

In this assignment you will implement different register allocators and compare the generated MIPS code in terms of the number of cycles required for execution.

1 Motivation

Register allocation is arguably the most important program optimization in any compiler. The MiniJava compiler version being provided either uses the stack to evaluate expressions (`-regalloc-stack`) or create the Assem intermediate representation with temporaries and spills all temporaries to memory (`-regalloc-spillAll`). Using the stack to perform expression evaluation prevents instruction level parallelism and require frequent memory loads and stores. The implications of spilling all the variables is that a MIPS instruction where temps have not been allocated

```
add    t0, t1, t2
```

will be expanded into four MIPS instructions using the load-load-compute-store concept

```
lw     $s0, -16($fp)
lw     $s1, -20($fp)
add    $s2, $s0, $s1
sw     $s2, -24($fp).
```

The code shown above assumes that the temporaries `t0`, `t1`, and `t2` are placed at memory locations `-24(%fp)`, `-16(%fp)`, and `-20(%fp)` respectively. The MIPSframe `tempRegSet()` method provides temporary registers for use when generating load-load-compute-store code. Currently three registers are provided, but the load-load-compute-store paradigm can be implemented with as few as two temporary registers.

Due to the major code expansion that occurs when spilling all temporaries, register allocation can reduce the number of cycles executed by up to 75%!

2 The Assignment

Your assignment is to design and implement two different versions of register allocation, compare the performance of the generated MIPS code amongst the different versions and the provided MiniJava compiler, and report on your design decisions, testing strategies, and performance results. Of the two versions of register allocation at least one of the versions must be based on graph coloring techniques and all versions must be able to handle spilling. You will design each of the versions by selecting amongst the following options:

- Basic approach:
 - Use a greedy algorithm to assign registers to variables, and spill when run out of registers.
 - Use a greedy coloring algorithm that only builds the interference graph once.
 - Use a simplifying graph coloring algorithm.
- Possible spilling heuristics: random spill choice, the temporary with the fewest uses, some combination of the number of uses and defs and degree in the interference graph, other possibilities that you design
- Type of spilling: optimistic or pessimistic
- Coalescing options: no coalescing, Briggs, other possibilities that you design
- Other options that you design

It should be possible to select each of your versions on the command line. For example,

```
> java -jar MJ.jar --two-pass-mips --regalloc-v1 file.java
> java -jar MJ.jar --two-pass-mips --regalloc-v2 file.java
...
```

You will need to create at least two test input programs. 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 register allocation. Feel free to share such test cases with other people in the class, but each person must submit their own two test cases. Also, performance in terms of dynamic cycles executed should be better when using register allocation than the provided MiniJava compiler.

You will also need to create two benchmark input programs. The benchmark programs should be designed to show off one or more of your register allocation implementations. The benchmark programs from the whole class will be run through each submitted register allocator. There are things you can do that are not described here that will result in better code than what is currently generated. If you discover these and decide to do them, they can be considered as “other options that you design”.

Graph the dynamic instruction count for the code generated by your versions of register allocation and the provided MiniJava compiler for both of your benchmark programs (see the Instruction Counter under the MARS tools). Explain the results.

3 Getting Started

Register allocation occurs after code generation and before the prologue and epilogue are added. Register allocation operates on instructions that are in the Assem representation. In this representation, each instruction has a list of source and destination temps associated with it. The problem that register allocation solves is that not all of the temporaries are associated with real registers.

The goal is therefore to replace all of the Temp instances in the Assem instructions with Temp instances directly associated with registers. If a Temp is to be spilled instead of replaced by a register Temp, then extra instructions to load a temporary upon use and store a temporary upon definition. See the spill-all code to see an example of adding Assem instructions.

The MJRegalloc.java file in the provided MJDFASStart (see google mailing list uploaded files) contains a driver that can be run with a `-regalloc-spillAll` option to generate code where register allocation occurs via spilling all Temps. The `-regalloc-color` option will cause a control flow graphs and liveness analysis results to be written to separate files for each method. `Liveness.getLiveOut(n)` returns a list of temps that are live after the execution of the instruction in the given node n. `FlowGraph.getInstr(n)` returns the Assem instruction for the given control flow graph node.

In MIPSframe.java, the sets of callee-saved and caller-saved registers have been reduced for testing purposes. You will want to expand them for running your register allocation on benchmarks.

Your interference graph data structure can inherit from the provided graph data structure. Just keep in mind that the interference graph is undirected.

Work out some examples by hand before jumping into the implementation of any one version.

I recommend using dot to visualize the control-flow graph and the interference graph.

4 Your Report

The report is an essential part of your completed assignment. Use it to describe your reasoning behind each of your register allocation designs, 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.
- Describe the options that you used in each of your register allocation implementations. Motivate your selection of the various options.
- Present and explain graphs that exhibit the performance difference between MIPS programs generated by the provided MiniJava compiler and the two or three versions of register allocation in your extended MiniJava compiler.
- How did you test your register allocation implementations?
- What problems did you encounter while developing your program? If you knew someone who was just about to start work on this assignment, what advice would you give them?
- 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 two and four then you're in the right ballpark.

5 Hints for doing the assignment

I can not emphasize the importance of your report enough. It is entirely possible that I will grade your whole project based solely on your report.

You should start this assignment early!! Consider the following possible schedule for completing this project:

1. Write some smaller test cases.
2. Look at the control flow graph and liveness results generated by the given compiler for the smaller test cases. Figure out how to access liveness results in the compiler.
3. Design and writeup your first register allocation version.
4. Write test cases designed to find mistakes in the register allocator.
5. Implement and test your first register allocation version.
6. Write your benchmarks.
7. Figure out how to grab the cycle count information from MARS, design the graphs, and generate the graphs.
8. Writeup the results from your first register allocation version.
9. Design, writeup, implement, and test your second register allocation version.
10. Finish your report.

You will probably have to redo steps when you find errors or problems. Also, notice in the above suggested sequence that you could turn in a report after only doing one register allocation implementation. It would be MUCH better to do that than to get all the register allocation implementations done, but not finish your report in time.

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 tar 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 a jar file on your provided test cases and benchmarks.

7 Due date

This assignment is due Thursday October 29th, **at 2:00pm** and is worth 10% of your final grade. Late assignments will be penalized 10% per day.