Phase 4 Report

Evan Andrews, Nitya Babbar, Sean Huckleberry

I. Design Choices and Implementation

A. Framework for dataflow optimizations

Our dataflow optimization infrastructure is primarily implemented in *state.rs* and *dataflow.rs* (and, notably, our submission branch for this phase is *copy_prop*, not *main*!). The former contains helper functions and data structures required for dataflow analysis, including functions to compute predecessors/successors of a basic block in the CFG, and a *CopyMap* structure which maintains information about the program across blocks (e.g., variable liveness). The entry point to *dataflow.rs* is *optimize_dataflow*, which takes in the CFG and a set of optimizations, and then runs a fixed point loop to continue applying our optimizations until our CFG reaches a steady state. We made this design decision to ensure that we get the most out of our optimizations, as applying certain optimizations (e.g., copy propagation) can pave the way for other optimizations (e.g., dead code elimination), and vice versa.

Each of our optimizations is a separate function in *dataflow.rs*, and runs the optimization in multiple passes over the basic blocks of a given function. *optimize_dataflow* is called from within our main code generation function *generate_assembly*, to optimize the CFG before producing the final x86 code. Once we have implemented more dataflow optimizations, we intend to perform benchmarking to determine the optimal order for running each optimization pass within *optimize_dataflow.*

We also implemented the basic interface in *cli.rs* to enable running our compiler with all (*"-O all")* or a subset of the optimizations *(e.g., "-O cp")*.

B. Optimizations

1. Copy propagation

Scope: Our current implementation takes place in the method scope; it performs propagation over multiple blocks in a method CFG. We currently only optimize for local variables (no support yet for global or array variables).

Dataflow equations:

- IN[B] = ∩ OUT[P] for all predecessors P of B
- OUT[B] = GEN[B] ∪ (IN[B] - KILL[B])

Sample test case: Our sample test case for copy propagation is *cp-sample.dcf*. For convenience, the source code for this test is copied below in the appendix as Figure 1, and it can also be found in /doc/phase4-code/. The generated x86 assembly and CFG for this sample test cases are also captured below Figure 1 and Figure 2, respectively. This test case demonstrates copy prop over a few different scenarios, including for both binary and unary operations, and over branching control flow. For example, the assignment of *k* to *z* is propagated after diverging control flow since the assignment is constant, while propagation is unable to be performed over *e.*

Discussion: Our copy propagation algorithm is similar to the fixed point algorithm presented in lecture. It is performed with two passes over the CFG: the first computes the steady-state IN and OUT sets for each basic block, and the second applies the actual substitutions. The first pass is done in *compute_maps*, which first computes the predecessors and successors for each basic block. It then iterates over the maps until hitting a fixed point, simulating the execution of the program in a forward direction. We update the maps *copy_to_src* and *src_to_copies* by generating entries on any direct assignment and killing the destination of any assignment, and these maps end up being our IN and OUT maps for each basic block. Our second pass is done within *copy_propagation* which simply applies the substitutions based on whatever was considered "safe" in the steady state of the previously computed IN and OUT maps.

A design alternative we considered was performing the copy propagation substitutions concurrently while computing the IN and OUT sets. In other words, this design attempted to perform the substitution as soon as it was deemed possible. This ended up leading to some correctness issues with, for example, loops, where an iterator value was propagated in the first iteration, but then killed in subsequent iterations. This kind of situation was not handled properly by this design, which is why we decided to compute all steady state maps before performing any substitutions.

2. Common subexpression elimination (in progress)

Our implementation of CSE is still in progress, but very close to being complete. It is essentially identical to copy propagation in scope, dataflow equations, and implementation, so we won't go into full detail here. That being said, the main difference with CSE is that, instead of our IN and OUT sets storing mappings between aliases and their sources, we store mappings between an expression and the variable it is currently stored in, where an expression is either an addition, subtraction, multiplication, division, or modulus of two temp variables. If we see that an expression is already computed and in our IN map, we replace the expression with that variable instead of computing it again. We kill expressions that involve variables that were reassigned to. This algorithm is different from the *value numbering* approach that was taught in lecture, and is instead more similar to the *available expressions* analysis discussed in recitation.

