Phase 5 Report

Sean Huckleberry, Evan Andrews, Nitya Babbar

I. Phase 5: Design Choices and Implementation

A. -O all flag

In its final state, the *-O all* option turns on all optimizations implemented for our compiler, which includes copy propagation, constant propagation, common subexpression elimination, dead code elimination, and register allocation. The four dataflow optimizations (copy & const prop, CSE, DCE) are repeatedly applied in a loop that terminates only when the code reaches a fixed point. We made this design decision to be able to get the most out of our dataflows, and eliminate as many common expressions and dead code as possible. While we experimented lightly with reordering the passes, we found that by optimizing to a fixed point, we didn't need to worry much about ordering since each optimization got its turn to respond to any changes to the code. The dataflow optimizations directly modified our CFG, and register allocation was performed over the optimized CFG.

Some optimizations, such as instruction selection and peephole optimizations, were applied directly to our code generation file, and are therefore not toggled by the flag. That being said, these optimizations were thoroughly tested for correctness and speedup.

Here are all the flags turned on by *-O all: cse, cp, dce, regalloc, constprop.*

B. Optimizations

B.1) Register allocation

The majority of our time on Phase 5 was spent implementing a register allocator, and this is where we also saw the most speedup. We implemented the web-based coloring algorithm from lecture almost directly. In particular, within *regalloc.rs*, we iterate through the method CFGs, and call the function *compute_webs,* whose result we pass into *compute_use_liveness_spans* in order to *build_interference_graph.* Finally, *assign_registers* performs the coloring-based allocation step presented in lecture, and *apply_reg_assignments* modifies the operands in the CFG to incorporate the register assignments for each web.

The main data structures for register allocation are contained within *web.rs*, webs, interference graphs, and an *InstructionIndex* struct, which introduces a global numbering scheme over each method CFG to play well with Rust.

Our initial stages of register allocation utilized *%r12–%r15,* which we identified as not being used for any other purpose. Since these are callee saved, we were able to pre-allocate stack space for the method based on the number of these registers it used, and then just *push* and *pop* in the method prologue and epilogue, respectively.

A bit more care was required for allocating the five registers used for arguments in the x86 calling convention. The basic update process was to *push* and *pop* these registers before and after any function call. However, we found weird bugs where, e.g., after pushing *%rdi* and loading its argument value, we had a subsequent argument *%rsi* whose value relied on the original value stored in *%rdi* pushed to the stack. We solved this by modifying the operand to load from the memory location during this process.

More specifically, for caller saved registers, we stored information about caller saved registers that were allocated to a specific *method_cfg*. If that method was called, the values of the registers would be stored to the stack first (Figure 1) and then referenced from there-on if we needed to get the value from one of the registers right before the function call. We then load all of the arguments as defined by the x86 ABI and would restore the value of such registers after the function call. Additionally we had to take care with the *%rdx* register which was being used in division; to handle this we push and pop *%rdx* from the stack before and after any division instructions.

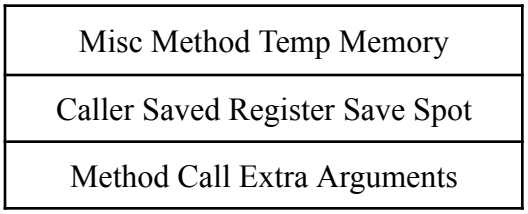

Figure 1: Method Stack Layout

We were also able to recover *%r10* by being more consistent and efficient about our use of scratch registers throughout *codegen.rs.*

In total, register allocation resulted in about a *0.8x* speedup. See *Table I* in the appendix for more detailed speedup information. See *Code Sample I* for a sample test case demonstrating the impact of our register allocator

B.2) Constant propagation

The implementation of constant propagation was similar to our dataflow analysis for copy propagation in Phase 4. The main difference was just keeping track of whether an operand was assigned to a register containing a constant, rather than whether it was a live copy.

Constant propagation's effect was twofold: it enabled more dead code elimination (300% increase of instructions eliminated across our local test suite), and also enabled the magic number division optimization presented below. See *Code Sample II* in the appendix for an example of constant propagation in action.

B.3) Peephole optimizations

We implemented some simple peephole optimizations based on the suggestions from Recitation 10 by scanning through our x86 assembly code 1-3 instructions at a time. These included removing unnecessary unconditional jumps (i.e. jumps at the end of a basic block to the label right after it), removing redundant push/pops and moves, replacing "mov $0, %reg" instructions with "xor %reg, %reg", and replacing multiplication by powers of 2 with a left shift.

