Research Statement

Xiang (Jenny) Ren

Performance makes or breaks a software system. Severe performance issues render systems unusable; Even small performance degradation can be costly— Google finds that a 0.5-second delay in page load time caused a 20% drop in repeat traffic. [15] Despite its importance, there are many obstacles to building performant systems. Modern software systems have complex performance characteristics that are difficult to understand, and users and developers still invest significant time manually diagnosing critical performance issues.

The goal of my research is to help developers build performant software systems. My approach is to understand the system performance characteristics comprehensively, then examine where they break existing assumptions and identify opportunities to create automated techniques to improve performance.

I started by studying how the performance of Linux's core functionalities evolved [16]. The study discovered that the performance of many core kernel functionalities worsens or fluctuates significantly over time, due to adding new features and security guarantees, as well as misconfigurations. Despite unavoidable trade-offs between performance and functionality or security, I found much of the slowdown can be improved through proactive testing, diagnosis and optimization, or avoided through custom kernel configuration. The findings sparked many discussions on Hacker News and Reddit, and were featured by the *Morning Paper*. The benchmark designed for the study was deployed at Amazon.

The previous study motivated me to create an automated tool to diagnose performance root causes. I found that existing root cause diagnosis tools are *not* built for performance bugs—they check for broken correctness guarantees, which should hold *regardless* of the inputs; however, the expected performance outcome can only be determined *relative* to the inputs. Based on this observation, I invented relational debugging. Inspired by the idea of choosing the right frames of observation in the physical world, relational debugging chooses the appropriate fine-grained events as the basis of analysis, and captures performance root causes in terms of occurrences relative to these events, independent of the effects of changing inputs. We implemented relational debugging with Perspect [17], which is able to automatically and efficiently pinpoint the root causes of complex performance bugs, such as Go-909 [1], which took developers a year to diagnose through a trial-and-error process. I also used Perspect to diagnose two open bugs (MongoDB-57221 [5], MongoDB-56274 [6]) whose root causes were confirmed by MongoDB developers. Perspect is requested by developers of the MMTk memory management framework and has attract interest from CockroachDB.

1 Current Research

A study of the performance evolution of Linux's Core Operations. I carried out a study to understand the performance of Linux's core operations, such as system calls and context switches, as the system evolved for seven and half years [16]. Surprisingly, the performance of many core operations has worsened or fluctuated significantly over time. I found that the need to accommodate an increasing number of new features and security enhancements contributed to the increased overhead of these core operations. Further, simple misconfigurations have also caused significant performance fluctuation. I further attributed the causes of these slowdowns to 11 changes. By disabling these root causes, I was able to speed up Redis, Apache, and Nginx benchmark workloads by as much as 56%, 33%, and 34%, respectively. These findings point to the feasibility and importance for Linux users to custom configure their kernels balance between performance, functionality and security.

Moreover, I found performance costs are paid in addition to the required trade-off for increasing security and offering new features. Out of the 11 root causes, one was optimized later by Linux developers, and we found further optimizations for two; Linux/Ubuntu developers eventually fixed three misconfigurations, and we found better configurations for two. Indeed, better performance can be gained from proactive testing, diagnosis, and optimization.

Perspect: Automating the diagnosis of performance root causes. After manually diagnosing the root causes of performance changes in the previous study, I noticed a lack of tools that automate performance diagnosis: Profilers identify performance symptoms, such as hotspots, but do not link them to the root causes. Existing root cause diagnosis tools [11, 14, 18, 19, 21] are also ineffective for performance bugs. Such tools check for broken correctness guarantees to identify the presence of bugs. For example, if the program crashes at line n, then bug x has been triggered; Otherwise, it has not. The tools then compare and analyze buggy and non-buggy executions or inputs to identify the root cause. On the other hand, performance bugs break this type of design because the expected performance must be determined *in relation to* input workloads. For example, a program may have consumed 10GB of memory because it has a memory leak or simply because it has processed more requests. Existing tools would treat both outcomes as equivalent.

