CSE 501: Compiler Construction (Winter ’95)

Instructor: Craig Chambers

- office hours: TuTh 11:00-12:00 (Sieg 309)

TAs: Jeff Dean and Dave Grove

- office hours: MF 10:30-11:30 (Chateau 110)

Goals:

- Understand how languages get implemented efficiently
- Understand what can and can’t be done, with how much effort
- Understand state of current research in efficient language implementation

Prerequisites:

- CSE 401 or equivalent
- CSE 505 or equivalent

Readings:

- Dragon book
- important papers from the literature

Grading:

- Midterm: 25%
- Final: 40%
- Project: 35%
- (there may also be homework)

Course Outline

Week	Topic
1/2	***Holiday*** Intro; structure of compilers; standard optimizations Intermediate representations: ASTs, CFGs, BBs, DAGs, RTLs Control flow, data flow; dependence Iterative global data flow analysis Reaching definitions, reaching constants
1/9	Common subexpression elimination Lattice-theoretic data flow analysis framework Integer range analysis Data flow analysis generators
1/16	***Holiday*** Advanced intermediate representations: def/use chains, SSA form, PDGs Loop-invariant code motion
1/23	Inlining Interprocedural analysis Partial evaluation
1/30	Alias and pointer analysis Dependence analysis; automatic parallelization
2/6	Global register allocation
2/13	Calling conventions; callee-save vs. caller-save Link-time register allocation Instruction scheduling
2/20	***Holiday*** Profiling Source-level debugging
2/27	Garbage collection Implementing functional languages
3/6	Implementing object-oriented languages

The Project

Description

As part of this class you will complete an implementation project as part of a team. The main goal of the course project is to reinforce the lectures and readings with practical experience designing, implementing, and assessing compiler analyses and optimizations.

You will form your own teams of 2-5 people and each team will negotiate a suitable course project with me and the TAs. Each project should include some sort of analysis component (something the compiler needs to extract from the program) and a transformation component (an improvement the compiler makes to the program based on the extracted information). Each project will include a written report describing the analyses performed, the transformations made, the generality of the optimizations, and the bottom-line improvement in program performance made by the optimizations. The CSE 501 project can be joint with the CSE 548 course project, if appropriate. Some sample projects are listed below.

In order to focus on the interesting aspects of optimizations and minimize less interesting grunt work, the project will be done in the context of the Vortex optimizing compiler for Cecil, written in Cecil. [Cecil is an object-oriented language we've developed here at UW. We will give tutorials on Cecil and the Vortex infrastructure during the first few weeks of the quarter.] The Vortex compiler includes a lot of support for performing optimizations, including an intermediate representation, a framework for iterative data flow analysis, some simple optimizations such as constant propagation, dead assignment elimination, and live range analysis, and back-ends generating C and SPARC assembly code. By leveraging off of the Vortex infrastructure, I hope that your work can focus on your analyses and optimizations, leading to a higher-quality result and better learning with less time cost. I also hope that you will see good examples of how to structure compilers so that they are extensible. Note, however, that the Vortex infrastructure is a research product still under dev elopment, and as such is likely to have bugs and some amount of ugly code. Please bear with us and help us to improve the infrastructure.

Time Schedule

Over the course of January, you should have formed project teams and negotiated with me about the details of the project. By **January 30**, at the start of class, you should turn in a formal project proposal, which is essentially a draft of your final report, excluding the results (but including a plan for what measurements you expect to take). By **February 27**, you should turn in another draft of this report, including preliminary results, and make your current implementation available to us. The final version of the project report and your implementation is due **March 8**. The final report can be no longer than 5 pages per team member.

Some Sample Projects

- Design and implement some standard optimization(s), such as loop-invariant code motion, induction variable elimination, or partial redundancy elimination.
- Design and implement some interprocedural optimization, such as interprocedural constant propagation. A framework for interprocedural analyses analogous to the existing intraprocedural iterative data flow analysis framework would be particularly cool.
- Implement several global (or interprocedural) register allocation schemes, and compare their effectiveness. This might make a cool paper.
- Implement an effective instruction scheduler, for some modern pipelined architecture(s). Again, a comparative study of scheduling algorithms might make a cool paper.
- Implement an effective automatic loop unroller.
- Design and implement dependence analysis of array references. Some alterations of the intermediate representation of array references might be needed.
- Design and implement some sort of pointer analysis and one or more optimizations using the results of the analysis.
- Design and implement an optimization to keep object such as floats and characters in an unboxed, native form for as long as possible, rather than boxing them eagerly into their standard, heap-allocated representation.
- Implement support for dynamic compilation. E.g., the compiler reasons about variables whose values will be constant at run-time, performs optimizations under the assumption that the variable is an (unknown) constant, and generates simple directives to patch in the real constants at run-time. Definite paper potential here.
- Adapt Jeff Dean’s inlining database to support effective automatic inlining decisions.
- Implement fully non-LIFO closures (upward funargs), using analyses to avoid their expense where possible.
- Implement compiler-assisted “accurate” garbage collection and some garbage-collection optimizations, and measure its effectiveness versus the existing conservative GC system.
- Design and implement mechanisms to support source level debugging without run-time cost.
- Implement a different intermediate representation of dependences, such as SSA form, sparse data flow evaluation graphs, or the PDG, and use it to perform some optimization(s).