In order to make the first optimization more effective, we reordered our basic block layout so that we could maximize the number of jumps that could be removed. We also attempted to implement fallthrough; however, we found that it didn't help performance, or even made it worse. Originally, we tried falling through to the "true" case for all conditionals, but this made almost every derby test case slower. We then tried the optimization just for loops, which was helpful for some test cases but made others worse, so in the end we eliminated this optimization. We are not sure why it didn't help, but we assume it has something to do with the way the branch predictor works.

The final and most effective peephole optimization we implemented was removing chains of moves such as: `mov a,b; mov b,c`. We noticed that this pattern appeared very frequently throughout our generated code. However, naively replacing all of these chains with a single move led to correctness issues because sometimes the eliminated operands weren't actually just "intermediates". To fix this, we performed a sort-of liveness analysis on our generated x86 code for each basic block, by keeping track of which x86 operands were live at each program point, so we knew which ones were safe to eliminate. We also used this information to replace mem-reg moves with reg-reg moves when possible. Therefore this wasn't entirely a peephole optimization; we're not sure if this is an actual compiler technique or something we just made up, but we found it helpful in achieving about 0.2x speedup.

B.4) Dead Code Elimination

We completed our DCE implementation from Phase 4 using a liveness analysis very similar to the one presented in lecture. Our implementation was slightly conservative in the sense that it sometimes fails to catch dead code in loops, but we found it effective enough for our purposes. The impact of dead code elimination was greatly boosted when we also implemented constant propagation in this stage. This can be seen in *Table I* in the appendix.

B.5) Magic number division

One of our attempted instruction selection optimizations was to eliminate divisions by a constant with shifts and adds using magic number division. In particular, we used the equation $a / c = (a * magic) >> shift$. We attempted to take the absolute value of the divisor, which enabled us to perform a simpler computation for the *(magic, shift)* pair and then reintroduce the negation with the x86 *neg* instruction. Unfortunately, the magic number division

was failing only on the *durp* test case, and we were unable to find the bug since the derby test case was private. That being said, looking at teams with similar runtimes for everything except *jurp* (the derby case most heavily relying on division), we would imagine this optimization bringing us up to about 3.0x speedup. Given how close this optimization was to being correct, we still kept the code samples.

II. Phases 1-4: Recap

A. Phase 1: Scanner & Parser

A.1) Data Structures (token.rs, ast.rs)

The token file contains the data structures used for scanning and producing the output token string. It consists mostly of enums for the major tokens in the decaf language, including keywords, literals, and punctuation. TokenInfo is a wrapper for Token that contains display, line, and col information so that the scanner can produce more useful lexing errors. There is also a TokenSlice struct, which is a wrapper for Token that implements the nom::Input trait, allowing me to use more powerful combinators like many0 and separated_list0.

The AST contains structures for the parser to construct the syntax tree. It includes the entire hierarchy and largely matches the terminals and non-terminals within the grammar, including program, method declarations, expressions, bool literals, etc.

A.2) Scanner (scan.rs)

The scanner was implemented by hand to gain more understanding and control over Rust. It relies heavily on pattern matching by precedence to generate individual tokens from text. At a higher level, the main function iterates through the string and keeps calling get_next_token, which invokes lower-level functions to either gobble whitespace or match certain token types. Token metadata is stored in the TokenInfo struct, which is then useful to print the results of the scanner to the output.

A.3) Parser (parse.rs)

The parser was written with nom, a parser combinator library for Rust that enables building simple parsers and composing them to parse more complex grammars. nom provided some nice abstractions in the default combinators, which enabled me to focus less on the specifics of backtracking or lookahead parsing, and instead on the basics of recursive-descent parsing and hacking the grammar to make it unambiguous. nom was found to be a good middle-ground between writing the parser by hand and using a parser generator. Working with precedence and punctuation consistency were difficult to get right for this part.

B. Phase 2: Semantic Checker

B.1) Data structures

The main data structures supporting the Phase 2 semantic checker are contained in *symtable.rs* and *scope.rs. symtable.rs* contains definitions for the semantic AST IR, which augments the raw AST from the parse tree with symbol tables. Symbol tables are defined as scopes in *scope.rs;* they each contain a hash table of defined variables, methods, and imports, as well as an optional parent pointer to the enclosing scope. There is also an implementation of *Scope* that provides methods for inserting and doing recursive lookups from a given instance. Lookups return a *TableEntry,* which contains the relevant information for semantic checks.

