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 has caused a 20% drop in repeat traffic [18]. 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.

For my dissertation, I carried out a study [19] that discovered many of Linux's core kernel operations had slowed down over time or had unstable performance due to adding new features and security guarantees, as well as misconfigurations. Despite unavoidable performance trade-offs, I found much of the slowdown can be improved through proactive testing, diagnosis, and optimization or avoided through custom kernel configuration. The study motivated me to create an automated tool to diagnose the root causes of performance issues. I recognized that existing root cause diagnosis tools make fundamental assumptions that are broken by performance issues. Such tools only work for functional bugs that cause clear-cut failures, *e.g.*, crashes, and have *absolute* root causes, *e.g.*, broken invariants or predicates, when the expected performance outcome must be determined *relative* to inputs workloads and performance root causes tend to be abnormal *relative* occurrences of events. I designed the relational debugging algorithm, which caters to the "relative" characteristic of performance issues, and we implemented the algorithm with Perspect [20], which can effectively diagnose complex performance issues that human developers struggle with and existing tools are ineffective against.

In addition to my main thesis work, I have also worked on improving system reliability by automating failure reproduction in distributed systems [20], including failures caused by concurrency [16]. I have also worked on creating a novel performant file system for persistent memory [17].

For future work, I am inspired by the observations from my Linux performance study to create an automated pipeline for performance bug detection, diagnosis, and patching. Given that existing tools that serve similar purposes are often built for functional failures, I see potential for novel techniques that cater to the characteristics of performance issues. In addition to creating an external toolchain, I am also interested in identifying ways that software systems can be designed such that they become easier to observe for the purpose of performance debugging. Previous efforts to make systems more observable [14, 22] have been focused on improving reliability. In addition, I am also excited to identify new opportunities to improve performance by uncovering unknown performance phenomena and studying emerging workloads, such as serverless computing and machine learning workloads. My most recent research is on studying performance creep, a gradual form of performance degradation that results in significant impact in the long run. Creep is challenging to diagnose manually because it is caused by code changes of fine granularity. The project aims to create novel techniques to accurately profile and diagnose performance creep and identify ways to mitigate and prevent it.

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 over seven and a half years [19]. 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 have 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.

In addition to those eventually carried out by Linux/Ubuntu developers, I found optimizations for two root causes and better configurations for two more. These findings suggest the importance of proactive testing, diagnosis, and optimization in improving performance and the benefit of custom configuring kernels in achieving a balance between performance, functionality, and security.

Impact The findings from the study 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.

Perspect: Automating the diagnosis of the root causes of performance issues. While carrying out the previous study, I manually diagnosed a number of performance changes and found no readily available tools that automate the diagnosis process for performance issues. Profilers identify performance symptoms, such as hotspots, but do not link them to the root causes. Existing root cause diagnosis tools are *not* built for performance issues—they only work for bugs that cause clear-cut failures deemed incorrect regardless of inputs, *e.g.*, crashes; however, the expected performance outcome must be determined *relative* to input workloads. 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 one that is processing a few requests but experiencing memory leaks. However, if we compare the memory consumption of the two executions at the request level, the memory leak issue can be exposed.

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 automatically chooses the appropriate fine-grained events as reference points to serve as the basis of comparison, enabling it to work independently of changing inputs. To identify these reference points, relational debugging starts from those farthest away from the performance symptoms in program dependency. It then iteratively moves to reference points closer to the performance symptoms and can still capture the performance difference.

Relational debugging returns the root causes of the performance issues in the form of a distribution of event occurrences relative to the reference point events, called relations. Relations can generalize to diverse forms of performance root causes while existing root cause representations cannot. For example, MongoDB-57221 [6] occurred because including an additional query plan non-linearly increased the number of evictions per record deleted. This case breaks no invariants or predicates but does result in abnormal relation changes.

We implemented relational debugging with Perspect [20], which 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 trial and error. We used Perspect to diagnose 10 out of 12 real-world performance bugs.

Impact I used Perspect to diagnose two challenging open bugs (MongoDB-57221 [6], MongoDB-56274 [7]), and MongoDB developers confirmed the root causes reported by Perspect. A developer commented, "(Perspect's result) ties all the pieces together into a nice explanation." Perspect is requested by developers of the MMTk memory management framework [9] and has attracted interest from CockroachDB.

Pensieve: Automating failure reproduction in distributed systems. When failures occur in distributed systems, it is key to promptly diagnose and resolve them . Often, developers struggle with failure diagnosis because they cannot reproduce the failure. Failure reproduction tends to be a cumbersome process where developers carry out guesswork with limited information from execution logs. Our study found that developers spend 69% of the failure resolution time on reproducing the failures [23]. We built Pensieve [23] to automatically construct near-minimal reproductions of failures using the event-chaining approach. Event-chaining scales to complex distributed systems while existing techniques like symbolic execution [12] scale poorly. The key idea of event-chaining is to restrict the analysis to events that most likely caused the failure and skip program logic that is likely irrelevant but is expensive to analyze, such as loops. The reproduction program inferred by event-chaining is then refined based on feedback from dynamic execution.

