Linhai Song - Research Statement

My main research interests are software system, program analysis, and software engineering. The goal of my research is to help programmers build more efficient software systems.

Everyone wants software to run faster. Slow and inefficient software can easily frustrate end users and has already caused several highly publicized failures. Although researchers devoted decades to improving software performance transparently, *performance bugs* continue to pervasively degrade performance and waste computation resources in the field. Performance bugs are software implementation mistakes, where relatively simple source code changes can speed up software, while preserving functionalities. The current tool support for performance bug is preliminary due to the poor understanding of performance bugs. There are three trends making performance-bug problems even more urgent in the future. Firstly, in the multi-core era, when each core is unlikely to become faster, performance bugs are particularly harmful. Secondly, the increasing complexity of software systems, rapidly changing workloads and frequently updated hardware provide new opportunities for performance waste and new challenges in diagnosis. Thirdly, increasing energy costs provide a powerful economic argument for avoiding performance bugs. Combating performance bugs is critical.

My Ph.D. research has mainly focused on performance bugs. My experience spans different stages of combating software bugs: real-world bug understanding, bug detection, compiler optimization, failure diagnosis, and patch generation. In particular, I conduct the first characteristics study on 110 real-world performance bugs from 5 representative open-source software suites [6]. I build a static performance bug detection tool suite, based on extracted efficiency rules from fixed performance bugs [6], and help to build a dynamic performance bug detection tool for inefficient nested loops [5]. These two techniques find hundreds of previously unknown performance bugs, many of which have already been confirmed and fixed by developers. I design three source-to-source compiler optimizations to transparently eleminate performance bugs in applications offloading computation to Intel Xeon Phi Coprocessor [3], and this work won MICRO'2014 best paper nomination. For production-run performance failure diagnosis, I provide a statistical debugging solution to conduct fault localization [4], and design static-dynamic hybrid analysis to further diagnose identified inefficient loops [2]. Besides combating performance bugs, I also work on a concurrent bug fixing system [7], which won a SIGPLAN CACM research highlight nomination [1]. My future research will continue to focus on improving the understanding of performance problems and providing better tool support for developers to build efficient software, including an on-line algorithmic profiling system, a tool chain to better balance precision and performance for floating-point applications, and an empirical study for performance problems in web applications.

1 Dissertation Research

Real-world Performance Bug Understanding. Many empirical studies have been conducted for functional bugs. These studies have successfully guided the design of functional software testing, functional bug detection, and failure diagnosis. Poor understanding and wrong perceptions of performance bugs are partly the causes of today's performance-bug problem. The lack of empirical studies on performance bugs has severely limited the design of performance bug avoidance, testing, detection, and fixing tools.

I co-lead the first, to the best of our knowledge, comprehensive study of real-world performance bugs based on 110 bugs randomly collected from the bug databases of five representative open-source software suites [6]. The study is mainly conducted through 4 dimensions: what are the root causes of performance bugs, how performance bugs are introduced, how to expose performance bugs, and how to fix performance bugs. The study has made many interesting and important findings, including that two thirds of studied bugs are introduced by developers' wrong understanding of workload or API performance features, that more than one quarter of the bugs arise from previously correct code due to workload or API changes, and that almost half of the studied bugs require inputs with both special features and large scales to manifest. These findings provide guidance for future work to avoid, expose, detect, and fix performance bugs.

Static/Dynamic Performance Bug Detection. The empirical study on performance bug finds that almost half of the examined bug patches include reusable efficiency rules that can help detect and fix performance bugs. Inspired by this finding, I co-lead a rule-based performance bug detection project [6]. We collect rules from 25 Apache, Mozilla, and MySQL bug patches and build static checkers to find violations to these rules. Our checkers automatically find 125 potential performance problems (PPPs) in the original buggy versions.

