Phase 4 Design Document

General Framework for Dataflow Optimizations

We first implemented single static assignment (SSA) on our control flow graph (CFG), which transforms every variable in the cfg so that each variable is only assigned to once. We achieved this by introducing versions for every variable ie. t1 - SSA -> t1_v2. We implemented SSA using the dataflow equations specified in section 3 of the SSA book. We did not perform SSA on global variables, allowing them to be reassigned throughout the program.

Our strategy for performing optimizations on our cfg was to first convert to SSA, optimize the code, then convert back into flat variables. We have implemented our optimizations such that when specified, we perform a given optimization once. In the next phase, we will stack the optimizations until the code no longer changes.

We implemented copy propagation and dead code elimination. In the next phase, we will implement constant propagation and common subexpression elimination.

Copy Propagation

In SSA, copy propagation was straightforward to implement: every time a variable is copied (i.e. t1_v0 <- t2_v0), the value of t1_v0 never changes. We iterate through the tree to find all of the copy statements, then iterate again to propagate the copies. We handled chained copies like by replacing copies with their most fundamental values. For example, given the statements t1 <- t2 and t2 <- t3, we would replace t1 and t2 by t3 everywhere in the cfg.

Performing copy propagation across phi expressions makes the SSA non-conventional as defined in section 2.6 of the SSA book. Converting from conventional SSA to flat variables involves giving all versioned variables that are related via phi expressions a unique name. This transformation does not work in non-conventional SSA, and applications of it give rise to the lost copy problem.

To convert back to conventional SSA, we must implement parallel copy instructions into the CFG. Like phi instructions, parallel copy instructions can’t be implemented directly in assembly, so we must cleverly convert them into a set of regular copy instructions. This process potentially introduces many temporaries. Using algorithm 3.6 in the SSA book we minimized the number of these temporaries created.

We did not propagate copies of global variables since we allow them to chance throughout the program while in SSA form.

Example:

```
1   ---------------------------------
2   Original Code
3   ---------------------------------
4   import printf;
5
6   void main() {
7     int a, b, c;
8
9     b = 1;
10    a = b;
11    c = a;
12
13    printf("%d %d %d", a, b, c);
14  }
15
16  ---------------------------------
17
18  method after ssa construction:
19  Method: main
20  -------------------------
21  BasicBlock 0
22    Parents: []
23  | t4_v1 <- $1 |
24  | t2_v1 <- t4_v1 |
25  | t1_v1 <- t2_v1 |
26  | t3_v1 <- t1_v1 |
27  | call printf(t"%d %d %d", t1_v1, t2_v1, t3_v1) |
28  | nowhere |
29  -------------------------
30
31  method after ssa destruction:
32  Method: main
33  -------------------------
34  BasicBlock 0
35    Parents: []
36  | t4 <- $1 |
37  | call printf(t"%d %d %d", t4, t4, t4) |
38  | nowhere |
39  -------------------------
```

In the examples, t4_v1 is copied into t2_v1 and implicitly copied into t1_v1 and t3_v1. After copy propagation the copy instructions are removed and the three copies of t4_v1 are replaced with the same value.

Dead Code Elimination

We performed dead code elimination in two steps. First, we compute the set of all variables that are sources for instructions. Then, we remove all instructions that store results in non-global variables that are never used.

Our current version of dead code elimination does only one iteration, however we ideally want to remove as much dead code as possible.

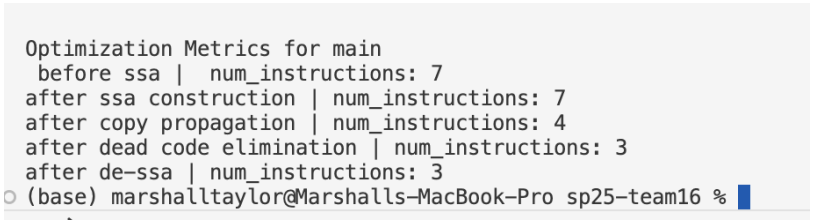
Example

```
Original Code
import printf;

void main() {
  int a, b, c;

  a = 5;
  b = 6;
  c = a + b;

  printf("%d %d", a, b);
}

method after ssa construction:
Method: main
-------------------------
BasicBlock 0
  Parents: []
| t4_v1 <- $5 |
| t1_v1 <- t4_v1 |
| t5_v1 <- $6 |
| t2_v1 <- t5_v1 |
| t6_v1 = t1_v1 + t2_v1 |
| t3_v1 <- t6_v1 |
| call printf(t"%d %d", t1_v1, t2_v1) |
| nowhere |
-------------------------

method after dead code elimination:
Method: main
-------------------------
BasicBlock 0
  Parents: []
| t4_v1 <- $5 |
| t1_v1 <- t4_v1 |
| t5_v1 <- $6 |
| t2_v1 <- t5_v1 |
| t6_v1 = t1_v1 + t2_v1 |
| call printf(t"%d %d", t1_v1, t2_v1) |
| nowhere |
-------------------------
```

The single round of dead code elimination accurately removed the definition of t6_v1 since it is never a source for another instruction. In the next phase, we will modify it to continue to remove dead code until all definitions are used in functions.

Extras

For this phase, we introduced metrics that helped us see how effective our optimizations were at reducing the number of instructions. Here's an example:



We would like to expand this in the next phase to compute what is expected number of assembly instructions of the method after each optimization.

Difficulties

We were able to pass all of the test cases at the end of phase 3 and are able to pass them all after introducing SSA and dataflow optimizations in our cfg. One current issue with our cfg is that immediates do not appear as a source in any instruction besides constant loads. This makes it very complicated to implement constant propagation and algebraic simplification.

Contribution

Marshall and Owen wrote the code for converting the CFG to and from SSA form. Marshall wrote the code for copy propagation and dead code elimination.Owen wrote the command line interface for activating the optimizations.