CS 744 - Advanced Compiler Design
Course Project

Timeline:

- Brief project choice e-mail due May 17
- Project proposal due May 31
- Progress report e-mail due June 23
- Presentations approximately July 19, 21
- Final report due July 26

Overview

The project is an important part of the course, and makes up the bulk of your final grade. The project gives you a chance to explore a topic of your interest in depth. In particular, you will be expected to become familiar with **existing research** on your topic, **implement** some ideas (either suggested by existing work or devised on your own), and **experimentally evaluate** them. You may do your implementation within any freely-available compiler infrastructure (e.g. Soot, JikesRVM, Wala, LLVM, Scala, gcc, Eclipse, abc, SUIF, . . .).

When you have selected a project topic, send me an e-mail describing the proposed project or discuss it with me to make sure that it is appropriate and that no other student has chosen the same project. For some project ideas, I also list other mentors who are likely to have useful insights about the project. This informal project choice should be done by May 17. Then write a more formal project proposal, which should address the following:

- Motivation (why is this interesting?)
- Related Work (what has already been done? include references to specific papers)
- Specific Proposal (what exactly do you plan to implement, and how will you do it?)
- Experimental Evaluation (how will you evaluate your implementation?)
- Expected Results (what do you hope your experiment will show?)
- Timeline (is the project feasible in the time available?)

You may send me e-mail about your progress at any time, but you must e-mail me at least one progress report no later than June 23. If you find you are falling behind schedule, we may have to discuss changes to your project.

Near the end of the term, you will report on your project in a paper in the style of a PLDI research paper (approx. 10 pages), and give an overview of your project in a 20-minute presentation to the class.

Suggested Projects

You may choose a project topic from the following list, or one of your own choice. Many of these suggested topics are vague; it is up to you to study the relevant systems and literature to fill in the details. Please contact me if you are interested in discussing one of these topics further.

1. Optimization of Scala code:
   Scala is a statically typed object-oriented language with many features that promote writing abstract, reusable code. Such abstract code requires a good optimizing compiler to match the efficiency of code written in a more concrete style. The Scala compiler compiles Scala source code into Java bytecode that executes on any Java Virtual Machine. Compare the efficiency of some Scala programs and their Java equivalents.
   (a) Suggest and implement an optimization in the Scala compiler to improve the performance of these programs.
   (b) Suggest and implement an optimization in a Java Virtual Machine that improves the performance of the Java bytecode typically generated by the Scala compiler.
2. Call graph construction [9]:
   A call graph is a pre-requisite for almost every interprocedural analysis for an object-oriented language such as Java. Many open problems remain:
   (a) Integrated development environments (IDEs) are one context in which call graphs are useful. However, precise call graph construction algorithms that construct a call graph from scratch are too slow to be executed after every change made to the program being edited. Design and implement an incremental call graph construction algorithm that could execute continuously in an IDE while the program is being modified.
   (b) Even when using precise analyses, call graphs for Java programs constructed by static analysis tend to contain many more methods than are actually executed when a benchmark is run. This project involves comparing dynamic (run-time) and static call graphs, finding causes of the differences, and suggesting improvements to either the test suite (to increase the dynamic call graph size) or to the static analysis (to reduce the static call graph size) to reduce the discrepancy.
3. Presentation of analysis results in Eclipse:
   The results of static analyses, especially interprocedural ones, are often difficult to understand and interpret because the information they provide relates to large portions of a program. Within an IDE such a Eclipse, plugins can be developed to provide an interactive interface for browsing the results of an analysis.

   (a) Create an Eclipse plugin for browsing the results of a typestate analysis [6], to help programmers determine where a program violates temporal specifications. Possible mentor: Nomair Naeem.
   (b) Create an Eclipse plugin for browsing a call graph, including browsing of the run-time type information that was used to resolve dynamic dispatch in the construction of the call graph. Possible mentor: Karim Hamdan Ali.

4. Summarizing method side-effects for interprocedural analysis:
   Interprocedural analysis of Java programs generally requires knowledge of the effects of all methods in the whole program. This requirement is impractical: some methods (such as native methods) may be unavailable for analysis, or there may be too many methods to efficiently analyze all of them. One solution is to define a language for summarizing or simulating the effects of these methods, so that the simulations can be substituted in the analysis in place of the original methods.

   (a) Define such a framework, and use it to create summaries of the native methods in a recent version of the Java Standard Library. For inspiration, Soot already includes such a framework for version 1.3 of the Java Standard Library.
   (b) Define such a framework, and write an analysis that automatically summarizes the externally visible effects of a set of classes. For example, the analysis could be applied to the Java Standard Library to create a small set of methods that have the over-approximate the effect that the whole library has on an application. Possible mentor: Pavel Parizek.

5. Improvements to Wala/LLVM/JikesRVM/Pin/etc.:
   Wala is an open source Java analysis framework led by IBM Research. There was a request on the Wala mailing list for small projects that could be done to improve Wala. The reply from Stephen Fink, one of the key Wala developers, can be found at: `http://tinyurl.com/3o3b6y`
   You could also contribute to some other project, such as LLVM, JikesRVM, Pin, etc.

6. Applications of SAT and SMT solvers:
   SAT and SMT solvers (such as Z3: `http://research.microsoft.com/en-us/um/redmond/projects/z3/`) have found many applications in program analysis (for example [3], among many others), verification, specification, testing, and symbolic execution (for example [2]). Study some interesting application of these solvers.

7. Evaluation of parallel programming models:
   Several programming models have been proposed for clearly and simply expressing concurrent computations, such as Actors, parallel collection libraries (e.g. in Scala 2.9.0), revisions [1], amorphous data-parallelism [5], and more. Investigate the usability and performance of one of these models by implementing and tuning some interesting algorithm(s). Possible mentor: Jonathan Rodriguez.

8. Assessment of the benefits of flow-sensitive points-to analysis of C:
   Traditionally, most practical points-to analysis algorithms have been flow-insensitive because of the high cost of flow sensitivity. Recently, several efficient flow sensitive points-to analysis algorithms have been proposed, for example [4]. How, it remains unknown how much of an

improvement in precision flow sensitivity actually provides. Evaluate this benefit by measuring the effect on analyses and optimizations that make use of points-to information, and by qualitatively inspecting the parts of programs on which the results of flow-insensitive and flow-sensitive analysis differ.

9. Points-to analysis in gcc:
   Begin by studying and thoroughly understanding the points-to analysis algorithms that are implemented in gcc. Identify ways in which they could be improved and implement them. A good article to get started is [8].

10. Pure method annotations for Scala:
    Pure methods are those that do not modify any objects that existed before the method was called. Design a system of automatically checkable annotations that is sound (i.e. every method annotated as pure is actually pure) and as precise as possible (i.e. many methods that are actually pure can be correctly annotated as pure according to the rules of the system). As inspiration, a relatively simple such annotation system has recently been proposed for Java [7].

References

[1] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 691–707. ACM, 2010.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[3] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 187–200. ACM, 2011.

[4] Ondřej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, New York, NY, USA, 2011. ACM.

[5] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2010. ACM.

[6] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 347–366, New York, NY, USA, 2008. ACM.

[7] David J. Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2011.

[8] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[9] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293. ACM Press, 2000.