UNIVERSITY OF CALIFORNIA

Los Angeles

Practical Dependable Systems with OS/Hypervisor Support

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Diyu Zhou

2020

ABSTRACT OF THE DISSERTATION

Practical Dependable Systems with OS/Hypervisor Support

by

Diyu Zhou
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Yuval Tamir, Chair

Critical applications require dependability mechanisms to prevent them from failures due to faults. Dependable systems for mainstream deployment are typically built upon commodity hardware with mechanisms that enhance resilience implemented in software. Such systems are aimed at providing commercially viable, best-effort dependability cost-effectively.

This thesis proposes several practical, low-overhead dependability mechanisms for critical components in the system: hypervisors, containers, and parallel applications.

For hypervisors, the latency to reboot a new instance to recover from transient faults is unacceptably high. *NiLiHype* recovers the hypervisor by resetting it to a quiescent state that is highly likely to be valid. Compared to a prior work based on reboot, *NiLiHype* reduces the service interruption time during recovery from 713ms to 22ms, a factor of over 30x, while achieving nearly the same recovery success rate.

NiLiCon, to the best of our knowledge, is the first replication mechanism for commercial off-the-shelf containers. *NiLiCon* is based on high-frequency incremental checkpointing to a warm spare, previously used for VMs. A key implementation challenge is that, compared to a VM, there is a much tighter coupling between the container state and the state of the underlying platform. *NiLiCon* meets this challenge with various enhancements and achieves performance that is competitive with VM replication.

HyCoR enhances *NiLiCon* with deterministic replay to address a fundamental drawback of high-frequency replication techniques: unacceptably long delay of outputs to clients. With deterministic replay, *HyCoR* decouples latency overhead from the checkpointing interval. For a set of eight benchmarks, with *HyCoR*, the latency overhead is reduced from tens of milliseconds to less than 600μs. For data race-free applications, the throughput overhead of *HyCoR* is only 2%-58%.

PUSh is a dynamic data race detector based on detecting violations of the intended sharing of objects, specified by the programmer. *PUSh* leverages existing memory protection hardware to detect such violations. Specifically, a key optimization in *PUSh* exploits memory protection keys, a hardware feature recently added to the x86 ISA. Several other key optimizations are achieved by enhancing the Linux kernel. For a set of eleven benchmarks, *PUSh*'s memory overhead is less than 5.8% and performance overhead is less than 54%.

The dissertation of Diyu Zhou is approved.

Milos D. Ercegovac

Songwu Lu

George Varghese

Yuval Tamir, Committee Chair

University of California, Los Angeles

2020

To my wife, Junchi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am fortunate to receive enormous support from many people during my pursuit of this Ph.D. degree. This thesis would not have been possible without these people.

First and foremost, I am forever grateful to my advisor, Yuval Tamir, for his guidance, immense knowledge, and commitment to my success. Yuval has endless patience for me, especially when I struggled with my research in the early stages of the Ph.D. journey. He has taught me how to conduct rigorous research and is always helpful with his research and life advice. I hope that in the future, I can help others in the same way as he has done to help me.

I would also like to thank my other committee members: Dr. Ercegovac, Dr. Lu, and Dr. Varghese for their helpful insights and constructive comments on my research.

I am grateful to the computer science department and the staff at UCLA. I deeply appreciate the fellowship I received in my first year and last year of the Ph.D. program. This allowed me to focus on research in the most critical stages of the Ph.D. journey. Special thanks to Joseph Brown for his invaluable help.

Many thanks to my friends, colleagues, and fellow graduate students at UCLA: Yuan He, Zhaowei Tan, Pei Wu, and Tianyi Zhang for sharing the joy and the despair in the Ph.D. journey. I enjoy every moment with you and will never forget the encouragement and help received from you. Thanks should also go to Israel Hsu, Siyuan Qi, Vince Rabsatt, and Jiewen Tan for their sincere friendship. Special thanks to Michael Le for his guidance, humor, and optimism that had taught me so much when I started my Ph.D. A significant part of this thesis is also built upon his work.

I very much appreciate the faculty members at Peking University, especially Professor Kaigui Bian, Professor Yingwei Luo, Professor Xiaolin Wang, and Professor Zhen Xiao for introducing me to the academic world and making me curious and enthusiastic about systems research.

