Huang J, Zhang C. Debugging concurrent software: Advances and challenges. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(5): 861–868 Sept. 2016. DOI 10.1007/s11390-016-1669-8

Debugging Concurrent Software: Advances and Challenges

Jeff Huang [1] and Charles Zhang[2], *Member, ACM, IEEE*

[1] *Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843, U.S.A.*
[2] *Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China*

E-mail: jeff@cse.tamu.edu; charlesz@cse.ust.hk

Received May 18, 2016; revised July 7, 2016.

Abstract Concurrency debugging is an extremely important yet challenging problem that has been hampering developer productivity and software reliability in the multicore era. We have worked on this problem in the past eight years and have developed several effective methods and automated tools for helping developers debugging shared memory concurrent programs. This article discusses challenges in concurrency debugging and summarizes our research contributions in four important directions: concurrency bug reproduction, detection, understanding, and fixing. It also discusses other recent advances in tackling these challenges.

Keywords debugging, concurrency, record and replay

1 Introduction

Detecting and repairing software defects, or bugs, have been the most expensive parts of the software development process[1]. In many real-world software systems, new bugs are typically being reported faster than developers can handle them, and many known bugs are never fixed due to the lack of human resources[2]. However, over the past decade, the debugging crisis has been exacerbated by the multicore revolution, which has dramatically increased the number of developers who face the challenge of debugging concurrent software.

Concurrent software is notoriously difficult to debug because of the complexity in reasoning about concurrency. Due to the fact that computations from concurrent threads can interleave with each other, developers can no longer reason in a sequential way but have to reason about the often astronomically large thread interleaving space. The number of different thread interleavings is typically exponential in both the number of threads and the length of program execution. If any of the interleaving corner cases are missed (e.g., not tested), the bugs may be triggered in unfortunate situations and may lead to fatal failures. What is worse is that thread interleavings are often non-deterministic. Due to the scheduling non-determinism and the timing differences between different hardware cores, multiple runs of a concurrent program on the same machine with the same input can exhibit different behaviors. Consequently, Heisenbugs[3] such as race conditions widely plague concurrent software systems and complicate debugging, because they may "disappear" when developers want to understand them.

The concurrency debugging challenge has attracted significant research attention in the past few years, and researchers have proposed a wide spectrum of approaches to help developers debugging concurrent software and to improve the overall software safety and reliability. In the past eight years, we have worked on this problem and developed several effective approaches and automated tools for concurrency debugging. This article discusses challenges in debugging concurrent software with examples and presents our research contributions in these areas as well as other recent advances in concurrency debugging.

Survey
Special Section on Software Systems 2016
This work is supported by the United States NSF CAREER Award under Grant No. CCF-1552935 and a Google Faculty Research Award to Jeff Huang, and the Hong Kong SAR RGC/GRF under Grant Nos. 622909 and 621912 to Charles Zhang.

2 Challenges

Concurrency Bugs Are Difficult to Reproduce. When a failure occurs, developers first need to reproduce it in order to understand the bug. However, the non-deterministic interleaving makes reproducing concurrency bugs extremely difficult. Consider a simple multithreaded example in Fig.1. There are two threads T_1 and T_2 accessing two different shared variables x and y, and there is an error at line `4`. Because these two threads can execute concurrently on different cores, their execution order may follow different interleaving sequences. For example, execution may follow either the interleaving `A` or `B`, represented by the statement line numbers `1-2-6-7-3-4` and `1-2-3-5-6`, respectively. If the program execution follows the interleaving `A`, the error at line `4` is triggered. However, if it follows the interleaving `B`, the error does not manifest. To reproduce this bug, not only the same program input, but also the same thread interleaving is required. Unfortunately, it is very challenging to capture thread interleavings on multicore computers. Recording thread interleavings at runtime inevitably hampers the execution parallelism, often incurring unacceptable program slowdown and is hard to deploy in production.