B.2) Semantic checking

The checking is split into two phases: during semantic AST construction, and after construction. *semcheck.rs* contains the code to build the scope-augmented AST. Some of the semantic checks are implemented here for convenience. For example, it makes more sense to ensure the method of a method call is defined during linear construction of the AST, then afterwards where we have less understanding of when method calls occur relative to method definitions. Finally, *traverse.rs* implements a traversal of the semantic tree, performing any of the checks for the remainder of the 23 rules.

B.3) Error reporting

Span information was calculated during the scanning phase, and propagated through the parser to the semantic AST, giving us direct access to the spans for each token and expression, and allowing us to give specific error messages. Upon any semantic check failing, the offending checker makes a call to a variant of *format_error_message,* which formats a message for the given semantic error. It also sets the flag *error_found* within a context object that is passed between methods. The implication of this flag is that we can report as many error messages as possible, and delay panicking until after all semantic checks occur.

B.4) Testing and Debugging

Testing relied on a combination of local testing infrastructure, and the set of private test cases provided on Gradescope. The team attempted a test-first approach, beginning with generating basic test cases for each of the 23 semantic rules. We tried to keep each of these test cases simple to use to ensure that each of the checks were performing as expected, on their own. After completely implementing all of the rules, we noticed that we were missing some of the invalid cases in the private suite on Gradescope. To this end, we closely analyzed each of the semantic rules and wrote some more advanced tests that exercised edge cases of the expected behavior. *cargo test* was used to run our test cases, as well as a script that let us examine the outputs to standard error for each of the runs. Debugging was easy once we had noticed holes in our semantic checking and created regression tests.

Pretty printing the tree, print statement debugging, and some use of Graphviz was useful in debugging our checker.

C. Phase 3: Code Generation

C.1) Starting Point: Semantic IR

During Phase II, we converted our AST representation into another high-level IR to perform semantic checking. This IR, known as the *symtable IR* ("symbol table IR"), is defined in `symtable.rs` and constructed in `semcheck.rs`. In comparison with the AST representation, this IR prioritizes information central to semantics, primarily by augmenting the AST with scopes. Each `SymProgram`, `SymMethod`, or `SymBlock` has an accompanying scope, which supports the insert and recursive lookup of local variables. The scopes have parent pointers to enable this recursive lookup.

C.2) CFG / low-level IR

Our CFG IR was generated directly from the semantic IR discussed above. The main supporting structures to our CFG are `BasicBlock` and `CFGScope` implemented in `cfg.rs` as well as `Instruction` implemented in `tac.rs`. `Instruction`s were designed to map cleanly to one or more x86 instructions, `BasicBlock`s were vital for organizing labels and code blocks within our methods, and `CFGScope`s were used to keep track of the temporary variables that we assigned to each local variable in the program. Originally, we were simply using the original names of local variables, but we realized that we could have problems with shadowing in the generated code.

Our CFG is created in `buildcfg.rs`. The bulk of the CFG construction is driven from `destruct_method`, which recursively calls `destruct_statement` and `destruct_expression`. Each of these functions adds instructions to the given `BasicBlock`, creates new ones as necessary, and returns the final `BasicBlock` at the end of its execution. Global *temp* and *block* counters are used to give each temporary variable and `BasicBlock` a unique ID. Each time we add a new temp variable, we assign a stack offset for it. For now, we allocate 8 bytes for everything no matter its type (we do not differentiate ints and longs yet). We also allocate an additional 8 bytes at the beginning of each array to store its length.

As we build our CFG, if we encounter any string literals, we store them in a data structure so that we can add them to the data section during code generation. Similarly, we have a data structure for all the global variables in the program.

Short-circuiting is implemented on all conditionals (not only in control flow statements). We originally considered splitting the CFG generation and linearization of our IR into two stages, but found that this wasn't necessary since our CFG was robust enough to use directly for code generation.

C.3) Assembly code generator

After generating the CFG IR, assembly code generation primarily consisted of matching on instructions within our CFG and mapping them to concrete x86 instructions. Important data structures defined for code generation involved `X86Insn`, `X86Operand`, and `Register`. Defined in `x86.rs`, these structures represent the final state of the code, which can be directly emitted by the display formatting that we defined for each of them.

Of course, some additional key considerations in code generation included adhering to calling convention and maintaining stack alignment for imported function calls. For now, most of our instructions use `%rax` as a working register (and `%r10` to store indices for array accesses) with all intermediate values stored on the stack, so we didn't have to consider caller or callee-saved registers, though we did need to consider where to place our function call arguments and return values. A key design choice in CFG construction made stack alignment much more manageable. Instead of pushing and popping from the stack constantly as temps were allocated, we stored the total stack size needed for a given method during CFG construction, and allocated that entire stack frame at the start of the method call. This meant that we could simply 16-byte align that single allocation instead of worrying about aligning every variable allocation!