I must also thank my family, especially my mom and dad, mother-in-law, and father-in-law, for their limitless love and unwavering support.

Last but not least, I'm deeply indebted to the love of my life, my wife, Junchi. Junchi, thank you for your accompany and for always being there for me in my darkest moment. This has been quite a long journey for both of us but it is just the end of the first adventure we would go through together in our life. I hold your hand, we grew old together till death do us apart.

VITA

2013	B.S., Computer Science Department of Computer Science and Technology Peking University
2018	M.S., Computer Science Department of Computer Science University of California, Los Angeles
2013–2020	Graduate Student Researcher/Teaching Assistant Department of Computer Science University of California, Los Angeles

PUBLICATIONS

Diyu Zhou and Yuval Tamir, “Fast Hypervisor Recovery Without Reboot,” 48th IEEE/IFIP International Conference on Dependable Systems and Networks, Luxembourg City, Luxembourg, pp. 115-126 (June 2018)

Diyu Zhou and Yuval Tamir, “PUSh: Data Race Detection Based on Hardware-Supported Prevention of Unintended Sharing,” ACM/IEEE 52nd Annual Symposium on Microarchitecture, Columbus, OH (October 2019).

Diyu Zhou and Yuval Tamir, “Fault-Tolerant Containers Using NiLiCon,” 34th IEEE International Parallel and Distributed Processing Symposium, New Orleans, LA, pp. 1082-1091 (May 2020).

CHAPTER 1

Introduction

Modern data centers consist of tens of thousands of machines. With such a large number of machines, during each hour of the day, there is a relatively high probability of failure of certain machines in the data center. For example, for a cluster of 1800 server machines in Google, typically there are 1000 individual machines failures in the first year of deployment [goo]. Data centers commonly host a large number of critical services. Hence, there is a need to deploy dependability mechanisms that prevent service failure due to hardware or software faults.

There are two approaches to increase the dependability of a system. One is to enhance the reliability of the applications by minimizing design faults and thus reducing the possibility of encountering failures. The other approach recognizes the fact that faults cannot be entirely eliminated and will eventually occur and cause system failures. This leads to the development of fault-tolerance mechanisms that allow the system and the applications to continue to operate correctly despite component failures.

A common method to reduce design faults is through the use of debugging tools to detect programming errors in the code. Fault tolerance mechanisms are either application-specific or application-transparent. Compared to their application-specific counterparts, application-transparent fault-tolerance mechanisms require only a one time development cost, avoiding an extra development cost for each new application. Application-transparent fault-tolerance mechanisms can be developed by enhancing the software infrastructure or the underlying hardware. This thesis presents a debugging tool that is designed to detect an important type of programming errors in parallel applications and several application-transparent fault-tolerance mechanisms based on the enhancement of the software infrastructure: hypervisors and containers.

Building dependable systems involves a tradeoff between soundness and overhead. Soundness represents the effectiveness of the dependability mechanisms and is measured by metrics such as detection rate and recovery rate. Overhead is the cost of building the dependable system and the cost of running applications under the dependable system. Overhead is measured by metrics such as development costs, throughput overhead, latency overhead, and resource overhead. Dependable systems for mainstream deployment are typically built upon commodity hardware with mechanisms that enhance resilience implemented in software. The goal of such dependable systems is to provide commercially viable, best-effort dependability cost-effectively. The focus of this thesis is on dependability mechanisms that can be deployed in this type of systems.

This thesis proposes several practical, low-overhead dependability mechanisms for critical components in the system: hypervisors, containers, and parallel applications. The approach towards building such dependability mechanisms involves first, identifying sweet spots in the design space that balance soundness and overhead. Second, novel use of hardware to optimize critical operations in the dependability mechanisms. Third, dedicated optimizations of the internals of operating systems/hypervisors. Based on these approaches, each of the proposed dependability mechanisms in this thesis reduces some aspects of overhead by orders of magnitude while still maintaining nearly the same level of soundness compared to directly-comparable prior works.

The rest of the chapter is organized as follows. Section 1.1 discusses a key tradeoff involved in dependability mechanisms. It argues that for mainstream deployment, it is appealing to trade a small reduction in soundness for a significant reduction in overhead. Section 1.2 discusses the use of existing hardware features and/or dedicated optimizations of the internals of operating systems and hypervisors for implementing dependability mechanisms.

