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 [16]. 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 [17]. The study discovered that the performance of many core kernel functionalities worsened or fluctuated 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 the paper was featured by the *Morning Paper* [5]. The benchmark designed for the study was deployed by Amazon.

The previous study motivated me to create an automated tool to diagnose the root causes of performance issues. I found that existing root cause diagnosis tools are *not* built for performance issues—they work for incorrect execution outcomes that are input independent, *e.g.*, crashes; However, the expected performance outcome is *relative* to inputs workloads. 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 [18]. Perspect is able to automatically and efficiently pinpoint the root causes of complex performance bugs. I used Perspect to diagnose Go-909 [1], which took developers a year to diagnose through a trial-and-error process. I also used Perspect to diagnose two challenging open bugs (MongoDB-57221 [6], MongoDB-56274 [7]), and the root causes reported by Perspect were confirmed by MongoDB developers. Perspect is requested by developers of the MMTk memory management framework and has attracted 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 [17]. Surprisingly, the performance of many core operations had worsened or fluctuated significantly over time. I found the increase in overhead was caused by the need to accommodate an increasing number of new features and security enhancements. Further, simple misconfigurations had also caused significant performance fluctuations. I attributed the causes of these fluctuations and 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.

Moreover, I identified performance costs paid in addition to those required to increase security and accommodate new features. Out of the 11 root causes, four can be optimized and five can be avoided through better configuration. I found optimizations for two root causes and better configurations for two more, In addition to those eventually carried out Linux/Ubuntu developers, These findings suggest that better performance can be achieved through proactive testing, diagnosis, and optimization, and custom configuring kernels can help achieve a balance between performance, functionality and security.

Perspect: Automating the diagnosis of the root causes of performance issues. While carrying out the previous study, I manually diagnosed the performance changes and found no readily available 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 [12, 15, 19, 21, 23] are ineffective for performance issues; They are designed for bugs that result in execution outcomes undesirable for any inputs, *e.g.*, crashes, whereas performance expectations are relative to inputs. For example, existing tools cannot automatically distinguish between an execution that is consuming excess memory because it is processing a large number of requests from an execution consuming excess memory that is processing few requests but experiencing memory leaks.

Motivated by the previous observations, I created relational debugging [18], which is able to effectively pinpoints the root causes of complex performance issues. While designing relational debugging, I drew inspiration from the analogy that one can choose many different points in the execution to compare performance, just like choosing different frames of reference to observe motion in the physical world. In the previous example, the memory leak issue can be exposed if one looks at the memory consumed at the per request level. According this idea, I designed relational debugging to identify the appropriate fine-grained events to serve as the basis for comparison and analysis. This design allows relational debugging to capture violated performance expectations, as well as the root causes, regardless of the input workload. For example, relational debugging captures the root cause of the previous example by examining number of objects freed relative to the number of objects allocated, which becomes much lower in the case of a severe memory leak. Specifically, relational debugging 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 considered a potential root cause. In addition, relational debugging pinpoints a small number of abnormally changed relations as root cause candidates by analyzing the program logic and differentiating cause and effect, for example, determining that the increase in memory consumed per request is due to the decrease in objects freed per object allocated.

We implemented relational debugging with Perspect. Perspect scales well in real-world systems, and is able to capture the root cause in 8 minutes on average for most of the cases we evaluated. Perspect is able to effectively diagnose the root causes of some of the most complex real-world performance bugs, such as Go-909 [1], which previous root cause diagnosis tools are ineffective against. I used Perspect to diagnose two open bugs (MongoDB-57221 [6] and 56274 [7]) which developers struggled to diagnose, and the root causes reported by Perspect 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 [22], which automatically constructs near-minimal reproductions of failures with the scalable event-chaining approach. During my internship at Microsoft Research, I applied the event-chaining algorithm to automatically reproduce flaky tests that are caused by concurrency [13]. 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 [14]. ctFS removes file indexing overhead and directly manages persistent memory using the hardware memory management unit (MMU) 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. 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 [10] 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 [9] and GCC [8]. 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 identify and develop techniques 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 the root causes of performance outliers in situ, and across software components. Performance outliers are impactful yet elusive. They are hard to reproduce and capture using profilers [11]. Further, built-in checks, such as asserts, are also limited because they only work for *absolute* conditions; When debugging performance issues, whether the execution is expected must be determined *relative* to input workloads. Similar to relational debugging [18], I propose *relational asserts*, which check for the expected execution relative to fine-grained program events and hold regardless of input workloads. I propose to automatically learns what to assert relationally by observing normal executions.

Moreover, different software components often cannot agree on the same set of performance expectations. For example, Go-8832 [4] was famously hard to diagnose because the memory bloat was *not* caused by the Go application but by the OS, which excessively promoted hugepages. With relational asserts systematically inserted into an application, I propose to automatically translate these asserts into different software components. For Go-8832, relational asserts on memory usage in the Go application can be translated into the OS to check the frequency of hugepage promotions.

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 performance 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 may have observed an abnormal `heap_size` but has no information on the execution that led to it; Further diagnosis with very little lead can be challenging, 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 [18] found that for many of the performance bugs, the root causes are also triggered in "healthy" runs, only less often. For example, the memory leak bug from Go-909 [1] also occurred in the good run. For future work, I plan to design an automated tool to detect such "hidden" root causes and identify input workloads that can cause them to manifest into performance issues.

Extending relational debugging to diverse performance diagnosis scenarios. Relational debugging is currently implemented with Perspect [18], which diagnoses the root causes of performance issues offline in a single-machine setting. 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. [20]—correlate with different types of root causes identified by Perspect [18]. 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] 2019. An analysis of performance evolution of Linux's core operations. (2019). https://blog.acolyer.org/2019/11/04/an-analysis-of-performance-evolution-of-linuxs-core-operations/

[6] 2021. Inconsistent delete performance on collections with multiple secondary indexes on same key. (2021). https://jira.mongodb.org/browse/SERVER-57221

[7] 2021. TTL deletes are much slower on descending indexes than ascending indexes. (2021). https://jira.mongodb.org/browse/SERVER-56274

[8] 2023. GCC, the GNU Compiler Collection. (2023). https://gcc.gnu.org/

[9] 2023. MongoDB: The Developer Data Platform — MongoDB. (2023). https://www.mongodb.com/

[10] 2023. MySQL. (2023). https://www.mysql.com/

[11] 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

[12] 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)*.

[13] 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

[14] 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

[15] 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)*.

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

[17] 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`

[18] 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`

[19] 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)*.

[20] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 421–426. `https://doi.org/10.1145/1882362.1882448`

[21] 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`

[22] 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`

[23] 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`