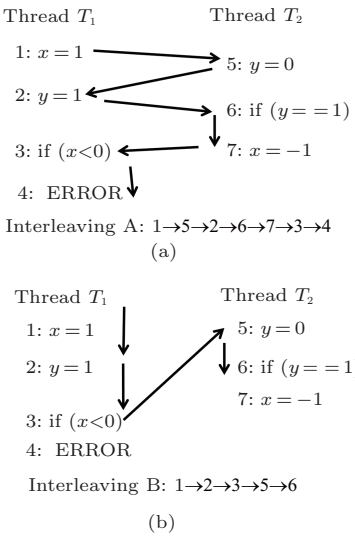

Fig.1. The same program exhibits different behaviors with different thread interleavings. The error at line 4 manifests with (a) the interleaving A but not (b) the interleaving B.

This problem is further complicated by the fact that most modern hardware can re-order instructions, which may make the bugs disappear while trying to capture them. Consider the two assertions ①`assert` and ②`assert` in Fig.2. On the sequential consistency (SC) model[4], ①`assert` will be violated if the two threads execute following the annotated interleaving. However, ②`assert` will never be violated under SC, but can be violated under the partial store order (PSO) model[5], which allows the reordering of writes to different memory addresses. For example, suppose lines 4 and 5 are re-ordered, ②`assert` will be violated following the interleaving shown in Fig.2(b). To reproduce this PSO bug, having a small runtime perturbation with reliable bug reproducibility is very challenging. For instance, there are at least 12 race pairs in this program that need to be tracked. A more subtle but critical point is that the PSO bug might never be captured if locks are used to track the interleaving. The memory fencing effect of locks can prevent the reordering to happen in test runs. And, if the tracking is disabled in production runs, the bug will re-appear.

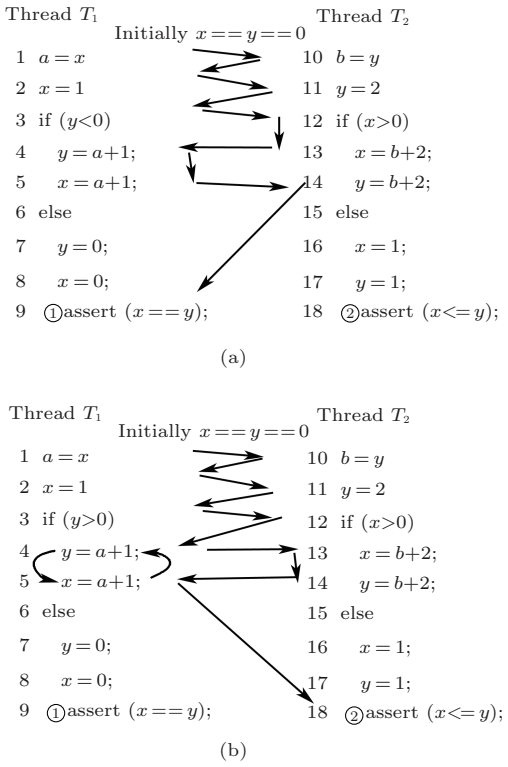

Fig.2. Concurrency errors on (a) sequential consistency and (b) partial store order memory models.

Concurrency Bugs Are Difficult to Detect. Due to the astronomically large number of thread interleavings, detecting concurrent bugs is challenging. Traditional program testing techniques for sequential programs do not work well for concurrent programs because they do not take the interleaving into account. Moreover, as the interleaving space is huge, testing is

often far from sufficient to cover the entire interleaving space. It is hard for static program analysis or model checking techniques to explore all thread interleavings in real-world programs, because there are just too many interleavings. Traditional dynamic analyses do not work well because only a limited size of paths and schedules are observed. Furthermore, traditional program analyses tend to report quite a large number of false alarms, further impeding the debugging process.

