Research Summary

Diyu Zhou

My research is in the general area of computer systems with specific interests in operating systems and dependable computing. My thesis is focused on practical, low-overhead dependability mechanisms, suitable for deployment in mainstream systems. Specifically, my thesis presents a data race detector; a low-latency recovery mechanism that I deployed for hypervisor recovery; and two container replication mechanisms. To build these dependability mechanisms, I identified sweet spots in the design space of dependability mechanisms that balance soundness and overhead and found novel ways to leverage existing hardware features and/or dedicated optimizations of the internals of operating systems and hypervisors.

Fast Hypervisor Recovery

Hypervisors demand fault tolerance since they are single points of failure in virtualized systems: a fault in the hypervisor can lead to the failure of all the VMs running above it. The prior state of the art, *ReHype* [2], recovers the hypervisor from transient faults (e.g., transient bit-flips in memory or register) by rebooting a hypervisor instance, which takes hundreds of milliseconds, an unacceptably long service interruption time in many deployment scenarios. **Based on my analysis of the operation of *ReHype*, I introduced a novel recovery technique, called *microreset*, that, for suitable software systems, performs recovery *without* reboot, significantly reducing the recovery time while achieving almost the same level of soundness. I applied *microreset* to the Xen hypervisor and built a hypervisor fault tolerance mechanism: *NiLiHype*.** This work [6] was published at *DSN 18*, the top conference on dependable computing.

Microreset is based on the observation that almost all the global state in a failed hypervisor/OS is valid. This is supported by the successful recovery rate results with *ReHype* and by fault injection studies of the Linux kernel reported by others [3] With *microreset*, upon error detection, the processing of all current requests in the failed component is abandoned and the component is reset to a quiescent state that is highly likely to be valid. Next, the *microreset* mechanism fixes the inconsistencies in the failed component to make it ready to handle new or retried requests from the rest of the system and thus finishes the recovery. A key implementation challenge of *NiLiHype* is to identify the inconsistencies in the hypervisor state after the hypervisor is reset to the quiescent state. *NiLiHype* overcomes this challenge using an incremental procedure, based on fault injection results. This process results in improving the recovery rate from 0% to >95%. **Compared to *ReHype*, *NiLiHype* reduces the service interruption time during recovery from 713ms to 22ms, a factor of over 30x. *NiLiHype*'s recovery rate is only 2% lower than the rate achieved by *ReHype*.**

Data Race Detection by Hardware-Supported Prevention of Unintended Sharing

A data race occurs when two threads access the same memory location concurrently without proper synchronization and at least one of the accesses is a write. Data races are causes of many concurrency bugs. A majority of prior data race detectors rely on instrumenting global memory accesses and, for each instrumented access, performing complex analysis to reason about concurrent conflicting memory accesses. This approach often results in tens of or even thousands of times increases in the execution time as well as extra memory usage. **I developed *PUSh* [7, 4], a novel dynamic data race detector with a similar level of soundness as the aforementioned detectors but with orders of magnitude lower performance/memory overhead.**

PUSh is a dynamic data race detector that requires the programmer to annotate the code to specify the intended sharing of each global object (e.g., private to one thread or protected by a lock). *PUSh* uses existing memory protection hardware to detect memory accesses which violate the specified sharing policy. **A key novel feature of *PUSh* is its use of memory protection keys (MPK), a hardware feature recently available in the x86 ISA.** With MPKs, each virtual page is tagged, in its page table entry, with a single protection domain number. A per-thread register: PKRU, controls the access permissions to each protection domain for the current thread. With MPK, *PUSh* efficiently enforced different access permissions for different threads, without requiring a separate page table for each thread. Critically, 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 *PUSh*, this is achieved by putting all the objects protected by the same lock into one protection domain and simply modifying the PKRU register, which only takes 13ns. *PUSh* thus avoids the prohibitive overhead that, without MPK, would be caused in this case by the need to invoke system calls to change access permissions in the page table.

PUSh also solves a fundamental problem in prior data race detectors that are based on programmer annotations to specify intended sharing policies. Specifically, the problem is that incorrect annotations (e.g., grant write permissions to two threads to a shared object without synchronization) can hide data races. *PUSh* includes a novel and efficient algorithm that detects such incorrect annotations based on happens-before analysis.

