't do something it is/isn't supposed to do.

The typical objective for compiler optimizations is to minimize the execution time of the compiled code. Sometimes, for specialized domains, optimizations might focus on things such as the size of the binary produced. This used to be a key consideration for programs intended for embedded systems and micro-controllers with previous little memory. While not as big a concern anymore, sometimes this is still a consideration.

In CS241, we approximate runtime with the size of the assembled MIPS code. Notice that this is a very coarse metric; a loop that runs a million times will take much longer than straight-line code with an equal number of instructions. However, since our intent is to introduce the reader to the field of compiler optimizations we consider the approach "good enough".

The most important rule of compiler optimizations is simple: an optimization should not change the intended semantics (behaviour) of the program. For `WLP4` the behaviour of the program includes the output it produces, the final value in register 3 and any program crashes that might result in running the program.

In this module, we look at some optimizations that can be applied to the `WLP4` compiler. We also consider some optimizations that are not applicable to `WLP4` but are nevertheless interesting.

1 Constant Folding

We show below the code that would be generated for the expression `1+2`.

```
lis $3
.word 1         ; code(1)
sw $3,-4($30)
sub $30,$30,$4 ; push($3)
lis $3
.word 2         ; code(2)
lw $5,0($30)
add $30,$30,$4 ; pop($5)
add $3,$5,$3   ; 1+2
```

First, recall why we push the value 1 on to the stack. In our code generation module, we had decided that we want to be able to generate code for arbitrarily complex expressions without needing to use too many registers to hold temporary values. We had chosen the simplest approach; all temporary

values are pushed on the stack. We could implement a compiler optimization which recognizes that this is un-necessary. This would lead us to generate the following optimized code:

```
lis $3
.word 1       ; code(1)
add $5,$3,$0  ; move temporary to $5
lis $3
.word 2       ; code(2)
add $3,$5,$3  ; 1+2
```

While this is certainly shorter than our original code, we can do much better. The runtime result of evaluating this expression can actually be computed at compile time. The compiler already knows the values of the two operands of the add expression; the compiler can, at compile-time, perform the addition, and generate code that stores the resulting value it computed:

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

In other words, we have folded the constants. There are some dangers to this; for example integer overflow might be different in the compiler's environment as opposed to the MIPS processor. This would mean that we would have changed the behaviour of the program. However, one could argue that since overflow is undefined behaviour, whatever the compiler ends up doing is acceptable behaviour.

2 Constant Propagation

Consider the `WLP4` code below to the left and the output generated by the compiler on the right.

```
int x = 1;
return x + x;
```

```
lis $3
.word 1
sw $3, -12($29)  ; assume x is at -12 wrt $29
lw $3, -12($29)  ; load operand 1
sw $3, -4($30)
sub $30, $30, $4 ; push operand 1
lw $3, -12($29)  ; load operand 2
lw $5, 0($30)
add $30, $30, $4 ; pop operand 1
add $3, $5, $3   ; x + x
```

The Constant Propagation optimization recognizes that variable is being used as a constant, i.e., from the time it is initialized to the time it is used, its value does not change. If the compiler can determine that a variable has a constant value that is known at compile-time, the compiler can replace the use of the variable with the known constant value. In the example above, the compiler knows that the variable `x` is initialized to the constant 1. So every use of the variable can be replaced with this constant value. This leads to the expression `x+x` changing to `1+1`. At that point, Constant Folding from above would simplify this expression to the value 2.

```
lis $3
.word 1
sw $3, -12($29) ; assume x is at -12 wrt $29
lis $3
.word 2
```

Notice also that if all uses of the variable `x` have been replaced by its constant value, there is no actual need to keep the variable anymore and the above code degrades to just loading the value 2 into register 3.

3 Common Sub-expression elimination

Consider the expression `x+x` from above but assume this time that the compiler cannot ascertain a constant value for `x` which means that constant propagation and folding is not possible. The compiler can still recognize that the two operands have the same value. The code on the left shows what our code generator would produce in general. The code on the right shows the optimization:

```
lw $3, -12($29)  ; load operand 1
sw $3, -4($30)
sub $30, $30, $4 ; push operand 1
lw $3, -12($29)  ; load operand 2
lw $5, 0($30)
add $30, $30, $4 ; pop operand 1
add $3, $5, $3   ; x + x
```

```
lw $3, -12($29) ; load operand 1
add $3, $3, $3  ; x + x
```

Another example would be a more complicated expression such as `(a+b)*(a+b)`. Once the expression `a+b` has been computed and stored in $3, it can be multiplied by itself rather than recomputing the expression and then multiplying.

