CSC 431 Final Paper

Ben Sweedler
Nathaniel Welch
June 8th, 2011

Architecture Overview

Our approach to the Evil Compiler followed a modular approach. With the exception of code transformation, each step in the compiling process is handled by one or two Java classes. This overview will explain the job and design choices of each module: type checking, control flow graph construction, ILOC output, Sparc transformation, register allocation, and optimization.

We started out with Dr. Keen's basic layout: Evil.java creating the user interface to our compiler and using Antlr to generate a walkable tree of the Evil code. We wrote two more Antlr files, TypeCheck.g and CFG.g, to type check the tree and then build a control flow graph from the tree respectively.

TypeCheck.g provides basic compile time error checking, warning the user of undeclared variables and functions, incorrect assignments, missing return statements, and mismatched function arguments. The other job of our type checker is to build a symbol table, which is used during ILOC code generation. Our type checker does not do syntax checking, and it is Antlr that will throw errors if the Evil code does not match the grammar.

CFG.g builds a graph table. The graph table has one entry per function from the Evil code, and each entry is a graph matching the control flow of the function. Each node in these graphs represents a block of code that will be executed together. We build these graphs based on the conditional, loop, and return statements in the tree. The rest of the Evil statements translate directly into one or more ILOC instructions. The symbol table is consulted when translating assignment instructions. We follow most of the advice provided by Dr. Keen for inserting instructions into our control flow graph, including the decision to use the same conditional instructions in the nodes both before and inside a while loop.

After the control flow graph is built, we don't use the Evil code tree again. Optimizations, register allocation and Sparc transformation are completed using the control flow graph only.
We took an object oriented approach to representing the instructions inside the nodes of the control flow graph. Each ILOC and Sparc instruction has it's own Java class. These classes, and the mappings from ILOC to Sparc, are generated by the generate_instructions.py script. We initially chose to write the script in Python because the Cal Poly servers don't have Ruby,

and Python has better string manipulation than Bash. The script starts out with a gigantic array filled with Python hashes. Each hash matches an ILOC instruction. It contains the name of the instruction, the types of sources the instruction has, the types of destinations the instruction has, and the Sparc instructions the ILOC instruction transforms into. The script loops through all of these hashes, and writes out a Java file based on a template string and the data in the hash. Some instructions have custom hand coded Sparc transformations, and for these classes we set a modified field in their hashes, so that their source files are not overwritten by the script.

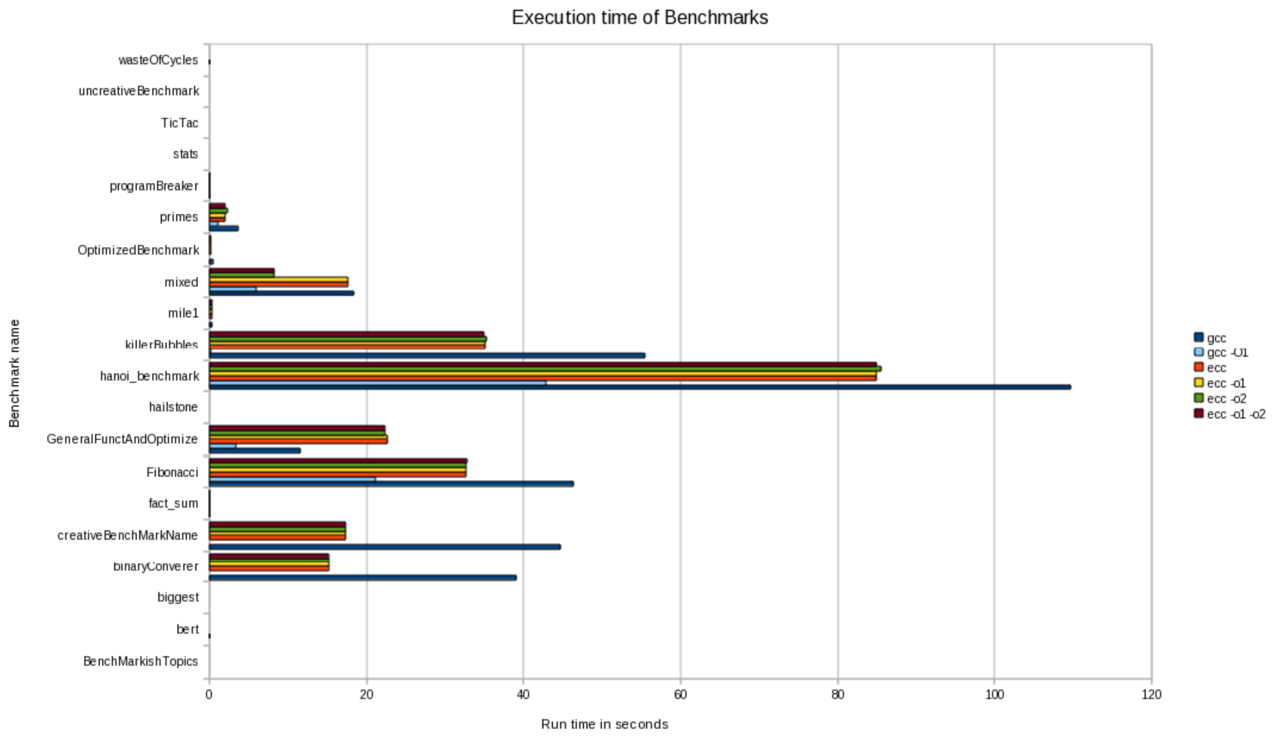
For ease of printing and transforming the compiled Evil, many of our classes have a toILOC() and toSparc() function. These functions explicitly exist in each instruction class, the node class which is used in the graph table, the graph table itself, and the symbol table class. While toILOC() and toSparc() both work differently at different levels of the code, their goal is to return a string of syntactically correct code for their scope. For example, Node prints out its label, along with the proper instruction code. GraphTable makes sure that all of the nodes are topologically sorted and prints extra data such as start and end data for functions. Each instruction just returns itself, properly formatted.

