Code optimization

Handout written by Maggie Johnson and revised by me.

Optimization is the processing of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. It is a really a misnomer that the name implies you are finding a "optimal" solution— in truth, optimization aims to improve, not perfect, the result.

Optimization is the field where most compiler research is done today. Although there are still some advanced problems left, the tasks of the front-end (scanning, parsing, semantic analysis) are well understood, and unoptimized code generation is relatively straightforward. Optimization, on the other hand, still retains a sizable measure of mysticism. High-quality optimization is more of an art than a science. Compilers for mature languages aren't judged by how well they parse or analyze the code—you just expect it to do it right with a minimum of hassle— but instead by the quality of the object code they produce.

Many optimization problems are NP-complete, and for most examples here, it is possible to come up with a case where the algorithm fails to produce better code or perhaps even makes it worse. However, these algorithms tend to do rather well overall.

It's worth reiterating here that efficient code starts with intelligent decisions by the programmer. No one expects a compiler to be smart enough to replace BubbleSort with Quicksort. If a programmer uses a lousy algorithm, no amount of optimization can make it zippy. In terms of big-O, a compiler can only make improvements to constant factors. But, all else being equal, you want an algorithm with low constant factors.

First let me note that you probably shouldn't try to optimize the way we will discuss today in your favorite high-level language. Consider the following two code snippets which each walk through an array and set every element to one. Which one is faster?

```
int arr[10000];

void Binky() {
   int i;
   for (i=0; i < 10000; i++)
       arr[i] = 1;
}
```

```
int arr[10000];

void Winky() {
  register int *p;
  for (p = arr; p < arr + 10000; p++)
      *p = 1;
}
```

You will invariably encounter people who think the second one is faster. And they are probably right….if using a compiler without optimization. But, many modern compilers emit the same object code for both, by use of clever techniques (in particular, this one is called "loop-induction variable elimination") that work particularly well on idiomatic usage. The moral of this story is that most often you should write code that is easier to understand and let the compiler do the optimization.

Correctness above all!

If may seem obvious, but it bears repeating that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of little value. It is expected that the unoptimized and optimized variants give the same output for all inputs. This may not hold for an incorrectly written program (e.g., one that uses an uninitialized variable).

When and in what representation to optimize

There are a variety of tactics for attacking optimization. Some techniques are applied to the intermediate code, to streamline, rearrange, compress, etc. in an effort to reduce the size of the abstract syntax tree or shrink the number of TAC instructions. Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to spill, and the like. And still other optimizations may occur after final code generation, attempting to re-work the assembly code itself into something more efficient.

Optimization can be very complex and time-consuming, it often involves multiple sub-phases, some of which are applied more than once. Most compilers allow optimization to be turned off to speed up compilation (gcc has specific flags to turn on and off individual optimizations even).

Control-flow analysis

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation. The compiler has done an enormous amount of analysis, but it still doesn't really know how the program does what it does. In control-flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

As an example, here is Decaf code and its corresponding TAC that computes fibonacci numbers:

```
int fib(int base)
{
   int result;

  if (base <= 1) {
      result = base;
   } else {
      int i;
      int f0;
      int f1;

      f0 = 0;
      f1 = 1;
      i = 2;
      while (i <= base) {
          result = f0 + f1;
          f0 = f1;
          f1 = result;
          i = i + 1;
      }
   }
   return result;
}
```

```
_fib:
        BeginFuncWithParams base ;
        _tmp0 = 1 ;
        _tmp1 = base < _tmp0 ;
        _tmp2 = base == _tmp0 ;
        _tmp3 = _tmp1 || _tmp2 ;
        IfZ _tmp3 Goto _L0 ;
        result = base ;
        Goto _L1 ;
_L0:
        _tmp4 = 0 ;
        f0 = _tmp4 ;
        _tmp5 = 1 ;
        f1 = _tmp5 ;
        _tmp6 = 2 ;
        i = _tmp6 ;
_L2:
        _tmp7 = i < base ;
        _tmp8 = i == base ;
        _tmp9 = _tmp7 || _tmp8 ;
        IfZ _tmp9 Goto _L3 ;
        _tmp10 = f0 + f1 ;
        result = _tmp10 ;
        f0 = f1 ;
        f1 = result ;
        _tmp11 = 1 ;
        _tmp12 = i + _tmp11 ;
        i = _tmp12 ;
        Goto _L2 ;
_L3:
_L1:
        Return result ;
        EndFunc ;
```

