6.035 Project Optimizations

Nicolaas Kaashoek, Endrias Kahssay, Tony Wang

1 SSA Form

To start with, we converted our CFG from what we had in project 3 to Static Single Assignment (SSA) form. In SSA form each variable can only be defined once. At join points in the program, ϕ-instructions are inserted to represent the joining of definitions that originally corresponded to the same variable. Assembly generation does not occur on an SSA-d CFG, so we destruct the SSA before this generating any assembly.

SSA form is used by almost all modern compilers, and makes later optimizations to the code much much simpler. Specifically, because there is only one def per variable we don't have to worry about variables getting redefined, which simplifies many dataflow-analysis optimizations.

Also,when a CFG is in SSA form, register allocation can be solved optimally in polynomial time, as the graph becomes chordal. While we lose the optimal solution when converting back to non-SSA, it is still reasonably accurate. We believe this trade-off is worth it based on how much simpler it made later optimizations.

In order to convert to SSA we wrote a variety of helper functions that became useful later as well, such as determining the dominator tree of our CFG, as well as functions that maximize a CFG and assert that def-use chains are valid (defs before uses).

2 Optimizations

This section details all the optimizations we added to our compiler.

2.1 Common Sub Expression Elimination (CSE)

Originally, we wrote Common Sub Expression elimination exactly as described in lecture. However, we found that there were some cases that became difficult to handle. For instance, if a common sub-expression is evaluated in both branches of an if statement, it is hard to recognize where the result of the expression is stored because of SSA form. For this reason, we went with a different approach.

Instead of doing a bit-vector analysis, we instead find all expression that are evaluate more than once and track the block in the CFG where they are used. We then use the dominator tree to find the least common ancestor of all these blocks and move the evaluation of the expression to this block. We then replace all previous evaluations with mov instructions that place the destination of the evaluation into the old destinations. This ensures that later uses of that expression don't break.

This method works because we move the evaluation of the expression to a dominator of each of the other evaluations. This ensure that it will always be evaluated when the program gets to where the old evaluations were.

This optimization generates redundant code, but this is eliminated through a combination of copy propagation and dead code elimination in a later optimization pass.

2.2 Copy/Constant Propagation/Folding

SSA form makes this particular optimization very straightforward. Because each variable in the CFG can only ever be defined once, we just scan the CFG for every def instruction and note the source and dest of

that instruction. Then, whenever we see that destination again, we can replace it with the source of the def instruction. We make sure to propagate chains as well, so if we have something like `A = B; C = A; D = C` we can turn this into `A = B; C = B; D = B;`. Again, this generates redundant code that is remove later with dead code elimination.

2.3 Dead Code Elimination (DCE)

Like CSE, we originally approached DCE with the strategy described in class. However, some quick research revealed that DCE in SSA is done using a different method. Similar to the idea of a mark and sweep garbage collector, we begin by marking every critical instruction as critical. These instructions are put onto a queue, which we pop from until it is empty. Whenever we pop from the queue, we find all variables that instruction uses and add the defs for those variables the queue. This is made straightforward because in SSA each variable is only defined once.

This algorithm terminates because there is a clear upper bound, as once all instructions have been marked there is no way to proceed any further. Additionally, we are guaranteed not to delete anything critical, as we move from the critical instructions outward, ensuring that anything they rely on will also get marked. This only works in SSA form however, as we know each variable has only one definition.

2.4 Register Allocation

We perform SSA based register allocation closely following `http://compilers.cs.ucla.edu/fernando/projects/soc/reports/short_tech.pdf`. To give some context, our CFG uses virtual registers which must get mapped to physical registers via register allocation. Our register allocation algorithm determines for each virtual register whether it gets spilled or what physical register it gets mapped to. No range splitting is performed.

The register allocation roughly happens in the following stages for each function:

1. We first convert into CSSA form, in which the live ranges of phi webs do not overlap. During this conversion we buffer each phi instructions with moves to control its live range. We first buffer every phi instruction and then selectively eliminate buffers while the phi webs do not overlap with respect to liveness.

2. Next we compute the live ranges of variables, constructing a data structure that tells us for every program point how many registers are live.

3. With the computed live ranges and the handy data structure mentioned above, we look at every program point where the register pressure is too high and mark virtual registers as spilled until the register pressure is reduced sufficiently. We use Chaitin's of $\frac{\text{cost}}{\text{degree}}$ to determine what registers to spill, where the cost is the sum of $10^{\text{loop nesting depth}}$ for every location where the virtual register is used or assigned.