The optimization is termed common subexpression elimination in that the compiler identifies common subexpressions and eliminates the redundant computation of the same expression multiple times.

4 Dead Code Elimination

Dead code refers to code that will never execute. From a runtime perspective, having dead-code does not effect performance. However, from a size perspective dead code is just dead weight; there is no upside to having it in the generated binary. A compiler can identify dead code and just not generate any output for that part of the parse tree. For example, consider the following code:

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

The innermost block of code is dead; there is no way for `a` to be less than `b` and `b` to be less than `a` at the same time. The compiler can therefore ignore that block of statements. What is more interesting is the nested `if` condition which checks $b < a$. At runtime, when the program is about to evaluate this condition, we already know that `a` must be less than `b`, otherwise why would this code even be executing. Well, then we know that this condition is going to evaluate to false. So, why should we even evaluate it. In other words, we can eliminate the nested `if` statement completely. This does has a minor performance improvement as we have eliminated the evaluation of a condition. Of course, for programs other than the silly example shown above, there is a high chance that the compiler would need something more sophisticated to keep track of what conditions are known to be true/false to achieve this.

It is also worth noting that many of the optimizations already discussed interact with each other. For example, consider the following code:

```
int x = 5;
int y = 10;
if ( y < 2 * x ) {
  .....
}
```

The compiler can propagate constants `x` and `y` to produce the condition $10 < 2 * 5$. It can then perform constant folding to generate the expression $10 < 10$. It can then do condition evaluation to determine that this condition is false. It therefore can infer that the condition and the true block is dead code.

Another example of dead code is variables that are assigned values that are never used. Since the computed value is never used, there is no need to compute it to begin with.

5 Register Allocation

If a variable is permanently stored in a register, its value can be directly used in MIPS instructions. If the variable is in RAM, the value must first be loaded (and later stored). Not only does this increase the number of instructions in the program, but instructions that touch RAM are some of the slowest to execute. Therefore, for efficiency it would be nice if the compiler could store all, or as many as possible, variables in registers. For many `WLP4` procedures, this is indeed possible since MIPS has 32 general purpose registers and our code generation still does not use most of these registers for any specific task. However, given that a procedure might call other procedures and therefore increase the need for additional registers, it might not be possible for a register to continue to hold a value indefinitely. For that purpose, it becomes important to decide which variables deserve to be in registers and which must reside in the RAM. If a variable cannot stay in a register all the time, ideally it makes sense to load it when it is first needed and then to keep it in the register until its last use. However, this too might not always be possible. In such situations there are different heuristics that can be used e.g. most frequently used, most recently needed in the future, etc. Such heuristics require knowing the `live range` of a variable. A live range of a variable is how long the current value of the value lives, i.e., the range from the time of initialization/assignment till the time that a use of the variable uses the value the variable was given. This is often achieved through an analysis called `Live Variable Analysis`.

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

The live ranges can be used to compute `register pressure` by determining how many live ranges intersect with each other. If the register pressure never exceeds the number of available registers, all variables can be assigned to registers. If register pressure is higher, then a smarter allocation scheme is needed that determines which variables should be in registers and which in RAM. Finding

the optimal register allocation is a difficult problem[1].

Recall, we had discussed that all variables reside in memory and are therefore lvalues. If we are to optimize this, and permanently place a variable in a register, that variable would no longer be an lvalue; registers do not have addresses. The compiler can ensure that only variables whose address is never taken are stored in registers. Alternate solutions, where a variable is moved from a register to a RAM location only when its address has been taken, are also possible.