Control-flow analysis begins by constructing a *control-flow graph*, which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into *basic blocks*. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively if the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it.

A basic block begins in one of several ways:

- the entry point into the function
- the target of a branch (in our example, any label)
- the instruction immediately following a branch or a return

A basic block ends in any of the following ways:

- a jump statement
- a conditional or unconditional branch
- a return statement

Using the rules above, let's divide the Fibonacci TAC code into basic blocks:

```
_fib:
     BeginFuncWithParams base ;
     _tmp0 = 1 ;
     _tmp1 = base < _tmp0 ;
     _tmp2 = base == _tmp0 ;
     _tmp3 = _tmp1 || _tmp2 ;
     IfZ _tmp3 Goto _L0 ;
```

```
     result = base ;
     Goto _L1;
```

```
_L0:
     _tmp4 = 0 ;
     f0 = _tmp4 ;
     _tmp5 = 1 ;
     f1 = _tmp5 ;
     _tmp6 = 2 ;
     i = _tmp6 ;
```

```
_L2:
     _tmp7 = i < base ;
     _tmp8 = i == base ;
     _tmp9 = _tmp7 || _tmp8 ;
     IfZ _tmp9 Goto _L3 ;
```

```
     _tmp10 = f0 + f1 ;
     result = _tmp10 ;
     f0 = f1 ;
     f1 = result ;
     _tmp11 = 1 ;
     _tmp12 = i + _tmp11 ;
     i = _tmp12 ;
     Goto _L2 ;
```

```
_L3:
```

```
_L1:
     Return result ;
     EndFunc
```

Now we can construct the control-flow graph between the blocks. Each basic block is a node in the graph, and the possible different routes a program might take are the connections, i.e. if a block ends with a branch, there will be a path leading from that block to the branch target. The blocks that can follow a block are called its successors. There may be multiple successors (e.g. if ends with a conditional branch) or just one. Similarly the block may many, one, or no predecessors.

Connect up the flow graph for Fibonacci basic blocks given above. What does an if-then-else look like in a flow graph? What about a loop?

You probably have all seen the gcc warning or javac error about: "Unreachable code at line XXX." How can the compiler tell when code is unreachable?

Local optimizations

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. We'll consider some of the more common local optimizations as examples.

Constant folding

Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant. In its simplest form, it involves determining that all of the operands in an expression are constant-valued, performing the evaluation of the expression at compile-time, and then replacing the expression by its value. If an expression such as `10+2*3` is encountered, the compiler can compute the result at compile-time (16) and emit code as if the input contained the result rather than the original expression. Similarly, constant conditions, such as an conditional branch `if a < b goto L1 else goto L2` where `a` and `b` are constant can be replaced by a `Goto L1` or `Goto L2` depending on the truth of the expression evaluated at compile-time.

The constant expression had to be evaluated at least once, but if the compiler does it, it means you don't have to do it again as needed during runtime. One thing to be careful about is that the compiler must obey the grammar and semantic rules from the source language that apply to expression evaluation, which may not necessarily match the language you are writing the compiler in. (For example, if you were writing an APL compiler, you would need to take care that you were respecting its Iversonian precedence rules). It should also respect the expected treatment of any exceptional conditions (divide by zero, over/underflow).