C.4) Testing infrastructure

Our local testing infrastructure consists of cargo testing and shell script testing. For earlier phases, the infrastructure provided by cargo was incredibly useful and easily augmentable. The code generation tests were a little more difficult to run since most of our teammates were on arm architecture, so we designed a shell script to be executed from a virtual machine (OrbStack or Athena) that generated x86 code using our compiler, compiled it with gcc, and then executed it and compared the results to the expected results. Both testing strategies proved effective, efficient to run, and easily expandable for regression testing.

Due to the addition of private test cases, our test coverage became much stronger during Phases II and III (many of our test cases can be seen in *tests/semantics* and *tests/phase3*). For Phase II, we created test cases corresponding to each semantic rule to ensure basic coverage of each rule. Then, we also created tests for larger programs that violated many semantic rules to ensure that our compiler was able to recognize and note them all. For Phase III, in particular, we recognized that our initial test cases weren't thorough enough, as even after passing all of our tests locally we were missing quite a few private test cases on Gradescope. To this end, we used the names of the test cases we were failing on Gradescope as a guide to create additional test cases, some of which caught additional errors, while others didn't.

D. Phase 4: Dataflow Optimizations

D.1) Framework for dataflow optimizations

Our dataflow optimization infrastructure is primarily implemented in *state.rs* and *dataflow.rs*. 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")*.

D.2) Optimizations
As noted previously, the effect of all dataflow and other optimizations are summarized in *Table I* in the appendix.

D.1.1) 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] $= \cap$ OUT[P] for all predecessors P of B
- OUT[B] = GEN[B] $\cup$ (IN[B] - KILL[B])

Sample test case: Our sample test case for copy propagation is *copyprop-sample.dcf*. The generated x86 assembly and CFG for this sample test case are also captured below *Code Sample 4* 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.

D.1.2) Common subexpression elimination

Our implementation of CSE 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.

III. Extras

Our most powerful tool for visual debugging was using GraphViz software to be able to visualize both our CFGs and register web interference graphs. This enabled us to walk through control flow and understand dependencies in a much more efficient way than by directly looking at the assembly or reading print statements. Once *surp.dcf* was released, we were able to recreate the dependencies so that we were able to test the case locally. This was invaluable in both correctness testing, and having a more complex program enabling us to produce more realistic code and perform peephole optimizations. We also set up hyperfine, but didn't find much use for it, as the Gradescope testing proved more reliable and accessible enough.

IV. Difficulties

We expect our compiler to be able to pass all Phase 1-4 and derby test cases, both with optimizations enabled and disabled. One issue we resolved was a non-deterministic failing of one of the fall-through test cases due to mistakenly loading the exit code into *%rax* instead of *%rdi*.

We attempted to get magic number division working in the compiler and we were able to see large speedups on the derby test cases. However, we were failing only one of the derby test cases (*durp*), and we were not able to fix the division to get it to pass the test case. We were able to find one small bug related to division of Longs that were close to the largest Long size allowed. Had we been given more time we would likely have been able to nail down this bug and significantly get more speedup on *jurp*. According to our calculations this would have brought us to over 3.0x speedup.

V. Contribution Statement

The two main tasks for Phase 5 were implementing a register allocator and performing peephole optimizations. All three members collaborated to implement and debug the most basic form of the register allocator (4 registers). From that point on, two members worked on extending the register allocator, while the other group member worked primarily on peephole and instruction selection optimizations.

VI. Appendix

Specs	**Results**
No optimizations	739.13, 2871.57, 3719.79, 2076.06, 3120.39, 4324.16, 1021.44, 4729.95, 2391.1, **1.123**
Copy propagation only	728.38, 2911.32, 3633.84, 2021.11, 3072.27, 4255.80, 1001.72, 4672.73, 2356.9, **1.139**
Integer vs. long differentiation only	504.69, 3188.94, 3617.29, 2006.59, 3004.51, 4165.95, 966.11, 4445.46, 2237.4, **1.200**
DCE only	735.60735, 2967.30, 3689.89, 1888.99, 3105.02, 4327.60, 1029.00, 4654.27, 2365.10, **1.135**