Section 1.3 presents the contributions of this thesis. Section 1.4 presents the organization of the rest of the thesis.

1.1 A Key Tradeoff in Dependable Systems

As mentioned earlier, the two ways to enhance the dependability of a system is by minimizing design faults and building fault-tolerance mechanisms to allow applications to survive failures. Software bugs are a significant source of design faults. Unfortunately, some bugs, especially those in multithreaded applications, are hard to find manually since the manifestation of the bugs is highly dependent on deployed environments, event timing, and thread interleaving. This has motivated the development of debugging tools that aid programmers in finding such bugs. In some cases, debugging tools are used only during the development phase. If the overhead of the debugging tools is low enough, they can be used in deployed systems to further detect and diagnose bugs that escape detection in the development phase.

Ideally, a dependability mechanism should have the following two characteristics.

(1) Maximum soundness: For a debugging tool, this means that it should have no false positives (i.e., does not produce false alarms) nor false negatives (i.e., does not miss any bugs that may eventually lead to failure). For a recovery mechanism, this means that after a failure occurs, the mechanism should always result in a successful recovery.

(2) Minimal overhead: The mechanism should not rely on any customized hardware to avoid the design and manufacturing overheads for new hardware. It incurs minimal performance, memory, and resource overheads during normal operation. It incurs minimal service interruption when recovery is necessary. Minimizing overhead is critical for any fault-tolerance mechanism since for it to be effective, it needs to be on all the time. For a debugging tool, as discussed earlier, low overhead enables it to be deployed in production runs.

The design of dependability mechanisms involves a tradeoff between soundness and overhead. An extreme example is life-critical systems, such as those used to pilot airplanes. Such systems are typically built using custom software and hardware. They often adopt formal verification to minimize design faults in the system. The goal of such dependable systems is to provide an extremely high level of soundness. This is typically associated with a high level of overhead.

Dependable systems for mainstream deployment are built upon commodity hardware with mechanisms that enhance resilience implemented in software. Extensive testing, with or without debugging tools, is used to reduce the number of programming errors. The goal of such dependable systems is to provide commercially viable, best-effort dependability cost-effectively. Hence, for mainstream deployment, the sweet spot in the design space generally involves accepting a reasonable reduction in soundness in order to reduce design, implementation, and operational overheads.

An example of design tradeoff for mainstream deployment of dependability mechanisms involves debugging tools. A debugging tool that incurs false positives is often not acceptable to the programmers since it typically takes a significant amount of effort to investigate bugs in applications. Programmers are unwilling to spend such an amount of effort on potentially correct code. On the other hand, debugging tools that incur false negatives can be tolerated by the programmers. The reason is that, for any debugging tool, it is impossible to guarantee detection of all the bugs in one execution. This is because only a subset of the possible execution paths is exercised in any single execution. Hence, a common practice is to run the applications with the debugging tool multiple times with diverse inputs to ensure that most, if not all, of the bugs, are caught. As a result, for a debugging tool, although not ideal, it is likely to be acceptable that some bugs may be missed during any single execution as long as there is a high probability that the bugs are detected in multiple executions.

The above discussion suggests that, for debugging tools, in order to reduce overhead, there are at least two ways to reduce soundness that may be acceptable for mainstream deployment: (1) accept a low rate of false negatives, and (2) accept that the tool may fail to detect some bugs in a single execution, as long as there is a high probability of detecting them in multiple executions.

Another example of design tradeoff for mainstream deployment of dependability mechanisms involves fault-tolerance mechanisms. Every fault-tolerance mechanism is based on some assumptions regarding the *failure model* of system components – how the components behave as a result of faults. For example, the failure is fail-stop – the faulty component immediately stops before corrupting any data in storage or communicating with other components. No fault tolerance mechanism can guarantee recovery from all possible component failures. Furthermore, for any particular component failure model, it is typically the case that, as the recovery rate approaches 100%, each small increase in recovery rate involves significant increases in design, implementation, and operational overheads. Hence, it is often the case that, especially for deployment in mainstream systems, the sweet spot in the design space is at a lower recovery rate than for life-critical systems in order to lower the various overheads to commercially viable levels.