I created relational debugging [17], which is able to effectively pinpoint the root causes of complex performance issues, because it captures violated performance expectations regardless of the input workload. Instead of comparing the outcomes of entire executions like existing root cause diagnosis tools, relational debugging breaks down an execution and identifies the appropriate fine-grained events to serve as the basis for comparison and analysis. Specifically, it represents a root cause with a relation, which is the number of times an event occurred relative to another causally related event. If a relation changes abnormally across executions, it is a potential root cause. For example, a severe memory leak causes the number of objects freed relative to the number of objects allocated to become much lower than 1. Secondly, relational debugging pinpoints a small number of abnormal relations as root cause candidates. The algorithm works similarly to the process of choosing the right frame of observation in the physical world: If a passenger appears to move faster in the train, but the train is still moving at the same speed, then the cause of the acceleration is from the passenger's own motion. Likewise, if an abnormal change in relation $r1$ is explained by another change in relation $r2$, then $r1$ is discarded as a root cause candidate. In practice, this algorithm scales well in real-world systems, and is able to capture the root cause in 8 minutes for average on most cases we evaluated.

Perspect can diagnose the root causes of some of the most complex real-world performance bugs that were previously impossible to diagnose with existing root cause diagnosis tools, including Go-909 [1]. I used Perspect to diagnose two open bugs MongoDB-57221 [5], 56274 [6], where the root causes were confirmed by the developers.

Improving the reliability of distributed systems. I have also worked on projects aimed at improving the reliability of distributed systems, including Pensieve [20], which automatically constructs near-minimal reproductions of failures with the scalable event-chaining approach. Subsequently, during my internship at Microsoft Research, I applied the event-chaining algorithm to automatically reproduce flaky tests that are caused by concurrency [12]. The prototype is able to reproduce 26 out of 31 flaky tests in production codebases in under 6 minutes on average.

Building performant systems for emerging hardware. I have also worked on building novel file systems for persistent memory [13]. ctFS removes file indexing overhead and directly manages persistent memory using the hardware memory management unit by allocating files as contiguous virtual memory chunks.

2 Future Research

For future work, I plan to take a multi-pronged approach to improving system performance. (1) I'm interested in studying the performance of emerging workloads, including serverless computing and machine learning workloads, etc. I'm also interested in uncovering unknown performance phenomena in system software in general, including database software, distributed systems, operating systems, etc. (section 2.1). Through the studies, I plan to re-examine fundamental assumptions around system performance and identify opportunities for better system design or automated techniques that enhance performance testing, diagnosis, or optimization. (2) I plan to build more observable software systems to enhance performance diagnosability (section 2.2) and (3) create novel techniques that form a pipeline that automates performance bug detection, diagnosis, and fixes (section 2.3). My vision is to create a self-adaptive system that detects performance issues and creates patches or reconfigures the system automatically.

2.1 Uncovering and studying new performance phenomena

Understanding performance creep. I'm currently carrying out a comprehensive study of performance creep. Performance creep refers to gradual performance degradation that is allowed to persist over time; however, its impact is often significant in the long run. For example, A basic `SELECT` workload in MySql [9] experienced a 34% regression from version 4.1.22 to 5.7.2 [2]. We find creep also causes significant regression over time in popular systems, including MongoDB [8] and GCC [7]. Unfortunately, the causes of creep remain largely mysterious. The main reason is that performance creep is very challenging to diagnose, as creep consists of fine-grained changes, each of which is very short-running and easily confused with background noise from the OS, etc.; Existing profiling strategies are inadequate as they are too coarse-grained or have too much overhead. I plan to develop novel approaches to measure small performance differences accurately, including execution noise reduction and high-precision profiling. Subsequently, I plan to comprehensively study the root causes of creep and identify opportunities to optimize performance. Ultimately, my goal is to enable creep to be automatically detected, diagnosed, and optimized.

2.2 Enhancing performance diagnosability by building more observable systems

Capturing performance outliers root causes in situ (across software components). Performance outliers are highly impactful yet very elusive. They are hard to reproduce and hard to catch using conventional profilers that aggregate execution, such as `perf` [10]. Further, built-in checks for abnormal program conditions, such as asserts or error logging, are also limited because they only work for *absolute* conditions. Since expected performance is relative to input workloads, whether the execution is expected must be considered in relation to inputs. Similar to relational debugging [17], I propose relational asserts, which check for the expected execution in relation to fine-grained program events and hold regardless of input workloads. I plan to design an automated technique that "learns" what to assert relationally by observing normal executions.