Concurrency Bugs Are Difficult to Understand. Typical executions of real-world concurrent programs often contain a large number of threads, thread interleavings, shared-memory dependencies and thread synchronizations. Even if a concurrency bug can be reproduced deterministically, it is still very challenging for developers to locate and understand the cause of the bug. Moreover, the performance of replay is often significantly slower than native execution. For long running programs, the bug reproduction process may take a long time. Furthermore, the bug reasoning process based on the trace often involves frequent context switches between the executions of different threads. These frequent context switches can significantly impair the effectiveness of concurrent program debugging, because developers can no longer reason sequentially to understand a concurrency bug.

Concurrency Bugs Are Difficult to Fix. After understanding a concurrency bug, fixing the bug is still challenging. Empirical studies[6-7] show that 39% of patches to concurrency bugs in operating systems are incorrect, and it takes more than two months on average to correctly fix a concurrency bug. A common way to fix concurrency bugs is to add the synchronization that prohibits the erroneous thread interleavings. However, facing the huge interleaving space and the large number of thread contexts, it is usually difficult to find the proper type of synchronizations and the proper location to place the synchronization. The improper placement of synchronization can not only incur prohibitive program slowdown, but also introduce new bugs such as deadlocks. Moreover, even if the proper synchronization is placed at the right location to rule out the manifested erroneous interleavings, it does not necessarily guarantee that the bug is fixed. Because the interleaving space is enormous, it is possible that some other unmanifested interleavings which can still trigger the bug are not forbidden by the added synchronization. Consequently, to validate the fixes, developers attempt to create thousands of threads and run tests millions of times — a tedious and ineffective practice.

3 Advances

Fig.3 shows an overview of our work. Our research has made contributions in four directions to make concurrency debugging easier: multiprocessor deterministic replay to reproduce concurrency bugs, predictive trace analysis to detect concurrency bugs, static and dynamic trace simplifications to help in understanding concurrency bugs, and data sharing reduction and synchronization verification to help in fixing concurrency bugs.

3.1 Concurrency Bug Reproduction

The technique of deterministic replay is one of the most important techniques for program understanding and debugging. We developed a lightweight record and replay system, called LEAP[8-9], that supports the deterministic replay of concurrent programs in general multicore and multiprocessor environments. LEAP is fast, portable, and deterministic. As long as a Heisenbug manifests once, LEAP is able to deterministically reproduce it in every subsequent execution. Moreover, underpinned by a new local-order based replay theorem, LEAP is able to deterministically reproduce errors in parallel programs with much lower overhead compared with previous approaches. LEAP is the first public available deterministic replay system for multithreaded

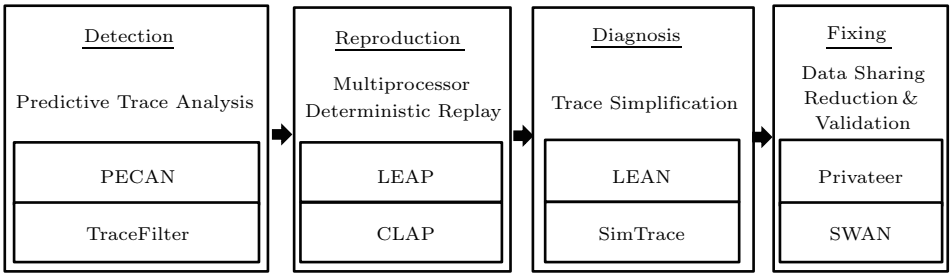

Fig.3. Overview of our work on concurrency debugging.

Java programs and has been used by several research groups worldwide.

We use the example in Fig.1 to illustrate the key idea of LEAP. Intuitively, we can record Interleaving A and use it to re-execute the program to reproduce the error at line 4. However, this is at the cost of seven global synchronization operations. In LEAP, we observe that thread accesses to different shared variables need not to be tracked together. Instead of recording a global interleaving, it is sufficient to record the thread access order that each shared variable sees. Specifically, we use two access vectors (***x.vec*** and ***y.vec***) for shared variables x and y and record $< t1, t2, t1 >$ and $< t2, t1, t2 >$ respectively. During replay, we associate x and y with condition variables to enforce the access order of threads to be identical to what is recorded in their respective access vectors. This guarantees to reproduce the error, but it requires zero global synchronization and only two groups of local synchronizations executed in parallel. We refer interested readers to [8-9] for the detailed design and implementation of LEAP that uses static analysis and bytecode instrumentation to transparently provide the capability of deterministic replay for Java programs.

