PP5: Register Allocation and Optimization

Submission Open on 4/8/2015
Due at Midnight 4/20/2015

Goal

In this project, you will improve the back end for your compiler, by optimizing the code that it generates. The goal is to reduce the running time of the generated code. You will achieve this by implementing compiler optimizations and by improving the register allocation schemes using both simple and advanced techniques.

You will start with your solution code generation pass to PP4, and modify the existing the MIPS code generator, which internally performs register allocation (extremely poorly!). There are 2 aspects of this project.

1) Register Allocation - You will add another pass to perform dataflow analysis, specifically live variable analysis, to help improve the register allocation scheme even more. This part is compulsory for all groups.

2) Optimization – You can *optionally* implement any optimization techniques learned in class or elsewhere. It would be a good idea to generate a common infrastructure for these optimizations and the live-variable analysis above.

Starter files

The starting project contains the following files (the boldface entries are the ones you are most likely to modify, depending on your strategy you may modify others as well):

`Makefile`	builds project
`main.cc`	`main()` and some helper functions
`scanner.h/.l`	our scanner interface/implementation
`parser.y`	yacc parser for Decaf (replace with your PP4 parser)
`ast.h/.cc`	base AST node class (replace with your PP4 code)
`ast_type.h/.cc`	AST type classes (replace with your PP4 code)
`ast_decl.h/.cc`	AST declaration classes (replace with your PP4 code)
`ast_expr.h/.cc`	AST expression classes (replace with your PP4 code)
`ast_stmt.h/.cc`	AST statement classes (replace with your PP4 code)
`codegen.h/.cc`	code generator class (replace with your PP4 code)
`tac.h/.cc`	interface/implementation of TAC class and subclasses
`mips.h/.cc`	interface/implementation of our TAC-to-MIPS translator
`errors.h/.cc`	error-reporting class for you to use
`hashtable.h/.cc`	simple hash table template class
`list.h`	simple list template class
`location.h`	utilities for handling locations, `yylloc`/`yyltype`
`utility.h/.cc`	interface/implementation of our provided utility functions
`samples/`	directory of test input files
`run`	script to compile and execute result on SPIM simulator

Use `make` to build the project. It reads input from stdin and you can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
$ ./dcc < samples/program.decaf > program.asm
```

The output is MIPS assembly that can be executed on the SPIM simulator. The `spim` executable, which actually executes the code, is in `~ansingh/Public/spim-install/`. We've also provided a convenient shell script at `~ansingh/Public/spim`, which adds the command-line option to preload the exception handling library for SPIM, so you don't need to do it by yourself. You can run your output assembly program through the `-file` argument:

```
$ ~ansingh/Public/spim -file program.asm
```

The SPIM simulator we provided is a special version (applied the modifications suggested in http://www.cs.colostate.edu/~mstrout/spim/keepstats.html) that supports the `-keepstats` option, which prints the number of instructions executed:

```
$ ~ansingh/Public/spim -keepstats -file program.asm
```

This option should be specified before the `-file` option and will make SPIM to output the following message after the program's normal output:

```
Stats -- #instructions : 85662
         #reads : 32274  #writes 22039  #branches 9003  #other 22346
```

We have also included a `run` script that compiles and runs a Decaf program in the SPIM simulator with the above commands:

```
$ ./run samples/program.decaf
```

Register Allocator Implementation

The current `Mips` class only makes use of 2 registers out of the 31 possible and always spills registers to memory for assembly code emission. This strategy is very conservative and leads to many unnecessary loads and stores. To make your code more efficient, you need to write a register allocator that uses all *general-purpose* registers to avoid spilling. To make this job easier, we have provided a list of general-purpose registers in `mips.cc`. In addition, the `Location` class now supports two helper functions, `SetRegister()` and `GetRegister()`, so your register allocator assign registers to the locations in the 3-address code via these functions, and the `Mips` class would take the assigned registers and generates appropriate code.

Your register allocation process should consist of the following 3 steps:

1) Liveness analysis – you should implement a live variable analysis to understand the live range of each variable. When the live ranges of two variables are overlapping, they *interfere* with each other and thus cannot be allocated to the same register.
2) Interference graph construction – in the *interference graph*, each node represents a variable. When two variables interfere with each other, an edge between these two variables is added into the graph.

3) Graph coloring – once the interference graph is constructed, you can find a *k*-coloring for this graph via *Chaitin's algorithm*, where *k* is the number of general-purpose registers, and variables of the same color would be assigned the same register. Occasionally, Chaitin's algorithm may fail to find a k-coloring. When it happens, you need to pick one variable to *spill* and assign no register to it.

Your first goal is to implement a live variable analysis for each function. This analysis requires you to construct the *control flow graph* (CFG) of the function. In the control flow graph, a TAC is represented as a node, and there is a directed edge from one TAC to another if the latter is either textually after the former, or can be jumped from the former. Once you have the control flow graph, you can use the following algorithm to compute the live variables at each program point:

```
Algorithm LiveVariableAnalysis(CFG):
Initialize: In(BB) = {} for each BB in CFG
1.  changed := true
2.  while changed:
3.      changed := false
4.      for each basic block, X, in CFG:
5.          old_IN = IN(X)
6.          OUT(X) = Union(IN(Y)) for all successors Y of X
7.          IN(X) = GEN(X)  + (OUT(X) – KILL(X))
8.          if old_IN ≠ IN(X):
9.              changed := true
10.         endif
11.     endfor
12. endwhile
```

In the above algorithm, KILL(X) and GEN(X) are the variables that are defined and used in basic block X respectively. When the algorithm ends, IN[X] and OUT[X] would be the sets of live variables before and after TAC. This algorithm iteratively updates IN[X] and OUT[X] till there is no more variable to be added into these sets.

Your second goal for the project is to build the interference graph and apply Chaitin's *k*-coloring algorithm on the graph. Since the IN and OUT sets are computed, building the interference graph is straightforward: for each X, the variables in KILL(X) would interfere with the variables in OUT[X], so we should add an edge for each pair of variables in the two sets. To find a *k*-coloring, you can pick any node whose degree is less than *k*, remove that node from the graph, recursively find a *k*-coloring for the resulting subgraph, then put the node back and assign a color different from all of its neighbors. If there is no such node to remove, you should choose a variable to *spill* and remove the variable without assigning any register to it. A node with highest degree is usually a good candidate to spill, as removing it maximizes the chance to find a node with degree less than *k*.

Note that the live variable analysis need only be done inside each function. You are not required to go across functional boundaries. This may require you to be conservative when estimating the effects of a function call. For example, when there is a function call within the live range of a variable, the value of the variable needs to be preserved across the function call. There are two policies to preserve the values: the ***caller-save*** policy and the ***callee-save*** policy. The caller-save policy requires that all registers whose values should be preserved are written to memory before a function call and read back after the call; the callee-save policy asks each function to backup all registers it uses and restore them back before returning. Conventionally, certain registers are callee-saved, and the others are caller-saved, and it is up to the compiler to decide whether a variable should be assigned a caller- or callee-saved register. To simplify the design, you can just stick with the caller-save policy for all registers. You will need to make necessary changes to `tac.cc` to save and restore the registers for all variables living after a function call.

Once you finish your register allocator, you should use the `-keepstats` option on SPIM to see how much your register allocator improves the execution. Specifically, as you increase the number of registers used, you will reduce the number of register spills, hence generate fewer load and store instructions. Your grade will be partially based on how fast your generated programs are. We will use a simple model to compute the performance cost of your output MIPS program: an ordinary instruction takes 1 cycle, a branch takes 2, and a load/store takes 10 cycles. However, keep in mind that your optimizations cannot affect the output of the program. If your optimized code no longer matches the expected output, you will lose points.

You are free to implement any optimizations, such as copy propagation and common subexpression elimination. It will probably be easier to make it a separate pass that executes just before the code generation pass. That way, the code generator can use the results to intelligently allocate registers. You may need to create additional structures to store data between passes. After one optimization is done it may so happen that another optimization that was not feasible earlier is possible. So it might be a good idea to rerun an already run optimization pass. Some of the recommended optimizations that can be implemented are:

4) Dead Code Elimination.
5) Constant Folding.
6) Common Subexpression Elimination.
7) Constant Propagation.
8) Forward Copy Propagation.

Combination of well thought optimization can also lead to better results.

Grading

The final submission is worth 100 points. We will compare the performance cost of each MIPS program output by your compiler with that by our reference solution, which only does the register allocation described above. The performance cost of your optimized programs should be no more than 20% worse that ours. The performance cost is computed as follows:

```
#reads * 10 + #writes * 10 + #branches * 2 + #other * 1
```

For example, if the MIPS program output by the reference solution executes 50 loads/stores, 40 branches and 220 arithmetic instructions, then the performance cost is 800 cycles, and your output cannot use more than 960 cycles.

In addition, we will have a little competition for this project. We have picked 10 programs to make a benchmark suite. For each program in the benchmark suite, the top 5 groups will get 1 bonus point! We have also prepared a real-time score board at the following URL. Try to make your compiler generate the fastest code!

```
http://www-personal.umich.edu/~ansingh/pp5-scoreboard.html
```

Submission

Use the following command on any CAEN's Linux server to submit your programs:

```
~ansingh/Public/submit.sh 5 <path_to_your_project_directory>
```