Consider the Decaf code on the far left and its unoptimzed TAC translation in the middle, which is then transformed by constant-folding on the far right:

```
a = 10 * 5 + 6 - b;     _tmp0 = 10 ;                  _tmp0 = 56 ;
                        _tmp1 = 5 ;                   _tmp1 = _tmp0 – b ;
                        _tmp2 = _tmp0 * _tmp1 ;       a = _tmp1 ;
                        _tmp3 = 6 ;
                        _tmp4 = _tmp2 + _tmp3 ;
                        _tmp5 = _tmp4 – b;
                        a = _tmp5 ;
```

Constant-folding is what allows a language to accept constant expressions where a constant is required (such as a case label or array size) as in these C language examples:

```
int arr[20 * 4 + 3];

switch (i) {
   case 10 * 5: ...

}
```

In both snippets shown above, the expression can be resolved to an integer constant at compile time and thus, we have the information needed to generate code. If either expression involved a variable, though, there would be an error. How could you re-write the grammar to allow the Decaf grammar constant folding in case statements? This situation is a classic example of the gray area between syntactic and semantic analysis.

Constant propagation

If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable. Consider this section from our earlier Fibonacci example. On the left is the original, on the right is the improved version after constant propagation, which saves three instructions and removes the need for three temporary variables:

```
_tmp4 = 0 ;
f0 = _tmp4 ;
_tmp5 = 1 ;
f1 = _tmp5 ;
_tmp6 = 2 ;
i = _tmp6 ;
```

```
f0 = 0 ;
f1 = 1 ;
i = 2 ;
```

Constant propagation is particularly important in RISC architectures because it moves integer constants to the place they are used. This may reduce both the number of registers needed and instructions executed. For example, MIPS has an addressing mode that uses the sum of a register and a constant offset, but not one that uses the sum of two registers. Propagating a constant to such an address construction can eliminate the extra add instruction as well as the additional registers needed for that computation. Here is an example of this in action. First, we have the unoptimized version, TAC on left, MIPS on right:

```
_tmp0 = 12 ;
_tmp1 = arr + _tmp0 ;
_tmp2 = *(_tmp1) ;
```

```
li $t0, 12 ;
lw $t1, -8($fp)
add $t2, $t1, $t0
lw $t3, 0($t2)
```

A bit of constant propagation and a little rearrangement on the store instruction, cuts the number of registers needed from 4 to 2 and the number of instructions likewise in the optimized version:

```
_tmp0 = *(arr + 12) ;
```

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```

Algebraic simplification and reassociation

Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions. Reassociation refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimizations such as constant-folding or loop-invariant code motion.

The most obvious of these are the optimizations that can remove useless instructions entirely via algebraic identities. The rules of arithmetic can come in handy when looking for redundant calculations to eliminate, such as these below, which allow you to replace an expression on the left into a simpler equivalent form on the right:

```
x+0  = x
0+x  = x
x*1  = x
1*x  = x
0/x  = 0
x-0  = x
b && true  = b
b && false = false
b || true  = true
b || false = b
```

The use of algebraic rearrangement can be used to restructure an expression to group constants to allow constant-folding or enable common sub-expression elimination and so on. Consider the Decaf code on the far left, unoptimized TAC in middle, and rearranged and constant-folded TAC on far right:

```
b = 5 + a + 10 ;      _tmp0 = 5 ;                  _tmp0 = 15 ;
                      _tmp1 = _tmp0 + a ;          _tmp1 = a + _tmp0 ;
                      _tmp2 = _tmp1 + 10 ;         b = _tmp1 ;
                      b = _tmp2 ;
```

Operator strength reduction

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below) which operations are the most and least expensive, assuming f is a float and i is an int? (Trick question: it may differ for differing architectures— you need to know your target machine to optimize well!)

```
a) i*2  =  2*i  =  i+i  =  i<<2
b) i/2  =  (int)(i*0.5)
c) 0-1  =  -i
d) f*2  =  2.0 * f  = f + f
e) f/2.0 = f*0.5
```