One of the most important applications of replay is to reproduce program failures. However, for programs with heavy shared-memory dependencies, large runtime recording overhead (e.g., `>6X` by LEAP) is still incurred due to the challenging problem of using the synchronization on multiprocessors. How to achieve low overhead is critically important and challenging. CLAP[10] is a technique that leverages thread local path profiling[11] and constraint solving[12] to compute bug reproducing schedules. CLAP does not use any synchronization at runtime, and does not log any thread interleaving or any program state. As path profiling is featherweight (even less than 1% with hardware approaches[13]), CLAP is significantly more efficient than previous approaches that track shared-memory dependencies at runtime. Moreover, CLAP works not only for sequentially consistent executions, but also for a range of relaxed memory models such as PSO illustrated in Fig.2.

As illustrated in Fig.4, CLAP has two key phases.

1) Monitoring an instrumented execution of the program. Unlike most dynamic techniques that collect a global trace, this phase records only the local control-flow choices of each thread. In threads that exhibit bugs, these local traces lead to the occurrence of the bug.

2) Assembling a global execution that exhibits the bug. This phase in turn has several key steps.

• Find all the possible shared data access points (called SAP — a read, write, or synchronization) on the thread local paths that may cause non-determinism, via a static escape analysis.

• Compute the path conditions for each thread with symbolic execution. Given the program input, the path conditions are all symbolic formulae with the unknown values read by the SAPs.

• Encode all the other necessary execution constraints — i.e., the bug manifestation, the synchronization order, the memory order, and the read-write constraints — into a set of formulae in terms of the symbolic value variables and the order variables.

• Use an SMT solver to solve the constraints, which computes a schedule represented by an ordering of all the SAPs, and this schedule is then used by an application-level thread scheduler to deterministically reproduce the bug.

CLAP achieves several important advances over previous approaches.

1) CLAP obviates the logging of shared memory dependencies and program states, and completely avoids adding extra synchronizations. This not only substantially reduces the logging overhead compared with shared memory recorders such as LEAP, but also minimizes the perturbation that extra synchronizations

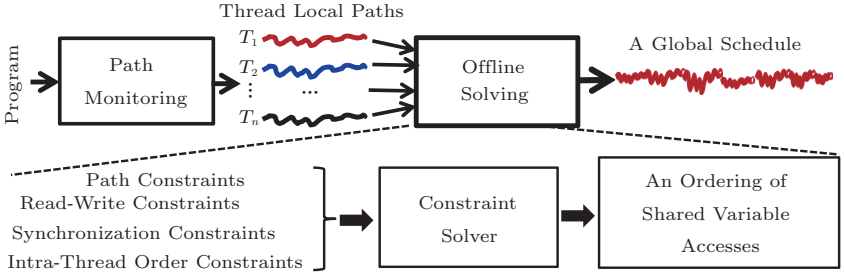

Fig.4. CLAP technical overview.

foreclose certain racy behaviors.

2) CLAP works not only for sequential consistent executions, but also for a range of relaxed memory models such as TSO and PSO[5]. We show that the memory order constraints between SAPs can be correctly modeled to comply with the memory model relaxation. This is of tremendous importance, because it makes CLAP applicable for the real production setting on commodity multiprocessors that allows the reordering of instructions.

3) CLAP can produce simpler bug-reproducing schedules than the original one. We are able to encode preemption bounding constraints over the order of shared data accesses to always produce a schedule with the minimal number of thread context switches. With this property, it becomes much easier to understand how the bug occurs due to the prolonged sequential reasoning. Moreover, through preemption bounding, the complexity of constraint solving is dramatically reduced from exponential to polynomial with respect to the execution length.

