Programming Assignment 6 - Optimization

Collaborators

- Michael Maitland, mtm68
- Scott Bass, sb2383
- Michael Tobin, mat292

Running the Program

Main class: `mtm68.Main.java`

Run `xic-build` (this requires having Maven installed). Then `xic [options...] <source-files>` becomes available.

Summary

In this assignment we implemented register allocation with move coalescing. We used a live variable analysis, enabling reusing of the same register for multiple variables. We also handled spilling. In addition, we implemented common subexpression elimination (CSE) which used an available expressions analysis. We also implemented copy propagation and dead code elimination which required available copies and live variable analysis. We also implemented inlining function definitions. Lastly, we implemented constant propagation which required a reaching definitions analysis.

We did our testing separately via unit tests, but verified that everything worked together using our integration tests from the previous assignment.

Specification

The following are choices we made regarding specification:

We decided that, if –optir or –optcfg were passed as parameters and errors occur in the lexing, parsing, or typechecking stage, no .dot files would be outputted. This did not seem necessary as a .s file will always be outputted with the error message. Therefore, this would be wasteful saving.

Also, if -O is used, then all optimizations will be turned off regardless of the presence of other -O parameters.

Lastly, we combined the command line options "Omc" and "Oreg". The reason for this is because the register allocation was implemented with move coalescing built-in, so there's no difference between register allocation and register allocation with move coalescing. As a result, both options will perform optimal register allocation with move coalescing.

Design and Implementation

Architecture

The key classes and packages we created or updated for this assignment are the following…

- mtm68.Main.java
  - Here passed XI files run through the entire pipeline of compilation, from parsing to optimizing to assembly generation.
- mtm68.ir.cfg.AvailableCopies
  - This class performs an available copies analysis on the given IR
- mtm68.ir.cfg.AvailableExps
  - This class performs an available exprs analysis on the given IR
- mtm68.ir.cfg.ConstantPropTransformer
  - This class performs constant propagation on the program.
- mtm68.ir.cfg.CopyPropTransformer
  - This class performs copy propagation on the program
- mtm68.ir.cfg.CSETransformer
  - This class performs CSE on the program
- mtm68.IRCFGBuilder
  - This class converts IR to CFG and allows that CFG to be converted back to IR form, useful for getting the transformed IR back
- mtm68.ir.cfg.LiveVariables
  - This class performs a live variables analysis on the given IR
- mtm68.ir.cfg.ReachingDefns
  - This class performs a reaching definitions analysis on the IR
- mtm68.visit.VariableRenamer
  - This class renames all of the variables within an AST to tree to fresh names. This is crucial in function inlining.
- mtm68.visit.FunctionInliner
  - This class carried out the function inlining at the AST level. It would modify SingleAssign, MultipleAssign, and ProcedureCall nodes to become a sequence of statements that included the function body and appropriate assignments for args and rets.
- mtm68.assem.cfg.AssemCFGBuilder
  - This class builds control flow graphs from abstract assembly code.
- mtm68.assem.cfg.Graph
  - This class is used as a data structure for reprsenting directed graphs. It's used in several data flow analyses.
- mtm68.assem.cfg.Liveness
  - This class performs live variable analysis at the assembly level used for register allocation.
- mtm68.assem.cfg.RegisterAllocation
  - This class performs optimal register allocation with move coalescing.

Code Design

- For this assignment we were able to split our code into three main parts: optimization on the AST, on the IR, and on the assembly.
- Function inlining was implemented at the AST level. This level was preferred as our function calls were conveniently represented as a single object. However, in the IR and beyond, a function call could be split into multiple instructions (such as a call and moves). This would have made implementation much more complicated. Inlining required taking the body of a function definition and putting it in the place of the call to the function. This required renaming all of the variables with novel names in the function body (as to not interfere with the variables already in scope). Then, return statements were able to be turned into simple assign statements. We made the design decision to only conservatively inline functions that had a single return statement at the end of the function's body. Dealing with embedded returns would have been complicated since we do access to explicit jumps at the AST level. In order to handle the space tradeoff, we limited function inlining to functions with a maximum of 10 AST statements.
- With regard to optimization on the IR, we had a class CFGBuilder that did work on the basic block and also performed transformation. Although this built the CFG, it had extra functionality that we did not need when doing our IR optimizations. Therefore, we opted to make a new IRCFGBuilder that was responsible for creating a CFG and converting back to IR. This made it reusable for many different CFG transformations that used the IR.
- Once we had a CFG, we were able to split optimizations into two main parts: an analysis and a transformation. We did CSE which required an available expressions analysis. Copy propagation required an available copies analysis. Dead code removal required live variables analysis, and constant propagation required a reaching definitions analysis. The analysis all used the Data Flow Analysis framework. Although each of these classes have lots of similar functionality, we found that they did have enough differences where it was easier to reason about by having some duplicate code. In the future, we would love to refactor similar functionality out into a DFAFramework class. However, keeping things in separate classes made it very clear exactly what was going on in that analysis and made it easy to reason and test our code.
- Our transformer classes would take an IR, convert to CFG, get the needed analysis, do the transformation, and convert back to IR. This made it very easy to test since we could pass in IR and assert against the output IR. All the transformations had the same form which makes it easy to read, reason, and add in new transformations.
- To implement optimal register allocation, we heavily relied on the pseudo-

