Research Summary

Diyu Zhou

My research is in the general area of computer systems with specific interests in operating systems and dependable computing. My dissertation work is focused on building dependability mechanisms for large, complex, parallel software components: hypervisors, containers, and server applications. Specifically, my thesis presents a debugging tool that is designed to detect an important type of programming errors: data races in parallel applications and several application-transparent fault-tolerance mechanisms.

The implementation of dependability mechanisms involves a tradeoff between soundness and overhead. If soundness is prioritized too highly, the result is unacceptably high overhead during normal operation and/or unacceptably long service interruptions when an error occurs. An important contribution of my dissertation work is to show that by identifying sweet spots in the design space that balance soundness and overhead, in many cases, it is possible to build **practical dependability mechanisms that reduce some aspects of overhead by orders of magnitude while providing nearly the same level of the soundness compared to directly-comparable prior works**. The approaches towards building such dependable systems are based on novel ways to leverage existing hardware features and/or on dedicated optimizations of the internals of operating systems and hypervisors.

Fast Hypervisor Recovery

A hypervisor demands fault tolerance since the hypervisor failure would lead to the failure of all the VMs running on top of it. The prior state-of-the-art work: *ReHype* [2] recovers the hypervisor from transient faults (e.g., transient bit-flips in memory or register) by rebooting the hypervisor, which takes hundreds of milliseconds, an unacceptably long service interruption time for many deployment scenarios. **Motivated by this, I have introduced a novel recovery technique: *microreset*, that performs the recovery significantly faster while almost preserving the same level of soundness. I have further applied *microreset* to the Xen hypervisor and build a hypervisor fault tolerance mechanism: *NiLiHype*.** This work [6] has been published at *DSN 18*, the top conference for 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 other fault injection studies [3]. With *microreset*, upon error detection, the processing of all current requests is abandoned from the failed component and this resets the failed component to a quiescent state that is highly likely to be valid. Next, the *microreset* mechanism fixes the inconsistent state 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 inconsistent state in the hypervisor after resetting it to the quiescent state, which *NiLiHype* overcomes by using an incremental procedure, based on fault injection results. With this approach, the successful recovery rate of *NiLiHype* increases 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 such access is a write. Data races are causes of many concurrency bugs. The majority of prior data race detectors rely on instrumenting global memory accesses and for each instrumented access, perform complex analysis to reason about concurrent conflicting memory accesses. This approach often results in tens of or even thousands of times of execution slowdown and extra memory usage. **I have developed *PUSh* [7, 4], a novel dynamic data race detectors with a similar level of soundness but orders of magnitude lower performance/memory overhead.**

PUSh is a dynamic data race detector based on requiring the programmer to annotate the code to specify intended sharing of each global object (e.g., private to one thread or protected by a lock) and the use of existing memory protection hardware to detect memory accesses which violates the specified sharing policy.

A key novelty in *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 PKRU, *PUSh* eliminates the need of adopting a separate page table for each thread. More 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 MPK, this can be 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. It thus avoids the prohibitive overhead caused by system calls that change the page table permission.

I have adopted many other optimizations to *PUSh*. Among them, the most critical one involves a simple and efficient enhancement to the Linux kernel to make it 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]. With Linux, a single lock serializes all page table changes invoked by threads in a process, which significantly limits the scalability of *PUSh*. The above kernel modification also eliminates such serialization. *PUSh* contributes a novel algorithm that prevents incorrect annotations from hiding data races and this can be applied to other annotation-based data race detectors. It also adopts a periodical rehash mechanism to reduce the chances of missing data races due to the limited number of protection domains in MPK. 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 overhead. 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 all the data races. **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 failure and this has motivated a plethora of application-transparent techniques adopting a virtual machine (VM) as the replication unit. Containers are often used as an alternative approach to VMs to provide an isolation and multitenancy layer. Despite the advantage of containers in smaller sizes, faster startup, there has been very little fault-tolerance work based on containers. **Motivated by this, I have proposed *NiLiCon* [8], which to the best of my knowledge, is the first replication mechanism for commercial off-the-shelf containers**.

NiLiCon adopts to containers *Remus* [1], a widely used VM replication mechanism that serves as the basis for many follow-up works. *NiLiCon* involves a primary and a backup container. The primary container is periodically paused and its state is checkpointed to the backup physical machine every tens of milliseconds. **A key challenge to implement *NiLiCon* is that, compared to a VM, there is much tighter state coupling between a container and the underlying kernel**. Specifically, *NiLiCon* needs to checkpoint all in-kernel state of the primary container. With the current Linux interface, this often involves a large number of expensive system calls, leading to prohibitive overhead. *NiLiCon* overcomes this challenge with various kernel enhancements, but only incurring minor changes to the kernel. For example, based on the observation that most of the in-kernel state rarely changes within a checkpointing interval, *NiLiCon* caches the in-kernel state from the previous checkpoint and instruments kernel functions transparently with *ftrace* to detect in-kernel state changes. If there is no state change since the last checkpoint, the cached state is sent to the backup and thus avoids the expensive operation to retrieve those in-kernel state. **Overall, *NiLiCon* achieves competitive performance as *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 all *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 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 deployment scenarios. **I have developed *HyCoR* [5], which uniquely combines deterministic replay with container replication to decouple checkpointing interval with incurred delay time and thus addresses this problem. *HyCoR* only incurs sub-milliseconds delay**.

With *HyCoR*, during normal execution, the primary takes periodic checkpoints, logs the outcomes of sources of non-determinism, and sends them to the backup. Upon failure, backup restores from the latest checkpoint and then deterministically replay 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 log to the backup. To ensure practical performance, *HyCoR* only records the outcomes of synchronization operations to capture memory non-determinism. This would cause a replay failure for buggy or legacy applications with data races. *HyCoR* adopts 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, for data-race-free applications, *HyCoR* could adopt a much longer checkpointing interval for better performance. **Specifically, with a one-second checkpointing interval, *HyCoR* incurs a performance overhead of 2%-58% versus 18%-139% with *NiLiCon***. Fault injection campaign shows 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. Under Review.

[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.