Today's server computing is undergoing a drastic shift; exponential growth in users, requests, and data poses an ever increasing demand on the performance of computer servers. This challenge has resulted in two major trends. First, data centers *scale out* the computation with more resources. A typical approach is to employ more cores in a single server and more servers in the data center. Second, blazingly fast storage devices with unique characteristics are developed to meet the exponential growth in data. Unfortunately, traditional systems software was not designed to support these trends and thus prevent server applications from fully realizing the potential of these developments.

My research designs system software that allows applications to fully benefit from these computing trends by supporting three critical application requirements. The first requirement is **storage I/O efficiency**. Traditional storage stacks are designed with the assumption of slow devices with block interfaces. Therefore, they cannot unleash the full potential of modern storage devices. I led a research team with graduate students and faculties to design *OdinFS* [3], a novel file system for emerging storage class memory by taking into account its unique characteristics, thus achieving tens to hundreds of times better performance than prior state of the art. The second requirement is **multicore scalability**. Modern applications must leverage the multicore architecture to meet the performance demand. Synchronization primitives are thus critical to application performance. I collaborate with a group of graduate students, researchers, and faculties to design *SynCord* [2]. *SynCord* allows applications to safely customize the kernel lock on the fly and thus maximizes application performance. Programming for multicore architecture is notoriously error-prone. I designed a novel data race detector: *PUSh* [5], which leverages existing memory management hardware, instead of memory instrumentation, to detect conflicting memory accesses. *PUSh* achieves two to three orders of magnitude lower performance overhead compared to the widely used memory instrumentation-based approaches. The third requirement is **fault tolerance**. With the exponentially increasing number of hardware components, practical fault tolerance mechanisms are a must. I collaborated with my Ph.D. advisor to design *RRC* [7], an application-transparent replication system for commercial off-the-shelf containers. *RRC* only incurs a 400μs latency overhead (vs. tens of milliseconds from competitors) while also achieving a significantly lower throughput overhead, enabling its deployment for critical server applications.

My approach to research is rooted in an application-driven method, often referred to as "measure, then build". My research project typically starts with a process to thoroughly understand application requirements, hardware characteristics, and the limitations of the existing systems. This ensures my work addresses real problems and often reveals novel insights for system design. Many of my systems are based on a design philosophy that involves identifying and breaking the hidden couplings in systems to overcome design challenges. The systems I designed typically connect various subsystems in the operating system, including I/O (storage and network) stacks, memory management systems, and synchronization primitives.

I am a strong believer in research reproducibility. I have fully open-sourced my research artifacts and ensured a third party can successfully reproduce the results. Major industry practitioners, such as Huawei, Oracle, and Apple, have shown interest in supporting and adopting my research.

Unleashing the performance potential of SCM

Maximizing SCM performance. The emerging storage class memory (SCM), e.g., Intel Optane persistent memory and future CXL-based devices, break the traditional dichotomy of storage and memory. Similar to storage devices, SCM offers data persistence, while similar to DRAM, it also offers ultra-low latency and direct accessibility through load and store instructions.

Recent characterization studies show that real commercially available SCM is not simply slower DRAM that persists data but instead has many subtle performance characteristics. Taking Intel Optane PM as an example, it has two crucial characteristics. The first one is the tension between concurrent accesses and device performance. Specifically, a small number of threads underutilizes device bandwidth, while a high number of threads leads to a performance meltdown. This is because excessive concurrent access renders on-DIMM caching and prefetching inefficient. The second factor is that remote NUMA accesses on PM are much slower than DRAM. This is because the current hardware maintains the cache coherence information in the memory and the relatively slower PM performance compared to DRAM leads to the performance difference. I studied the state-of-the-art file systems for SCM and found that they mostly exploit their low-latency and byte-addressable access. However, existing systems do not consider the above two performance factors, leading to severely underutilized PM performance; my evaluation shows that existing file systems can only utilize up to 47% and 19% of the PM read and write bandwidth, respectively.