While our CSE optimization passes all test cases (for phase 4 on Gradescope), we ran into issues while trying to merge it with our copy propagation optimization. This led to infinite loops in some of the test cases, which we have yet to debug. We also noticed that we had the same problem with this optimization as we did with copy propagation (discussed at the end of the last section), where we prematurely performed expression substitutions that were supposed to be killed later on. An example of this bug is shown in Figure 4; originally, we were substituting the addition expression inside the loop with variable *c* even though the expression should be killed by the reassignment to variable *a*. This led to incorrect output before we fixed our approach, by completely generating IN and OUT sets before performing any expression substitutions. However, even with this bug, we were still passing all 125 Gradescope tests.

A sample test case for CSE is provided below for reference in Figure 4, but was not included in our final submission since we haven't fully integrated.

II. Extras

The graph visualization tool Graphviz proved very useful in debugging by allowing us to visually examine our CFG. We also wrote a test script *testopt.sh* to run all of our test cases with optimizations and match their output to the expected output.

III. Difficulties

1. *Phase 3 test cases:* The four test cases from Phase 3 *(sp25-01-longs.dcf, sp25-10-side-effects.dcf, sp25-11-weird-for.dcf,* and *sp25-12-stack-recursive.dcf)* were successfully debugged after receiving the source code.
2. *Int/long and CSE integration:* While int/long differentiation and the CSE optimizations are mostly complete and correct in isolation, they have not yet been properly integrated with the submission branch with copy propagation. This is a merge we intend to complete soon, and then we are hoping to implement dead code elimination to see the speedup effects of all of these optimizations. One caveat is that our int/long differentiation code fails on the derby server while passing all tests locally and on Gradescope, so this is something we will need to debug. Another problem we are running into with intlong is related to our behavior on the phase 5 Gradescope submission for test cases. We have gone into office hours to determine the source of one of our bugs, however, the other bug seems more serious and is causing curp.dcf to sometimes fail and sometimes succeed when TAs attempt running it locally for debugging help. Going forward, we will visit office hours again to attempt to resolve the issues with integration as we are hopeful that it will provide more opportunities for improvement later in phase 5.

IV. Contribution Statement

The three primary tasks for our Phase 4 consisted of int/long differentiation, copy propagation, and common subexpression elimination. Each teammate had a significant contribution to at least 1-2 of these tasks, and often collaborated for debugging help.

(See appendix below)

V. Appendix

Figure 1: Decaf source code for *cp-sample.dcf*, our sample test case for copy propagation

```
// Our sample test case to demonstrate copy propagation
import printf;

void main() {
   int a;
   int b;
   int c;
   int d;
   int e;
   int f;
   int z;
   int k;
   bool cond1;
   bool cond2;
   bool notcond;

   a = 5;
   k = 999;
   b = a;          // copy: b = a
   c = b + 2;      // propagation: b -> a
   d = c % 3;
   cond1 = true;
   cond2 = cond1;

   if (cond1) {
       // should be e = (7 % 3 = 1)
       e = d;      // copy across branch
       z = k;
   } else {
       e = 0;
       z = k;
   }

   a = z + 3; // propagation: z -> k
   notcond = !cond2;      // propagation: cond2 -> cond1
   printf("%d\n", e); // no propagation: should be 1
}
```