code found in Appel's book. The explanation and code provided in the register allocation chapter was immensely helpful when it came to doing our own implementation. As a result, our implementation is very similar to the one presented in the book adapted to our classes and assembly model.

Programming

- One challenge during this project was the dependency on the CFGBuilders at the start. We wanted the CFG builders to feel the same at the IR and Assem level. We had a CFG builder from the past that contained transforming functionality. We decided it would be best to start fresh, but this meant that we had to do this all together and was more difficult to work in parallel until this was finished. We ended up being able to write analysis and transformation but needed the CFG conversion in order to test. Once we finished the CFG builders this was no longer a problem and it was very easy to work in parallel.
- The following is the team coding/responsibility breakdown for this assignment...
  - **Tobin:**
    - Optimizer
    - Function inlining + tests
    - Command line options
    - IRCFGBuilder + tests
  - **Maitland:**
    - CSE + tests
    - Available Expressions + tests
    - Dead Code + tests
    - Available copies + tests
    - Reaching defns + tests
    - Copy prop + tests
    - Constant prop + tests
    - live vars + tests
    - IRCFGBuilder
  - **Bass:**
    - AssemCFGBuilder
    - Register Allocation
    - Live vars (assem level)
    - Benchmark programs
- We used our previous code for lexing, parsing, typechecking, and code generation. Fortunately, we have been on top of correcting our errors after each assignment so there were very few changes that needed to be made to this code.

Testing

In this assignment testing was very helpful, although sometimes we wrote tests that output a CFG and we would analyze the CFG ourselves.

When testing the DFA analysis, we would write tests cases to make sure in and out were as they were supposed to. When testing the IR transformations, we wrote tests cases that asserted a given IR was optimized into a new IR. This testing helped find bugs in optimizations and transformations and let us know exactly what was going wrong when something broke.

Our integration tests stepped through the entire pipeline of compilation. First, we turned the xi program into IR code and tested the output of this IR code against the expected output using the IRSimulator. Then, we generated assembly from the IR code and automatically ran it using linkxi.sh. Setting up this architecture in JUnit allows us to very easily add more programs as test cases. It simply requires writing xi programs and determining their expected output. It allows us to check to make sure optimizations didn't break the pipeline. We also set up boolean toggles for each of the optimizations. This made it very easy to turn opts on and off for testing which helped to isolate which optimization was the culprit when we encountered an error.

When we ran into bugs in our integration tests, we would write a simpler program that isolated the issue. Then, we would generate the executable for this smaller program and step through the execution using gdb. Although being very painful, this was effective in helping us isolate a number of issues (such as improper stack pointer alignment and register value clobbering).

It should also be said that we utilized the xth test suite to make sure our output was compliant with the autograder's expectations.

Benchmarking

In our IntegrationTests, you can see unit tests that are made specifically for our benchmarking (duplicate xi programs were correctly placed in the benchmarks directory). We created 3 per optimization and timed both unoptimized and optimized results. We observed that, in each case, our optimized code had faster performance. The only optimizations we do not have benchmarks for are copy propagation and constant propagation. These two optimizations generally require other optimizations for their potential to be actualized. They create dead code which can be removed by dead code removal to speed up the code. Therefore, we believe isolated benchmarks do not make much sense.

Work plan

We found the requirements to be fairly parallelizable once we sorted out the foundational design. Scott was able to tackle the register allocation at the assembly level. Mike and Mike diviied up optimizations to complete and worked closely to design the infrastructure surrounding the optimizations.

We did a great job of communicating frequently, working together to clarify things, and adapting to challenges we faced. Our plan allowed us to work efficiently and effectively without being held up by each other's progress.

Known Problems

We currently are not aware of any issues with our compiler.

Comments

Thank you for a great year, this class was a bunch of fun and we learned a crazy amount!!!