1.2 Commodity Hardware and Dedicated Privileged Software Support for Practical Dependable Systems

The dependability mechanisms developed in this thesis involve repurposing features in commodity hardware and/or implementing dedicated optimizations of the internals of operating systems and hypervisors. This section motivates this approach.

Modern computer systems have a rich set of hardware features for many different purposes. Some of these hardware features can be reused for optimizing the implementation of dependability mechanisms. For example, the memory management unit, which is originally used to virtualize the memory, is often used in the dependability mechanisms to track the dirty pages between checkpointing. As shown in this thesis, novel reuse of hardware features can significantly reduce the overhead of dependability mechanisms.

Privileged software, such as operating systems and hypervisors, plays a critical role in implementing dependability mechanisms. This is due to the following three reasons. Firstly, as discussed above, hardware features are often reused to implement dependability mechanisms. Privileged software manages and provides access to hardware. To exploit these hardware features, dependability mechanisms must interact with the privileged software. Secondly, applications often have certain state (e.g., file descriptor tables) maintained by the privileged software. A dependability mechanism often needs to interact with the privileged software to access, restore, or modify such state. Thirdly, dependability mechanisms often require changing the normal behavior of the privileged software. For example, many checkpoint-based recovery mechanisms require network traffics to clients to be buffered until the backup acknowledges the receipt of the corresponding checkpoint. In such a case, the privileged software needs to provide interfaces, in this example, to buffer and release network packets.

Unfortunately, commodity privileged software is often not tailored for implementing dependability mechanisms. Thus, it either does not provide interfaces to perform the operations discussed above or provides highly inefficient ones. Hence, the overhead of dependable systems can be significantly reduced, if the privileged software is enhanced to efficiently support these operations.

1.3 Thesis Contributions

Table 1.1: Summary of the four dependable systems presented in the thesis.

Mechanism	Fault/Failure Model	Protected Component
NiLiHype	Transient hardware and software faults	Hypervisors
NiLiCon	Fail-stop failures	Containers
HyCoR	Fail-stop failures	Containers
PUSh	Data race bugs	Applications

This thesis develops and evaluates four practical, low-overhead dependability mechanisms that protect critical components in the system as summarized in Table 1.1. Each of these mechanisms enhances the reliability of a different layer of the multilayer system architecture used in data centers.

1.3.1 *NiLiHype*

NiLiHype is a recovery mechanism that recovers hypervisors from failures due to transient hardware and software faults. *NiLiHype* is related to ReHype [LT11, LT14], a prior work that similarly recovers hypervisors from failures due to these faults.

ReHype is built upon a component-level recovery mechanism: *microreboot* [CKF04]. Upon a hypervisor failure is detected, ReHype preserves all the VMs and boots a new hypervisor instance. The state of the hypervisor is then updated to make it consistent with the states of the other system components (e.g., the guest VMs). This requires ReHype to reuse a significant amount of the state from the previous (failed) hypervisor instance. However, it has been shown that recovery success rates above 85% are achieved. Achieving such recovery rates despite reusing state from the failed instance is an indication that there is a relatively low probability of a fault corrupting state critical to the survival of the entire system.

A key drawback of ReHype is that for many important applications, the latency of the reboot results in unacceptably long service interruption. Motivated by this and the above observation that the critical state in the failed hypervisor is unlikely to be corrupted, this thesis proposes a component-level recovery scheme: microreset. With microreset, a failed component is reset to a quiescent state that is highly likely to be valid and where the component is ready to handle new or retried requests from the rest of the system. By avoiding the reboot step of microreboot, the recovery latency is dramatically reduced.

NiLiHype (Nine Lives Hypervisors) is an implementation of microreset on the Xen [BDF03] hypervisor. Both *NiLiHype* and *ReHype* incur negligible overhead during normal operation. During recovery, compared to *ReHype*, *NiLiHype* trades a small reduction in recovery rate (<2%) for a significant reduction in service interruption time. Specifically, the service interruption time is reduced from 713ms with *ReHype* to 22ms with *NiLiHype*, a factor of over 30x.

