

In this assignment you will implement different program optimizations that use the results of data-flow analysis and compare the generated MIPS code in terms of the number of cycles required for execution. This assignment should be done with groups of two or three. Each program optimization will be assigned a weight that estimates relative implementation difficulty. Your final group project will require implementing a total weight of 4 per person in your group. (Groups of size 2 must do a project of weight 8, groups of 3 must do a project of weight 12.)

1 Motivation

Program optimizations such as dead-code elimination, copy propagation, constant propagation, common subexpression elimination, code motion, PRE, value numbering, and induction variable elimination are commonly applied to low-level intermediate representations to reduce the overall instruction count for a program. The reduction in instruction count resulting from these program optimizations typically results in a performance improvement as well. Also, higher-level transformations often assume various subsets of these optimizations for cleaning up the code after parts of the high-level transformation have been applied.

2 The Assignment

Your assignment is to implement *dead-code elimination* and other program optimizations adding to your group's required implementation weight in the MiniJava compiler.

- Copy propagation (weight: 1)
- Constant propagation without constant folding (weight: 2)
- Constant propagation with constant folding (weight: 3)
- Common sub-expression elimination (weight: 3)
- Code motion or hoisting (weight: 5)
- Induction variable elimination (weight: 6)
- Value numbering (weight: 7)
- Partial redundancy elimination (weight: 8)

It should be possible to specify each optimization you implement on the command line. For example,

```
% java -jar MiniJavaCompiler.jar --copyprop file.java
% java -jar MiniJavaCompiler.jar --copyprop --deadcode file.java
...
```

You should also have a “-fast” option that performs your optimizations in an order that you believe will provides the best possible performance in most cases.

The compiler should just print one line of output indicating each program optimization that is being applied. For example,

```
Applying constant propagation
Applying dead-code elimination
Applying copy propagation
Applying dead-code elimination
Applying register allocation via graph coloring
```

Your group will need to create at least two test input programs per optimization implemented. To check correctness, compare the output of the MIPS code running with SPIM to the output of the same program run with the JVM. Design your test cases so that they will break an incorrect implementation of the implemented program optimizations. Feel free to share such test cases with other groups in the class, but each group must submit their own two test cases. Individuals not working with a partner must also generate two test cases per optimization.

Your group will also need to create and submit two benchmark input programs. The benchmark programs should be designed to show off one or more of your program optimization implementations. The benchmark programs from the whole class will be run through each submitted compiler using the “-fast” option. We will compare the results from each group in class.

Graph the results for each program optimization you implement, your “-fast” option, the MiniJava compiler provided for the register allocation project, and your version of the compiler that performs register allocation for both of your benchmark programs. Explain the results.

3 Getting Started

1. You will need to implement the following to support many of the possible program optimizations:
 - A method in FlowGraph for removing a node. See FlowGraph.rmFlowGraphNode().
 - A method in FlowGraph for generating a new list of instructions after performing an optimization. See Liveness.show() for an example.

- A data-flow analysis framework has been provided and the Liveness package exhibits usage of the framework.
 - A DeadCodeElim source file has been posted at the class mailing list site, but you must implement it.
 - New subclasses for Assem.Instr and a Mips/Codegen.java that generates the new instruction subclasses have been posted at the at the class mailing list site. You must implement the method in Codegen.java that regenerates MIPS code for the Assem.Instr subclasses.
2. I recommend working out some examples by hand before jumping into the implementation of any program optimization.

4 Your Report

The report is an essential part of your completed assignment. Use it to describe your reasoning behind each program optimization you implement, assumptions, difficulties, insights, and results. Organize and present your document as if it were the only basis for your assignment's grade. The format of your writeup is up to you, but it should minimally include the following:

- Introduce the main goals of the project and in a couple of sentences summarize what you have accomplished.
- Briefly describe the optimizations you chose to implement. Motivate your selection of those optimizations and the ordering you decide to use for the `-fast` command-line option.
- Present and explain graphs that exhibit the performance difference between MIPS programs generated by the provided MiniJava compiler and and your extended MiniJava compiler.
- How did you test your program optimization implementations?
- What problems did you encounter while implementing the program optimizations? If you knew someone who was just about to start work on this assignment, what advice would you give them? Do you agree with provided weightings?
- What, if any, outside sources did you use (e.g., articles, books, other students)? This is particularly important. It's OK to look at books and articles and speak with your professor and fellow students (although sharing code and working together is strictly forbidden), but as with any scientific document, you should always cite your references and collaborations. You can either cite collaborations in footnotes or in a separate Acknowledgment section.

There's no exact number of pages you should write, but if you've got between four and six then you're in the right ballpark.

5 Hints for doing the assignment

You should start this assignment early!! Make sure to start a subversion log at least a week before the assignment is due so as to make a last minute 24-hour extension possible. I would recommend spending a couple hours a day on the project starting now. Start writing your report as soon as you have dead-code elimination implemented. In fact, write the whole thing based on dead-code elimination and then add the other optimizations in as they are completed. A well-written report with some missing implementation guarantees a much higher grade than a poorly written report with all the implementation.

I can look at your code and help point you in the right direction, but the amount of help I can give may be inversely proportional to the amount of time until the due date. By no means should you spend several hours trying to figure out a weird bug; consult me for help. When e-mailing me about the project, send a copy of the relevant section of your code (not as an attachment; send it as text pasted into your message). Give a good description of the problem including information about the stack in a debugger.

6 What to turn in

Turn in a hard copy of the report and email a copy in pdf format to mstrout@cs.colostate.edu.

Create a jar file that includes all of the bytecode files, the source files, your test cases, and your benchmarks. Also submit a README that gives specific command-lines for running the jar file on your provided test cases and benchmarks.

7 Due date

This assignment is due Friday November 2rd, **at 2:00pm** and is worth 10% of your final grade. Late assignments will be penalized 10% per day.