# Research Statement

John Whaley

January 2005

## 1 My Approach to Research

As a child, I was always interested in building things. When I was six years old, I taught myself programming by typing in programs that were printed in magazines and modifying them to add new features. To my six-year-old mind, computers were a source of endless possibilities; whatever I could conceive, I could build. Over twenty-two years later, I still have that same enthusiasm. My interest in computer science research is fundamentally driven by the desire to build better systems — faster, more reliable, more maintainable, or that make new things possible — and to help others do the same.

At the same time, I love intellectual challenges. Nothing satisfies me more than coming up with an elegant solution to a difficult problem. I was fortunate enough to have a solid foundation in theory from my studies at MIT and Stanford. Although I am not the type to do theory for theory's sake, I find theory and formalism indispensable in clarifying ideas, finding unseen corner cases, suggesting new directions to explore, and in the fundamental understanding of an algorithm.

The combination of engineering and theory is what first attracted me to the area of compilers and program analysis. It is one of the areas in computer science where theoretical results are very often put into practice. Also, work in compilers and program analysis can have a great impact in improving many systems. Any improvement I make instantly applies across a wide variety of programs; the impact is increased because I operate at the meta level. Finally, there are many challenging problems in compilers and program analysis: Most of the problems are NP-complete or undecidable.

My approach to research is influenced by both engineering and theory. When I attack a new research problem, I start by thinking deeply about the problem. I will often build a small prototype to test out an idea or to gain insight. After I feel I understand the problem sufficiently, I research what others have published on the problem. I find that the only way I can fully understand and evaluate other's approaches is if I first understand the problem myself. After that, I start implementing. My first attempt is rarely successful, but my experience in building the implementation quickly exposes what the real issues are. Then, I throw away my first implementation and use what I have learned to do it right. Finally, after I have a working implementation, I sit down and try to document and formalize the ideas. I feel it is important to have a working implementation that you can experiment with before you formalize the system because otherwise your formulation may not match reality or miss some key subtleties. The process of documentation and formalization often suggests new ways of simplifying or improving the implementation and exposes corner cases that I would not have considered. Thus, both engineering and theory play complementary roles in my approach to research.

Finally, I believe strongly in the openness of research and the free flow of ideas. I find communication with others invaluable in the development and understanding of research ideas. Most of my research ideas were not sudden personal inspirations but were the result of interacting with others. I often take ideas from different areas of computer science and apply them to new areas. To facilitate cooperation, communication, and the advancement of research, I release all of my implementations as open source so that other researchers can build upon my work. I use a public CVS repository for much of my day-to-day research, so anyone can find out what I am working on and have access to my latest research.

## 2 Research Interests

My primary research interest is in program analysis. I am interested in using advanced program analyses to help find bugs and security flaws, increase software reliability, improve performance, assist the software development process, and enhance software understanding. I want to build tools and techniques that are usable by others and that work on real-life programs.

The remainder of this section outlines some of the specific research areas I am interested in.

### 2.1 Pointer Analysis

Pointer analysis is one of the most fundamental problems in program analysis. Pointer analysis attempts to answer the basic question, "What can a pointer point to?" (and the related question, "Can these two pointers point to the same thing?") It has applications and implications across almost all program analyses and optimizations. Understanding pointers is essential to the understanding of almost any program.

Despite its importance, pointer analysis is still one of the most vexing problems in program analysis. Early techniques were flow-sensitive but context-insensitive, which lead to poor accuracy. Context-sensitive techniques were too slow to be usable on even moderate-sized programs. Later, Steensgaard and Andersen each devised flow- and context-insensitive algorithms that could handle large programs.

In 1999, Martin Rinard and I developed a flow-sensitive, context-sensitive pointer alias analysis[19]. Our analysis was compositional, which means that (in the absence of recursion) it analyzed each method once and generated a summary for that method, which was then applied to each of its callers. (Recursive cycles were iterated until they reached a fixpoint.) Our analysis was also unique in that it supported partial program analysis by explicitly representing the interactions between analyzed and unanalyzed regions of the program. Our OOPSLA paper describing the algorithm has been very well-cited, and there have been at least a dozen publications on extensions to our algorithm. We later extended the algorithm to handle interactions between multiple threads[7].

