Linhai Song - Research Statement

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

My dissertation research has mainly focused on software performance. Everyone wants software to run fast. Slow and inefficient software can easily frustrate end users and cause economic loss. The software-inefficiency problem has already caused several highly publicized failures. My research philosophy is to view software-inefficiency problem from the perspective of combating *performance bugs*. Performance bugs are software implementation mistakes that can cause inefficient execution. Performance bugs cannot be optimized away by state-of-practice compilers. Many of them escape from in-house testing and manifest in front of end users, causing severe performance degradation and huge energy waste in the field. Performance bugs are becoming more critical, with the increasing complexity of modern software and workload, the meager increases of single-core hardware performance, and pressing energy concerns. It is urgent to combat performance bugs.

To fight performance bugs, my methodology is investigate existing approaches originally designed for functional bugs and try to apply, adapt, and extend them for performance bugs. My experience spans different stages of combating performance bugs: real-world bug understanding, bug detection, failure diagnosis, and automated bug fixing. In particular, I conduct the first characteristics study on performance bugs, based on 110 bugs randomly collected from 5 representative open-source software suites [7]. I build a static performance bug detection tool suite by using extracted efficiency rules from fixed performance bugs [7], and help build a dynamic performance bug detection tool for inefficient nested loops [6]. I explore how to apply statistical debugging to performance failure diagnosis [5], and design a series of static-dynamic hybrid analysis routines to provide more detailed diagnosis information for inefficient loops [2]. I design three source-to-source code transformations to automatically fix performance bugs in parallel applications, which offload computation to Intel Xeon Phi manycore coprocessor [4].

Besides software efficiency, I also work on two software reliability projects. It is challenging to fix concurrency bugs. Long repair times and wrong patches are common for concurrency bugs. I help build a concurrency bug fixing system [8], which can automatically eliminate one common type of concurrency bug: single-variable atomicity-violation. Multi-threaded programs are difficult to write. Missing proper synchronization would lead to correctness bugs and over synchronization would lead to performance problems. I help study change histories of critical sections in open-source software [3] to provide better understanding of synchronization challenges faced by real-world developers.

My research work has already led to real-world impact. My two performance bug detection techniques [6, 7] find hundreds of previously unknown performance bugs in mature open-source software, and many of them have already been confirmed and fixed by developers. My performance bug fixing project [4] won MICRO'14 best paper runner up, and my concurrency bug fixing system [8] won ACM SIGPLAN Research Highlights Award [1]. According to Google Scholar, all my papers have 294 citations.

1 Dissertation Research

To address software-inefficiency problem, my research efforts cover the whole stack of software systems, from hardware to software, and cover all aspects of combating software bugs, from understanding, to detecting, diagnosing, and fixing.

Real-world Performance Bug Understanding. Like functional bugs, research on performance bugs should also be guided by empirical studies. Poor understanding of performance bugs is part of the causes of today's performance bug problem. In order to improve the understanding of real-world performance bugs, I co-lead the first, to the best of our knowledge, empirical study on real-world performance bugs, based on 110 bugs randomly sampled from five open-source software suites [7]. Following the lifetime of performance bugs, our study is mainly performed in four dimensions. We study the root causes of performance bugs, how they are introduced, how to expose them, and how to fix them. The main findings of our study include that (1) performance bugs have dominating root causes and fix strategies, which are highly correlated with each other; (2) workload mismatch and misunderstanding of APIs' performance features are two major reasons why performance bugs are introduced; and (3) around half of the studied performance bugs require inputs with both special features and large scales to manifest.

Our empirical study can guide future research on performance bugs. According to Google Scholar, our paper has already been cited for 87 times since 2012. Our empirical study has already motivated our own bug detection and diagnosis projects.

Static/Dynamic Performance Bug Detection. Inspired by our empirical study, we hypothesize that efficiency rules widely exist in software, rule violations can be statically checked, and violations also widely exist. To test our hypothesis, we manually inspect final patches of fixed performance bugs from Apache, Mozilla, and MySQL in our studied performance bug set, extract efficiency rules from 25 bug patches, and implement static checkers to detect rules' violations [7]. In total, our static checkers find 332 previously unknown performance bugs from the latest versions of Apache, Mozilla, and MySQL. Among them, 101 were inherited from the original buggy versions where final patches are applied. Tools are needed to help developers automatically and systematically find all similar bugs. 12 new bugs were introduced later. Tools are needed to help developers avoid making the same mistakes repeatedly. 219 new bugs are found by cross-application checking. There are generic rules among different software. We report some of identified bugs to developers. 77 reported bugs have already been confirmed by developers, including 15 reported bugs already fixed by developers. Overall, all our hypotheses are verified by our experimental results. Rule-based performance-bug detection is a promising direction.

