Compiler Optimizations

Compiler Optimizations

- The goal of optimizations is generally to make the code run faster.
    - For the Project 5 bonus component, we're instead concerned about the *size* of the generated code, since this is easier to measure.
    - Size may sometimes be important in specialized domains like embedded systems or micro-controllers which have very limited memory.
- We will just discuss the high-level ideas behind optimizations.
- Implementing these optimizations is easier said than done.
    - Typically, compilers create **intermediate representations** of the program that are somewhere in between a parse tree and assembly code.
    - Many optimizations become easier to implement with an appropriate IR.

Constant Folding

- Our code generator would produce the following code for 1+2:

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

- But 1 and 2 are constants that we know at compile time!

Constant Folding

- Our code generator *could* produce the following code for 1+2:

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

- If we notice that each element of the expression is a constant, we can add the constants *at compile time* and output code for the final value.
- Note that if the expression was 1+x, we would need to know the value of the variable x.
- If the value of x depends on the input to the program, we cannot determine it at compile time.

Constant Propagation

- Sometimes the value of a variable *is* known at compile time:

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

- We can replace x with its known value, so this is equivalent to:

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

- Now we can apply constant folding!

```
int x = 1; return 2;
```

- In this code snippet, x isn’t used anywhere else, so we could even eliminate the variable declaration entirely:

```
return 2;
```

Constant Propagation

- Constant propagation is more difficult than constant folding.

```
int wain(int x, int y) {
  println(x + x); // Constant propagation cannot be applied
  x = 1;
  println(x + x); // Constant propagation can be applied
  x = y;
  return x + x;   // Constant propagation cannot be applied
}
```

- We can only apply it if we know the variable's value does not depend on the input *during the part of the program we're processing*.

Common Subexpression Elimination

- Even if the value of x is not known, there is a simplification we can make when generating code for x+x.
- Here is the "naïve" code (assuming x is at offset 0 from $29):

```
lw $3, 0($29)    ; $3 = x
sw $3, -4($30)   ; push($3)
sub $30, $30, $4
lw $3, 0($29)    ; $3 = x
add $30, $30, $4 ; pop($5)
lw $5, -4($30)
add $3, $5, $3   ; $3 = x + x
```

Common Subexpression Elimination

- Even if the value of x is not known, there is a simplification we can make when generating code for x+x.
- Since we're adding the same variable twice, we can just do this!

```
lw $3, 0($29)    ; $3 = x
add $3, $3, $3   ; $3 = x + x
```

- We can do the same trick with larger expressions, e.g., if we have (a*b-c)+(a*b-c):

```
[block of code that computes a*b-c]
add $3, $3, $3
```

Common Subexpression Elimination

- Can we apply common subexpression elimination to this code?

```
int f(int x) { println(x); return 2*x; }
int wain(int a, int b) {
  return f(a) + f(a);
}
```

- No! CSE must not eliminate side effects.
- If the procedure had no side effects, you technically could, but this complicates your analysis (you now need to determine whether procedures have side effects before applying CSE!)

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.

```
int wain(int a, int b) {
  if (a < b) {
    if (b < a) {
      b = 0;
    } else { }
  } else { b = 0; }
  return a + b;
}
```

- The code inside the innermost if can be ignored.

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.

```
int wain(int a, int b) {
  if (a < b) {
    if (b < a) {
      // dead code
    } else { }
  } else { b = 0; }
  return a + b;
}
```

- Deleting this code has a size benefit, but no real performance benefit.

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.

```
int wain(int a, int b) {
  if (a < b) {
    // if condition eliminated
  } else { b = 0; }
  return a + b;
}
```

- However, since the else condition of the innermost if was empty, we can now simply eliminate the innermost if entirely!

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.
- Dead code elimination interacts with other optimizations.

```
int wain(int x, int y) {
  int releaseVersion = 0;
  if (releaseVersion == 1) {
    x = 1;
  } else { x = 0; }
  return x * y;
}
```

- Normally, we can’t apply constant propagation to x in the return.

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.
- Dead code elimination interacts with other optimizations.

```
int wain(int x, int y) {
  int releaseVersion = 0;
  x = 0;
  return x * y;
}
```

- Constant propagation + dead code elimination results in this.
- Now constant propagation can be used on x as well.

Dead Code Elimination

- Sometimes the compiler can determine that certain code will never execute, and eliminate this code.
- Dead code elimination interacts with other optimizations.

```
int wain(int x, int y) {
  return 0;
}
```

- The program could ultimately be simplified to this if the compiler uses the rule that anything times zero is zero.
- So DCE can allow constant propagation to occur. Conversely, constant propagation can allow the compiler to prove code is dead.

Register Allocation

- We repeatedly ran into the issue that for sufficiently complicated code, it is not possible to store all values in registers.
  - If there are more variables than registers, some must be stored on the stack.
  - The same was true for temporary values of expressions.
  - The same was true for arguments passed to procedures.
- Our solution was to simply put *everything* on the stack because this makes generating code simpler and more consistent.
- But using registers for storage is much faster than using RAM.
  - When using RAM, we need extra sw/lw instructions. Not only does this increase the number of instructions, but these instructions are slow.

