Research Statement – Heming Cui

Real-world programs are undergoing paradigm shifts, and these shifts introduce reliability and security problems. One paradigm shift is that programs are moving from single-threaded to multi-threaded. However, multi-threaded programs are notoriously hard to get right, and a key reason is that these programs have *too many* possible thread interleavings, which aggravates understanding, testing, debugging, and can lead to wrong outputs and security breaches. Another paradigm shift is that programs are becoming larger and more complicated, while still having to obey critical rules (e.g., allocated memory must be freed, and file updating and disk syncing must be done consistently), as violating these rules can cause resource leaks and data losses. Unfortunately, existing techniques can not feasibly check these rules on real-world programs due to program path explosion. My research is focused on creating effective systems to attack these reliability and security problems.

My approach is that, starting from a fresh and fundamental insight to a critical problem, I search for an effective solution combined from multiple fields such as systems and program analysis. A fresh and fundamental insight helps me understand the key challenge and steer my direction, and a combined solution enables me to leverage the strengths of different fields, such as the practicality of systems skills and rigor of program analysis techniques. This approach has helped me achieve important research advances: §1 describes three systems that can greatly improve reliability of multi-threaded programs by reducing the number of thread interleavings, §2 introduces compelling applications of these systems on advancing two research fields called static analysis and model checking, and §3 presents results on detecting new security violations in widely used programs.

1 Stable Multi-threading (StableMT)

My primary research is focused on improving reliability of multi-threaded programs by reducing the number of thread interleavings in these programs. To realize this technique, I have built three systems by collaborating with colleagues from Columbia University and Carnegie Mellon University.

Multi-threaded programs have become pervasive due to the accelerating computational demand and the rise of multi-core hardware. Unfortunately, despite much research and engineering effort, these programs are still notoriously difficult to get right, leading to severe problems including wrong outputs, program crashes, and security breaches. Our research [5] reveals that a key reason of this difficulty is that multi-threaded programs have *too many* possible thread interleavings, or schedules. Even given only a single input, a program may run into excessive schedules, depending on factors such as hardware timing and OS scheduling. Considering all inputs, the number of schedules is even much greater. Finding a buggy schedule in this huge schedule set is like finding a needle in a haystack, which aggravates understanding, testing, and analyzing of programs. For instance, testing is ineffective because the schedules tested in the lab may not be the ones run in the field.

To reduce the number of schedules for all inputs, we have studied the relation between inputs and schedules of real-world programs, and made a surprising discovery: many programs require only a small set of schedules to efficiently process a wide range of inputs. Leveraging this discovery, we have proposed the idea of stable multi-threading (StableMT) [5] that reuses each schedule on a wide range of inputs. StableMT conceptually maps all inputs to a greatly reduced set of schedules, drastically shrinking the "haystack", making the "needles" much easier to find. StableMT can greatly benefit understanding multi-threaded programs and many reliability techniques, including testing, debugging, replication, and verification. For instance, testing schedules in such a much smaller schedule set becomes a lot more effective.

StableMT is complementary to another idea called deterministic multi-threading (DMT) that enforces the same schedule on each input. Our research [5] has shown that although DMT is useful, it is not as useful as commonly perceived, because a DMT system can map slightly changed inputs to very different schedules, which can easily cause very different program behaviors. Furthermore, the number of schedules for all inputs in DMT can still be extremely large, which is the key problem that StableMT mitigates. Notably, StableMT has attracted

the research community's interest since it was initially proposed by us in 2010 [3], and most subsequent systems are both StableMT and DMT.

To realize StableMT, we have built three systems, TERN [3], PEREGRINE [2], and PARROT [1], with each addressing a distinct challenge. Unless specified, these systems use a common definition for "schedule": a total order of synchronization operations (e.g., lock acquisitions).

The first challenge of implementing StableMT is how to find highly reusable schedules for different inputs. The more reusable a schedule is, the fewer schedules are needed. However, finding highly reusable schedules is hard with existing static or dynamic techniques, because statically computed schedules are in general not guaranteed to work at runtime due to the halting problem, and dynamically computing schedules may be slow.

TERN [3], our first StableMT system, addresses the schedule-finding challenge by proposing a technique that records a set of past, working schedules, and then reuses these schedules on future inputs when possible. This technique is inspired by a real-world analogy that human and animals tend to migrate along past, familiar routes and avoid possible hazards in unknown ones. To find a schedule suitable for an input, TERN leverages a set of advanced program analysis techniques to compute preconditions on inputs that match a schedule. Evaluation on a diverse set of popular programs showed that TERN can reuse a small set of schedules to process a wide range of inputs. For instance, just 100 schedules for the `Apache` web server can process 90.3% of a 4-day trace (122K requests) from the Columbia CS department website.