FlakeRepro: Automating the reproduction of concurrency-related flaky tests. We built FlakeRepro [16] during my internship at Microsoft Research to automatically reproduce flaky tests that fail non-deterministically due to concurrency. Flaky tests are among the top two causes of software development slowdown at Microsoft because they are hard to reproduce. In our evaluation, FlakeRepro is able to reproduce 26 out of 31 flaky tests in production codebases in under 6 minutes on average. FlakeRepro statically identifies critical memory accesses causally relevant to a failure in a flaky test and dynamically explores interleavings between the memory accesses to reproduce the failure.

ctFS: a novel performant file system for persistent memory. Persistent byte-addressable memory (PM) is significantly faster than disks, shifting the bottleneck from IO to the file system software. Existing file systems use expensive file indexing logic originally designed for disks. Different from the disk, accesses to persistent memory are governed by the hardware memory management unit (MMU), like volatile RAM. ctFS [17] removes most of the software file indexing overhead. It achieves this by allocating files as contiguous virtual memory chunks, such that translating the file offset to an address no longer needs expensive file index lookups and only requires a simple virtual memory offset calculation. The rest of the lookup is performed by the MMU hardware at a fraction of the cost of software indexes. ctFS outperforms ext4-DAX and SplitFS [15] by 3.6x and 1.8x, respectively.

2 Future Research

For future work, I plan to take a multi-pronged approach to improving system performance. (1) As new hardware, software, and workloads constantly arise, so do new patterns of performance problems. I am interested in studying the performance of emerging workloads, such as serverless computing and machine learning workloads. I am also interested in uncovering unknown performance phenomena in system software in general. My current focus is on understanding performance creep (section 2.1). Based on the findings from 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 that provide better diagnostic information when performance issues occur (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 patches the system automatically.

2.1 Uncovering and studying new performance phenomena

Understanding performance creep. I am 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 [11] experienced a 34% regression from version 4.1.22 to 5.7.2 [2]. We find creep is a common cause of significant long-term regression in popular software systems, including MongoDB [10] and GCC [8]. Unfortunately, the causes of performance creep remain largely mysterious. The main reason is that performance creep is very challenging to diagnose, as it 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 for diagnosing performance creep as they are too coarse-grained or have too much overhead. I plan to identify and develop techniques to measure small performance differences accurately through 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, the goal is to enable performance 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. Performance outliers such as tail latency may break service agreements and are highly impactful. Yet the causes of performance outliers are more elusive, for example, when extra processing or slow paths are triggered, which are hard to reproduce or capture using profilers [13]. 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 relations, which represent performance root causes [20], I propose *relational asserts*, which check for the expected execution relative to fine-grained program events and hold regardless of input workloads. The goal is that when performance outliers occur, failed relational asserts can give developers effective clues about the root causes. I plan to design automated techniques to identify what to assert relationally accurately by analyzing normal executions and to decide where to insert the asserts such that they cover a maximal number of performance corner cases while incurring minimal overhead.

Capturing the root causes of performance outliers in situ *across software components*. Because of the difficulty in representing and communicating performance expectations, 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 base pages to huge pages against the application's "wishes." With relational asserts systematically inserted into an application, I propose to automatically translate them into corresponding asserts in 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 huge page 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], which was eventually diagnosed through a trial-and-error process. I plan to create informative performance metrics that encode relevant diagnostic information, such as variable values and execution paths while causing minimal execution overhead. The goal is that when developers observe abnormal metrics, they are also provided with sufficient information to understand how the execution led to abnormal metrics.

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

Detecting performance bugs before they cause severe consequences. Catching performance bugs early during the testing phase prevents undesirable performance or outages in production. Unfortunately, performance tests have limited coverage and frequently fail to catch performance bugs that can lead to severe consequences. I am interested in automating performance bug detection based on findings from my previous work [20], which suggests 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 but did not cause noticeable memory over-consumption. Such 'hidden" root causes create opportunities for early detection of performance issues. For future work, I plan to create automated techniques to modify the workloads of "healthy" runs that already trigger such root causes so that they manifest into detectable performance issues.

Extending relational debugging to diverse performance diagnosis scenarios. Performance diagnosis is a cumbersome task that developers need to perform under an array of different settings. We implemented Relational debugging with Perspect [20] to automate root causes diagnosis of performance issues offline in a single-machine setting. Relational debugging is adaptable to more diverse scenarios: I plan to build a lightweight online tool that diagnoses the root causes of performance issues by carrying out relational debugging incrementally to achieve low overhead. I also plan to adapt relational debugging to work on distributed systems as well as the Linux kernel.

Coupling performance diagnosis with patch generation. In addition to performance issues that require complex fixes designed by human developers, many causes of bad performance are possible to mitigate through automatically generated patches because they follow a set of patterns, such as lack of caching, redundant work, and sub-optimal configuration [21]. I find that well-known patterns of non-optimal performance correlate with different types of root causes identified by Perspect [20]. 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. Such a pipeline can then realize a variety of goals. For example, it can be used to automatically adapt a system based on changing workloads or to automatically detect and fix pervasive patterns of non-optimal performance that are too cumbersome to fix individually.

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. MMTk is a memory management toolkit. https://www.mmtk.io/. (2023).

[10] 2023. MongoDB: The Developer Data Platform — MongoDB. (2023). https://www.mongodb.com/

[11] 2023. MySQL. (2023). https://www.mysql.com/

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[13] 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

[14] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[15] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 494–508. https://doi.org/10.1145/3341301.3359631

[16] 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

[17] 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

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

[19] 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

[20] 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

[21] 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

[22] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. *SIGARCH Comput. Archit. News* 39, 1 (mar 2011), 3–14. https://doi.org/10.1145/1961295.1950369

[23] 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