Programmers failed to fix them together with the original 25 bugs where the rules come from. Our checkers also find 332 previously unknown PPPs in the latest versions of Apache, Mozilla, and MySQL. These include 219 PPPs found by checking an application using rules extracted from a different application. We report some of found problems to developers. Developers have confirmed 77 reported problems, and have fixed 15 reported problems. This static bug detection work demonstrates the existence and value of efficiency rules: efficiency rules in the study are usually violated at more than one place, by more than one developer, and sometimes in more than one program. Rule-based performance bug detection is promising. Our empirical study also finds that 90% performance bugs involve loops, and 50% performance bugs involve at least two-level of loops. Motivated by this finding, I help a fellow graduate student build a dynamic bug detection tool named Toddler for inefficient nested loops [5]. Toddler has found 42 new bugs in six Java projects, and developers have fixed 10 bugs and confirmed 6 more as real bugs.

Source-to-Source Compiler Optimization for Intel Xeon Phi Coprocessor. Xeon Phi coprocessors are introduced by Intel recently as new members of the manycore family. Compared with GPU, Xeon Phi coprocessors are easier to program, since they provide x86 compatibility and support many different programming models. In order to offload existing parallel loops, developers just need to add simple pragmas. However, my recent experience shows that simply adding pragmas does not give performance on the Xeon Phi coprocessor, and too many performance bugs contained in simply offloaded parallel loops.

After careful investigation, I design three source-to-source compiler optimization to transparently eliminate performance bugs contained in offloaded parallel loops [3]. The first optimization, data streaming, is designed to reduce the overhead of transferring data between CPUs and coprocessors. The optimization automatically transfers offloaded codes to stream data to and from coprocessors, which overlaps data transfers with computation to hide data transfer overhead and reuse memory on coprocessors. The second optimization, regularization, is design to handle loops with irregular memory accesses. This optimization identifies irregular memory access patterns in a loop and re-arranges the order of computations to regularize memory accesses. It enables data streaming and vectorization for manycore processors in the presence of irregular memory accesses. The last optimization is designed to support the efficient transfer of large pointer-based data structures between CPUs and coprocessors. An augmented design of pointers is introduced for fast translating pointers between their CPU and coprocessor memory addresses. The designed optimizations can benefit 9 out of 12 benchmarks in the experiments, and improve the performance by 1.16x - 52.21x. This work won MICRO'2014 best paper nomination.

Performance Failure Diagnosis. The state of the art of performance failure diagnosis is preliminary. Profilers are the most commonly used tools, but profilers can only tell where computation resources are spent, but not where or why computation resources are wasted. I conduct two separated projects to improve performance failure diagnosis from two aspects:

Performance fault localization targets to point out where are the root causes, referring to static code regions that can cause inefficient execution. In the fault localization project [4], I first conduct an empirical study to understand how performance problems are observed and reported by real-world users. My study shows that statistical debugging is a natural fit to localize root causes for performance problems, which are often observed through comparison-based approaches and reported together with both good and bad inputs. Then I thoroughly investigate different design points in statistical debugging, including three different predicates and two different types of statistical models, to understand which design point works the best for performance diagnosis. Experimental results show that branch predicate under two statistical models can effectively localize root causes for performance bugs caused by wrong branch selection or inefficient loops. These two types can cover almost all user-perceived performance bugs in my study. Finally, I study how some unique nature of performance bugs allows sampling techniques to lower the overhead without extending the diagnosis latency.

Fine-grain root cause identification targets to explain why inefficient code regions waste performance. I build LDoctor [2], which can automatically and accurately identify the fine-grained root-cause type for inefficient loops, which dominate user-perceived performance bugs in my study. LDoctor is built in three stages. Firstly, a taxonomy for the root causes of common inefficient loops is figured out by a thorough study of 45 real-world inefficient loop problems. I find that all inefficient loops are caused either by resultless or by redundancy. Secondly, static-dynamic hybrid analysis is built to balance performance and accuracy. Thirdly, sampling is used to further lower the run-time overhead of LDoctor, without degrading diagnosis capability. I use 18 real-world

performance problems to evaluate LDoctor. The evaluation shows that LDoctor can accurately identify the detailed root causes for all 18 benchmarks, and provide correct fix-strategy suggestions for 16 benchmarks. All these are achieved with low run-time overhead.

