Engineering a Compiler, 2nd Edition

Keith D. Cooper and Linda Torczon
MKP/Elsevier
February 2011

Graduate Course Lab Notes

COMP 512 is a graduate course on advanced topics in compiler construction that we teach at Rice. COMP 512 focuses on scalar analysis and transformation—that is, on classic optimizations other than transformations to discover and capitalize on loop-level parallelism. See the document "Graduate Course Lecture Notes" for more information on the course and its content.

A major component of COMP 512 is its programming exercise, affectionately referred to as a "lab" by the students. We vary the actual assignment from year to year as the interests of the students and the course staff change. In the last five years, the lab has consisted of an implementation project in some compiler infrastructure. We have used our own research compiler infrastructure (the "MSCP" tools) and the LLVM compiler infrastructure. The same lab could be done in the Gnu Compiler Collection infrastructure; we have not used the Gnu infrastructure because of local familiarity with LLVM, and because LLVM operates on an SSA-like intermediate form, which lowers the learning curve for students.

In general, the goal of the exercise is to have the students download and build a reasonably complex compiler; to modify that compiler by adding one or more passes to it; and to study the impact of their modifications by running some suite of benchmark codes through the compiler.

In the 2011 edition of Comp 512, the students are working in pairs to implement an optimization in LLVM. They will be expected to implement their transformation and to test it with standard set of LLVM test codes. For this exercise, the students will choose one of the following transformations:

1. Dominator-based value numbering (§ 10.5.2 in EaC2e)
2. DEAD (§ 10.2.1 in EaC2e) plus CLEAN (§ 10.2.2 in EaC2e)
3. Loop invariant code motion based on the Cytron, Lowry, and Zadeck algorithm ([111] in the EaC2e bibliography)
4. Register promotion (Cooper & Lu, PLDI 97 and Sastry and Ju, PLDI 98 [306] in EaC2e bibliography)
5. Algebraic reassociation of expressions (Briggs & Cooper, PLDI 94 and Cooper, Eckhardt, & Kennedy, PACT 2008)

This set of transformations provides a broad range of difficulties to suit the abilities and ambitions of the different teams of students. From my perspective as an

instructor, I am more concerned that the students have an immersion experience in a real compiler system than I am concerned that they implement an incredibly intricate optimization.

In some years, I have had them all implement the same optimization — OSR (§ 10.7.2 in EaC2e) or Sparse Conditional Constant Propagation (§ 10.7.1 in EaC2e). We found that the OSR implementation was somewhat easier than the SCCP implementation. Other years, we had them build SSA form and use it—a three-part lab that began with live analysis, progressed to the SSA construction and then used the SSA form to perform either constant propagation (§ 9.3.6 or § 10.7.1 in EaC2e), or useless code elimination (§ 10.2.1 in EaC2e).

Finally, many years ago, we had a programming exercise where we presented the students with code in ILOC form (similar to the ILOC used in EaC2e) and had them construct a five-pass optimizer. Students picked five transformations and an order in which to apply them. Students were given several large (~2,000 ILOC operation) programs. Students were graded competitively based on the speed with which their programs worked on an ILOC simulator (see the undergraduate course materials), with zero credit for incorrect answers. This lab inspired a strong competition among the students. It seems to me, in retrospect, that having them do less implementation but do it in a system such as LLVM or GCC is probably a more valuable use of their time.