Compiler Project II

CS 526 — Advanced Compiler Construction
Spring Semester 2019

Due: End of Semester.
The exact date will depend on the Final Exam schedule.

Goal

The primary goal of this project is to give you experience with designing and writing a major analysis or transformation in an optimizing compiler, by implementing (and preferably, extending) techniques in the literature. A secondary goal is to give you exposure to topics of current research interest in the compiler area – you are welcome (and highly encouraged) to include a research component in your project!

This is a team project. You must work in teams of 2 (unless you have a strong reason to work alone; in that case discuss with me). All members of a team get the same grade for the project. Because these are long team projects, they will be expected to meet a high standard for completeness, testing, and documentation. *Correctness, modularity, and code quality are more important than the amount of functionality you implement, so start simple and add functionality incrementally, testing thoroughly at each step.* If you plan to include a research-based component, you must discuss it with the instructor and get the approval before submitting the proposal. Overall, the research-based proposal should be directly related to the topic of the course (program analysis, program transformations), comparable in effort with the topics we discuss later in this document.

Testing and Evaluation. We will use LLVM as the common infrastructure for implementing the tasks. You will submit both source and executable code that can execute on the EWS machines.

The code that you turn in should be thoroughly tested. This includes small unit tests, but also medium-to-large real-world programs between 10K and 100K lines of code. You can use some of the real programs in `llvm/project/-llvm-test/{MultiSource,External}`. Consider testing your analysis/optimization on the common benchmark suites, such as SPLASH (for CPU-intensive tasks), PARSEC (for multicore performance, also includes the updated version of SPLASH), Rodinia (for heterogeneous architectures), or MediaBench (for multimedia applications). You are encouraged to find and try public-domain C or C++ programs, e.g., GNU utilities (e.g., coreutils package, tar, grep, awk, sed), and larger applications often used in analyzing compilers, such as Apache (httpd), OpenSSH, Squid, MySQL, Povray, etc. Before trying your code on any such program, make sure they can be compiled with Clang or Clang++ (just to make sure there are no unexpected LLVM failures).

You are free to select a subset of such applications if your approach is domain-specific (e.g., analyzing multimedia or machine learning applications). Clearly, all compiled applications should pass *all* their tests after your optimizations.

Deliverables

Thursday, Feb 28, 11:59pm (Urbana Time Zone):

Short project proposal, as ascii text via e-mail (2 pages max). This proposal should include:

- a short description of the problem to be solved;
- a list of references;
- a short summary of what algorithm(s) you will implement (if they are taken from existing papers), *or* a short summary of the state of the art, and how you want to improve on this.
- a break-down of the work into a short list of tasks, *credible* (but tentative) completion dates and how the tasks will be divided between the two team members.

Your list of tasks should include two explicit tasks for testing: one for creating and testing with small unit tests, and another for setting up and testing with larger programs. Don't underestimate these tasks: they can each take one person a week or more, just to do a minimally adequate job. Moreover, *each of these two tasks should be shared by both team members*, i.e., both write unit tests and both set up and run real programs.

Tuesday, April 9, 11:59pm: **30% of the grade**

The progress report, should be an ASCII text in the body of an e-mail message (2 pages max). By this stage you should have a reasonably complete suite of tests (including unit tests and medium and large programs). You should also have *partial working code for a subset of proposed functionality* that has been tested on all those test cases. The functionality can be small, but it should be solid. Describe what you have accomplished, including any relevant preliminary results for programs that work. 30% of the overall grade is reserved for your progress accomplished during these first 5 weeks after sending me your project proposal.

Wednesday May 1[1] (Officially), 11:59pm Urbana Time: **70% of the grade**

Final (valgrind-clean, well documented) working code (10%), test suites (20%), experimental results (20%) and project report (20%). *Unless noted otherwise in the project description, it is important that your project should work on several moderately large programs between 10K and 100K LOC.* The final report should describe the status for such programs in some detail, including programs that work and ones that don't. Your compiler passes should be "*valgrind-clean*": no errors or warnings from valgrind for *any* of your input tests. The code you submit should include your source code, an executable that can be run on the EWS machines, as well as *all* the input programs you used for your tests. Include any modified LLVM source files. The report should explain where the source code, executables, and test programs are, and how to run your code. The report format is described below.

Final Report

Your final project report should be submitted to me via email in PDF form. It should be maximum 5 pages (10 pt font, single-column article format or two column ACM format), not including References and Appendix.