4) The constraint solving in CLAP is much easier to scale. The solver does not need to directly solve the complex path constraints (such as non-linear arithmetic or string constraints), but only to find a solution for the order variables that satisfies the path constraints. Thus, the solving task can be divided into two parts: generating candidate schedules (that comply with the memory order) and validating them (using the other constraints). The first part (which is searching possible executions) does not have complex constraints, and the second part (which does have complex constraints) is focused on a single execution. Moreover, generating and validating multiple candidate schedules can be done in parallel, which theoretically scales CLAP to programs with arbitrary execution length when there are sufficient computation cores.

Experimental results show that CLAP is effective in reproducing concurrency bugs, incurring only 10%∼3X runtime overhead on real-world systems with intensive shared-memory dependencies. Additionally, the computed schedules by CLAP typically contain less than three preemptive thread context switches, which is much easier to reason about for diagnosing the bug.

3.2 Concurrency Bug Detection

It is always preferable to detect software defects early. Predictive trace analysis (PTA)[14-17] is such a powerful technique that can predict concurrency bugs from normal executions. Generally speaking, a PTA technique records a trace of execution events, statically (often exhaustively) generates other permutations of these events under certain scheduling constraints, and exposes concurrency bugs unseen in the recorded execution. Compared with dynamic analysis, it is capable of exposing bugs in unexercised executions and, compared with static analysis, it incurs much fewer false positives because its analysis is based on the dynamically collected traces. Moreover, when used for software assurance, it is able to prevent software failures by predicting and fixing them before they occur.

PECAN[14] is a PTA system that predicts concurrency access anomalies such as data races and serializability violations in Java programs, from normal executions, with the appealing feature that it can not only predict those Heisenbugs, but also generate concrete executions that deterministically expose the bugs. With PECAN, developers are provided with the full execution history and context information to understand the bug, which dramatically expedites the debugging process. PECAN has revealed several serious and previously unknown bugs in large open source concurrent systems.

PTA in general faces considerable challenges in scaling to large traces because of the state space explosion problem and the algorithmic complexity of detecting concurrency errors. Nevertheless, often a large percentage of events in the trace are redundant for presenting useful analysis results to the end user, and those redundant events can be safely removed without affecting the trace analysis results. TraceFilter[15] is a system that automatically removes such redundant events for detecting concurrency access anomalies based on a trace redundancy theorem, while guaranteeing the redundancy-removed trace produces the same analysis result as that from the original trace. TraceFilter improved the scalability of PTA by orders of magnitude in many real-world concurrent systems.

Data races are among the most common bugs in modern concurrent software. Although there exist numerous race detection tools, all techniques suffer from either the unsoundness (we mean that the techniques may produce false alarms) or the incompleteness (miss real races). RVPredict[16] is a sound dynamic race detection technique that is provably complete with respect to the observed dynamic execution trace, i.e., it never misses any data race that can be found by other sound techniques based on the same trace. RVPredict is underpinned by a sound and maximal causal model

with the inclusion of a novel control-flow event, and leverages existing advancements of theorem provers and decision procedures to explore the causal traces and prove the existence of data races. RVPredict has been shown to detect significantly more races than previous techniques, and it scales to executions of real-world concurrent applications with tens of millions of critical events. RVPredict has also revealed many previously unknown races in real systems (e.g., Eclipse) and has been adopted by Eclipse developers.

In addition to detecting data races and access anomalies, we have also developed a system GPredict[17] that further generalizes PTA to handle high-level generic concurrency property violations. GPredict allows the users to define arbitrary ordering properties over the execution events, such as, "a resource must be authenticated before use" and "a collection cannot be modified if it is being iterated over", with a pattern specification language, and to predict violations of the specified properties from normal executions. By uniformly formulating the property violations and a sound causal model as first-order logic constraints, GPredict is able to predict all property violations captured by the causal model using off-the-shelf SMT solvers.