We implemented register allocation pretty much exactly as taught in class. The only significant difference between the suggested implementation and ours, is that we included each Sparc register in our interference graph, and then added extra edges to our graph according to the Sparc register usage rules. This let us prevent global registers being stored across call instructions and so forth. We colored the graph by giving each real Sparc register a color to start, and then colored the virtual nodes to create a mapping of virtual to real registers. Spills are handled by holding three Sparc registers, %g1, %g2 and %g3, for use only during spill scenarios. Then after register allocation, we check for instructions that use those registers and add in load and stores from the stack for these instances.

We implemented two optimizations which can be turned on with command line flags. The -o1 flag turns on Local Value Numbering with Copy Propagation. Copy Propagation was included in this optimization because LVN needs Copy Propagation to lower the instruction count. The -o2 flag turns on Dead Code Removal. We implemented this by looking at the live sets for each node, and finding instructions whose targets were overwritten by other instructions before being used as a source. We choose these optimizations because they work well together in theory: LVN turns arithmetic instructions into move instructions, Copy Propagation propagates these extra move instructions and Dead Code Removal gets rid of any now useless moves.

Finally, we heavily modified Dr. Keen’s Makefile to make automated testing easier. We also created a variety of Bash scripts to automate common task. Our script ecc wraps the compiler into a single file and provides a nice access point for the compiler because we were unable to get the program into a single executable. The script build.sh is similar to one that Keen provided, but gives the user a nice interface to run one benchmark at a time. It provides a diff output depending on pass or fail of the test. timing.sh is similar to build.sh, but instead it compares the run time against pure C for a single benchmark. The script test.sh loops through all of the benchmarks and runs build.sh for them. If they pass, it also runs timing.sh on them. Our final script, typetest.sh runs all of the tests from the first milestone and does nice error reporting.