I designed *OdinFS* [3], a novel file system designed to maximize the performance of SCM. The key insight behind *OdinFS* is that SCM accesses must be *decoupled* from the application threads to maximize performance. Specifically,

on each NUMA node, *OdinFS* creates multiple background kernel threads (delegation threads). Application threads cannot directly access SCM but instead find out the NUMA node the SCM block belongs to and delegate access to one of the corresponding delegation threads on that NUMA node. This approach has the following benefits. First, by controlling the number of delegation threads, *OdinFS* therefore limits the maximum access concurrency within a NUMA node, avoiding the performance meltdown. Second, delegation threads always access their own local SCM, thus avoiding the pronounced NUMA impact. Third, delegation threads always access devices in different NUMA nodes in parallel, enabling efficient use of aggregated SCM bandwidth across all the NUMA nodes. *OdinFS* constantly outperforms other file systems by several orders of magnitude in terms of I/O bandwidth and latency.

Resilient SCM. Like traditional storage devices, SCM is vulnerable to media errors (e.g., device failures). Unlike traditional storage devices, SCM is also vulnerable to software bugs (e.g., memory scribbles). The high performance of SCM demands an efficient mechanism to handle both types of errors. I collaborate with a team of researchers to contribute Tenet [1], a resilient persistent transactional memory system. Leveraging the Intel memory protection key (MPK) and transactional memory semantics, Tenet only needs to detect data corruption upon first accessing or committing an object. Based on an insight that transactional memory systems already provide data redundancy for crash consistency, Tenet replicates data off the critical path to an SSD. As a result, Tenet offers stronger protection with a significant lower performance (¡8% vs. 60% - 67%) and storage overhead (60X reduction) than prior works.

Towards Reliable and Customizable Scalability

Efficient and sound data race detection. Programming parallel programs is the primary way to achieve scalability. However, parallel programs introduce concurrency bugs, which are notoriously difficult to debug. Data races are a major cause of concurrency bugs. A data race occurs when two threads access the same memory location concurrently without synchronization, and at least one of the accesses is a write. Most prior data race detectors, such as ThreadSanitizer (TSan), rely on instrumenting memory accesses and perform complex analysis to reason about concurrent conflicting memory accesses for each instrumented access. This approach often results in thousands of times increases in performance and memory overhead.

I designed *PUSh* [5], a novel dynamic data race detector with a similar level of soundness as the aforementioned detectors but with orders of magnitude lower performance and memory overhead. The efficiency of *PUSh* comes from two key insights. First, instead of reasoning about the conflicting memory accesses, *PUSh* requires programmers to add lightweight annotation (typically less than 5% of the total code) to specify the intended sharing of each global object (e.g., private to one thread or protected by a lock). Second, instead of relying on code instrumentation, *PUSh* uses existing memory management hardware (e.g., page table) to detect memory accesses that violate the specified sharing policy.

A key novelty of *PUSh* is its use of memory protection keys (MPK), a hardware feature recently available in the x86 ISA. With MPK, *PUSh* eliminates the overhead of adopting a separate page table for each thread. Furthermore, in many cases, MPK allows *PUSh* to avoid the prohibitive overhead that invokes system calls to change access permissions in the page table. For example, every time a lock is acquired or released, the access permissions to the objects protected by the lock need to change to grant or revoke the thread's access permission to the objects. With MPK, this can be achieved by putting all the objects protected by the same lock into one protection domain and simply modifying the userspace PKRU register, which only takes around ten nanoseconds. *PUSh* also addresses a critical drawback in all prior annotation-based data race detectors by ensuring incorrect annotations (e.g., grant write permissions to two threads to a shared object simultaneously) cannot hide data races. This is achieved by performing happens-before analysis on annotations instead of memory accesses. *PUSh*'s memory overhead is less than 5.8% (versus 54%-11000% with TSan) and performance overhead is less than 54% (versus 304% - 36000% with TSan).

