Tiger Compiler Design Report: Phase II

Luis Pastrana, Ben Templin, Rahul Tholakapalli
CS 4240: Compilers and Interpreters
25 October 2019

Overview

This document outlines the design, use, and features of the backend of our Tiger language compiler. This compiler takes an already generated intermediate representation file for a Tiger program and converts it into MIPS assembly.

Contents

Overview 2

Contents 3

Building and Running the Tiger Compiler 4

Design Internals 5

- Control Flow Graph Generation 5
- Register Coloring 5
- Naive MIPS Assembly Generation 6
- Register Coloring MIPS Assembly Generation 6

Code Quality Comparison 7

Building and Running the Tiger Compiler

To build the backend for our Tiger Compiler, run `make backend`. To run the backend of the compiler, put the `.ir` file to be compiled in the `P2Testing` directory and from the top level project directory run `java TigerBackend <filename>`. This will output both a naive allocation and register coloring version to the directory `P2Output` with `_naive` or `_colored` appended to the original filename. Note that the supplied filename should be the name of the file only and not include “P2Testing” in the path.

Design Internals

This section will outline some of the internal design choices that we made for our compiler backend. The general flow of the backend compiler is that it takes a `.ir` file, uses that to create a control flow graph for the program and calculate live ranges for each variable, uses that graph to create an interference graph for each variable, map each variable in each line of the program to a register, and create the naive and colored MIPS assembly for the file.

Control Flow Graph Generation

A Control Flow Graph (CFG) for each program is created by the `CFGGenerator` class. Each instance of this object takes in a filename and creates the CFG for the program when the `generateBlocks` method is called. After creating the CFG, the `generateInOutSets` and `createLiveRanges` methods are used to perform liveness analysis. Finally, the class provides some utility methods to retrieve parts of the CFG or results of the liveness analysis for use in other parts of the program.

CFG generation is performed linearly on the text of the `.ir` file by identifying basic blocks in the CFG to be the nodes of the graphs and adding edges depending on branches or changes in control flow. Liveness analysis is conducted recursively based on the paths control can follow through the CFG.

Register Coloring

After a CFG is created for a given program and liveness analysis is performed on the CFG, the `InterferenceGraph` class provides functionality to determine which variable interfere with each other. An interference graph can be created for each function in a program by instantiating an interference graph with the root block of the function and the associated `CFGGenerator` object. Calling the color function colors each node in the interference graph with a register, spilling the value if necessary. Finally, the `generateRegisterMap` function creates a `HashMap` associating each line of the program with which registers map to which variables used in that program line.

The interference graph is created linearly by determining the overlaps in all variable live ranges in the program. Coloring is performed using the Briggs' optimistic coloring algorithm. The register map generation is performed linearly by using the colored graph to determine which variable corresponds to which register at each point in the program.

Naive MIPS Assembly Generation

Instead of generating pseudocode, our design turns the IR directly into MIPS. `Allocator` contains the main code that is common between both the naive assembly generation and the colored assembly generation. `NaiveAllocator` contains code specific to naive allocation, and `ColoringAllocator` contains code specific to coloring allocation.

The first step is to build the ".data" section. This is done by doing one pass through all the instruction in each respective function to find global variables. A global variable is found if it is not declared in the functions "int-list", "float-list", or parameters. Additionally, all arrays are global variables. All floats and ints are given four bytes, while arrays are given four bytes multiplied by their size.

The next step is to build the ".text" section. When we go into a new function, we execute several function setup steps. We set a global variable currentFunction to track which function we are currently in. Additionally, we setup the stack offset with a dummy offset value that we fill in later once we know how many registers we need to save on a function call (this value is in addition to the offset required to store all declared variables in the function). Finally, we store all function parameters from the argument registers onto the stack and set all registers to inactive/usable (e.g. $s0, $s1, etc.) because each function should have a fresh set of registers when called.

The next part of building the ".text" section is to convert each type of instruction in our IR into its MIPS equivalent. For naive allocation, this is fairly straightforward. For every variable needed, we get an available register and use the generateLoad function to load that variable into the appropriate type of register depending on whether its an int or float. The regular assign and operator instruction conversions are all relatively straightforward. The array assign is slightly different because we need to loop through the array and assign every value individually. We use temporary registers to calculate the offsets and assign every value. Array loads and stores are done in a similar fashion, using temporary registers to calculate the offsets and load/store values into their appropriate places. Branches are fairly straightforward, with some additional logic implemented that is required specifically for floats. The function call instructions are a bit more involved. Here, we load all the necessary arguments into the argument registers, save all registers in use to the stack, jump to the function, and finally restore all registers from the stack. This is also where we get the total stack offset needed for a function, and so we then go back and fill in this value everywhere we add or subtract the stack pointer value itself. The last instruction is the return instruction, which is relatively straightforward and simply involves loading the return value into the return register ($v0 or $f0).

Register Coloring MIPS Assembly Generation

Most of the instruction conversions in the colored MIPS is similar to that of the naive MIPS. The only difference is that unlike the naive version, where we constantly load/store from the stack using the same $s0, $s1, $s2 etc. registers, we use the registers specified from the coloring section described earlier. In the case of a spill, we treat that variable like in naive analysis where it is loaded/stored from/to the stack. Additionally, we must still load parameters from memory on their initial use. Global variables are still loaded and stored from their location in memory as well (like in naive allocation).

Code Quality Comparison

Testing of both MIPS generation algorithms showed marked efficiency improvements when using the register coloring algorithm to generate the MIPS assembly. The table below details our code quality comparisons and shows that using register coloring resulted in an average of a 28.95% decrease in total instructions, with a 78.26% and 72.97% decrease in read and write instructions, respectively.

Test File	Register Allocation Sheme	Instruction Count	Reads	Writes	Branches	Other
factorial.ir	Naive	261	90	86	23	62
factorial.ir	Coloring	172	31	30	23	88
perfect_sqrt.ir	Naive	324	9	8	8	299
perfect_sqrt.ir	Coloring	139	2	1	8	128
sort.ir	Naive	2574	670	431	257	1216
sort.ir	Coloring	1799	188	137	247	1227
spill.ir	Naive	350	105	64	45	136
spill.ir	Coloring	151	7	6	45	93
test1.ir	Naive	6347	1104	806	605	3832
test1.ir	Coloring	4742	202	203	605	3732
Averages	Naive	1971.2	395.6	279	187.6	1109
	Coloring	1400.6	86	75.4	185.6	1053.6
Average % Change (Naive to Coloring)		-28.94683442	-78.26087	-72.97491	-1.0660981	-4.9954914