Moreover, it is often hard for different software components to agree on the same set of performance expectations. Indeed, Go-8832 [4], was famously hard to diagnose because the root cause of the memory bloat lies *not* in the Go application itself but in the OS, which excessively promoted huge pages against the application's "wishes". With relational asserts systematically inserted into an application, I plan to design solutions to automatically translate these into corresponding performance expectations in different software components. For example, in Go-8832, relational asserts in the go runtime can be translated into asserts in the OS about how aggressively to promote huge pages.

Embedding execution information into built-in performance metrics. Real-world software systems have large collections of built-in performance metrics to aid in debugging. For example, the go runtime has 70 metrics [3]. Developers often look for abnormal performance metrics when diagnosing performance issues, but performance metrics alone have limited explanatory power. For example, a developer might observe an abnormal `heap_size` but has no further information on what caused it. Manually diagnosing this issue with very little lead can be extremely time-consuming, like in the case of Go-909 [1]. I plan to design informative performance metrics that maximally encode relevant diagnostic information, such as variable values and execution paths while causing minimal execution overhead.

2.3 Building a pipeline of performance bug detection & diagnosis & patch generation

Detecting performance bugs before they cause severe consequences. My previous work [17] found that for many of the performance bugs, the root causes are triggered in "healthy" runs, too, only less often. For example, the memory leak bug from Go-909 [1] got triggered a small number of times in the good run as well. For future work, I plan to design an automated tool to detect such latent root causes and identify input workloads that can cause latent root causes to manifest into performance issues.

Extending relational debugging technique to diverse use cases. Relational debugging is currently implemented with Perspect [17], which pinpoints the root causes of performance issues offline in a single-machine setting. I plan to apply relational debugging to more diverse scenarios: I plan to build an online root cause diagnosis tool that carries out relational debugging incrementally to achieve low overhead. Secondly, I plan to adapt relational debugging to work on distributed systems as well as the Linux kernel.

Coupling performance diagnosis with patch generation. I find that well-known patterns of non-optimal performance — such as lack of caching, redundant work, sub-optimal configurations, etc. — correlate with different types of root causes identified by Perspect [17]. For future work, I'm interested in creating a pipeline where the performance root causes diagnosed by Perspect are then used as feedback to automatically generate patches in a principled manner.

References

[1] 2010. runtime: garbage collection ineffective on 32-bit 909. (2010). `https://github.com/golang/go/issues/909`

[2] 2013. [MDEV-5333] Single-threaded performance regressions - Jira. (2013). `https://jira.mariadb.org/browse/MDEV-5333`

[3] 2013. [runtime/metrics. (2013). `https://pkg.go.dev/runtime/metrics`

[4] 2014. runtime: GC freeing goroutines memory but then taking it again 8832. (2014). `https://github.com/golang/go/issues/8832`

[5] 2021. Inconsistent delete performance on collections with multiple secondary indexes on same key. (2021). `https://jira.mongodb.org/browse/SERVER-57221`

[6] 2021. TTL deletes are much slower on descending indexes than ascending indexes. (2021). `https://jira.mongodb.org/browse/SERVER-56274`

[7] 2023. GCC, the GNU Compiler Collection. (2023). `https://gcc.gnu.org/`

[8] 2023. MongoDB: The Developer Data Platform — MongoDB. (2023). `https://www.mongodb.com/`

[9] 2023. MySQL. (2023). `https://www.mysql.com/`

[10] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance Comprehension at WiredTiger. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 83–94. `https://doi.org/10.1145/3236024.3236081`

[11] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*.

[12] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. `https://doi.org/10.1145/3540250.3558956`

[13] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. 2022. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 35–50. `https://www.usenix.org/conference/fast22/presentation/li`

[14] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*.

[15] Greg Linden. 2017. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. (Nov. 2017).

[16] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 554–569. https://doi.org/10.1145/3341301.3359640

[17] Xiang (Jenny) Ren, Sitao Wang, Zhuqi Jin, David Lion, Adrian Chiu, Tianyin Xu, and Ding Yuan. 2023. Relational Debugging — Pinpointing Root Causes of Performance Problems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 65–80. https://www.usenix.org/conference/osdi23/presentation/ren

[18] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

[19] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[20] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3132747.3132768

[21] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 131–146. https://doi.org/10.1145/3341301.3359650