Application-informed kernel synchronization primitives. Synchronization primitives, such as locks, are crucial to achieving multicore scalability. Except for the locks used in the applications, kernel locks also significantly affect application performance. Unfortunately, kernel locks are designed for general cases, and their implementation is baked into the kernel, leaving applications minimal control over its behavior. Such a semantic gap leads to pathological cases where the general kernel locks incur an unacceptably high overhead.

I collaborated with a group of researcher to design *SynCord* [2], that bridges this semantic gap by allowing applications to deploy workload-specific lock implementations to the kernel. Specifically, *SynCord* abstracts key behaviors of kernel locks and exposes them as APIs. The application implements these APIs with C-style code, and *SynCord* applies the code without recompiling or rebooting the kernel. A key challenge in *SynCord* is to prevent the user-provided code from corrupting the kernel, which we overcome with a combination of static verification by

leveraging the existing eBPF framework, lightweight runtime checking to ensure liveness, and careful API designs to provide mutual exclusion. *SynCord* increases the application performance by up to three orders of magnitude.

Fault-Tolerant Virtualization

Responsive Replicated Containers. Critical services deployed in the cloud demand high reliability to survive failures. Most replication mechanisms are based on checkpointing to a passive backup, where the primary replica is periodically paused, and its state is checkpointed to the backup. A fundamental drawback of this scheme is the prohibitively long delay of output to clients. Specifically, to ensure consistency upon failure, the outputs generated by the primary to the external world, including to clients, cannot be released until the corresponding checkpoint has been sent to and committed by the backup. Since checkpointing is an expensive operation, the checkpointing interval is typically set to tens of milliseconds, leading to an incurred delay of tens of milliseconds. Such delay is unacceptable for many server applications.

I had the insight that the disadvantages of existing replication mechanisms come from the *undesirable coupling* between replication-related operations and normal operations (e.g., the coupling of checkpointing interval and the output delay, as explained above). I designed *RRC* [7], a container replication mechanism that targets server applications with minimal response latency overhead by breaking the undesirable couplings. *RRC* first decouples checkpointing interval from output delay with a combination of periodic checkpointing and deterministic replay. Specifically, during normal execution, the primary sends periodic checkpoints to the backup and logs to the backup outcomes of non-deterministic events. Upon failure, the backup restores the latest checkpoint and then deterministically replays the execution up to the last external output. Hence, external outputs only need to be delayed by the short amount of time it takes to send and commit the relevant portion of the non-deterministic event log to the backup.

RRC also minimizes tail latency overhead by decoupling the time to take a checkpoint from service interruption. Specifically, *RRC* introduces a new kernel primitive: container fork. For checkpointing, *RRC* pauses the primary container, forks a shadow container, and resumes execution. The checkpoint is obtained from the shadow container, resulting in a minimal service interruption time. *RRC* thus achieves two orders of magnitude lower latency overhead (¡300μs vs. tens of milliseconds) and significant advantages in throughput overhead compared to the passive replica scheme. Huawei has shown interest in deploying *RRC*.

Resilient virtualization infrastructure. In addition to individual components, I also design resilience mechanisms for virtualization infrastructures. I proposed a novel component-level recovery scheme called *Microreset* and designed a resilient hypervisor [4] based on it. To realize *RRC*, I have designed *NiLiCon* [6], which to the best of my knowledge, is the first replication mechanism for commercial off-the-shelf containers.

The Road Ahead

Looking forward, I plan to continue designing system software that provides applications with high performance, scalability, and reliability. I am interested in a cross-layer design, from user-level frameworks to hardware architecture, to fully unleash the potential of modern hardware.