The second challenge of implementing StableMT is how to efficiently make executions follow schedules without deviating. This challenge has existed in the area of deterministic execution and replay for decades. Previous work typically enforces two types of schedules: a total order of shared memory accesses (mem-schedule), and a total order of synchronization operations (sync-schedule). The mem-schedules are fully deterministic even with data races, but they are several times slower than traditional multi-threading. The sync-schedules incur only modest overhead because most code is not synchronization and thus can still run in parallel, but these schedules may deviate if there are data races. Overall, despite much research effort, people can only choose either full determinism or efficiency, but not both.

To tackle this challenge, our second StableMT system, PEREGRINE [2], leverages the following observation: although data races exist in some programs, the races tend to occur only within minor portions of an execution, and the majority of the execution is still race-free. Therefore, we can enforce a sync-schedule in the race-free portions of an execution and resort to a mem-schedule only in the racy portions, combining both the efficiency of sync-schedules and determinism of mem-schedules. PEREGRINE implements hybrid-schedule with a new technique that first records an execution trace of all executed instructions on a new input, and then relaxes the trace into a highly reusable hybrid-schedule. Evaluation on a diverse set of programs showed that PEREGRINE provides both determinism and efficiency, and can frequently reuse schedules for half of the evaluated programs. PEREGRINE has been featured in media such as `ACM Tech News`, `TG Daily`, and `Physorg`.

In the last four years, StableMT has achieved promising advances and attracted the research community's interest. Numerous StableMT systems have been built, including our TERN and PEREGRINE systems. However, it remains an open challenge whether StableMT can be made simple and deployable. Previous StableMT systems either run into slow schedules that *serialize* parallel computation, or are fairly hard to deploy due to their high complexity (e.g., TERN and PEREGRINE require sophisticated program analysis).

To address this challenge, we have built our third and latest StableMT system, PARROT [1], a simple, deployable runtime that enforces a well-defined round-robin schedule for synchronization operations, vastly reducing the number of schedules. To mitigate the serialization problem that causes big slow-down, PARROT uses an insight based on the 80-20 rule: most threads spend most execution time in only a few core computations, and PARROT only needs to make these core computations parallel. Accordingly, PARROT provides a new abstraction called *performance hints* for developers to annotate core computations. These hints, which are intended to improve parallelism of core computations, are not real synchronization, and can be safely ignored without

affecting correctness of a program. Evaluation on a wide range of 108 popular programs (e.g., `Berkeley DB` and `MPlayer`), which is roughly 10× more programs than any previous StableMT or DMT evaluation, showed that, these hints were easy to add and made PARROT fast (12.7% mean overhead on 24-core machines). Due to PARROT's simplicity and high practicality, we have made it open source for deployment.

2 Applications of StableMT

To demonstrate the potential of StableMT, we have applied PEREGRINE [2] to improve static analysis, a popular technique that analyzes a program and gets high coverage (e.g., covers all possible schedules) without executing code. One shortcoming of static analysis is that it suffers from poor precision: to get high coverage without executing code, static analysis has to over-approximate the huge schedule set and includes many schedules that will never occur. Therefore, static analysis often raises excessive false reports from the impossible schedules, which buries real bugs in noise. Fortunately, StableMT can drastically shrink the schedule set for static analysis. We have created a static analysis framework [6] that uses PEREGRINE to greatly reduce the number of possible schedules. This reduction enables the framework to soundly avoid most false reports without missing bugs, and to detect seven new harmful data races in heavily studied programs.

In addition, PARROT [1] has demonstrated that StableMT can greatly advance model checking, a dynamic testing technique that systematically explores schedules for errors on an input. Although industry and academia have made model checking effectively find errors in real-world programs, this technique can check only a very tiny fraction of schedules even after checking for thousands of years, as the number of possible schedules can be very large. Because StableMT vastly reduces the number of possible schedules, with it model checking enjoys much higher coverage (i.e., the ratio of checked schedules over all possible schedules). We have integrated PARROT with a state-of-the-art model checking tool called DBUG. Evaluation on 61 popular programs containing large schedule set showed that, PARROT improved DBUG's coverage by many orders of magnitude for 56 programs, reducing the schedule set in most of these 56 programs to only *one* schedule under our input setup.