The report should include:

1. The problem statement and motivation (1/3 page).
2. Brief summary of existing work in the literature, with citations (1/2 page).
3. High-level overview of your algorithm or design (1 page).
4. Implementation details (1-1.5 pages):
   (a) Prior analyses or transformations required by your code.
   (b) Major code components (passes, data structures, and functions).
   (c) Testing strategy and status: unit tests, small source programs, larger (multi file) source programs of 10K-100K LOC (as counted by `sloccount`).
5. Experimental results: (1-2 pages): Choose results appropriate for your project.
6. References.
7. Appendix:
   - A description of where to find your source code, executable, and input programs, and how to run the executable for example inputs.
   - An Extended Example (as long as necessary): Use part of one of the benchmarks (or design your example based on one of the benchmarks) to illustrate what your code does. Choose the example carefully to highlight the major technical capabilities and limitations. You can use more than one example if needed, but no more than absolutely necessary.

[1] **Automatic extension until Sunday May 5.**

Working in Teams

Some things to keep in mind when working in a team for this project:

- *All members of a team will receive the same grade.* You are responsible for monitoring each other's progress. Discuss any potential problems between yourselves first to try to resolve them. If that doesn't work, bring me into the discussion.
- Plan your tasks so that you can make substantial progress independently.
- At the same time, plan to integrate your pieces in phases, at least 3-4 times during the course of the project. Integrating everything all at once for the final experiments may raise too many difficult problems, when you don't have the time to tackle them.
- Use *Pair Programming* for key stages of the project, including designing details of all the interfaces between the key components, integrating your pieces, and tracking down difficult bugs. If you are not familiar with Pair Programming, see `http://en.wikipedia.org/wiki/Pair_programming`. Whether or not you are familiar with it, see `http://www.wikihow.com/Pair-Program` for advice on how to go about it effectively.
- Be sure to divide up some of the key phases of the project: writing the unit tests; setting up larger programs for testing (do a few each); designing the overall code organization and major interfaces; running the experiments; writing the final report.

Suggested Projects Involving Known Techniques in the Literature

This is a list of suggestions by Vikram Adve for the earlier versions of the course. These topics are given mainly as a reference of the scope of the project. For more suggestions and inspiration, talk to me about some interesting ongoing work connecting compilers/PL and machine learning or see recent proceedings of the PLDI, OOPSLA, and CGO conferences.

1. **Partial Redundancy Elimination via Lazy Code Motion**

   This is an elegant iterative bit-vector dataflow algorithm for PRE. The implementation should use the improvements to PRE described by Briggs and Cooper, which also describes how to perform PRE effectively starting with code in SSA form.

   References:

   (a) J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," In *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992.

   (b) Preston Briggs and Keith D. Cooper, "Effective partial redundancy elimination," In *Proc. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

2. **A State-of-the-art Algorithm for Inclusion-based Interprocedural Alias Analysis**:

   Inclusion-based (also known as Anderson's) algorithms are among the more powerful *practical* algorithms for interprocedural alias analysis. The approach has been improved significantly over time, with fast and highly scalable algorithms having been proposed. Implement one of the state of the art algorithms, listed below. If you choose, your implementation can use Binary Decision Diagrams (BDDs), using an existing BDD package (e.g., BuDDy), or Datalog, using an available datalog solver (e.g., `bddbddb`).

   Choices of References:

   (a) "Strictly declarative specification of sophisticated points-to analyses," M. Bravenboer and Yannis Smaragdakis, OOPSLA 09.

   (b) "The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code," Ben Hardekopf and Calvin Lin, PLDI 2007.

(c) "Semi-Sparse Flow-Sensitive Pointer Analysis," Ben Hardekopf and Calvin Lin, POPL 2009.

3. **An Array Dependence Analysis Algorithm**

   Implement an array dependence analysis pass by choosing an algorithm or a combination of algorithms from the literature. Most of the implementation effort is in extracting the input information for each subscript pair (and loops bounds), and this has already been done for you in LLVM. Because of that, you should aim to implement at least two powerful tests, plus two simple ones. One or more simple tests such as the simple SIV tests can be used to handle the most common cases fast, and to fall back on a more powerful test when the simple test fails. Some possible choices for the powerful test are:

   - The Delta Test: Goff, Kennedy and Tseng [PLDI 1991]
   - The Range Test: Blume and Eigenmann [Supercomputing 1994]
   - The Omega Test (use the Omega library): Pugh [CACM, Aug. 1992]

4. **Interprocedural Slicing Using Dependence Graphs**

   Program slicing identifies the subset of a program that affects a particular value (backward slicing), or the subset that is affected by that value (forward slicing). The technique was originally described by Mark Weiser, and is an important technique in advanced programming environments, program verification, performance modeling, and other problems.