6 Strength Reduction

In the real world, addition is much faster than multiplication. This means that it might be a good idea to replace multiplying with addition. A classic example is that instead of generating code for `n*2`, it is better to generate code for `n+n`, which, with common sub-expression elimination can be done very efficiently:

```
;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
```

The entire code above can be replaced with the single instruction:

```
; load n into $3
add $3, $3, $3
```

7 Peephole optimizations

This is a post code generation optimization, i.e., once the compiler has decided what output it should generate, it does not simply output the instructions. Instead, the instructions are cached in some data structure and then analyzed. The `peephole` refers to the number of instructions the compiler looks at, at a time. The idea is to look for patterns in this "window" of instructions being analyzed and determine if a more efficient pattern is possible. For example, consider the code generated for `a+b`:

```
lw $3, offset_a($29)  ; code for a
push($3)
lw $3, offset_b($29)  ; code for b
pop($5)
add $3, $5,$3  ; a+b
```

Recall that we had chosen to push temporary values on to the stack so that our code generation strategy could handle arbitrarily complex expressions using just one temporary register. A peephole optimization can be implemented which checks for the following:

- A push of a register value (e.g. $3 above)

[1] In fact, optimal register allocation can be shown to be NP-Complete by transforming it to a k-coloring problem.

- Some code that does not update the stack
- A pop into a register (e.g. $5 above)

As long as the code in between the push and pop does not use $5, there was no need to push and then subsequently pop the value. Instead, we could directly move the value for `a` into $5 as shown below:

```
lw $3, offset_a($29)  ; code for a
add $5,$3,$0          ; we know $5 was free
lw $3, offset_b($29)  ; code for b
add $3, $5,$3  ; a+b
```

This optimization has saved a push and a corresponding pop. Of course, our code generation strategy could have been more sophisticated and noticed that this, being a straightforward expression, did not require us to use the stack to begin with.

8 Inlining Procedures

Method inlining is the process of replacing a call to a function with the body of the function. For example, the call to function `foo` below and to the left has been inlined in the code to the right.

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

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

The obvious advantage of method inlining is that code overhead necessary to make the call (such as saving registers and passing arguments) can be avoided. From a runtime efficiency perspective, this can be a win. However, what about the size of the binary? In addition to the instructions saved in setting up the call, if all calls to a function are inlined, there is no need to generate the code for the function. However, whether inlining would actually reduce the size of the generated output depends on the size of the function in comparison to the number of instructions needed to call the function. If the body of the function being inlined is longer than the number of instructions needed to call the function, inlining will increase the size of the output. In other words, inlining short functions makes more sense than inlining longer functions. Of course some functions, such as recursive functions, might be extremely complex to inline.

9 Tail Recursion

Consider the following function that implements factorial using an accumulator:

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

The function shown above is Tail Recursive; the recursive call is the last thing that the function does, i.e., there is no pending work to do when the recursive call returns. An elegant optimization for such tail recursive function calls exists that can reduce the amount of stack memory used. This does not actually reduce the number of instructions the compiler outputs and is therefore a purely

performance optimization.

The optimization is based on the observation that since a tail recursive function has no pending work to do when the recursive call returns, the contents of the stack frame for the function are no longer needed. The optimization is to reuse the current stack frame for the recursive call. To do this, the code generated for the rule `factor` $\rightarrow$ `ID`(`expr`$_1$, ..., `expr`$_n$) would need to change **for a recursive call**. Our pseudo-code for this rule (from the previous Module) is reproduced below:

```
code(factor) = push($29) + push($31) ; caller-saves
             + code(expr_1) + push ($3)
             + code(expr_2) + push ($3)
             + ...
             + code(expr_n) + push ($3)
             + lis $5
             +.word ID   ; where ID is the name of the WLP4 procedure to call
             + jalr $5
             + pop n times ; pop all arguments
             + pop($31) + pop($29) ; caller-saves
```

To reuse the stack frame, we will no longer need to save \$29, \$30 and would no longer use the `jalr` instruction. The pseudo-code for a tail recursive function call would be as follows:

```
code(factor) =
          code(expr_1) + sw $3, offset_of_param_1($29)
        + code(expr_2) + sw $3, offset_of_param_2($29)
        + ......
        + code(expr_n) + sw $3, offset_of_param_n($29)
        + add $30, $29, $4 // reset stack pointer to bottom of stack
        + lis $5
        + .word ID
        + jr $5 ; jr not jalr which is why we do not need to push $29 and $31
```

Notice what this does; instead of pushing the arguments on to the stack, the code replaces the values of the current arguments with the new arguments (remember this is a recursive call). The compiler uses its symbol table to retrieve where these arguments are placed using offsets with respect to register 29 as discussed in the code generation module.