Integer vs. long differentiation, CP, CSE	494.55, 3240.59, 3572.26, 2045.75, 2633.25, 4007.05, 948.16, 4452.44, 2186.1, **1.228**
Integer vs. long differentiation, CP, CSE, ***DCE***	485.09, 3250.80, 3500.70, 1850.25, 2659.91, 3958.48, 942.01, 4417.43, 2144.6, **1.252**
Integer vs. long differentiation, CP, CSE, DCE, ***basic regalloc (4 registers)***	**1.85** (other measurements weren't tracked)
Integer vs. long differentiation, CP, CSE, DCE, basic regalloc (4 registers), ***low-degree prioritization for register allocation***	282.37, 1412.64, 2162.75, 1944.75, 2059.63, 2939.60, 810.50, 3822.35, 1539.0, **1.745**
Integer vs. long differentiation, CP, CSE, DCE, basic regalloc (4 registers), ***argument register allocation (9 regs total)***	254.35, 1141.65. 1670.11, 1441.15, 1989.21, 2690.34, 719.84, 3882.89, 1341.0, **2.002**
Integer vs. long differentiation, CP, CSE, DCE, basic regalloc, argument register allocation, ***reclaim %rbx for regalloc (10 regs total)***	255.31, 1112.19, 1671.36, 1407.15, 2005.48, 2691.69, 711.57, 3550.68, 1318.0, **2.037**
Integer vs. long differentiation, CP, CSE, DCE, basic regalloc, argument register allocation, reclaim %rbx for regalloc (10 regs total), ***round 1 peephole optimizations (fallthrough, block reordering, etc.)***	237.16, 1264.21, 1824.45, 1215.22, 1888.42, 2722.42, 751.16, 3509.45, 1316.2, **2.040**
Integer vs. long differentiation, CP, CSE, DCE, basic regalloc, argument register allocation, reclaim %rbx for regalloc (10 regs total), round 1 peephole optimizations (fallthrough, block reordering, etc.), ***round 2 peephole optimizations (zeroing rax, mul strength reduction, redundant code elimination)***	199.86, 880.84, 1446.29, 1504.30, 1875.54, 2545.89, 703.64, 3457.31, 1205.1, **2.228**

Integer vs. long differentiation, CP, CSE, DCE, basic regalloc, argument register allocation, reclaim %rbx for regalloc (10 regs total), round 1 peephole optimizations (fallthrough, loop reordering, etc.), round 2 peephole optimizations (zeroing rax, mul strength reduction, redundant code elimination), ***constant propagation***	184.09, 757.43, 1369.46, 1351.13, 1531.24, 2485.51, 701.86, 2986.66, 1094.4, **2.453**

Table I: Optimizations and corresponding derby measurements.

```
str0:
   .string "%d %d\n"
str1:
   .string "%d %d\n"
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $160, %rsp
main0:
   movl $10, -32(%rbp)
   movl -32(%rbp), %eax
   movl %eax, -4(%rbp)
   movl $20, -36(%rbp)
   movl -36(%rbp), %eax
   movl %eax, -8(%rbp)
   movl $30, -40(%rbp)
   movl -40(%rbp), %eax
   movl %eax, -12(%rbp)
   movl -4(%rbp), %eax
   addl -8(%rbp), %eax
   movl %eax, -44(%rbp)
   movl -44(%rbp), %eax
   movl %eax, -16(%rbp)
   movl $3, -48(%rbp)
   movl -12(%rbp), %eax
   imul -48(%rbp), %eax
   movl %eax, -52(%rbp)
```

```
str0:
   .string "%d %d\n"
str1:
   .string "%d %d\n"
.globl main
main:
   pushq %rbx
   pushq %rbp
   movq %rsp, %rbp
   subq $152, %rsp
main0:
   movl $10, -32(%rbp)
   movl -32(%rbp), %edi
   movl $20, %ebx
   movl %ebx, %esi
   movl $30, %ebx
   movl %ebx, %ecx
   movl %edi, %ebx
   addl %esi, %ebx
   movl %ebx, %esi
   movl $3, %ebx
   movl %ebx, %ebx
   imul %ecx, %ebx
   movl %ebx, %ebx
   movl %esi, %ecx
   imul %ebx, %ecx
   movl $100, %ebx
   movl %ecx, %eax
```

```
movl -52(%rbp), %eax
movl %eax, -20(%rbp)
movl -16(%rbp), %eax
imul -20(%rbp), %eax
movl %eax, -56(%rbp)
movl $100, -60(%rbp)
movl -56(%rbp), %eax
subl -60(%rbp), %eax
movl %eax, -64(%rbp)
movl -64(%rbp), %eax
movl %eax, -20(%rbp)
leaq str0(%rip), %rax
movq %rax, -72(%rbp)
movq -72(%rbp), %rdi
movl -16(%rbp), %esi
movl -20(%rbp), %edx
xorq %rax, %rax
call printf
movl $16, -76(%rbp)
movl -16(%rbp), %eax
cdq
idivl -76(%rbp)
movl %edx, -80(%rbp)
movl -80(%rbp), %eax
movl %eax, -24(%rbp)
movl $100, -84(%rbp)
pushq %rdx
movl -20(%rbp), %eax
cdq
idivl -84(%rbp)
movl %eax, -88(%rbp)
popq %rdx
movl -88(%rbp), %eax
movl %eax, -28(%rbp)
leaq str1(%rip), %rax
movq %rax, -96(%rbp)
movq -96(%rbp), %rdi
movl -24(%rbp), %esi
movl -28(%rbp), %edx
xorq %rax, %rax
call printf
```

```
subl %ebx, %eax
movl %eax, %ebx
movl %ebx, %ecx
leaq str0(%rip), %rax
movq %rax, %rbx
movq %rdi, 0(%rsp)
movq %rsi, 8(%rsp)
movq %rcx, 24(%rsp)
movq %rbx, %rdi
movl 8(%rsp), %esi
movl 24(%rsp), %edx
xorq %rax, %rax
call printf
movq 0(%rsp), %rdi
movq 8(%rsp), %rsi
movq 24(%rsp), %rcx
movl $16, %ebx
movl %esi, %eax
cdq
idivl %ebx
movl %edx, %ebx
movl %ebx, %esi
movl $100, %ebx
pushq %rdx
movl %ecx, %eax
cdq
idivl %ebx
movl %eax, %ebx
popq %rdx
movl %ebx, %ecx
leaq str1(%rip), %rax
movq %rax, %rbx
movq %rdi, 0(%rsp)
movq %rsi, 8(%rsp)
movq %rcx, 24(%rsp)
movq %rbx, %rdi
movl 8(%rsp), %esi
movl 24(%rsp), %edx
xorq %rax, %rax
call printf
movq 0(%rsp), %rdi
```

```
   movl $0, -100(%rbp)
   movl -100(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   ret
```

```
   movq 8(%rsp), %rsi
   movq 24(%rsp), %rcx
   movl $0, -100(%rbp)
   movl -100(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   popq %rbx
   ret
```