3.3 Concurrency Bug Understanding

To address the difficulty of diagnosing concurrency bugs on a reproducible buggy trace, we have developed static and dynamic trace simplification techniques[18-19] that effectively reduce the complexity of the buggy trace and shorten the replay time without losing the determinism in reproducing concurrency bugs. A simplified trace with smaller size and fewer context switches greatly lessens the debugging effort by prolonging the sequential reasoning of concurrent program execution and reducing the number of places in the trace where we need to look for the cause of the bug. We next describe the key ideas and characteristics of our techniques SimTrace[18] and LEAN[19]. We refer our readers to the full papers[18-19] for detailed examples.

SimTrace[18] is a static technique that dramatically improves the efficiency of trace simplification through reasoning about the computation equivalence of traces offline. By constructing a dependence graph that encodes all the dependence relations between events in the trace, SimTrace reduces the trace simplification problem to a graph merging problem, of which the objective is to minimize the graph size. By merging consecutive nodes from the same thread that have no inter-thread dependencies, and by performing a topological sort on the reduced dependence graph, SimTrace generates a simplified trace with much fewer thread context switches compared with the original trace. Moreover, SimTrace scales linearly in the trace size and quadratic in the dependence graph size, making it attractive for practical use with traces containing millions of critical events.

LEAN[19] is a dynamic trace simplification technique that, building on top of a record and replay system, significantly reduces the complexity of the buggy replay trace and speeds up the replay process without losing the replay determinism. Based on a redundancy criterion that characterizes the redundant computation in a buggy trace, LEAN is able to simplify the buggy trace beyond data and control dependencies by effectively identifying and removing redundant threads and instructions that are not essential for understanding the bug. On several large systems, LEAN reduces the number of threads and thread context switches in the trace by 90%, and shortens the size of the replay trace and the length of replay time by as large as 300x.

3.4 Concurrency Bug Fixing

Recent research has developed a few effective concurrency bug fixing approaches[20-22] through inserting proper synchronizations that eliminate erroneous interleavings. Nevertheless, synchronizations are neither necessary nor sufficient for fixing concurrency bugs. In practice, many concurrency bugs are fixed through privatization, i.e., changing thread-shared data to private data. In Privateer[23], we observe and also formally prove that for a vast category of concurrent programs, called scheduler-oblivious programs, whose computation result is expected to be always deterministic regardless of the thread scheduling, it is safe and theoretically sound to privatize a subset of shared data accesses for repairing concurrency errors without using synchronization. Leveraging the privatizability property, Privateer is able to soundly eliminate false and unnecessary data sharing in concurrent programs that are expected to be scheduler-oblivious, and isolate all the potential erroneous interleavings on the privatized data without adding any synchronization. Privateer has fixed several real concurrency bugs without impairing the execution parallelism. Moreover, for a few benchmarks, Privateer improves the program performance by as large as 12%, because after privatization the heap accesses become local stack operations.

A key component missing in most bug fixing techniques is that, except repeated testing, there exists no systematic way to validate the correctness of the fixes. However, for a fix to be correct, not only the observed buggy interleaving, but all the other buggy interleavings should be eliminated. SWAN[24] is a technique that helps in validating the fixes of atomicity violations through constraint solving and replay. It encodes the buggy trace into a set of constraints according to a maximal causal model (MCM)[16], generates additional events for the fixes, such as lock acquire and release, and formulates them as additional constraints. If the additional constraints conjoined with the MCM constraints cannot be satisfied, then the bug is fixed for at least a large set of interleavings captured by MCM. However, if there still exists any solution to the new formed constraints, then the fix is incorrect. Moreover, a buggy execution can be created to demonstrate the incorrect fix by decoding the buggy schedule from the solution and replaying it in the re-execution.

