# 6.035 Phase 5 Documentation

*Decaf Optimizing Compiler*

An Bo Chen, Alex Dang, Youran (Yoland) Gao
{anbochen, alexdang, youran}@mit.edu

# Design & Implementation

## Overview

This section describes all of the optimizations performed by our Decaf compiler. The Phase 4 optimization descriptions have been summarized and copied into this document. For extensive details about each of the Phase 4 Optimizations, including detailed comparisons of CFG printouts, please refer to the Phase 4 Documentation document.

In this phase, we seek to benchmark each of the optimizations performed in Phase 4 as well as the optimizations in Phase 5. Benchmarks were performed on a local machine, a 2020 Macbook Pro running a 10th generation Intel Ice Lake Architecture (i5-1038NG7) x86 CPU on a 64 bit platform. Benchmarks were performed using `hyperfine` with 100 warmup runs and a minimum of 1000 timing runs. All benchmarks were run with register allocation enabled unless otherwise noted.

## Phase 4: Common Subexpression Elimination (CSE)

**Enable this optimization with flag**: `--opt cse`

Common subexpression elimination is performed as a CFG pass by first running an available expressions (AE) fix point algorithm similar to the one seen in lecture and then visiting all basic blocks in the program. Note that expressions that contain global variables do not participate in common subexpression elimination due to the possibility of side effects and the expressions in the bit vector are only bin op expressions. We use AE to find the set of the available expressions at the beginning of each basic block, which would be `IN[n]` for each block `n`. In the `GEN[n]` procedure, we make sure to not generate available expressions that we also kill in the same block.

At the beginning of the CSE analysis, we have `exprToTemp`, a mapping of `CfgExpression` to their corresponding unique temp variable, `varToExprs`, a mapping of `CfgVariable` to a set of `CfgExpressions` that have the variable as one of its operands, and `IN[n]` for each block `n`. For each assign statement in block `n`, if the right hand expression is available via `IN[n]` or a previously seen expression in the block and has not been killed yet, we replace right

hand expression with the corresponding temp variable using `exprToTemp`. If not, then we append a new assign statement afterwards with the corresponding temp variable = left hand variable of the current assign statement. Regardless of what the right hand expression is, we kill all available expressions and seen expressions corresponding to the left hand variable using `varToExprs`.

**Benchmarks:**

| | **No Optimizations** | **CSE Only** |
|---|---|---|
| **noise_median.dcf** | 36.2ms | 35.8ms (-1.10%) |
| **saman_negative.dcf** | 25.5ms | 25.0ms (-1.96%) |
| **segovia_blur.dcf** | 142.7ms | 142.0ms (-0.49%) |

Overall, CSE alone produced about a 1-2% performance improvement. This is expected, as copy propagation is likely required to produce a larger benefit.

## Phase 4: Copy Propagation (CP)

**Enable this optimization with flag**: `--opt cp`

Copy propagation is performed as a CFG pass by visiting all basic blocks in the program. We then visit all the statements in each basic block. Firstly, on all assign statements, the left hand variable is killed, regardless of what the right hand side is. When there is an assign statement such that both the left and right sides are of type `CfgVariable`, it is possible to add this mapping to `tmpToVars`. Global variables do not participate in copy propagation due to the possibility of side effects.

Then, for most uses of variables, we check the `tmpToVars` map to see if any copies can be propagated. Function parameters do not participate in copy propagation due to the possibility of side effects.

A change from Phase 4 is that this optimization is now performed globally between basic blocks.

**Benchmarks:**

| | **No Optimizations** | **CP Only** | **CSE + CP** |
|---|---|---|---|
| **noise_median.dcf** | 36.2ms | 35.2ms (-2.76%) | 34.3ms (-5.24%) |
| **saman_negative.dcf** | 25.5ms | 25.2ms (-1.17%) | 24.9ms (-2.35%) |
| **segovia_blur.dcf** | 142.7ms | 142.7ms (0%) | 141.6ms (-0.77%) |

CP alone produces anywhere between a 0-2% performance improvement. However, when combined with CSE, it produces a 1-5% performance improvement.

## Phase 4: Constant Propagation (CSP)

**Enable this optimization with flag**: `--opt csp`

Constant propagation is performed in the same file as copy propagation using almost exactly the same procedure, except with constants. If variable is assigned to a constant, it is added to the `tmpToVars` map, and when the left-hand variable is reassigned, it is removed from the map.

Then, for most `CfgExpression` types, we check the `tmpToVars` map to see if any constants can be propagated. Function parameters and global variables do not participate in constant propagation due to the possibility of side effects.

From Phase 4, we have made a change that allows constant propagation to be performed globally though all basic blocks.

**Benchmarks:**

| | **No Optimizations** | **CSP Only** | **CSE + CSP + CP** |
|---|---|---|---|
| **noise_median.dcf** | 36.2ms | 35.6ms (-1.65%) | 34.1ms (-5.80%) |
| **saman_negative.dcf** | 25.5ms | 25.1ms (-1.56%) | 25.0ms (-1.96%) |
| **segovia_blur.dcf** | 142.7ms | 142.8ms (+0.07%) | 141.3ms (-0.98%) |

Overall, CSP alone delivers a 0-2% improvement. All the optimizations together up to this point produce an improvement of 1-5%.