Figure 2: x86 code for *cp-sample.dcf* before (left) and after (right) applying copy propagation.

```
str0:
   .string "%d\n"
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $192, %rsp
main0:
   movq $5, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -96(%rbp)
   movq -96(%rbp), %rax
   movq %rax, -8(%rbp)
   movq $999, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -104(%rbp)
   movq -104(%rbp), %rax
   movq %rax, -64(%rbp)
   movq -96(%rbp), %rax
   movq %rax, -16(%rbp)
   movq $2, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -112(%rbp)
   movq -96(%rbp), %rax
   addq -112(%rbp), %rax
   movq %rax, -120(%rbp)
   movq -120(%rbp), %rax
   movq %rax, -24(%rbp)
   movq $3, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
```

```
str0:
   .string "%d\n"
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $192, %rsp
main0:
   movq $5, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -96(%rbp)
   movq -96(%rbp), %rax
   movq %rax, -8(%rbp)
   movq $999, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -104(%rbp)
   movq -104(%rbp), %rax
   movq %rax, -64(%rbp)
   movq -8(%rbp), %rax
   movq %rax, -16(%rbp)
   movq $2, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -112(%rbp)
   movq -16(%rbp), %rax
   addq -112(%rbp), %rax
   movq %rax, -120(%rbp)
   movq -120(%rbp), %rax
   movq %rax, -24(%rbp)
   movq $3, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
```

```
   movq %rbx, -128(%rbp)
   movq -120(%rbp), %rax
   cqto
   movq -128(%rbp), %rcx
   idiv %rcx
   movq %rdx, -136(%rbp)
   movq -136(%rbp), %rax
   movq %rax, -32(%rbp)
   movq $1, %rax
   movq %rax, -72(%rbp)
   movq -72(%rbp), %rax
   movq %rax, -80(%rbp)
   movq -72(%rbp), %rax
   cmpq $0, %rax
   jne main1

   jmp main2
main1:
   movq -136(%rbp), %rax
   movq %rax, -40(%rbp)
   movq -104(%rbp), %rax
   movq %rax, -56(%rbp)
   jmp main3
main2:
   movq $0, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -144(%rbp)
   movq -144(%rbp), %rax
   movq %rax, -40(%rbp)
   movq -104(%rbp), %rax
   movq %rax, -56(%rbp)
   jmp main3
main3:
   movq $3, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -152(%rbp)
   movq -104(%rbp), %rax
```

```
   movq %rbx, -128(%rbp)
   movq -24(%rbp), %rax
   cqto
   movq -128(%rbp), %rcx
   idiv %rcx
   movq %rdx, -136(%rbp)
   movq -136(%rbp), %rax
   movq %rax, -32(%rbp)
   movq $1, %rax
   movq %rax, -72(%rbp)
   movq -72(%rbp), %rax
   movq %rax, -80(%rbp)
   movq -72(%rbp), %rax
   cmpq $0, %rax
   jne main1

   jmp main2
main1:
   movq -32(%rbp), %rax
   movq %rax, -40(%rbp)
   movq -64(%rbp), %rax
   movq %rax, -56(%rbp)
   jmp main3
main2:
   movq $0, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -144(%rbp)
   movq -144(%rbp), %rax
   movq %rax, -40(%rbp)
   movq -64(%rbp), %rax
   movq %rax, -56(%rbp)
   jmp main3
main3:
   movq $3, %rbx
   movq $0, %rax
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -152(%rbp)
   movq -56(%rbp), %rax
```

```
addq -152(%rbp), %rax
movq %rax, -160(%rbp)
movq -160(%rbp), %rax
movq %rax, -8(%rbp)
movq -72(%rbp), %rax
xor $1, %rax
movq %rax, -168(%rbp)
movq -168(%rbp), %rax
movq %rax, -88(%rbp)
leaq str0(%rip), %rax
movq %rax, -176(%rbp)
movq -176(%rbp), %rdi
movq -40(%rbp), %rsi
movq $0, %rax
call printf
movq $0, %rbx
movq $0, %rax
shlq $32, %rax
orq %rax, %rbx
movq %rbx, -184(%rbp)
movq -184(%rbp), %rax
movq %rbp, %rsp
popq %rbp
ret
```

```
addq -152(%rbp), %rax
movq %rax, -160(%rbp)
movq -160(%rbp), %rax
movq %rax, -8(%rbp)
movq -80(%rbp), %rax
xor $1, %rax
movq %rax, -168(%rbp)
movq -168(%rbp), %rax
movq %rax, -88(%rbp)
leaq str0(%rip), %rax
movq %rax, -176(%rbp)
movq -176(%rbp), %rdi
movq -40(%rbp), %rsi
movq $0, %rax
call printf
movq $0, %rbx
movq $0, %rax
shlq $32, %rax
orq %rax, %rbx
movq %rbx, -184(%rbp)
movq -184(%rbp), %rax
movq %rbp, %rsp
popq %rbp
ret
```

Figure 3: CFG before (left) and after (right) applying the copy propagation optimization.

