6.035 Project 4 Design Proposal

Group le03

Introduction
As per the requirements of the previous project, we have built a dataflow analysis framework and implemented global common subexpression analysis on top of it. Ultimately, we left the bulk of the optimizations unimplemented due to time constraints. Thus, we've scheduled basic optimizations like copy propagation and dead code elimination for this project - we've begun work on those implementations already to compensate.

Planned Optimizations
Copy Propagation and Constant Propagation
Utilizing the same framework as CSE, copy propagation and constant propagation will be relatively straightforward implementations after experience implementing CSE. Certain modifications to how constants are tracked may be required, but otherwise the same concepts of mapping variables and merging states still apply.

Dead Code Elimination
Using our dataflow analysis framework from the previous project, we run a liveness analysis on our IR (this will also be useful later in register allocation). The results of this can be used to perform dead code elimination and prevent the execution of unnecessary instructions. The major implementation difference from CSE will be in running the analysis backwards.

DCE is perhaps one of the highest priority optimizations, since without it CSE and CP can be actually detrimental to performance.

Unreachable Code Elimination
This may require more aggressive algebraic simplification to take full advantage of unreachable code optimizations, but even with simple if(true) {} statements we can implement simpler levels of optimization relatively easily. Thus, we've scheduled this optimization at a higher priority than register allocation.

Register Allocation
Our top priority will be implementing a graph coloring-based register allocator. It has potential to save operations across the board, not only in certain cases, so this optimization is expected to produce huge performance gains. Our focus will be only implementing basic coloring and then using papers and research to optimize certain areas, especially with calling conventions, presplitting, and web merging.

Algebraic Simplification
Our current AS implementation simplifies the ANTLR tree prior to main optimizations. There is still room for algebraic simplification *after* optimizations, however. For example, constant propagation may simplify *x = 5; a = x + 5* to *a = 5 + 5*, which should be simplified to *a = 10*. Following code like *b = a + 5* would get simplified to *b = 10 + 5* after another round of CP and then *b = 15* with AS, so it is possible that we run these optimizations multiple times as long as

code gets compacted. We add this here because we can achieve certain optimizations almost for free.

Code Hoisting
As with similar optimizations, hoisting can be written as a transfer function that operates using our dataflow analysis framework. By searching for invariant expressions within loops, we can eliminate redundant operations. For example, consider a loop where each element of an array is set to the expression *x + y*. By hoisting, this expression could be computed outside of the loop and as many add operations as there are iterations of the loop will be saved.

Instruction Selection / Peephole Optimizations
Similar to algebraic simplification, we see instruction selection as a set of optimizations from which we can pick the "lowest hanging fruit." For one, we plan on using LEA for array offset calculations. Strength reduction is included in this section, replacing multiplies with shifts or adds when possible. Other optimizations will be implemented as our web research and remaining time dictate - we'd like to remove consecutive POP and PUSH operations from consecutive calls, for example.

List Scheduling
List scheduling is our lowest priority - given that register allocation is expected to have the greatest impact on our performance and the complexity of list scheduling, we hope to focus on a good implementation of register allocation in lieu of this optimization.

"Full Optimizations"
Full optimizations will need to consider the order in which to apply the optimizations. Based on whether optimizations work with ASTs, blocks, or assembly code as well as interactions between optimizations, we tentatively want optimizations applied in the following order:

- Algebraic simplification (ANTLR level, to make following optimizations more effective)
- Common subexpression elimination
- Copy propagation (fixes CSE assignments to temporary variables when possible)
- Constant propagation
- Dead code elimination (cleans up temporary variables created by CSE and CP)
- Unreachable code elimination
- Algebraic simplification (assembly level)
- Copy propagation
- Dead code elimination (once again, since optimizations like unreachable code elimination and copy propagation may have created dead code)
- Code hoisting (to the best of our knowledge, we are assuming prior optimizations don't complicate code hoisting and it doesn't matter when this occurs)
- Instruction Selection / Peephole Optimizations (these rely only on assembly-level code, and applying this ahead of time may obfuscate intent and complicate other optimizations)
- Register Allocation (since other optimizations may potentially add or remove code, we don't want to allocate registers until we are ready to generate code)

Plan of Action
Given this set of largely optimizations, we decided to parallelize work and distribute them based

on complexity and implementation time. Working off a deadline of 5/16, the division of labor is as follows:

- Saji: code hoisting by 5/2, basic instruction selection and peephole optimizations (including additional research) by 5/8
- David: dead code elimination by 4/27, assembly level algebraic simplification by 5/2
- Josh: copy propagation and constant propagation by 4/23
- Saif: unreachable code elimination by 4/25
- Josh and Saif: register allocation by 5/2
- Josh, Saif, and David: parallel enhancements to register allocation (calling conventions, presplitting, etc) by 5/12
- All: work on design doc (document individual parts) and finish by 5/16