## Phase 4: Algebraic Simplification

**Enable this optimization with flag**: `--opt asimple`

Algebraic simplification is implemented as a pass over the CFG. On each pass, it scans for `CfgExpression` that have `CfgValue` that are able to be simplified using simple algebraic rules. If the two sides are of type `CfgImmediate`, the value is computed and returned as another immediate. Otherwise, the following simplification rules are checked:

*Algebraic Simplification rules*:
`0 + x = x, a + 0 = x`

```
0 - x = -x, a - 0 = x
x * 1 = x, 1 * x = x, x * 0 = 0, 0 * x = 0
x / 1 = x, 0 / x = 0, x / 0 = 0 (undef)
x % 1 = 0
```

*Boolean Simplification rules*:

```
x && true = x, true && x = x
x && false = false, false && x = false
x || true = true, true || x = true
x || false = x, false || x = false
```

**Benchmarks:**
There was no discernable difference between the baseline and when only algebraic simplification is turned on. This is to be expected, as this optimization works noticeably well only when programmers write very unoptimized code.

## Phase 4: Dead Code Elimination (DCE)

**Enable this optimization with flag**: `--opt dce`

Dead Code elimination is implemented on the function scope level. Within this scope, it first passes over all blocks and computes the variables that are assigned to, and (upwards) used within each block. Then, it traverses the control flow graph to compute liveness as in class (though it differs slightly from the slides in that it does account for global variables). Then, it passes over all blocks and eliminates the dead code using the liveness information. There is a special caveat that dead code where variables are assigned to function calls are not eliminated, but just converted to the function call without the assignment since the function may have side effects.

Finally, it repeats the above steps once again until the code stops changing, since it is possible that eliminating some dead code exposes more dead code. There is an optimization possible here in terms of performance- I had an idea where the liveness computation could be modified so as to focus only on the “essential” statements i.e. returns and function calls- but this was not done due to time constraints.

**Benchmarks:**
There was no discernible difference between the baseline and when only dead code elimination is on. This makes sense because DCE requires CSP, CP, and CSP to create opportunities for dead code to be eliminated. Otherwise, DCE alone has very little benefit.

## Phase 5: Register Allocation

**Enable this optimization with flag**: `--opt regalloc`

## ScopeLiveness

ScopeLiveness takes in an entire scope of a function and reuses the liveness analysis from dead code elimination to produce a more granular liveness analysis between every statement for register allocation, rather than between every block. The only difference is that it deliberately ignores arrays and global variables as those are not assignable to registers. Essentially each statement has a set of CfgVariables that are live both before and after the statement so that we have a good representation to build the Register Interference Graph from.

## Interference Graph

At the method level, we convert the `Scope` into a `ScopeLiveness` and pass that into the `buildGraph` method of `Interference Graph` which builds an adjacency list representation of an undirected graph such that each node represents the web of a `CfgVariable` and the edges between each node represent an interference between webs, meaning two variables are live in the same program point at the same time. Using the `colorGraph` method, we assigned webs to registers using a graph-coloring algorithm similar to what we have seen in lecture. The only difference is that we prioritize using caller-saved registers to avoid saving to the memory due to the caller-callee convection.

## RegVisitor

A CFG visitor that performs the register allocation for each variable in the scope of each method. Using the `ScopeLiveness` and `Interference Graph`, we obtain a mapping of `CfgVariables` to `Real Registers` to be used by the codegen assembler.

The benchmark is run with all dataflow optimizations enabled. The test case with no register allocation uses only the stack for storing variables.

**Benchmarks:**

| | **No Register Allocation** | **Register Allocation** |
|---|---|---|
| **noise_median.dcf** | 39.8ms | 34.6ms (-13.6%) |
| **saman_negative.dcf** | 25.5ms | 25.0ms (-1.96%) |
| **segovia_blur.dcf** | 145.2ms | 140.2ms (-3.44%) |

Depending on the program, the register allocator either had a large impact, or a much smaller effect (2% on saman_negative). From the statistics on the leaderboard tests, we see that on the

hidden derby program, the register allocation was responsible for a 63% speedup (from 453ms to 167ms).

# Phase 5: Instruction Selection and Scheduling

**Enable this optimization with flag**: `--opt instsched`

This section comprises of a number of smaller optimizations that in total produced a significant benefit. These include various peephole optimizations and strength reduction that was performed on the assembly level.

## Efficient Array Reads / Writes

We know at compile time the length of all arrays that exist. Therefore, for loads and stores to an index in an array that is known at compile time (an immediate), a bounds check can be safely omitted if the immediate value is in bounds.

We also store local arrays on the stack in the direction such that higher indices have more positive stack offsets. In that case, we can avoid the use of `negq` for indexing into arrays on the stack for register reads. For local arrays reads and writes where the index is known at compile time, the exact stack location of the index can be easily calculated, and therefore can be done in one `movq` instruction.

