

Research Statement

Tok Teck Bok

Overview

Compilers play an instrumental role in improving the correctness, reliability and running time of programs. For example, they are used in detecting software errors, building fault tolerant programs, and optimizing software libraries for different architectures. As software programs grow in size and sophistication, such tools are also needed to improve programmer productivity. To produce high-quality outputs, the compilers often run for a long time and consume a lot of memory. My research strives to find new compiler techniques to improve on the performance of these high-quality compilers.

Current Research

My current research focuses on dataflow analysis. The purpose of dataflow analysis is to statically reason about the runtime behavior of programs. There is wide spectrum of analysis methods. Program understanding and error checking (such as security loophole or invalid state detection) are two high-level methods that are important for software quality and programmer productivity. In order to be useful, the analyses have to be precise. Unfortunately, precise whole-program analyses are often computationally expensive (e.g. NP hard).

We devise techniques that improve such high-precision and expensive program analyses. The key insight of our approach is that sometimes there is a high volume of unimportant computations—they do not improve the accuracies of the analyses. By identifying and reducing these computations, we improve analysis time without sacrificing accuracy. We evaluate our idea by applying new algorithms on different components of an analysis. Specifically, we evaluate these ideas:

1. We devise an improved worklist management algorithm that drastically reduces the amount of work that is placed on the worklist [2]. We achieve this by exploiting a kind of sparse dependences found in most programs, so that only necessary work are placed on the worklist. The technique enables us to reduce analysis time by 44% on average and up to 88% in the best case (actual analysis time before our optimization is as long as 6200 seconds).
2. We develop Relevance-Based Context-Sensitivity (RBCS), which groups similar contexts—a collective set of analysis steps—together in such a way that only important information will be computed precisely, and that the number of contexts to analyze is considerably smaller. The biggest benefit of our technique is we can now complete analysis on many large programs (they run out of memory without our technique). On the remaining programs, running time is $3.3\times$ faster on average and up to $260\times$ in the best case [3].
3. Interprocedural Finite Distributive Subset (IFDS) dataflow analysis is a well-known technique for precise program analysis. It is popular because its polynomial cost is considered relatively efficient. However, it is still expensive for large programs. We devise a new algorithm to exploit dependences in the programs (the same kind as in (1)), so that the graph-based internal data

structure becomes sparse. This new graph is faster to construct, and the subsequent program analysis (reasoning on this graph) is also more efficient with same level of precision. On our benchmark suite, the graphs are on average $10\times$ smaller, and the analysis time is $2.5\times$ faster [4].

Future Direction

Extensions to Other Dimensions of Precisions

Our new worklist algorithm and RBCS algorithm reduce unimportant work by focusing on two dimensions of a high-precision program analysis: flow- and context-sensitivities. The approach is applicable to other dimensions. Consider shape analysis, which attempts to determine properties of heap contents. Such information is useful for pointer analysis but its worst-case complexity is doubly-exponential. We hypothesize that most parts of a program do not need such precise solutions, and we propose identifying and selectively applying shape analysis to obtain that extra accuracy at a small additional cost.

Unimportant computations may also exist in other dimensions such as path sensitivity and heap model. (Additional examples can be defined with object-oriented programs.) In path sensitive analysis, many paths share common segments, so that computing solutions to these segments separately constitute redundant work. A heap-sensitive analysis models heap objects more precisely by using more abstract objects. However, sometimes solutions to these abstract objects are identical, so that only one representative is needed. To summarize, it seems possible to extend our proven techniques to other dimensions of precisions.

Client-Driven Analysis

Client-Driven analysis [1] is a proven technique for efficient and precise analysis, but it also has some promising extensions. I have—and plan to continue to—made improvements to the existing system, as well as explored new extensions. The next two paragraphs briefly explains what is Client-Driven analysis.

Program analyses often work together to produce some really useful results. For example, a pointer analysis determines what a pointer may point-to; this in turn is used in other analyses such as program verification. One important observation is that while it is important to know the value of some ‘key’ pointers, very often we do not care about many other variables in the program. This disparity depends directly on the needs of the client analysis (program verification in the above example). Client-Driven analysis builds on this insight, by applying high-precision pointer analysis on subset of variables, and low-precision analysis on the rest. In this way, the client analysis can produce high quality results without paying a high price.

This mixed-precision approach has previously been applied to pointer analysis with error checking (security loophole detection). This previous study [1] focuses on varying two precision dimensions (flow- and context-sensitivities). We have already conducted some initial studies on other dimensions (heap- and path-sensitivities). In addition, there are potentially many other client analyses that we can study (e.g. program verification, optimization). Another possible extension is incremental analysis, which is to increase precision in one dimension at a time. It can be shown that this approach can sometimes produce better results than when precision in all dimensions are increased at the same time.

Program Analysis and Model Checking

My current research has focused on reducing unimportant computations in expensive program analysis. The idea is similar to state space reduction in program abstraction, a common technique used in model checking. While the motivations are similar, the approaches are quite different. Nevertheless, there is a promising possibilities that some of the program analysis techniques can be useful for model checking. While we are still developing the ideas, we speculate that our notions of finding unimportant work, and the different dimensions of precision, are well-defined and systematic methods that can benefit model checking. Conversely, it is also an interesting question if any model checking techniques can benefit program analysis. Already my advisor and another graduate student have begun some initial study, and we all believe there is a lot of work to be done here, with many collaboration opportunities.

C-Breeze Compiler Infrastructure

C-Breeze is the compiler that we developed and maintained at our research group. It is a important tool in our research, but with some enhancements it may be possible to help us venture into other research areas such as programming languages, model checking, and simulations. The compiler is a small and flexible compiler infrastructure intended to be both a research and teaching tool. It is currently a source-to-source translator for ANSI C, and it also dismantles the source code into intermediate program representation. The compiler allows plug-in phases which perform various program analyses, optimizations, or transformations. We can add supports to the compiler such as front- and back-end for Java or other languages, or a back-end for generating machine code. Continuing to enrich the compiler helps to increase future research opportunities.

Other Research Interests

During my graduate student career I had brief but invaluable experience with the TRIPS project (Terap, Reliable, Intelligently adaptive Processing System, <http://www.cs.utexas.edu/~trips>), which is developing a new class of technology-scalable, power efficient, high-performance microprocessor architectures. I helped to investigate how to write a high-performance library routine for the new architecture. I find that experience a very fulfilling one, and I would be glad to be involved in similar projects again if given the chance.

My other interests include (not in any order) programming languages, parallel and distributed computing, fault tolerance, grid computing, security, algorithms, operating system, and numerical analysis. These are all exciting research areas, and at the same time they can also open up new opportunities when combined with compiler techniques.

References

- [1] Samuel Z. Guyer and Calvin Lin. Client Driven Pointer Analysis”, In Radhia Cousot, editor, *10th Annual International Static Analysis Symposium (SAS’03)*, volume 2694 of *Lecture Notes on Computer Science*” pages 214–236, June 2003.
- [2] Teck Bok Tok and Samuel Z. Guyer and Calvin Lin. Efficient Flow-Sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, number 3923 in *Lecture Notes in Computer Science*, pages 17–31, March 2006.
- [3] Teck Bok Tok and Calvin Lin. Relevance-based context-sensitivity. Under preparation.

[4] Teck Bok Tok and Calvin Lin. Improving the practical scalability of reachability-based dataflow analysis. Under preparation.