CS 8803-008: Project Phase 2 Report

Turn-in includes:

1, Source code: tiger.cpp, scope.h, RegisterAllocation.cpp, RegisterAllocation.h, OptRegisterAllocation.cpp, OptRegisterAllocation.h, LivenessAnalysis.cpp, LivenessAnalysis.h, CodeGeneraion.h, BackEndHelper.cpp, BackEndHelper.h, Makefile

2, An executable of the compiler: tiger

3, Generated MIPS code: naivefactorial.s, optfactorial.s, naivelotsoints.s, optlotsoints.s

4, This report.

Design internals

In naïve register allocator, there's no analysis at all. For each instruction, registers are allocated to its operands. Then after it is executed, its result is stored back into memory and the allocated registers are released.

In whole function register allocator, liveness analysis is done for all variables in each function with the following equations

$$in[I] = (out[I] - \text{def}[I]) \cup use[I]$$

$$out[B] = \bigcup_{B' \in succ(B)} in[B']$$

Live range and interference graph for variables are built based on liveness analysis. Then register allocation is performed using Briggs' optimistic coloring algorithm[1]:

While G cannot be R-colored
While graph G has a node N with degree less than R
Remove N and its associated edges from G and push N on a stack S
End While
If the entire graph has been removed then the graph is R-colorable
While stack S contains a node N
Add N to graph G and assign it a color from the R colors
End While
Else graph G cannot be colored with R colors
Simplify the graph G by choosing an object to spill and remove its node N from G
(spill nodes are chosen based on object's number of definitions and references)
End While

Both allocators support IR for Tiger Language, including assign, add/sub/mult/div/and/or, goto, breq/brneq/brlt/brgt/brgeq/brleq, return, call/callr, array_store/array_load/assign(array assignment)

Both allocators also supports operations on int and float type variables.

In instruction selection and code generation, we use one to one match method to generate one or more assembly codes for each IR code. It is very straightforward to find corresponding MIPS code for memory, control, arithmetic IR code.

One difficulty is to do function call, in our design, we record the total space, including the space of local variables, outscope variables and return address for every function. When meeting call/callr instruction, we reduce the $sp by the amount of space size of callee function in caller function, which means we allocate space for callee function, and store needed variable into that area. When meeting return, we add the $sp and deallocate the space. To ensure the correctness of program and avoid reusing the same register, we store all variables that reside in registers back to memory before function call and load them back after finishing function call.

How to Build:
By running **make** on **Ubuntu**, the executable file **tiger** will be generated in current directory.

How to Run:
Our executable program has three parameters
-f, input the name of ir file
-o, type ‘u’ means choosing the unoptimized naive register allocator; type ‘o’ means choosing the optimized whole function register allocator.
-h, print the help information
For example:
./tiger -f factorial.ir -o u
means we use factorial.ir as input to our compiler and choose naive register allocator
./tiger -h
will output help information,

```
#define USAGE
"usage:\n"
"    tiger [options]\n"
"options:\n"
"    -f [ir_file]    ir file to generate MIPS code\n"
"    -o [reg_allocator]    'o' means optimized register allocation, 'u' means unoptimized\n"
"    -h                    Show this help message\n"
```

The compiler will generate a file named MIPSCode.s which contains assembly code of the input ir file. MIPSCode.s can load and execute on the supplied SPIM simulator.

The next section shows the detailed code quality generated by our compiler.

Code quality comparisons

	factorial.ir					lotsoints.ir				
	#reads	#writes	#branches	#other	#instructions	#reads	#writes	#branches	#other	#instructions
Naive register allocator	143	82	23	116	364	109	54	5	156	324
Whole function register allocator	35	38	23	122	218	40	15	5	157	217

We did tests on two provided ir files: factorial.ir and lotsoints.ir. For each ir testing file, the output result of two register allocators matches, and whole function register allocator greatly reduce the number of reads and writes instructions. You can use the provided .s files (naivelotsoints.s, optlotsoints.s, naivefactorial.s, optfactorial.s) to reproduce these results.

Reference:

[1] https://courses.cs.washington.edu/courses/csep521/07wi/prj/pardoe.docx