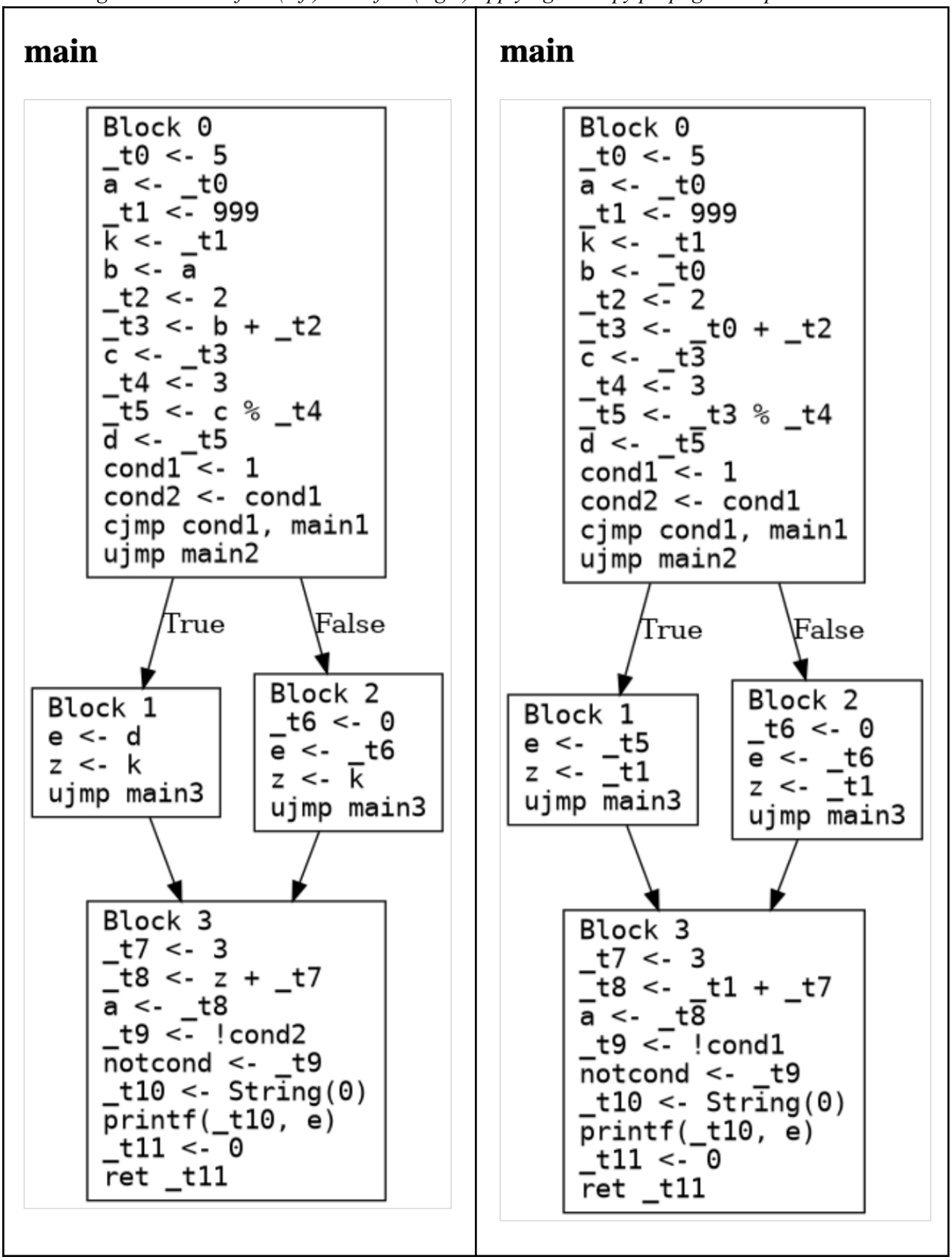

Figure 4: Decaf source code for a CSE test case.

```
import printf;

void main () {
 int a;
 int b;
 int c;
 int d;
 int i;

 a = 2;
 b = 3;
 c = a + b;

 for (i = 0; i < 10; i++) {
   d = a + b;
   a = 1;
   printf("%d\n", d);
 }
}
```