In 2002, Monica Lam and I designed and implemented an efficient context-insensitive inclusion-based points to analysis for Java programs[16]. My algorithm was based on a fast algorithm by Heintze and Tardieu[5]. It improved on their algorithm by adding field sensitivity and by treating local variables in a flow-sensitive manner.

Most recently in 2004, Monica Lam and I designed a context-sensitive inclusion-based pointer analysis using binary decision diagrams[17]. The algorithm achieves context sensitivity by creating a clone of a method for every acyclic path through the call graph and running a context-insensitive algorithm over the expanded call graph to get context-sensitive results. Normally, this formulation is hopelessly intractable as a call graph often has $10^{14}$ acyclic paths or more. But by using a clever encoding that exposes the commonalities among contexts, the exponential relations can be computed efficiently using binary decision diagrams (BDDs). The result is a highly accurate, yet very efficient pointer analysis that can scale to very large programs. This algorithm was a significant advance in the state-of-the-art in pointer analysis. The publication describing this algorithm was the winner of the best paper award at PLDI.

Formulating context sensitivity as cloning has other benefits. First, it greatly simplifies the development of context-sensitive analyses. Context-sensitive algorithms are notoriously difficult, while with cloning one can just use a context-insensitive algorithm on the expanded call graph to get context-sensitive results. Second, unlike most other techniques for context sensitivity, cloning calculates all of the answers for all of the contexts at once. With a cloning approach, it is possible for the analysis to answer questions such as "Under what contexts can this occur?", which is not possible when using, for example, a summary-based approach. Finally, the technique of cloning allows for fine-grained control over the amount of context sensitivity in different parts of the program. By cloning specific parts of the program more than others, one can focus the analysis effort on the parts that are more important.

Looking forward, I am interested in further improving the state-of-the-art in pointer analysis and finding new applications for analysis results. One of the weaknesses of my PLDI 2004 algorithm is in its treatment of object creation sites — the analysis represents all objects created at a particular site as a single location, so it cannot distinguish between multiple objects created at a single creation site. I have been investigating using different forms of context sensitivity to gain better precision with respect to object creation sites and also better precision within recursive cycles. I am also interested in new applications for accurate pointer analysis results. One of my colleagues has used my context-sensitive analysis to find SQL injection vulnerabilities in large programs; his results indicate that accurate context-sensitive analysis is absolutely essential to the problem[6]. Another has written a version of my analysis for C programs and used it to optimize away bounds checks in the CRED dynamic bounds checker[2]. I am also working on using pointer analysis results to guide the automatic translation of legacy Java code to use the Java generic libraries. In the future, I plan to use the analysis results to tackle many more difficult problems in the areas of bug finding, optimizations, and automated program transformations.

## 2.2 Optimizations and Program Transformations

I am interested in improving the performance of code through optimizations and automated program transformation. My primary interest is in improving performance where my changes can have a big impact. In 1999, Martin Rinard and I designed an escape analysis for Java programs[19]. Escape analysis discovers objects that do not escape a particular region of the program; these objects can be stack-allocated or do not need synchronization, which can lead to very large performance improvements. In my master's thesis, I investigated improving performance through dynamic compilation[10]. My thesis dealt with how to automatically find and exploit opportunities for runtime specialization based on profile data. My coworkers at IBM Japan implemented many of these ideas in their commercial JIT compiler[8].

I have also done work on less traditional areas of performance. One of the major issues with client-side Java is not the steady-state performance, but rather the startup time and memory footprint. In 2001 I published two papers attacking these problems in different ways.

In the first paper, I developed a technique of checkpointing a running Java virtual machine using reflection and program analysis[13]. With this technique, you run the system to a desired state and then use reflection to discover the state of threads and objects in the system. Given the current state, a program analysis determines the set of reachable objects and methods. The objects and code are serialized to a checkpoint file; unreachable objects, methods and fields are automatically trimmed. This technique improved startup time by more than an order of magnitude, and also reduced memory footprint by up to 39% and code size by up to 19%.