Strength reduction is often performed as part of *loop-induction variable elimination*. An idiomatic loop to zero all the elements of an array might look like this in Decaf and its corresponding TAC:

```
while (i < 100) {                    L0: _tmp2 = i < 100;
    arr[i] = 0                           IfZ _tmp2 Goto _L1 ;
    i = i + 1;                           _tmp4 = 4 * i ;
}                                        _tmp5 = arr + _tmp4 ;
                                         *_tmp5 =  0 ;
                                         i = i + 1 ;
                                     L1:
```

Each time through the loop, we are multiplying i by 4 (the element size) and adding to the array base. Instead, we could be maintaining the address to the current element and just adding 4 each time instead:

```
    _tmp4 = arr ;
L0: _tmp2 = i < 100;
    IfZ _tmp2 Goto _L1 ;
```

```
    *_tmp4 =  0;
    _tmp4 = _tmp4 + 4;
    i = i + 1 ;
L1:
```

This eliminates the multiplication entirely and reduces the need for an extra temporary. By re-writing the loop termination test in terms of arr, we could remove the variable i entirely and not bother tracking and incrementing it at all.

Copy propagation

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment `a = b` in our instruction stream, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between). Given the way we generate TAC code, this is a particularly valuable optimization since it is able to eliminate a large number of instructions that only serve to copy values from one variable to another.

The code on the left makes a copy of `tmp1` in `tmp2` and a copy of `tmp4` in `tmp3`. In the optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
tmp2 = tmp1 ;                    tmp3 = tmp1 * tmp1 ;
tmp3 = tmp2 * tmp1;              tmp5 = tmp3 * tmp1 ;
tmp4 = tmp3 ;                    c = tmp5 + tmp3 ;
tmp5 = tmp3 * tmp2 ;
c = tmp5 + tmp4 ;
```

We can also drive this optimization in a "backwards" direction, where we can recognize that the original assignment made to a temporary can be eliminated in favor or direct assignment to the final goal:

```
tmp1 = LCall _Binky() ;          a = LCall _Binky();
a = tmp1;                        b = LCall _Winky();
tmp2 = LCall _Winky() ;          c = a * b ;
b = tmp2 ;
tmp3 = a * b ;
c = tmp3 ;
```

Dead code elimination

If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream. So if we have

```
tmp1 = tmp2 + tmp3 ;
```

and `tmp1` is never used again, we can eliminate this instruction altogether. However, we have to be a little careful about making assumptions, for example, if `tmp1` holds the result of a function call:

```
tmp1 = LCall _Binky(a, b);
```

Even if `tmp1` is never used again, we cannot eliminate the instruction because we can't be sure that called function has no side-effects. Dead code can occur in the original source program but is more likely to have resulted from some of the optimization techniques run previously.

Common sub-expression elimination

Two operations are *common* if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An expression is *alive* if the operands used to compute the expression have not been changed. An expression that is no longer alive is *dead.*

```
main()
{
    int x, y, z;

    x = (1+20)* -x;
    y = x*x+(x/y);
    y = z = (x/y)/(x*x);
}
```

straight translation:

```
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
```

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined? Here is an optimized version, after constant folding and propagation and elimination of common sub-expressions:

```
tmp2 = -x ;
x = 21 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
z = tmp5 / tmp3 ;
y = z ;
```

Global optimizations and data-flow analysis

So far we were only considering making changes within one basic block. With some additional analysis, we can apply similar optimizations across basic blocks, making them *global* optimizations. It's worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Inter-procedural analysis is an even larger task, one not even attempted by a lot of compilers.

The additional analysis the optimizer must do to perform optimizations across basic blocks is called *data-flow analysis*. Data-flow analysis is much more complicated than control-flow analysis, and we can only scratch the surface here, but were you to take CS243 (a wonderful class, in my opinion) you will get to delve much deeper!

Let's consider a global common sub-expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be *available* at that point. Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis.

Each block is a node in the *flow graph* of a program. The successor set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is *defined* at the point where it is assigned a value and *killed* when one of its operands is subsequently assigned a new value. An expression is *available* at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed.

`avail[B]` = set of expressions available on entry to block B
`exit[B]` = set of expressions available on exit from B

`avail[B]` = $\cap_{x \in pred[B]}$ `exit[x]` (i.e. B has available the intersection of the exit of its predecessors)

`killed[B]` = set of the expressions killed in B
`defined[B]` = set of expressions defined in B