4. Once we have marked registers as spilled, SSA allows us to color the graph in perfect elimination order as described in Hack et. al. During coloring, we preference registers to be colored the same as registers which they are moved to, again using a weighting heuristic proportional to the exponential of the loop nesting depth.

2.4.1 SSA Destruction

As part of register allocation, we also perform SSA destruction. This just consists of replacing phi functions with an equivalent set of move instructions.

2.5 Loop Recognition

We recognize loops by looking for back edges as described in class. If two loops share the same header, we merge the loops.

2.6 Loop Invariant Code Motion

For loop invariant code motion we follow the technique discussed in class. We begin by locating any statement which relies only on constants or definitions from outside the loop. We mark these as invariant and then proceed to look at all the other instructions in the loop, progressively marking those which depend only on invariant statements as invariant themselves.

Once there are no more statements to mark as invariant, we create a new node in the CFG representing the preheader, and update all previous pointers to the loop header to point instead to the preheader. The invariant code is then moved to the preheader, at which point we run CFG maximization to collapse the CFG in case the preheader and its parent can be merged.

2.7 Loop Unrolling

Loop unrolling is an optimization done to convert a loop in the CFG into a repeated iterations of the same set of blocks. This opens up more optimization opportunities as the compiler is given more code to work with. Additionally, induction variabes can have their values precomputed in this case, providing some speed up.

We unroll loops that are executed a constant number of times in the CFG. To do this, we begin by recognizing base induction variables using the techniques discussed in class. Once we recognize these variables we can expand the loop however many times is necessary. Whenever we expand the loop we take the def statements for the induction variables and replace them with precomputed values.

For instance, in the loop

```
int i;
for (i = 0; i < 10; i++) {
    // Some code here
}
```

we can convert this into

```
int i;
i = 0;
\\ Code
i = 1
\\ Code
i = 2
\\ And so on and so forth
```

This then allows us to perform optimizations like copy propagation now that `i` has a constant value.

2.8 Algebraic Simplification

We perform basic algebraic simplification. We recognize addition and subtraction by 0, multiplication and division by 1. We also try to optimize division instructions in the assembly. When profiling our compiler, we found that division instructions incur a significant overhead. To address this, we convert them to multiplication and addition by magic numbers. The guide for doing so can be found here: HackersDelight

2.9 Assembly Level Optimizations

In our assembly from Project 3, we generated assembly pretty lazily. We modified our generation of assembly to eliminate most of this redundancy. For instance, we remove cases where we moved a value from `A` into `B` and then back again. Additionally, when we use an instruction that can take two registers as its arguments, we removed temporary registers when possible. Also we allow each function to use all possible registers, so we needed to add some code to restore x86 calling conventions at the end of the function.

3 Ordering Optimizations

We begin by doing alegebraic simplification, followed by common sub-expresssion elimination and copy propagation, followed by a pass of dead code elimination. We repeat these optimizations after all other optimizations we perform. After the first pass we do loop invariant code motion and then loop unrolling.

After this we perform register allocation as we destruct the SSA form, and then perform the assembly level optimizations. Dead code elimination is also called multiple times during the destruction process. We determined the order of optimizations through experimentation and by noting which optimizations might generate dead code. We always call CSE and CP just to be safe as well.

4 Optimization Evaluation

We maintained evaluations of the optimizations we implemented on our projects github wiki, located at the WIKI.

5 CFG Optimization Examples

Included here are some sample CFGs with optimizations highlighted. We produce these CFGs using the Graphviz exporter from project 3. None of these were hand constructed.

