Compiler Project II

CS 526 — Advanced Compiler Construction
Spring Semester 2014

Due: 5pm Friday, 9 May 2014

Goal

The aim of this project is to give you experience with designing and writing a major analysis or transformation in an optimizing compiler, by implementing (and optionally extending) techniques in the literature. A secondary goal is to gain exposure to topics of current research interest in the compiler area.

You must work in teams of 2 for this project (at most one team of 3 if there are an odd number of people in the class). All members of a team get the same grade, *without exception*. Because these are team projects spanning more than ten weeks, these projects 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.*

The following sections describe the deliverables for this project with their due dates, some issues in working as a team, and finally a list of suggested projects. You are also welcome to identify other problems of interest and discuss them with me. You must get my approval before submitting the proposal for such a problem. My only criteria for approving such a choice are that the problem should be (a) directly related to the topics of this class, i.e., program analyses or compiler transformations but not, e.g., other language implementation issues; and (b) comparable in effort to the other projects suggested here. All projects must use the LLVM system, in order to create a level playing field for grading: the complexity of a project depends heavily on the infrastructure used. You must submit both source and executable code that I can run on the EWS machines. I want to have the option of building it.

Testing and Evaluation

Whatever code you turn in should be as thoroughly tested as possible, first on a large number of small unit tests, and then on medium-to-large real-world programs up to 100K lines of code. In particular, *testing* a compiler is a difficult task that requires care and creativity, and cannot be taught in lectures: you must do it to learn it!

You can use some of the real programs in `llvm/project/llvm-test/{MultiSource,External}`. You are encouraged to find and try public-domain C or C++ programs, e.g., GNU utilities like tar, bzip, and grep, as well as more substantial applications like Apache (httpd), OpenSSH, Squid, MySQL, ClamAV, Povray, etc. Before trying your code on any such program, test it with Clang or Clang++ and check for unexpected compiler failures.

Deliverables

Note that a substantial part of the grade is reserved for each of the two progress reports (20% each), so do not take these milestones lightly.

Wednesday, Feb. 26, 5pm:

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.

Friday, April 4, 5pm: **20% of the grade**

First progress report, as ASCII text in the body of an e-mail message (2 pages max). By this stage you should have both 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. 20% of the overall grade is reserved for your progress accomplished during these first 5 weeks after sending me your project proposal.

Friday, April 25, 5pm: **20% of the grade**

Second progress report, as ASCII text in the body of an e-mail message (2 pages max). By this stage you should have *working code* for close to the full functionality of the project, though perhaps tested only on unit tests and small programs. (Unlike the first progress report, getting larger programs working is not necessary for the grade on the second report, but it will greatly reduce pressure on the final phase.) Describe what you have accomplished, including any relevant preliminary results for programs that work. 20% of the overall grade is reserved for your progress accomplished during these three weeks since the first progress report.

Friday, May 9, 5pm: **60% of the grade**

Final (valgrind-clean, well documented) working code (10%), test suites (10%), 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 up to 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 (*I do NOT want to have to compile your code*), as well as *all* the input programs you used for your tests (except LLVM bitcode for non-public programs, like SPEC, that I may give you). Include any modified LLVM source files; directory structure is unimportant here. 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.

Your functionality should be fully implemented by the time of the second progress report. The last phase, after the second progress report, should be used only for testing with larger programs, and for further experiments and writing the final report. Two weeks will be tight for completing these: don't underestimate the time required for fixing the numerous bugs inevitably uncovered by larger programs.

Final Report

Your final project report should be submitted to me via email in PDF form. It should be 5 pages max, not including the References and Appendix. The page estimates below assume 10 point font, 12 point spacing, and 1 inch margins.

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`).

(d) Implementation status describing what functionality works and what doesn't work, for arbitrary source programs.

(e) A description of where to find your source code, executable, and input programs, and how to run the executable for example inputs.

5. Experimental results: (1-2 pages): Choose results appropriate for your project, again for complete programs.

6. References.

7. Appendix: 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.

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

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. **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.

5. **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.

6. **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.

7. **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.

8. **A SAT-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.