Assembly instructions required for common array operations:

| | **Read (Imm)** | **Write (Imm)** | **Read (Reg)** | **Write (REG)** |
|---|---|---|---|---|
| **Global Arrays** | 2 | 2 | 4 | 4 |
| **Local Arrays** | 1 | 1 | 4 | 4 |

## Peephole Optimizations

The code generation occasionally generates assembly code that can be redundant. We perform several peephole optimizations where we read one or two lines of assembly at a time to see if any provably inefficient code can be optimized.

For this section, several optimizations were identified using Agner Fog's software optimization resources and latency tables. The test server operates on an AMD EPYC 7601 CPU, which is built on the Zen 1 microarcitecture from AMD. The instruction table we referred to is available here at this link.

***Instruction Optimization: enter/leave***

The `enter` and `leave` instructions are notoriously slow on x86 systems. The `enter` instruction has requires 12 ops and the `leave` instruction requires 2 ops. These both can be replaced by faster equivalents, namely:

```
enter $16, $0
leave
```

can be replaced by

```
push %rbp
movq %rsp, %rbp
subq $16, %sp

movq %rbp, %rsp
pop %rbp
```

Each of these instructions takes only 1 op, and therefore we are saving approximately 9 ops by using these instructions.

***Instruction Optimization: movq $0***

It is faster on x86 to use the xorq instruction instead of moving a 0 immediate to a register. Therefore, whenever the code does this, we emit a xorq instruction instead of movq. The only exception is when it is part of a comparison, as the xorq instruction overwrites the flag registers.

```
movq $0, %r10
```

Can be replaced, most of the time, by

```
xorq %r10, %r10
```

***Same location moves***

It is trivial to see that these instructions do nothing and can be safely removed.

```
movq %r11, %r11
movq -8(%rbp), -8(%rbp)
```

***Circular moves***

It can be seen that the move below from `%r11` back to `%r10` is not needed because the value of `%r11` is already in `%r10` since that's where it was copied from.

```
movq %r10, %r11
movq %r11, %r10
```

We can samely remove the second `movq` instruction from this sequence.

### Strength Reduction

For multiplication and division by a power of 2, it is often faster to emit a shift rather than using the `idivq` and `imulq` instructions.

According to the latency tables, `idivq` has a 14-47 cycle latency on Zen 1, and `imul` has a 4 cycle latency on Zen 1. In the strength reduction, we reduce all power of 2 multiplications to a shift left logical, `shl` instruction, and all power of 2 divisions to a shift right arithmetic, `sar` instruction.

This is only possible when the dividend or multiplicend is known at compile time. This optimization has much greater benefit when CSE, CP, and CSP are enabled to allow more operands to be known at compile time.

**Benchmarks:**

| | **All Dataflow Optimizations + Reg Allocation** | **With Assembly Optimizations** |
|---|---|---|
| **noise_median.dcf** | 34.6ms | 33.4ms (-3.48%) |
| **saman_negative.dcf** | 25.0ms | 24.5ms (-2.00%) |
| **segovia_blur.dcf** | 140.2ms | 137.0ms (-2.28%) |

The assembly optimizations and strength reduction described in this section produced a 2-3% benefit over the baseline when all dataflow and optimizations are enabled.

## Phase 5: No-Op Elimination

Essentially, go through the control flow graph and remove all blocks that are no-ops. This is because there may still remain some no-ops that cannot be simplified by just combining basic blocks in the process of maximizing basic blocks. This essentially is done by “shortcutting” any blocks that go to a no-op to go to the next block that is not a no-op.

**Benchmarks:**
There was no discernible difference between the baseline and when the no-op elimination optimization was enabled. This is likely due to the small size of the optimization (only removes `jmp` instructions).

## Phase 5: Fall-Through Optimization

Take advantage of the fact that falling-through a branch (i.e. branch not taken) is faster in assembly than branch taken. This optimization requires inverting some conditions like so:

| | | | |
|---|---|---|---|
| Loop | //a<br>while (x > 2) {<br>//b<br>}<br>//c | Yes | // a<br>(x <= 2) ? GOTO c<br>// b<br>GOTO a<br>// c |

Special care must be taken with short-circuiting, since it may be sometimes necessary to negate a condition/ not negate the condition for the best fall through.

**Benchmarks:**
We observed about a 5% performance increase on segovia_philbin.dcf.

## Phase 5: Loop Invariant Code Motion

Implement loop invariant detection, and code motion to move the invariant code out of the loop. Not much different than the slides in class. However, to ensure correctness, I also implemented a guard clause for the invariant code (since it is possible that the loop may never be run, in which case the invariant code will not be run either).

**Benchmarks:**
Unfortunately, we found that this optimization either made no difference or made the derby program slower on the leaderboard, so we opted not to include it.

# Contributions

In Phase 5, An Bo and Alex collaborated to write a graph-coloring register allocator, specifically Alex wrote the `ScopeLiveness` program and An Bo wrote the `InterferenceGraph` and `RegVisitor` program as well implemented global `CSE` and global `CP`. Yoland integrated the register allocator with the code generation, performed the benchmarks, and wrote the peephole optimizations. Alex wrote the no-op elimination code, fall-through optimization, and loop invariant optimization. We collaborated in writing the documentation.