Another important thing to notice is that this specialized pseudo-code for tail recursive function calls does not pop the arguments because they were never pushed by this code. The arguments will be popped by the code that made the first call to this function, i.e., the code that caused the stack frame that we are reusing to be created will pop the arguments, and the stack pointer and the return address.

Notice also that this does not affect how the code for the procedure is generated.

Now, the bad news; we cannot apply this optimization directly in `WLP4` programs due to the syntax restrictions in the language. Recall that `WLP4` only allows one return statement which must be the last statement in the function. This means that the factorial function shown above is actually not a valid `WLP4` function. The closest we could get to would be code that looks as follows:

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

```
  return retVal;
}
```

All hope is not lost! We could do some tree transformations to detect if the input function is in fact tail recursive. Remember that the restriction that a return must only occur once and as the last statement is a restriction imposed by the Context Free Grammar and validated by the parser. Once the parse tree is constructed, we can transform it such that it can have multiple returns if we want. We observe the valid `WLP4` implementation of the factorial function and notice that we had to create a variable (`retVal`) which was assigned the value returned by the recursive call. This variable was then returned by the unique return statement. Our first transformation is that whenever an if-then-else statement is followed by a return statement, we pull the return inside both branches. Based on this, the valid `WLP4` function can be written as the following equivalent but no longer valid function (while we show code here, the transformation is being made to the parse tree).

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

The next transformation is that whenever an assignment statement is followed by a return, we merge it. This transforms our code from above to:

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

At this point, we have the same code that we began this section with. The compiler of course needs to detect that there is indeed a recursive function call and that it is tail recursive.

A generalization to tail call recursion is to reuse the stack frame when a function's last action is any function call (not necessarily a recursive one).

10 Supporting overloading

This section does not really belong in this module. However, it is an interesting topic to consider so we discuss it.

The following code is not valid `WLP4` or C code:

```
int foo(int a) { .... }
int foo(int a, int b) { ..... }
```

Our WLP4 Context Sensitive Analysis would detect the duplicate procedure name and produce an error. But what if we wanted to allow this, i.e., what if we wanted to allow function overloading? (Note that this is legal C++ code which does allow function overloading).

To support function overloading, the Context Sensitive Analysis phase would need to be updated to not simply reject procedures which share the same name. Instead, a more detailed inspection of procedures with the same name would be required where the number and types of parameters would be considered. An error would only occur if two procedures with the same name and the same number and types of parameters is detected. The code generator would also need to be changed. The change would require a guarantee that the labels generated for two overloaded functions are unique. This can be done through a process called `name mangling` where the types of the parameters are encoded as part of the label for the procedure. For example, using the character `i` for the integer type and `p` for a pointer type, we could use a convention where each label is the concatenation of the character `F`, an `i` or `p` for each parameter and then the name of the procedure. Some examples are:

```
int foo() becomes F_foo
int foo(int a) becomes Fi_foo
int foo(int a, int *b) becomes Fip_foo
```

Since there is no standard name mangling convention, compilers use their own strategies. This does pose a problem since it makes it hard, if not impossible, to link code produced by different compilers. But that is actually a good thing. Since different compilers might use different code generation strategies (e.g. calling conventions), we do not really want to successfully link code that might have conflicting conventions.

An interesting exception to this is that while C has no overloading, and therefore no name mangling, we routinely call C code from C++ code which does mangle names since it does support function overloading. The question becomes: how does the compiler know when to look for a mangled name vs. an un-mangled name. The short answer: the programmer must tell the compiler what to do.
As an example, suppose we want to call the C function `foo` from C++. We must write the following code in C++:

```
extern "C" int foo(int n); // look for an un-mangled name when linking
```

Alternately, suppose we want to implement the function `bar` in C++ but want C code to be able to call it. We must write the following code in C++:

```
extern "C" int bar(int* p) {  //do not mangle bar's name
   .....
}
```

Obviously, this means that you cannot overload a function which has been externed "C".