```
exit[B] = avail[B] - killed[B] + defined[B]
avail[B] = ∩ (avail[x] - killed[x] + defined[x])
           x  pred[B]
```

Here is an algorithm for global common sub-expression elimination:

1) First, compute defined and killed sets for each basic block (this does not involve any of its predecessors or successors).
2) Iteratively compute the avail and exit sets for each block by running the following algorithm until you hit a stable fixed point:
   a) Identify each statement `s` of the form `a = b op c` in some block B such that `b op c` is available at the entry to B and neither `b` nor `c` is redefined in B prior to `s`.
   b) Follow flow of control backward in the graph passing back to but not through each block that defines `b o c`. The last computation of `b o c` in such a block reaches `s`.
   c) After each computation `d = b op c` identified in step 2a, add statement `t = d` to that block where `t` is a new temp.
   d) Replace `s` by `a = t`.

Try an example to make things clearer:

```
main:
   BeginFuncWithParams ;
   b = a + 2 ;
   c = 4 * b ;
   tmp1 = b < c;
   ifNZ tmp1 goto L1 ;
   b = 1 ;
L1:
   d = a + 2
EndFunc
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block. Then find an expression available in a block and perform step 2c above. What common sub-expression can you share between the two blocks?

What if the above code were:

```
main:
   BeginFuncWithParams ;
   b = a + 2 ;
   c = 4 * b ;
   tmp1 = b < c ;
   IfNZ tmp1 Goto L1 ;
   b = 1 ;
   z = a + 2 ; <========= an additional line here
L1:
   d = a + 2
   EndFunc ;
```

Code motion

Code motion (also called *code hoisting*) unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation. The most common form of code motion is *loop-invariant* code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. What statements inside the following TAC code can be moved outside the loop body?

```
L0:
   tmp1 = tmp2 + tmp3 ;
   tmp4 = tmp4 + 1 ;
   LCall _PrintInt(tmp4) ;
   tmp6 = 10 ;
   tmp5 = tmp4 == tmp6 ;
   IfZ Goto L0 ;
```

We have an intuition of what makes a loop in a flowgraph, but here is a more formal definition. A loop is a set of basic blocks which satisfies two conditions:

1. All are *strongly connected*, i.e. there is a path between any two blocks.
2. The set has a unique *entry point*, i.e. every path from outside the loop that reaches any block inside the loop enters through a single node. A block n *dominates* m if all paths from the starting block to m must travel through n. Every block dominates itself.

For loop L, moving invariant statement s in block B which defines variable v outside the loop is a safe optimization if:

1. B dominates all exits from L
2. No other statement assigns a value to v
3. All uses of v inside L are from the definition in s.

Loop invariant code can be moved to just above the entry point to the loop.

Machine optimizations

In final code generation, there is a lot of opportunity for cleverness in generating efficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

Register allocation

One machine optimization of particular importance is register allocation, which for all architectures, is perhaps the single most effective optimization there is. Registers are the fastest kind of memory available, but as a resource, they are scarce. The problem is how to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus.

Our Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them up and chooses ones to spill fairly arbitrarily. A much more effective strategy would be to consider which variables are more heavily in demand and keep those in registers and spill those that are no longer needed or won't be needed until much later.