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 novel automated test oracle named Toddler [6], which enables testings for performance bugs to use the well established and automated process of testing for functional bugs. Using Toddler, we find 42 new bugs in six Java projects: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. Based on our bug reports, developers so far fixed 10 bugs and confirmed 6 more as real bugs.

Performance Failure Diagnosis. Due to the preliminary tool support, many performance bugs escape from in-house performance testing and manifest in front of end users. After users report performance bugs, developers need to diagnose them and fix them. Diagnosing user-reported performance failure is another key aspect of fighting performance bugs.

We first investigate the feasibility and design space to apply statistical debugging to performance failure diagnosis [5]. After studying 65 user-reported performance bugs in our bug set, we find that majority of performance bugs are observed through comparison, and many user-filed performance bug reports contain not only bad inputs, but also similar and good inputs. Statistical debugging is a natural fix for user-reported performance bugs. We evaluate three types of widely used predicates and two representative statistical models. Our evaluation results show that branch predicate plus two statistical models can effectively diagnose user-reported performance failure. Basic model can help diagnose performance failure caused by wrong branch decision, and ΔLDA model can identify inefficient loops. We apply sampling to performance failure diagnosis. Our experimental results show that special nature of loop-related performance bugs allows sampling to lower runtime overhead without sacrificing diagnosis latency, and this is very different from functional failure diagnosis.

We then build LDoctor [2] to further provide fine-grained diagnosis information for inefficient loops through two steps. We first figure out a root-cause taxonomy for common inefficient loops through a comprehensive study on 45 inefficient loops. Our taxonomy contains two major categories: resultless and redundancy, and several subcategories. Guided by our taxonomy, we then design a series of analysis for inefficient loops. Our analysis focuses its checking on suspicious loops pointed out by statistical debugging, hybridizes static and dynamic analysis to balance accuracy and performance, and relies on sampling and other designed optimization to further lower runtime overhead. We evaluate LDoctor by using 18 real-world inefficient loops. Evaluation results show that LDoctor can cover most root cause subcategories, report few false positives, and bring a low runtime overhead.

Performance Bug Fixing. 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, our recent experience shows that simply adding pragmas does not give performance, and too many performance bugs contained in offloaded parallel loops.

After careful investigation, we design three source-to-source code transformations to automatically fix performance bugs contained in offloaded parallel loops [4]. The first transformation, data streaming, is designed to reduce the overhead of transferring data between CPUs and coprocessors. The transformation automatically changes 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 transformation, regularization, is design to handle loops with irregular memory accesses. This transformation identifies irregular memory access patterns inside a offloaded 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 transformation is designed to support the efficient transfer for 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 transformations can benefit 9 out of 12 benchmarks in the experiments, and improve the performance by 1.16x - 52.21x. This work won MICRO'14 best paper runner up.

2 Future Research

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

On-line Algorithmic Profiling. Many performance bugs in our study are caused by conducting computation in superlinear complexity. There are two possible reasons why these performance bugs are introduced. In one case, under the assumption that workload would be small, developers choose to use superlinear algorithms. In the other case, unnoticed superlinear codes escape from the testing process, due to the small size of test inputs. On-line algorithmic monitoring can detect performance bugs in both of these two situations. Previous work on algorithmic profiling or monitoring will bring more than 10X runtime overhead, which cannot be tolerated in production run. How to build a tool which can collect runtime information in a reasonable overhead from deployed software and infer approximate complexity remains an open question.

Test Input Generation for Performance Bugs. Our empirical study shows that almost half of studied performance bugs need inputs with both special features and large scales to manifest. Existing techniques are designed to generate inputs with good code coverage and focus only on special features. How to extend existing input generation techniques with emphasis on large scales remains an open issue. Another important problem during performance testing is to automatically judge whether a performance bug has occurred. How to leverage existing dynamic performance bug detection techniques to build test oracles for performance bugs also remains an open issue.

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 vectorization optimization, one SIMD instruction can process one time more float numbers than double numbers. Choosing between float numbers and double numbers must 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 Real-world Web Applications. Web applications are highly interactive. This feature makes performance and reliability problems in web applications different from software systems I have studied. Web applications are popular. Optimizing web applications and improving reliability in web applications are important. I would like to study performance problems and reliability issues in open-source web applications, and provide better understanding for the root causes, exposing conditions, and fix strategies of these problems, leveraging my previous experience of bug study.

References

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

[2] **Linhai Song** and Shan Lu. "Performance Diagnosis for Inefficient Loops". `http://songlh.github.io/ldoctor.pdf` (**Under Submission**).

[3] Rui Gu, Guoliang Jin, **Linhai Song**, Linjie Zhu, and Shan Lu. "What Change History Tells Us About Thread Synchronization". In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, **ESEC/FSE'15**, 2015.

[4] **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**).

[5] **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.

[6] 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.

[7] 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. *: equal contributions.

[8] 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`).