Register Allocation

- Real-world compilers will try to use registers as much as possible.
- We say a variable is **live** if the current value of the variable will be used at a later point in the program.
  - We can apply this definition to e.g. temporary expression values as well.
- Ideally, a variable should be in a register if and only if it is live.
- When the variable is no longer live, it should be removed from its register to make room to put other variables or values in registers.
- If too many variables or values are live at the same time, we have to choose which ones to put in RAM vs. registers.

Register Allocation: Live Ranges

```
x = 3;
y = 10;
println(x);
z = 7;
y = y - x;
y = y - z;
println(z);
return z;
```

Register Allocation: Live Ranges

```
1  x = 3;
2  y = 10;
3  println(x);
4  z = 7;
5  y = y - x;
6  y = y - z;
7  println(z);
8  return z;
```

- x becomes live on line 1, and is last used on line 5.

Register Allocation: Live Ranges

```
1  x = 3;
2  y = 10;
3  println(x);
4  z = 7;
5  y = y - x;
6  y = y - z;
7  println(z);
8  return z;
```

```
Live Ranges:
x: Lines 1 to 5
```

- y becomes live on line 2, and is last used on line 6.

Register Allocation: Live Ranges

```
1   x = 3;
2   y = 10;
3   println(x);
4   z = 7;
5   y = y - x;
6   y = y - z;
7   println(z);
8   return z;
```

```
Live Ranges:
x: Lines 1 to 5
y: Lines 2 to 6
```

- z becomes live on line 4, and is last used on line 8.

Register Allocation: Live Ranges

```
1  x = 3;
2  y = 10;
3  println(x);
4  z = 7;
5  y = y - x;
6  y = y - z;
7  println(z);
8  return z;
```

```
Live Ranges:
x: Lines 1 to 5
y: Lines 2 to 6
z: Lines 4 to 8
```

- Notice on lines 4 to 5, all three variables are live. If we only had two registers available, we would need to put one variable in RAM.

Register Allocation: Live Ranges

```
1  x = 3;
2  y = 10;
3  println(x);
4  z = 7;
5  y = y - x;
6  y = y - z;
7  println(z);
8  return z;
```

Live Ranges:
x: Lines 1 to 5
y: Lines 2 to 6
z: Lines 4 to 8

- We can use live ranges to construct a graph indicating which ranges overlap, and use graph coloring algorithms to allocate registers.

Register Allocation

- If the live range graph can be k-colored, where k is the number of available registers, we can allocate all variables to registers.
- Graph coloring can be slow (it is a NP-complete problem) so it is often approximated.
- If we cannot allocate all variables to registers, we need to decide which ones to “spill” into RAM.
  - No easy solution to this – heuristics are often used.
- Aside: What if the address-of operator is used on a variable?
  - We can’t take the address of a register, so this variable must go in RAM.

Simple Register Allocation

- For the Project 5 bonus, you can get significant gains by just implementing a basic register allocator.
  - Our recommended code generation strategy uses the stack heavily, and each push/pop takes two instructions.
  - Optimizations that eliminate pushes/pops or decrease the number of instructions for a push/pop are very effective on Project 5.
- Instead of a complex live range analysis, you can allocate variables and temporaries to registers on a "first-come, first-served" basis.
- In your code generator, keep track of which registers are free/unused and which are allocated to a variable or temporary value.

Simple Register Allocation

- Modify your offset table so that there are two kinds of "variable locations": offsets from the frame pointer, or registers.
- Allocate non-parameter local variables in registers whenever possible.
- Also allocate registers for temporary values in expressions, and return them to the "free registers" list when done.
- Procedures complicate things. Procedure calls (particularly recursive calls) should not mess up the values in allocated registers.
- Can you pass (some) parameters in registers? Probably, but this changes the calling convention.

Strength Reduction

- This optimization involves replacing costly operations with equivalent faster operations.
- For example, multiplication is slower than addition.
  - n * 2 could be replaced with n + n (or a left bit shift of n).
  - (x + y) * 2 could be replaced with (x + y) + (x + y), which can then be optimized further using common subexpression elimination!
- A more complex version involves optimizing loops which perform expensive operations involving the loop counter.
  - A loop that does multiplication on every iteration could potentially be transformed into a loop which computes the same thing with addition.

Peephole Optimization

- This optimization happens after code generation is finished.
- Instead of directly outputting the generated code, the code is placed in a data structure and subject to further analysis.
- The analysis tries to find sequences of instructions that can be replaced with simpler sequences.
- For example, I wrote a code generator that outputs a lot of stuff like:

```
add $3, $1, $0 ; $3 = a
add $7, $3, $0 ; copy $3 to temporary register
```

- Peephole optimization could change this to **add $7, $1, $0**. This might be easier than making the code generation step itself "smarter".

Inlining Functions

- This optimization consists of replacing a function call with the body of the function itself.

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

- This is equivalent to:

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

- This removes the overhead of doing a function call.
- As a *size* optimization, it maybe not be effective unless the function is shorter than the number of instructions needed to call it.

Tail Recursion

- A recursive function call is in *tail position* if it is the last thing the function executes before returning.
- In this case, what happens normally is:
  - The recursive call happens, and pushes local variables etc. to the stack.
  - The recursive call finishes, pops from the stack, then returns.
  - The original call finishes, pops from the stack, then returns.
- Tail call optimization is based on the observation that in this situation, the recursive call can *reuse the stack frame* of the original call instead of pushing its own stack frame, saving lots of stack space.
- The original call pops the reused stack frame.