In the second paper, I developed the technique of partial method compilation[12]. In partial method compilation, an online profiler is used to find code regions that are rarely executed and the method is recompiled and optimized without those regions. If a branch that was predicted to be rare is actually taken at run time, we fall back to the interpreter or dynamically compile another version of the code. By avoiding compiling and optimizing code that is rarely executed, we greatly improve compilation time with little or no degradation in performance. This paper was the winner of the best paper award at OOPSLA 2001. My coworkers at IBM Japan wrote a followup paper[9], which includes their experience with the technique in a commercial JIT compiler.

In the future, I would like to work on more optimizations that improve other nontraditional measures of performance, such as optimizing for power consumption, optimizing to reduce hardware requirements, or optimizing for reduced memory footprint. I would also like to investigate using program analysis for code transformations; for example, automatically updating code to match a new API or introducing dynamic checks where necessary to ensure safety.

## 2.3 Virtual Machines

Another research area I am interested in is that of virtual machines. Virtual machines provide a powerful platform for dynamically profiling and modifying software as it runs.

I have done research on profiling and online measurement within virtual machines. At IBM, I developed a portable sampling-based profiler for Java virtual machines[3, 11]. Because the profiler had very low overhead, it could be run continuously, providing a feedback mechanism to the dynamic compiler. The profiler used a novel data structure, the partial calling context tree (PCCT), which allowed the efficient encoding of approximate context-sensitive profile information.

While an employee at IBM, I was fortunate enough to be one of the first people to work on the Jalapeño virtual machine[1] (now known as the Jikes RVM). I was very active in its development and I wrote much of the optimizing compiler framework[4]. Jalapeño was unique in that it was the first real Java virtual machine that was written entirely in Java.

While a Ph.D. student at Stanford, I designed and implemented the Joeq virtual machine and compiler infrastructure[14], a system designed to facilitate research in virtual machine technologies such as Just-In-Time and Ahead-Of-Time compilation, advanced garbage collection techniques, distributed computation, sophisticated scheduling algorithms, and advanced run time techniques. Like Jalapeño, Joeq is entirely implemented in Java, leading to reliability, portability, maintainability, and efficiency. However, unlike Jalapeño, Joeq is also language-independent, so code from any supported language can be seamlessly compiled, linked, and executed — all dynamically. Each component of the virtual machine is written to be independent with a general but well-defined interface, making it easy to experiment with new ideas. Joeq is released as open source software, and is being used as a framework by many researchers all over the world. It is also the basis for the Advanced Compilation Techniques class at Stanford.

Virtual machines allow the easy introspection of components as they are executing. When you combine this with a static analysis, you can obtain both upper and lower bounds on behavior. In a publication that won an ACM SIGSOFT Distinguished Paper Award, Michael Martin, Monica Lam, and I combined static analyses to deduce illegal call sequences in a program, dynamic instrumentation techniques to extract models from execution runs, and a dynamic model checker that ensures that the code conforms to the model[18].

I foresee many new applications for virtual machines in the future. I am interested in using virtual machines for more sophisticated inspection of running programs, dynamic reoptimization, program shepherding, identifying and isolating compromised machines, better encapsulation, and easier state migration.

### 2.4 Program Analysis using BDDs

While working on the BDD-based context-sensitive pointer analysis, I realized there was a correspondence between the BDD operations and operations in relational algebra. I began to formulate the analysis using Datalog, a declarative logic programming language for talking about relations. I represent all program information and analysis results as relations in a database. Because Datalog is succinct and declarative, one can express points-to analyses and many other algorithms simply and intuitively in just a few Datalog rules. These rules correspond exactly to the inference rules one would use when writing a formal description of the algorithm. Using Datalog, I was able to encode pointer analysis in only four lines!