1.3.2 ***NiLiCon***

NiLiCon (Nine Lives Containers) is a mechanism for running duplicated containers, thus providing the ability to tolerate container fail-stop failures. Cloud computing typically relies on VMs or containers to provide an isolation and multitenancy layer [Ber14, Mer14, RG05]. Containers are often an attractive alternative to VMs since they have the benefits of smaller size, faster startup, and avoiding the need to manage updates of multiple VMs [Ber14, LKG15]. However, despite the advantages of containers, there has been very little work on fault-tolerance techniques for containers. To the best of our knowledge, *NiLiCon* is the first container fault-tolerance mechanism that is transparent to applications and clients and supports stateful applications.

NiLiCon applies the widely used VM replication technique: *Remus* [CLM08] to containers. Specifically, with *NiLiCon*, the applications run in a primary container, which is periodically paused every tens of milliseconds, so that its state can be checkpointed to a backup machine. If the primary container fails, the applications are started on the backup machine in a new instance of the container, from the checkpointed state.

A container has a much tighter state coupling between the container and the underlying kernel than between a VM and the underlying hypervisor. Specifically, there is much more container state in the kernel (e.g., the list of open file descriptors) than there is VM state in the hypervisor. Hence, a key challenge to implement *NiLiCon* is to efficiently checkpoint parts of the container state that are in the kernel. We overcome this challenge with various optimizations and among them, a key optimization is to enhance the kernel to identify the unchanged container state in the kernel since the previous checkpoint and skip the checkpointing of such state in the current checkpoint iteration.

We have validated the operation of *NiLiCon* and evaluated its overhead using seven benchmarks, five of which are server applications. The recovery rate with *NiLiCon* is 100% with a recovery latency of ~350ms. *NiLiCon* achieves performance that is competitive with *Remus*. Specifically, the performance overhead with *NiLiCon* is in the range of 19%-67% versus 13%-72% with *Remus*. During normal operation, the CPU utilization on the backup is in the range of 6.8%-40%.

1.3.3 ***HyCoR***

HyCoR builds upon *NiLiCon* and similarly provides the ability to tolerate container fail-stop failures. *HyCoR* addresses a fundamental disadvantage of Remus-based replication approaches [CLM08, LBV15, RZP19, WCJ18]: unacceptably long delay of outputs to the clients. Specifically, with *NiLiCon*, for consistency between the server applications and their clients after failover, outputs must be delayed and released only after the checkpoint of the corresponding epoch is committed at the backup. Since checkpointing is an expensive operation, for acceptable overhead, the epoch duration is typically set to tens of milliseconds. On average, outputs are delayed by half an epoch plus the time to take and transfer the checkpoint. This results in delays of tens of milliseconds.

HyCoR overcomes the above drawback by integrating deterministic replay and container checkpoint. Specifically, during normal operation, execution on the primary is divided into epochs and the primary state is checkpointed to an inactive backup at the end of each epoch. The primary also records its non-deterministic events. When the primary sends a reply to a client, the log containing the non-deterministic events is sent to the backup. Hence, the external outputs are delayed only by the time it takes to commit the relevant last portion of the log to the backup. In such a way, *HyCoR* decouples latency overhead from the checkpointing interval.

Upon failure of the primary, the backup begins execution from the last primary checkpoint and then deterministically replays the primary's execution of its last partial epoch, up to the last external output. The backup then proceeds with live execution.

To achieve a practical overhead, *HyCoR* records an incomplete set of non-deterministic events, namely the outcomes of synchronization operations and non-deterministic system calls, on the primary. Untracked non-deterministic events, such as those caused by data races, will cause the replay to fail and thus the recovery to fail. To overcome this challenge, *HyCoR* includes a simple timing adjustment mechanism that results in a high recovery rate for applications that contain data races, as long as their rate of unsynchronized write operations is low.

With *HyCoR*, various enhancements are added to the kernel to facilitate the implementation of deterministic replay, integration of deterministic replay and checkpointing, and preserving the network connection. We have evaluated *HyCoR* with eight benchmarks. Five of these benchmarks are specifically designed to stress *HyCoR*. *HyCoR* requires the applications to link with a modified glibc library that records, sends, and replays non-deterministic events. Furthermore, *HyCoR* incurs a small reduction in recovery rate (<0.5%) for applications with data races. In return, *HyCoR* has a much lower extra delay of outputs to clients (reduced from 36ms-51ms to 150-572μs) and much lower throughput overhead if the application is free of data races (2%-58% with *HyCoR* vs. 18%-139% with *NiLiCon*). For applications without data races, *HyCoR*'s recovery rate is 100% and it recovers within one second.