I designed and implemented many other optimizations to *PUSh*. Among them, the most critical one involves a simple enhancement to the Linux kernel that allows it to efficiently support mapping multiple virtual pages to the same physical page. This enables a key optimization used by *PUSh* to reduce the memory overhead [4]. *PUSh* also utilizes a mechanism that periodically changes (rehashes) the mapping of objects to domains to prevent the limited number of protection domains in MPK from hiding recurrent data races. Compared to other annotation-based data race detectors, *PUSh* does not miss data races due to incorrect annotations and has significantly lower performance and memory overheads. Compared to *ThreadSanitizer*(TSan), a commonly used data race detector, *PUSh* requires an extra effort of annotation and in rare cases, *PUSh* requires several executions to detect a data race. **In return, for a set of 10 real-world benchmarks, *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)**.

Fault-Tolerant Containers

Critical services deployed in the cloud demand high reliability to survive failures and this has motivated a plethora of application-transparent techniques that utilize a virtual machine (VM) as the replication unit. Containers are often used as an alternative to VMs for providing an isolation and multitenancy layer. Despite the advantages of containers in smaller sizes and faster startup, there has been very little work on fault tolerance for containers. **Motivated by this, I designed and implemented *NiLiCon* [8], which, to the best of my knowledge, is the first replication mechanism for commercial off-the-shelf containers.**

NiLiCon adapts to containers a VM replication algorithm, called *Remus* [1], that is the basis for several VM replication schemes. *NiLiCon* involves a primary container and a backup container. The primary container is periodically (tens of milliseconds) paused and its state is checkpointed to the backup physical machine. **A key challenge in the implementation of *NiLiCon* is that, compared to VMs and hypervisors, there is much tighter coupling between the container state and the underlying kernel.** Specifically, *NiLiCon* needs to checkpoint parts of the container state that are in the kernel. With the existing Linux interface, this often involves a large number of expensive system calls, which leads to prohibitive overhead. *NiLiCon* overcomes this challenge with various kernel enhancements, incurring only minor changes to the kernel. For example, based on the observation that most of the in-kernel state rarely changes between checkpoints, *NiLiCon* caches in-kernel container state from the previous checkpoint and instruments kernel functions transparently with *ftrace* to detect in-kernel state changes. If there are no state changes since the last checkpoint, the cached state is sent to the backup and thus avoids the expensive operation to retrieve the in-kernel state components. **Overall, *NiLiCon* achieves performance that is competitive with *Remus*. For a set of eight benchmarks, the overhead is 19% - 67% with *NiLiCon* vs. 13% - 72% with *Remus*.**

Low Latency Replicated Containers with Deterministic Replay

A fundamental drawback of *Remus*-based mechanisms 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 in many deployment scenarios. **I developed *HyCoR* [5], a container replication mechanism that uses deterministic replay to decouple the checkpointing interval from the incurred delay, thus enabling it to achieve sub-millisecond latency overhead.**

With *HyCoR*, during normal execution, the primary sends periodic checkpoints to the backup and logs to the backup the 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. *HyCoR* logs the outputs from system calls. In addition, in order to capture memory non-determinism while maintaining good performance, *HyCoR* records the order of synchronization events. This could cause replay failures for buggy or legacy applications with data races. *HyCoR* uses a simple timing adjustment mechanism that significantly increases the probability of correct replay as long as the rate of data races is small, which is likely the case for deployed applications. Specifically, the mechanism works by adjusting the relative timing of the application threads during replay to approximately match the timing during the original execution to preserve the order of racy memory accesses.

For a set of eight benchmarks, with *HyCoR*, the average incurred extra delay of outputs is less than 600μs versus 38ms-63ms with *NiLiCon*. Since the checkpointing interval does not affect the response latency, a unique feature of *HyCoR* is that the checkpointing interval can be adjusted to trade off performance and resource overheads with recovery latency and vulnerability to data races. For applications known to be data-race-free, *HyCoR* can utilize a much longer checkpoint interval for better performance. **Specifically, with a one-second checkpointing interval, *HyCoR* incurs a performance overhead of 2%-58% versus 18%-139% with *NiLiCon*.** A fault injection campaign has shown that the timing adjustment mechanism achieves a recovery rate of over 99.5% for two of the evaluated benchmarks that have low data race rates.

References

[1] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, April 2008.

[2] Michael Le and Yuval Tamir. ReHype:Enabling VM Survival Across Hypervisor Failures. In *7th ACM International Conference on Virtual Execution Environments*, pages 63–74, Newport Beach, CA, March 2011.

[3] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Can Linux be Rejuvenated without Reboots? In *IEEE Third International Workshop on Software Aging and Rejuvenation*, pages 50–55, Hiroshima, Japan, November 2011.

[4] Diyu Zhou and Yuval Tamir. Data Race Detection by Prevention of Unintended Sharing Using PUSh. In preparation.

[5] Diyu Zhou and Yuval Tamir. HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay. In preparation.

[6] 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.

[7] 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.

[8] 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.