I built a tool called bddbddb, which stands for “BDD-Based Deductive DataBase”. bddbddb automatically translates Datalog programs into highly efficient BDD implementations. Using bddbddb has a number of advantages. First, bddbddb makes development of BDD-based program analyses exceedingly easy. Whereas before you would have to write hundreds or thousands of lines of code to implement an analysis, with bddbddb you only need to write a few lines of Datalog. Second, it closes the gap between the algorithm specification and its implementation. In bddbddb, the algorithm specification is automatically translated into an implementation, so as long as your algorithm specification is correct you can be reasonably sure that your implementation will also be correct. Third, because BDDs can efficiently handle exponential relations, it allows us to solve heretofore unsolved problems in program analysis, such as context-sensitive pointer analysis for large programs. Finally, bddbddb makes advanced program analysis more accessible. Trying out a new idea in program analysis used to be confined to the realm of experts and compiler writers, and would take weeks to months of tedious effort to implement and debug. With bddbddb, writing a new analysis is simply a matter of writing a few straightforward inference rules. The tool takes care of most of the tedious part and helps you develop powerful program analyses easily.

One of the most difficult and critical issues with using BDDs is the question of BDD variable ordering. Different BDD variable orderings can change an operation that runs in linear time to one that runs in exponential time, or vice-versa. Unfortunately, finding the best BDD variable order is an NP-complete problem. Michael Carbin and I have developed an active learning algorithm to determine variable orderings in BDD representations[15]. It uses the execution times of a number of variable orderings to learn the important features of a good variable ordering, and it uses the learned features to direct which variable orderings to measure. We have implemented this algorithm in the bddbddb system, and experiments show that variable orderings generated by our algorithm can outperform those obtained after months of manual exploration.

Another hazard of using BDDs is the fact that they can be memory-intensive. Recently, I extended bddbddb to support distributed computation of Datalog queries. A problem that is too large to fit on one machine is partitioned into smaller pieces, each of which lives on a separate client. Each client calculates the fixpoint of its local data and communicates newly-generated relations to the other clients. Preliminary results indicate that this technique can speed up the computation and also allow larger problems to be solved.

## 3 Conclusion and Future Directions

After my experience with using bddbddb, I am now firmly convinced that the future of program analysis is in high-level algorithm specifications that are automatically translated into efficient implementations. BDDs are good for certain problems; however, for many problems other data structures are much more effective. I would like to extend bddbddb to support other underlying data structures, such as bit vectors and fast union-find data structures, and to be able to automatically choose the best data structure for the task at hand.

My goal is to improve program analysis technology to such a point that it becomes a standard and indispensable part of the software engineering process. Program analysis should be powerful enough to give precise answers to hard questions but simple enough so that any programmer can use it.

Program analysis is full of interesting research problems that have practical applications. I am very excited about the possibilities and I am looking forward to contributing whatever I can. Although I do not think I will necessarily stay in the area of program analysis for my entire career, there are plenty of interesting problems to keep me busy for the foreseeable future.

## References

[1] B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[2] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Using C pointer analysis to improve software security. In *International Conference on Software Engineering*, May 2005.

[3] J. J. Barton and J. Whaley. A real-time performance visualizer for Java. *Dr. Dobb's Journal of Software Tools*, 23(3):44, 46–48, 105, March 1998.

[4] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *ACM Conference on Java Grande*, pages 129–141, June 1999.

[5] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[6] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[7] M. Rinard and J. Whaley. Compositional pointer and escape analysis for multithreaded Java programs. Technical Report MIT-LCS-TR-795, MIT Laboratory for Computer Science, November 1999.

[8] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM SIGPLAN Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 180–194, 2001.

[9] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 312–323, 2003.

[10] J. Whaley. Dynamic optimization thru the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[11] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM Conference on Java Grande*, pages 78–87, June 2000.

[12] J. Whaley. Partial method compilation using dynamic profile information. In *ACM SIGPLAN Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 166–179, Oct. 2001.

[13] J. Whaley. System checkpointing using reflection and program analysis. In *Reflection, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 44–51, Sept. 2001.

[14] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *ACM SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2003.

[15] J. Whaley, M. Carbin, and M. S. Lam. Finding effective variable orderings for BDD-based program analysis, 2005. Submitted for publication.

[16] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *9th International Static Analysis Symposium*, pages 180–195, Sept. 2002.

[17] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[18] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis*, pages 218–228, July 2002.

[19] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 187–206, Nov. 1999.