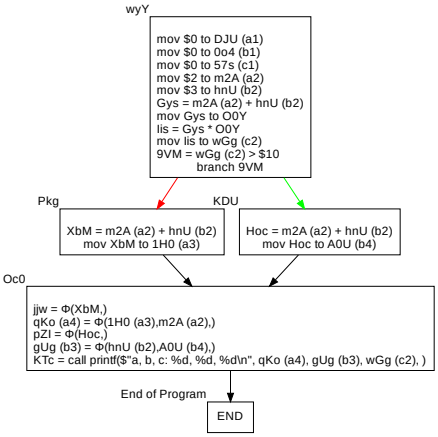

Figure 1: Before CSE

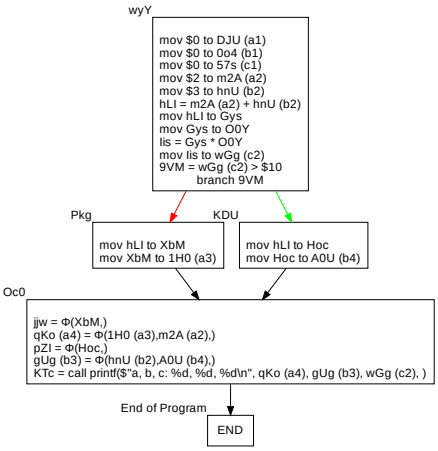

Figure 2: After CSE

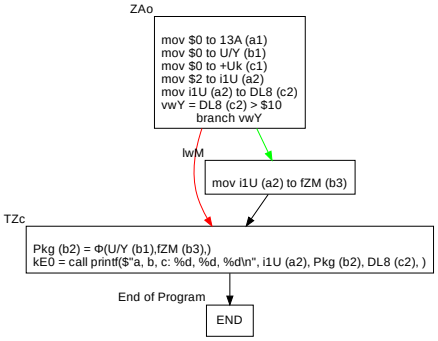

Figure 3: Before CP

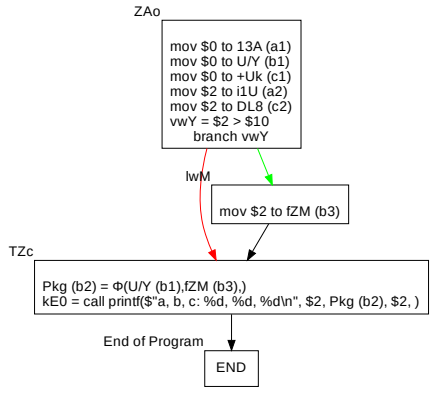

Figure 4: After CP

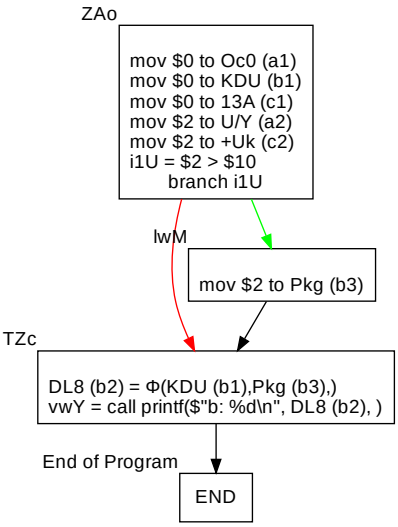

Figure 5: Before DCE

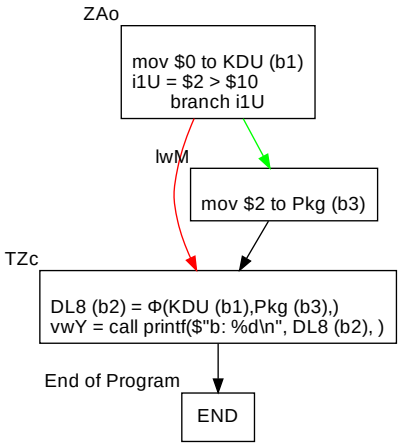

Figure 6: After DCE

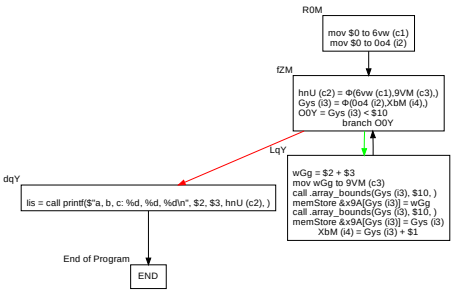

Figure 7: Before Loop Motion

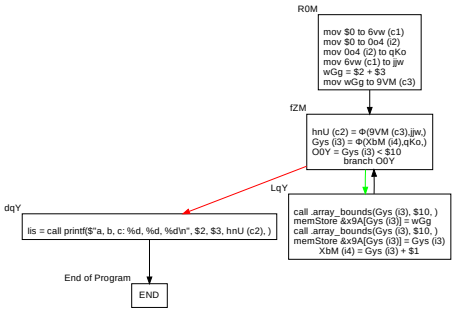

Figure 8: After Loop Motion