Although there are numerous register allocation algorithms out there, the most common is called “register coloring”, after the central idea to view register allocation as a graph coloring problem. If we have 8 registers, then we try to color a graph with eight different colors. The graph’s nodes are made of “webs” and the arcs are determined by calculating interference between the webs. A web represents a variable’s definitions—places where it is assigned a value (as in x = ….)—and the possible different uses of those definitions (as in y = x + 2). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition

reaches a use, there is an arc between the two nodes. If two portions of a variable's definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i = 10;
j = 20;
x = i + z
y = j + a;
```

We say that `i` interferes with `j` because at least one pair of `i`'s definitions and uses is separated by a definition or use of `j`. More formally, `i` and `j` are "alive" at the same time. A variable is alive between the time it has been defined and that definition's last use, after which the variable is dead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupy the same register.

Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register). You may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimal algorithm. Here is a simplified version of what is commonly used in modern register allocators:

1. Find the node with the least neighbors. (Break ties arbitrarily.)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty.
4. Now, rebuild the graph as follows:
   a. Take the top node off the stack and reinsert it into the graph
   b. Choose a color for it based on the color of any of its neighbors presently in the graph, rotating colors in case there is more than one choice.
   c. Repeat a and b until the graph is either completely rebuilt, or there is no color available to color the node.

If we get stuck, then the graph may not be r-colorable, we could try again with a different heuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable to memory.

Instruction scheduling

Another extremely important area of optimization in the final code generator is instruction scheduling. Because many machines, including most RISC architectures, expose some amount of pipelining to the user, the compiler writer must take to effective harness that capability with judicious ordering of instructions.

Consider an architecture like MIPS where each instruction is issued in one cycle, but some take multiple cycles to complete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reach its destination, but an instruction can be placed in the "delay slot" after a branch and executed in that slack time. On the left is one arrangement of a set of instructions that requires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the second and third slots while the load completes and has a dead cycle after the branch because the delay slot holds a noop. On the right, a more favorable rearrangement of the same instructions will execute in 5 cycles with no dead cycles.

```
lw $t2, 4($fp)
lw $t3, 8($fp)
noop
add $r4, $r2, $r3
subi $r5, $r5, 1
goto L1
noop
```

```
lw $t2, 4($fp)
lw $t3, 8($fp)
subi $r5, $r5, 1
goto L1
add $r4, $r2, $r3
```

Peephole optimizations

Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine-dependent code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination of load of a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations.

Much of the work done in peephole optimization can be though of as find-replace activity, looking for certain idiomatic patterns at the single or 2-3 instruction level than can be replaced by more efficient alternatives. For example, MIPS has instructions that can add a small integer constant to the value in a register without loading the constant into a register first, so the sequence on the left can be replaced with that on the right:

```
li $t0, 10
lw $t1, -8($fp)
add $t2, $t1, $t0
sw $t1, -8($fp)
```

```
lw $t1, -8($fp)
addi $t2, $t1, 10
sw $t1, -8($fp)
```

What would you replace the following sequence?

```
lw $t0, -8($fp)
sw $t0, -8($fp)
```

What about this one?

```
mul $t1, $t0, 2
```

Optimization soup

You might wonder about the interactions between the various optimization techniques. Some transformations may expose possibilities for others, and even the reverse is true, one optimization may obscure or remove possibilities for others. Algebraic rearrangement may allow for common subexpression elimination or code motion. Constant folding usually paves the way for constant propagation and then it turns out to be useful to run another round constant-folding and so on. How do you know you are done? You don't!

As one compiler textbook author (Pyster) puts it:

> "Adding optimizations to a compiler is a lot like eating chicken soup when you have a cold. Having a bowl full never hurts, but who knows if it really helps. If the optimizations are structured modularly so that the addition of one does not increase compiler complexity, the temptation to fold in another is hard to resist. How well the techniques work together or against each other is hard to determine."

Bibliography

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

R. Mak, Writing Compilers and Interpreters. New York, NY: Wiley, 1991.

S. Muchnick, Advanced Compiler Design and Implementation . San Francisco, CA: Morgan Kaufmann, 1997.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.