Performance

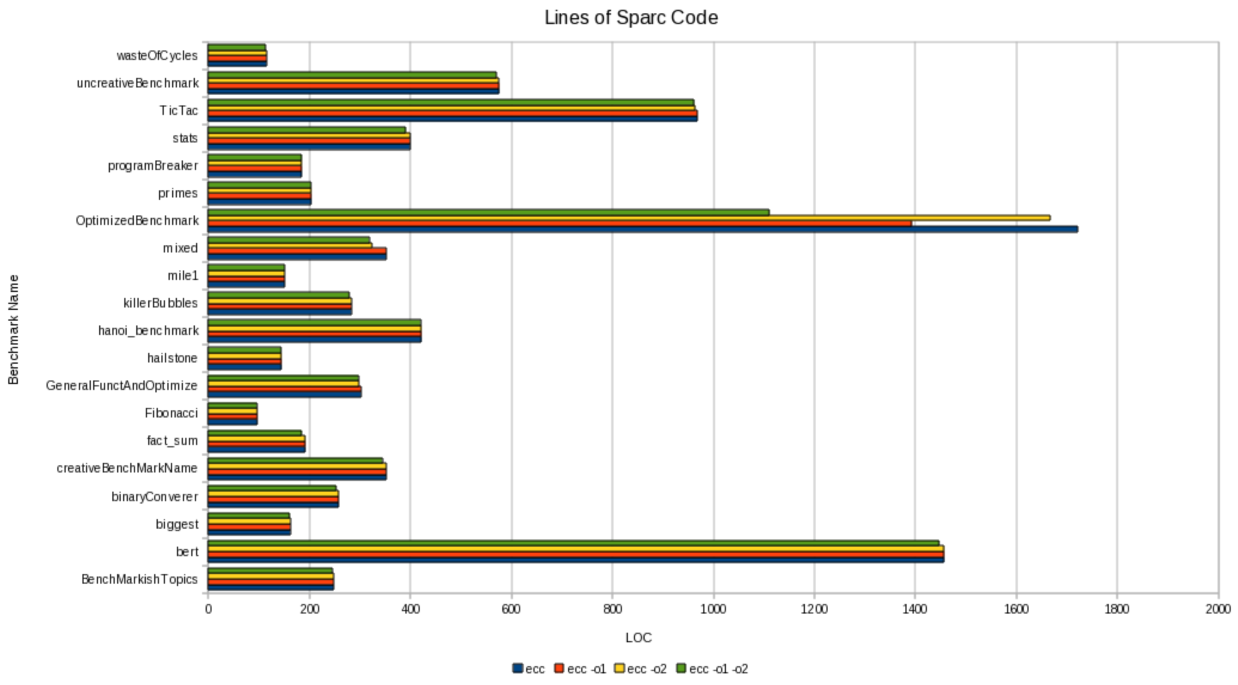


Our optimizations did not affect benchmark run time for the most part. As shown in our lines of code graph below, most benchmarks did not have any dead code, and only a couple instructions were removed when LVN was done.

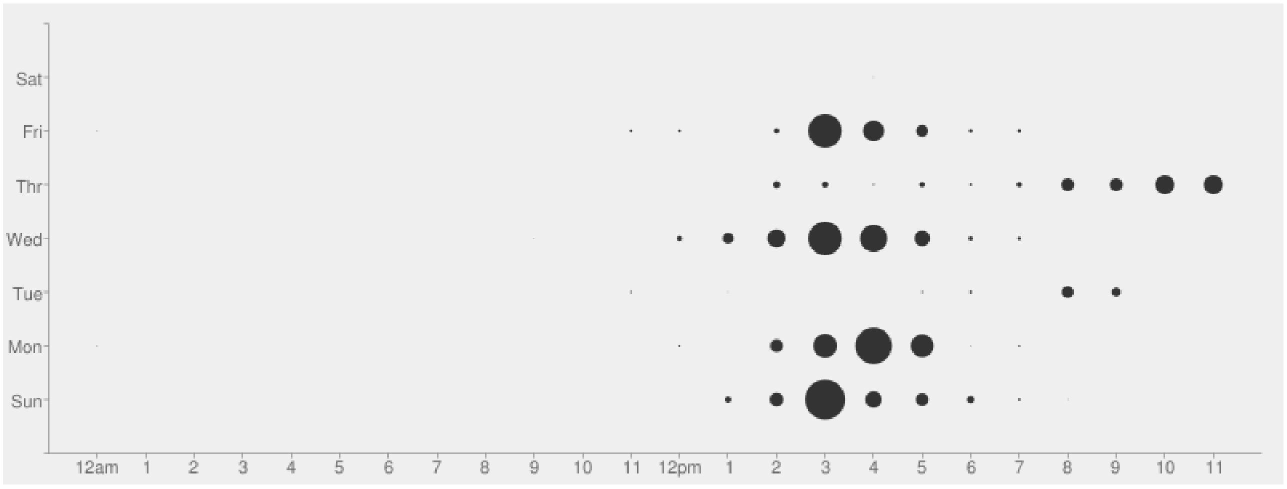
OptimizedBenchmark included both dead code and redundant arithmetic. For this benchmark, our optimizations took the time down from .19 seconds to .14. More impressive is number of Sparc instructions removed. The optimizations lowered the instruction count from 1721 to 1111. This makes sense as this benchmark was designed to be optimized. We can conclude that when applicable, our optimizations work really well. It is also worth noting that the improvement from running both optimizations is better than the improvement from either one by itself. In OptimizedBenchmark, dead code removal removes 54 instructions, LVN removes 323 instructions, but both together remove 610 instructions.

With or without our optimizations, our Evil compiler runs at the same speed or up to twice as fast as gcc without optimization. This is because gcc does so many loads and stores. If we run gcc with the O1 flag, it beats our times be at least a third, and usually way more.

Other Graphs



Our commits usually happened in the afternoon:


Sat
Fri
Thr
Wed
Tue
Mon
Sun
12am
12pm