2 Future Research

The goal of my research is to improve performance of various types of software systems. My future projects can be conducted along two lines. The first part of my future research is to combat studied performance bugs in different aspects from what I have already done, like on-line performance bug detection and automatic performance bug fixing. The second part of my future research is to explore performance problems in other areas. Although the performance bugs in my study provide a good coverage of real-world performance problems, there are still uncovered categories, like performance problems in scientific computation, distributed systems, mobile and web applications. The following are some immediate research opportunities:

On-line Algorithmic Profiling. Many performance bugs in my study are caused by conducting computation in superlinear complexity. There are two possible reasons why these performance bugs are introduced. In one case, developers know the algorithm is superlinear, but they assume the workload is small. In the other case, developers do not notice the algorithm is superlinear, and due to the small size of test inputs, the performance bug escapes from the testing process. On-line algorithmic monitoring can detect performance bugs under both of these two situations. Previous work on algorithmic profiling or monitoring will bring 10X run-time overhead, and this cannot be tolerated in production-run. I will build a tool which can collect run-time information in a reasonable overhead from deployed software, and infer approximate complexity for monitored computation.

Tool Support for Tuning Floating-point Applications. Floating-point arithmetic is used in a wide variety of applications, including high-performance computing, graphics and finance. Many programming language provides two types of floating-point numbers: double and float. Double numbers are represented by using more digits, and they are more accurate than float numbers. Insufficient accuracy may accumulate during computation, and make programs instable or generate final results with intolerable large relative errors. However, float numbers can be processed much faster than double numbers, because float numbers bring less memory traffic, and after accelerated by vectorization, one SIMD instruction can process one time more float numbers than double numbers. Choosing between float numbers and double numbers should balance precision and performance. I would like to provide tool support to help developers optimize floating-point arithmetic programs under given precision bounds.

Empirical Study on Performance Problems in Real-world Web Applications. Web applications are highly interactive, and latency for event handlers in web applications is more important than the whole execution time. This feature makes performance problems in web applications different from performance bugs I have studied. Web applications are popular. Optimizing web applications and combating performance bugs in web applications are important. I would like to study performance problems in open-source web applications from github, and provide better understanding for the root causes, exposing conditions, and fix strategies of performance problems in web applications, leveraging my previous experience of bug study.

References

[1] SIGPLAN. "SIGPLAN CACM Research Highlights Nominated Papers". URL: http://http://www.sigplan.org/Newsletters/CACM/Papers/.

[2] **Linhai Song** and Shan Lu. "Performance Diagnosis for Inefficient Loops". Under Submission.

[3] **Linhai Song**, Min Feng, Nishkam Ravi, Yi Yang, and Srimat Chakradhar. "COMP: Compiler Optimizations for Manycore Processors". In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, **MICRO'14**, December 2014. **(Best Paper Nomination)**.

[4] **Linhai Song** and Shan Lu. "Statistical Debugging for Real-world Performance Problems". In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, **OOPSLA'14**, October 2014.

[5] Adrian Nistor, **Linhai Song**, Darko Marinov, and Shan Lu. "Toddler: Detecting Performance Problems via Similar Memory-access Patterns". In *Proceedings of the 2013 International Conference on Software Engineering*, **ICSE'13**, May 2013.

[6] Guoliang Jin, **Linhai Song**, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and Detecting Real-World Performance Bugs". In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, **PLDI '12**, June 2012.

[7] Guoliang Jin, **Linhai Song**, Wei Zhang, Shan Lu, and Ben Liblit. "Automated Atomicity-Violation Fixing". In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, **PLDI '11**, June 2011. Won a **SIGPLAN CACM Research Highlights Nomination** `http://www.sigplan.org/Newsletters/CACM/Papers`.