Projects with Research Opportunities

The following projects are more open-ended than the ones above, and will not be expected to meet the same threshold of testing. It is sufficient to test these on programs up to a few thousand lines of code (though testing with larger programs is still encouraged).

1. **Improving the Effectiveness of Automatic Pool Allocation:**

   Automatic Pool Allocation is a compiler transformation that segregates heap objects into distinct pools at run-time, based on a partitioning computed by points-to analysis. It has been used by us and others for a wide range of purposes, including improving cache locality [PLDI 2005, ISMM 2008], reducing memory consumption on 64-bit systems [MSP 2005], optimizing the run-time overheads of array-bounds checks for C/C++ [ICSE 2006], enforcing the soundness of static analyses in C/C++ programs [PLDI 2006], optimizing Software Transactional Memory [SPAA 2008], etc. The paper describing this transformation and its uses for improving spatial locality of pointer-based data structures won a Best Paper Award at PLDI 2005.

   A major weakness of the current algorithm, however, is that it works with a unification-based (albeit context-sensitive) points-to analysis [PLDI 2007], which we have found (through experience) often severely limits the effectiveness of the memory partitioning. A natural next step in this research, therefore, is to develop a new version of *Automatic Pool Allocation* that can work with an inclusion-based points-to analysis. My suggestion is to use the existing code from Ben Hardekopf for the points-to analysis [PLDI 2007] so that the project can focus on how best to perform the pool allocation itself. Two possible improvements to explore are (a) since a pointer may point to multiple different points-to sets in an inclusion-based analysis (but only one at a time), use a run-time tag to track at run-time which pool a pointer actually points to; or (b) compute unified points-to sets from the output of the points-to analysis, so that each pointer only points to a single unified set, and then use the existing transformation to allocate these unified sets to pools. There may be other options, as well as combinations of these options for different subsets of a program. One part of the

research is designing such options (many details have to be filled in, even for the above two). A second key part of the research is exploring which options or combinations thereof are the most effective for a particular goal, say, optimizing spatial locality.

The Automatic Pool Allocation transformation and the pointer-analysis on which it is based are both available in the SAFECode software distribution: `http://safecode.cs.illinois.edu/downloads.html`.

The Ant and the Grasshopper implementation for LLVM is available on Ben Hardekopf's home page: `http://www.cs.ucsb.edu/~benh/research/downloads.html`.

References:

(a) "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap," Chris Lattner and Vikram Adve. PLDI 2005.

(b) "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World," Chris Lattner, Andrew Lenharth, and Vikram Adve. PLDI 2007.

2. **Efficient On-Stack Replacement for LLVM:**

One of the most important features of modern adaptive managed run-time systems, e.g., the Hotspot JVM and the Jikes RVM, is *on-stack replacement* (OSR): the ability to modify the code and rewrite the stack frame for a function while there are one or more invocations of the function active. There is no public implementation of this feature for LLVM, which severely limits the ability of the LLVM Just-in-time compiler to perform flexible adaptive optimizations, run-time instrumentation (e.g., for security), or any other run-time transformations that may wish to modify live functions. There is one paper on this topic, listed below, but that paper uses strictly LLVM-IR-level transformations to support OSR, which likely hurts its efficiency (nevertheless, it is a valuable introduction to the general challenges and the LLVM-specific issues in OSR).

The goal here would be to design, implement and evaluate an efficient OSR system for LLVM that uses native code where needed for efficiency. This project will likely require more architecture-specific (back-end and run-time) effort than the other projects. LLVM already provides mechanisms (intrinsic functions) to traverse the stack frames on a stack and to inspect them, which should simplify some of the implementation effort.

References:

(a) A modular approach to on-stack replacement in LLVM," Nurudeen A. Lameed and Laurie J. Hendren. VEE 2013.

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

Another major feature lacking in the LLVM JIT is run-time adaptive (profile-guided) optimization. This 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, MATTHEW ARNOLD, STEPHEN J. FINK, DAVID GROVE, MICHAEL HIND, AND PETER F. SWEENEY. Proceedings of the IEEE, 32 (2), Feb. 2005.