Code Sample 1: Register allocation comparison on test case 02-expr.dcf. Notice that fewer memory accesses occur on the right (with regalloc) compared with the left (no regalloc).

```
.comm x, 4, 4
.comm y, 4, 4
.comm z, 4, 4
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $112, %rsp
main0:
   movl $10, -16(%rbp)
   movl -16(%rbp), %eax
   movl %eax, -4(%rbp)
   movl $20, -20(%rbp)
   movl -20(%rbp), %eax
   movl %eax, -8(%rbp)
   movl -4(%rbp), %eax
   addl -8(%rbp), %eax
   movl %eax, -24(%rbp)
   movl -24(%rbp), %eax
   movl %eax, -12(%rbp)
   movl $2, -28(%rbp)
   movl -12(%rbp), %eax
   imul -28(%rbp), %eax
   movl %eax, -32(%rbp)
   movl -32(%rbp), %eax
   movl %eax, x
   movl $10, -36(%rbp)
   movl x, %eax
```

```
.comm x, 4, 4
.comm y, 4, 4
.comm z, 4, 4
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $112, %rsp
main0:
   movl $10, -16(%rbp)
   movl $10, %eax
   movl %eax, -4(%rbp)
   movl $20, -20(%rbp)
   movl $20, %eax
   movl %eax, -8(%rbp)
   movl -4(%rbp), %eax
   addl -8(%rbp), %eax
   movl %eax, -24(%rbp)
   movl -24(%rbp), %eax
   movl %eax, -12(%rbp)
   movl $2, -28(%rbp)
   movl -12(%rbp), %eax
   imul $2, %eax
   movl %eax, -32(%rbp)
   movl -32(%rbp), %eax
   movl %eax, x
   movl $10, -36(%rbp)
   movl x, %eax
```

```
   subl -36(%rbp), %eax
   movl %eax, -40(%rbp)
   movl -40(%rbp), %eax
   movl %eax, y
   movl $2, -44(%rbp)
   pushq %rdx
   movl y, %eax
   cdq
   idivl -44(%rbp)
   movl %eax, -48(%rbp)
   popq %rdx
   movl -48(%rbp), %eax
   movl %eax, z
   movl $0, -52(%rbp)
   movl -52(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   ret
```

```
   subl $10, %eax
   movl %eax, -40(%rbp)
   movl -40(%rbp), %eax
   movl %eax, y
   movl $2, -44(%rbp)
   movl y, -48(%rbp)
   sar $1, -48(%rbp)
   movl -48(%rbp), %eax
   movl %eax, z
   movl $0, -52(%rbp)
   movl -52(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   ret
```