User-level storage stack. Storage class memory (SCM), such as Intel Optane Persistent memory, provides direct accessibility through ordinary load and store instructions. Furthermore, the memory management unit protects access to SCM. These characteristics bring the opportunity to move most of the storage stack from the kernel to userspace, which minimizes the software overhead and thus fully exploits the ultra-low latency of these devices. Although several recent systems follow the same observation, they fail to resolve an inherent tension between security and performance, thereby underutilizing SCM performance. Specifically, for security, most of the systems rely on a privileged component to handle most of the metadata operations (e.g., creating a file), which significantly increases software overhead. A few systems perform most of the metadata operations in userspace at the cost of severely reduced security compared to the conventional kernel storage stack. I am currently designing a framework that enables secure user-level storage stacks while preserving the low latency benefit. This requires a novel system design that breaks the tension between performance and security, as mentioned above. Furthermore, I plan to leverage formal methods to prove that the user-level storage stack meets security requirements. I am also interested in extending such a design to distributed storage systems.

Microsecond-scale fault tolerance. Critical server applications demand low latency, and the microservice architecture further exacerbates such demand. Recent advances in I/O devices and system software have enabled applications to achieve microsecond-scale latency. Unfortunately, the advancement of fault-tolerant systems lags behind; none of the existing work can provide microsecond-scale fault tolerance for general multithreaded applications. My prior work: *RRC* took the first step toward this problem. Unfortunately, recording non-deterministic events (as in *RRC*) still incurs too high overhead for the microsecond-scale computing, rendering RRC unsuitable. I plan to continue pushing state of the art on this front by designing resilience systems for microsecond-scale computing platforms. A possible direction is to co-design hardware and software to achieve both the latency and the reliability goals.

Future operating system architecture for modern applications. Modern applications require performance, reliability, and customizability from the underlying operating systems. The "de-facto" operating system for server computing: Linux, has constantly failed to meet these requirements. Alternative proposals, such as Microkernel, only focus on one of the aspects: reliability, and cannot provide the two other requirements. Adapting existing OS architecture to meet these requirements is inherently challenging because their design is not rooted in these critical requirements. For example, despite an enduring engineering effort for decades, the Linux kernel is still known to be a performance and scalability bottleneck, forcing applications to bypass it to meet the performance demand. Similarly, providing efficient communications among Microkernel components is still an open problem. I believe that now is a suitable time to explore the right architecture that treats the aforementioned requirements as first-class citizens in the operating system design. All my prior and current works lead toward this direction, and designing such an operating system architecture is my long-term research goal.

Conclusion

A golden age for computer systems is ahead of us. There are major changes both below and above system software. Below the system software, emerging hardware devices provide enormous performance potential, whereas, above it, modern applications require critical properties that traditional designs inherently cannot provide. There is an urgent demand for system software to adapt to this new computing landscape. I am fortunate and enthusiastic as a systems researcher to contribute in this exciting time.

References

[1] R. Madhava Krishnan, Diyu Zhou, Wook-Hee Kim, Sudarsun Kannan, Sanidhya Kashyap, and Changwoo Min. TENET: Memory Safe and Fault tolerant Persistent Transactional Memory. In *21st USENIX Conference on File and Storage Technologies*, Santa Clara, CA, February 2023.

[2] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 667–682, Carlsbad, CA, 2022.

[3] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. Odinfs: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193, Carlsbad, CA, 2022.

[4] Diyu Zhou and Yuval Tamir. Fast Hypervisor Recovery Without Reboot. In *48th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 115–126, Luxembourg City, Luxembourg, June 2018.

[5] Diyu Zhou and Yuval Tamir. PUSh: Data Race Detection Based on Hardware-Supported Prevention of Unintended Sharing. In *ACM/IEEE 52nd Annual Symposium on Microarchitecture*, pages 886–898, Columbus, OH, October 2019.

[6] Diyu Zhou and Yuval Tamir. Fault-Tolerant Containers Using NiLiCon. In *34th IEEE International Parallel and Distributed Processing Symposium*, pages 1082–1091, New Orleans, LA, May 2020.

[7] Diyu Zhou and Yuval Tamir. RRC: Responsive Replicated Containers. In *2022 USENIX Annual Technical Conference*, pages 85–99, Carlsbad, CA, 2022.