3 Detecting Reliability and Security Rule Violations

Apart from StableMT, my research also involves checking reliability and security rule violations. Programs must obey many rules, such as assertions must succeed, allocated memory must be freed, and file updating and disk syncing must be done consistently. It is crucial to verify programs with these rules, because violating them can easily lead to critical failures such as program crashes, resource leaks, data losses, and security holes. Unfortunately, existing techniques can not precisely verify a program with high program path coverage. For example, although symbolic execution, a popular program analysis technique, can systematically explore program paths to find errors, this technique suffers from path explosion: it can rarely explore however a tiny portion of program paths, because a typical program is complicated and contains exponentially many paths. This poor path coverage makes rule violations extremely hard to check in real-world programs.

To address this problem, we propose WOODPECKER [4], a *rule-directed* symbolic execution system. The insight behind WOODPECKER is: only a small portion of paths are relevant to rules, and the rest (majority) of paths are irrelevant and do not need to be verified. WOODPECKER directs symbolic execution towards the program paths that are relevant to a built-in rule, and soundly prunes redundant paths without missing rule violations. Evaluation on 136 widely used programs showed that WOODPECKER effectively verified 28.7% of program and rule combinations, whereas a top-of-the-line symbolic execution system verified only 8.5%. In addition, WOODPECKER detected a total of ten serious data loss or security violations in `git`, `CVS`, and `useradd`. These violations can corrupt repositories or hack OS user accounts, and most of them have been quickly confirmed or fixed by the corresponding developers.

4 Future Plans

In the short term, I plan to explore the potential of StableMT to benefit other reliability fields such as concurrency error detection, debugging, verification, and state machine replication (SMR). For instance, SMR has become a dominant fault-tolerance technique in distributed systems, but SMR for multi-threaded programs, which is driven by the accelerating computational demand, is still difficult. A root cause of this difficulty is the execution divergence among replicas caused by the enormous number of thread schedules and the nondeterministic arrival timings of requests. I plan to attack this root cause by integrating StableMT with PAXOS: StableMT drastically reduces the number of schedules, and PAXOS coordinates the order of arriving requests among replicas as well as the deterministic logical timings of these requests in StableMT's schedules. By doing this, multi-threaded SMR may be made feasible.

In the medium term, I plan to design and implement a unified reliable and secure framework for parallel and distributed computing. Firstly, this framework requires a new language and synchronization primitive for developers to express the tradeoff between the amount of possible schedules and performance. Secondly, it requires a new program analysis engine that can conveniently plug in dynamic checking algorithms to analyze the whole system without interrupting normal executions. Last but not least, it requires a new runtime that can enforce developers' intention from the new language, and can automatically deploy reliability and security services including replication and auditing. Realizing this framework will encourage collaborations among diverse fields such as systems, programming languages, networking, algorithms, and security. I believe these exciting collaborations can realize a common goal: making programs reliable and secure.

In the long term, I am willing to attack a broad range of critical problems caused by the paradigm shifts of computing. For instance, I would like to explore how datacenters reshape existing software. If a datacenter continues to become a "big computer", how do we design the process and thread models in this new computer? Can we leverage some StableMT techniques to ensure good reliability for these future processes and threads? How do we reconstruct TCP and UDP to improve throughput for the aggregated workloads within a datacenter? Shall we reexamine existing distributed protocols within this approximately synchronous network? I think each of these questions can raise fundamental problems, and I look forward to applying all my enthusiasm, knowledge, and skills to fight against these problems.

References

[1] **Heming Cui**, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth Gibson, and Randy Bryant. "Parrot: a Practical Runtime for Deterministic, Stable, and Reliable Threads". Proceedings of *SOSP* 2013.

[2] **Heming Cui**, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. "Efficient Deterministic Multithreading through Schedule Relaxation". Proceedings of *SOSP* 2011.

[3] **Heming Cui**, Jingyue Wu, Chia-che Tsai, and Junfeng Yang. "Stable Deterministic Multithreading through Schedule Memoization". Proceedings of *OSDI* 2010.

[4] **Heming Cui**, Gang Hu, Jingyue Wu, and Junfeng Yang. "Verifying Systems Rules Using Rule-Directed Symbolic Execution". Proceedings of *ASPLOS* 2013.

[5] Junfeng Yang, **Heming Cui**, Jingyue Wu, Yang Tang, and Gang Hu. "Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading". In *Communications of ACM* 2014.

[6] Jingyue Wu, Yang Tang, Gang Hu, **Heming Cui**, and Junfeng Yang. "Sound and Precise Analysis of Parallel Programs through Schedule Specialization". Proceedings of *PLDI* 2012.