Code Sample 2: Constant Propagation.

```
str0:
   .string "b: %d, c: %d\n"
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $96, %rsp
main0:
   movl $2600, -16(%rbp)
   movl -16(%rbp), %eax
   movl %eax, -4(%rbp)
   movl $-50, -20(%rbp)
   pushq %rdx
   movl -4(%rbp), %eax
   cdq
   idivl -20(%rbp)
   movl %eax, -24(%rbp)
   popq %rdx
   movl -24(%rbp), %eax
```

```
str0:
   .string "b: %d, c: %d\n"
.globl main
main:
   pushq %rbx
   pushq %rbp
   movq %rsp, %rbp
   subq $104, %rsp
main0:
   pushq %rdx
   movl $2600, %r11d
   movabs $368934881474191033, %rax
   mul %r11
   movl %edx, %esi
   neg %esi
   popq %rdx
   pushq %rdx
   movl $2600, %r11d
   movabs $970881267037344822, %rax
```

```
   movl %eax, -8(%rbp)
   movl $19, -28(%rbp)
   pushq %rdx
   movl -4(%rbp), %eax
   cdq
   idivl -28(%rbp)
   movl %eax, -32(%rbp)
   popq %rdx
   movl -32(%rbp), %eax
   movl %eax, -12(%rbp)
   leaq str0(%rip), %rax
   movq %rax, -40(%rbp)
   movq -40(%rbp), %rdi
   movl -8(%rbp), %esi
   movl -12(%rbp), %edx
   xorq %rax, %rax
   call printf
   movl $0, -44(%rbp)
   movl -44(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   ret
```

```
   mul %r11
   movl %edx, %ecx
   popq %rdx
   leaq str0(%rip), %rax
   movq %rax, %rbx
   movq %rsi, 8(%rsp)
   movq %rcx, 24(%rsp)
   movq %rbx, %rdi
   movl 8(%rsp), %esi
   movl 24(%rsp), %edx
   xorq %rax, %rax
   call printf
   movq 8(%rsp), %rsi
   movq 24(%rsp), %rcx
   movl $0, -44(%rbp)
   movl -44(%rbp), %eax
   movq %rbp, %rsp
   popq %rbp
   popq %rbx
   ret
```

Code Sample 3: Magic number division.

```
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
```

```
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
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
```

```
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 -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 -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
   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
```

```
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 -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
```

```
   shlq $32, %rax
   orq %rax, %rbx
   movq %rbx, -184(%rbp)
   movq -184(%rbp), %rax
   movq %rbp, %rsp
   popq %rbp
   ret
```

Code Sample 4: x86 code for cp-sample.dcf before (left) and after (right) applying copy propagation.

```
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $80, %rsp
main0:
   movl $42, -4(%rbp)
   movl $1, -16(%rbp)
   movl -4(%rbp), %eax
   addl -16(%rbp), %eax
   movl %eax, -8(%rbp)
   movq %rbp, %rsp
   popq %rbp
   ret
```

```
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $80, %rsp
main0:
   movq %rbp, %rsp
   popq %rbp
   ret
```

Code Sample 5. Dead Code Elimination for deadcode-sample.dcf.

```
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $64, %rsp
main0:
   xorq %rax, %rax
   call example
   movl $0, -4(%rbp)
   movl -4(%rbp), %eax
```

```
.globl main
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $64, %rsp
main0:
   xorq %rax, %rax
   call example
   xorl %eax, %eax
   movq %rbp, %rsp
```

```
   movq %rbp, %rsp
   popq %rbp
   ret

example:
   pushq %rbx
   pushq %rbp
   movq %rsp, %rbp
   subq $88, %rsp
example0:
   movl $0, %ecx
   movl $2, %ebx
   movl %ebx, %esi
   addl %ecx, %esi
   imul $8, %ecx
   movl $5, %ebx
   cmpl %ebx, %ecx
   setl %al
   movzbq %al, %rax
   movl %eax, %ebx
   movl %ebx, %eax
   cmpl $0, %eax
   jne example1

   jmp example2
example1:
   jmp example3
example2:
   movl $4, -36(%rbp)
   movl -36(%rbp), %esi
   jmp example3
example3:
   movl %esi, %eax
   movq %rbp, %rsp
   popq %rbp
   popq %rbx
   ret
```

```
   popq %rbp
   ret

example:
   pushq %rbx
   pushq %rbp
   movq %rsp, %rbp
   subq $88, %rsp
example0:
   xorl %ecx, %ecx
   movl $2, %esi
   addl %ecx, %esi
   shl $3, %ecx
   movl $5, %ebx
   cmpl %ebx, %ecx
   setl %al
   movzbq %al, %rax
   cmpl $0, %eax
   jne example1

   jmp example2
example1:
   jmp example3
example2:
   movl $4, %esi
example3:
   movl %esi, %eax
   movq %rbp, %rsp
   popq %rbp
   popq %rbx
   ret
```