4 Other Important Directions

Stateless Model Checking. An alternative way that can eliminate concurrency bugs is to exhaustively verify the interleaving space with model checking. Stateless model checking (SMC) is a technique that explores the state-space systematically by driving concrete program executions via a dynamic scheduler without storing any states. Since the pioneering work of VeriSoft[25] and CHESS[26], SMC has been successfully applied in real-world programs and found many deep bugs through optimization techniques such as partial order reduction (POR)[27] and context (or preemption) bounding[26]. A key challenge in SMC is how to avoid redundant explorations of the same program state. Maximal causality reduction (MCR) is a recent technique that achieves a significant advance in SMC over POR and context bounding. MCR takes into account the value of reads and writes and exploits the maximal causality between redundant executions that lead to equivalent states by encoding them into first-order logical constraints. By solving the constraints together with new state-change constraints, MCR can generate new interleavings that drive the program to reach new program states. And by ensuring that in every explored execution there exists at least one read that reads a different value (from that in all the other executions), MCR minimizes the number of executions that are needed to explore for verifying concurrent programs.

Deterministic Multithreading. In contrast to detecting and fixing concurrency bugs under random schedules, deterministic execution techniques pioneered by DMP[28], DThreads[29], and Parrot[30] aim to make the execution deterministic by default, such that Heisenbugs either manifest themselves, or do not, on every execution. This approach if successful is promising to reduce the difficulty of concurrent software debugging since bugs become deterministic. A key challenge is how to minimize the performance degradation. Existing deterministic execution techniques usually incur large runtime overhead in order to enforce the determinism. Another practical challenge is how to identify those events that may introduce non-determinism. If those events are missed or not instrumented, deterministic execution may fail.

5 Conclusions

As long as the shared memory model continues to stay in the main stream for concurrent programming, the programmers will continue to demand high-quality techniques and tools to tackle the challenges in writing correct concurrent programs, understanding the errors they make, and tolerating the faults caused by these errors. Our effort was focusing on taming the interleaving complexities by making randomization deterministic, by the automated reasoning of interleaving behaviors, and by the semantic-preserving removal of interleavings. While many of our techniques are effective, there are still many obstacles caused by the performance requirement and the heterogeneity of today's complex systems. The road to bring these techniques to practice is still long. We are optimistic about that with advances of hardware-based debugging utilties and static program analysis, we can fully unleash the potential of these ideas and significantly improve the power of these tools to make programming concurrent programs more productive.

Acknowledgment We thank the anonymous JCST reviewers for their comments on an initial version of this paper.

References

[1] Britton T, Jeng L, Carver G *et al.* Reversible debugging software. Technical Report, Judge Business School, University of Cambridge, 2013.

[2] Guo P, Zimmermann T, Nagappan N *et al.* Characterizing and predicting which bugs get fixed: An empirical study of