   The first paper below describes an algorithm for computing interprocedural slices of whole programs, using a program dependence graph. The second paper replaces a key step of the algorithm with a much faster version.

   LLVM already includes all the infrastrucuture you need to build a simple, intraprocedural dependence graph: (a) SSA dataflow edges; (b) May-alias information for pairs of memory operations; and (c) control dependence information (via dominance frontiers for the reverse CFG).

   References

   (a) S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, 12(1), pp. 26–60, January 1990.

   (b) An important improvement to part of the algorithm is described in:
   T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up Slicing," *SIGSOFT '94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dec. 1994. ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), pp. 11-20.

5. **Automatic Parallelization Via Decoupled Software Pipelining**

   A state-of-the-art algorithm for automatic parallelization of complex loops is Decoupled Software Pipelining, described in the papers below. Implement either the original algorithm (PACT 2004) or the improved one (CGO 2008) in LLVM, using the existing SSA, memory- and control-dependence analyses.

   References:

   (a) "'Decoupled software pipelining with the synchronization array," Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. PACT 2004.

   (b) "Performance Scalability of Decoupled Software Pipelining," Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. ACM Transactions on Architecture and Code Optimization (TACO), 5(2), Aug. 2008.

   (c) "Parallel-Stage Decoupled Software Pipelining," Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David I. August. CGO 2008.

6. **Auto-vectorization through Superword Level Parallelism for SIMD Instruction Sets.**

   Generate vector code for short SIMD instruction sets (AVX / SSE* / Neon) using an algorithm that looks for isomorphic instructions within a basic block.

   References:

(a) Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, PLDI '00, pages 145–156, Vancouver, Canada, 2000.

7. **Memory optimizations for GPUs.**

   Implement two optimizations for OpenCL – *memory coalescing* and *memory prefetching* (a). Use the NVPTX back end in LLVM to compile and run on nVidia GPUs.

   References:

   (a) Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In Proceedings of the *2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2010, pages 86–97, New York, USA, 2010.

8. **KINT: Static analysis for integer overflow errors:** Use information flow tracking to identify integer operations that can be influenced by external program inputs (a). Use integer range analysis to determine which of those operations might overflow. Optional improvement: The KINT paper uses a trivial alias analysis that assumes that loads may return a value produced by *any* store. Evaluate more precise flow through memory using an existing pointer analysis called DBAA available for LLVM.

   References:

   (a) Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. *10th Usenix Symposium on Operating System Design and Implementation*, OSDI 2012, Hollywood, USA, 2012.

9. **A SAT/SMT-Based Program Analysis Framework:**

   Constraint-based analysis can be used for a wide range of static analysis problems. Boolean satisfiability is one kind of constraint-based analysis where program facts are represented as Boolean formulas and a Boolean satisfiability solver is used to prove the existence of solutions to the constraints. The SATURN system described in the paper below (and in its references) is one such system. Build a simplified version of a SATURN-like analysis system for LLVM, using a custom or off-the-shelf Boolean satisfiability solver.

   References:

   (a) "An Overview of the Saturn Project," A. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins and B. Hackett. PASTE 2007.

10. **Online profile-guided optimization for LLVM:**

    Run-time adaptive (profile-guided) optimization is a common feature in most modern JVM and .NET implementations. LLVM does have a profiling infrastructure, but it is likely too heavyweight for online profiling. The goals here are to (a) implement simple lightweight online profiling for LLVM (e.g., function call frequency and perhaps loop trip counts); (b) use this profile information as the basis for an adaptive optimization engine in the LLVM JIT. The JIT engine is production-quality and can run arbitrary LLVM optimizations, which also exist, so the JIT or optimizations do not have to be implemented. Instead, this project should focus on developing a robust adaptive algorithm that decides when a function (or loop) should be reoptimized, what optimizations to (re)run, and performs those optimizations. Since on-stack replacement does not exist for LLVM, it is fine to limit the optimizations to functions that are not live (e.g., to defer them until after the last invocation of a function returns).

    The following survey gives an excellent and fairly comprehensive overview of just-in-time optimizing compilers, including adaptive optimization technologies, through 2005.

    References:

    (a) A Survey of Adaptive Optimization in Virtual Machines, M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Proceedings of the IEEE, 32 (2), Feb. 2005.