Code Sample 6. Peephole optimizations for peephole.dcf.
moves of `$0` *have been replaced with* `xor` *instructions, moves through intermediate operands (*`-4(%rbp)` *and* `%ebx`*) have been compressed, multiplication by* `$8` *has been replaced by a shift, and a jump to* `example3` *has been eliminated.*

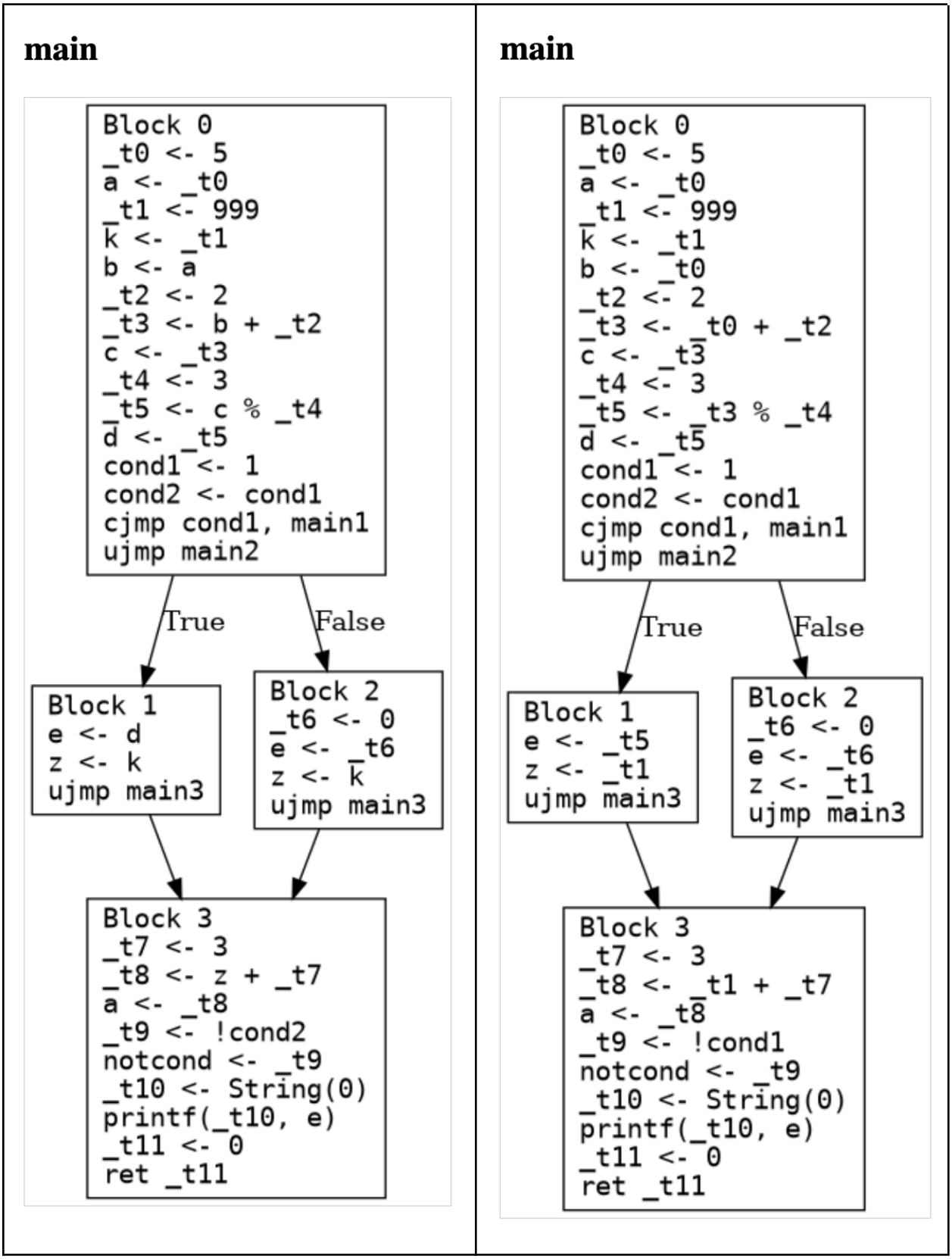

Figure 2: CFG before (left) and after (right) applying the copy propagation optimization.