Microsoft windows. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp.495-504.
[3] Gray J. Why do computers stop and what can be done about it? In *Proc. the 5th Symp. Reliability in Distributed Software and Database Systems*, Jan. 1986, pp.3-12.
[4] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 1979, 28(9): 690-691.
[5] Weaver D, Germond T (eds.). The SPARC Architecture Manual, Version 9. SPARC International, Inc., 1994.
[6] Lu S, Park S, Seo E *et al*. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. the 13th ASPLO*, Mar. 2008, pp.329-339.
[7] Yin Z, Yuan D, Zhou Y *et al*. How do fixes become bugs? In *Proc. the 19th ACM SIGSOFT Symp. the Foundations of Software Engineering and the 13th European Software Engineering Conference*, Sept. 2011, pp.26-36.
[8] Huang J, Liu P, Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proc. the 18th ACM SIGSOFT FSE*, Nov. 2010, pp.207-216.
[9] Huang J, Liu P, Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proc. the 18th ACM SIGSOFT FSE*, Nov. 2010, pp.385-386.
[10] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. In *Proc. ACM PLDI*, June 2013, pp.141-152.
[11] Ball T, Larus J R. Efficient path profiling. In *Proc. the 29th IEEE/ACM MICRO*, Dec. 1996, pp.46-57.
[12] de Moura L M, Bjørner N. Z3: An efficient SMT solver. In *Proc. the 14th TACAS*, March 29-April 6, 2008, pp.337-340.
[13] Vaswani K, Thazhuthaveetil M J, Srikant Y N. A programmable hardware path profiler. In *Proc. the 3rd IEEE/ACM CGO*, Mar. 2005, pp.217-228.
[14] Huang J, Zhang C. PECAN: Persuasive prediction of concurrency access anomalies. In *Proc. the 20th ISSTA*, July 2011, pp.144-154.
[15] Huang J, Zhou J, Zhang C. Scaling predictive analysis of concurrent programs by removing trace redundancy. *ACM Trans. Softw. Eng. Methodol.*, 2013, 22(1): Article No. 8.
[16] Huang J, Meredith P O, Rosu G. Maximal sound predictive race detection with control flow abstraction. In *Proc. ACM PLDI*, June 2014.
[17] Huang J, Luo Q, Rosu G. GPredict: Generic predictive concurrency analysis. In *Proc. the 37th ICSE*, May 2015, pp.847-857.
[18] Huang J, Zhang C. An efficient static trace simplification technique for debugging concurrent programs. In *Proc. the 18th SAS*, Sept. 2011, pp.163-179.
[19] Huang J, Zhang C. LEAN: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proc. the 27th OOPSLA*, Oct. 2012, pp.451-466.
[20] Jin G, Song L, Zhang W *et al*. Automated atomicity-violation fixing. In *Proc. PLDI*, June 2011, pp.389-400.
[21] Jin G, Zhang W, Deng D. Automated concurrency-bug fixing. In *Proc. the 10th OSDI*, Oct. 2012, pp.221-236.
[22] Liu P, Zhang C. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proc. the 34th ICSE*, June 2012, pp.299-309.
[23] Huang J, Zhang C. Execution privatization for scheduler-oblivious concurrent programs. In *Proc. OOPSLA*, Oct. 2012, pp.737-752.
[24] Shi Q, Huang J, Chen Z *et al*. Verifying synchronization for atomicity violations. *IEEE Trans. Software Eng.*, 2016, 42(3): 280-296.
[25] Godefroid P. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 2005, 26(2): 77-101.
[26] Musuvathi M, Qadeer S, Ball T *et al*. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. the 8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008, pp.267-280.
[27] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In *Proc. the 32nd ACM POPL*, Jan. 2005, pp.110-121.
[28] Devietti J, Lucia B, Ceze L *et al*. DMP: Deterministic shared memory multi-processing. In *Proc. the 14th ASPLO*, Mar. 2009, pp.85-96.
[29] Liu T, Curtsinger C, Berger E D. Dthreads: Efficient deterministic multithreading. In *Proc. the 33rd ACM SOSP*, Oct. 2011, pp.327-336.
[30] Cui H, Simsa J, Lin Y H *et al*. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proc. the 24th ACM SOSP*, Nov. 2013, pp.388-405.

Jeff Huang received his Ph.D. degree in computer science from Hong Kong University of Science and Technology, Hong Kong, and is currently an assistant professor in the Department of Computer Science and Engineering at Texas A&M University, College Station. His research focuses on developing techniques and tools for improving software performance and reliability based on fundamental program analyses and programming language theory. He has published at premium conferences and journals such as PLDI, OOPSLA, ICSE, FSE, IEEE TSE and ACM TOSEM. His research has won awards including ACM SIGSOFT Outstanding Dissertation Award, SIGPLAN PLDI Distinguished Paper Award, SIGPLAN Research Highlights, Google Faculty Research Award, and NSF CAREER Award.

Charles Zhang is an associate professor and director of the Cybersecurity Lab in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His major research interest is the use of program analysis techniques to improve software quality. He has published extensively at premium conferences and journals of programming languages and software engineering. He has served on many organizational and technical committees of international conferences. He is currently an associate editor of IEEE TSE. His research received many awards including PLDI Distinguished Paper Award, ACM SIGSOFT Doctoral Dissertation Award, and IBM Ph.D. fellowships. His research is supported by Research Grant Council, Innovation and Technology Fund, and grants from Microsoft and IBM. Charles obtained his Ph.D., M.S., and B.S. degrees with honours, all in computer science from University of Toronto.