1.3.4 ***PUSh***

PUSh is a dynamic data race detector. Most of the competitive approaches [thr, SBN97, FF09, ZLJ16, SI09] are based on memory instrumentation to perform complex happens-before or lockset analysis on global memory accesses to detect racy memory access pairs. A key disadvantage of these mechanisms is that they incur prohibitive performance and memory overhead (often in the scale of hundreds of times), limiting them to be used only for debugging with small workloads. *PUSh* overcomes the disadvantage mentioned above with two core ideas. First, the detection of data races can be facilitated by requiring programmers to explicitly specify any intended sharing of global objects and then verifying compliance with these intentions. Second, hardware, in the form of page-level protection, instead of memory instrumentation, can be used to efficiently enforce the specified sharing policies.

PUSh prevents the sharing of global objects unless the programmer explicitly specifies sharing policies that permit it. Annotation can be added when an object is created to specify its sharing policy, such as *private*: read/write accessible by one thread, or *read-shared*: potentially readable by all the threads. Subsequently, *change policy annotations* can be used to change the sharing policy of objects. If the policy changes on the same objects are unordered, data races might escape detection. *PUSh* uses happens-before tracking of the policy changes to detect such a scenario.

PUSh uses the conventional memory management unit to enforce sharing policies and includes a key performance optimization that exploits memory protection keys (MPKs), a hardware feature recently added to x86 platforms. Several key optimizations that significantly reduce the performance and memory overhead are enabled by enhancing the memory management subsystem in the Linux kernel. A drawback of *PUSh* is that it requires the programmer to annotate the sharing policy of each object in the code. However, with our benchmarks, the annotation is mostly <5% of the overall lines of code. Furthermore, in rare cases, due to the limited number of protection domains supported by MPK, *PUSh* might miss a manifested data race. Fortunately, *PUSh* has a so-called *pseudo completeness* property: with a sufficient number of different executions, all data races will eventually be detected. A key advantage of *PUSh* is that, compared to the competitive tools, such as ThreadSanitizer(TSan), a widely used data race detector, *PUSh* has much lower performance overhead (0%-54% with *PUSh* vs 304%-36000% with TSan) and memory overhead (0%-5.8% with *PUSh* vs 54%-11000% with TSan). These results indicate that, in many deployment scenarios, *PUSh* can be used in production runs.

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 presents the background and related work for this thesis. It starts with a brief introduction to virtualized systems and fault models. It then presents background and related work of OS/hypervisor resilience, duplication, deterministic replay, VM/process replication, container migration, and data race detection.

Chapter 3 presents *NiLiHype*, a hypervisor recovery mechanism with extremely small (22ms) service interruption time during recovery. Chapter 4 presents *NiLiCon*, to the best of our knowledge, the first replication mechanism for commercial off-the-shelf containers. Chapter 5 presents *HyCoR*, a container replication mechanism enhancing *NiLiCon* with deterministic replay, that almost eliminates the unacceptably long delay on outputs to clients. Chapter 6 presents *PUSh*, a low overhead data race detector based on hardware-supported prevention of unintended sharing. Chapter 7 concludes this thesis and discusses the directions of future work.

CHAPTER 2

Background and Related Work

This chapter presents the background knowledge and the related work to facilitate the discussion for the rest of this thesis. The rest of the chapter is organized as follows. The dependability mechanisms presented in this thesis are aimed at protecting critical components in data center machines. Virtual machines and containers are the key components of the software infrastructure of data centers. Section 2.1 presents the background knowledge of virtual machines and containers. Section 2.2 presents the background knowledge of fault models and failure models, focusing on those assumed by the dependability mechanisms presented in this thesis. Related work on operating system/hypervisor dependability is presented in Section 2.3.

Section 2.4 presents an overview of fault-tolerance mechanisms based on duplication to facilitate the discussion of the rest of the chapter. Section 2.5 presents the background knowledge and related work of deterministic replay. Section 2.6 presents the background knowledge and related work of VM/process replication, which facilitates the discussion of the container replication mechanisms presented in this thesis. Section 2.7 presents prior work on container and process migration, with a focus on CRIU [cria], which is the starting point of the container replication mechanisms presented in this thesis. Section 2.8 presents related work on data race detection.

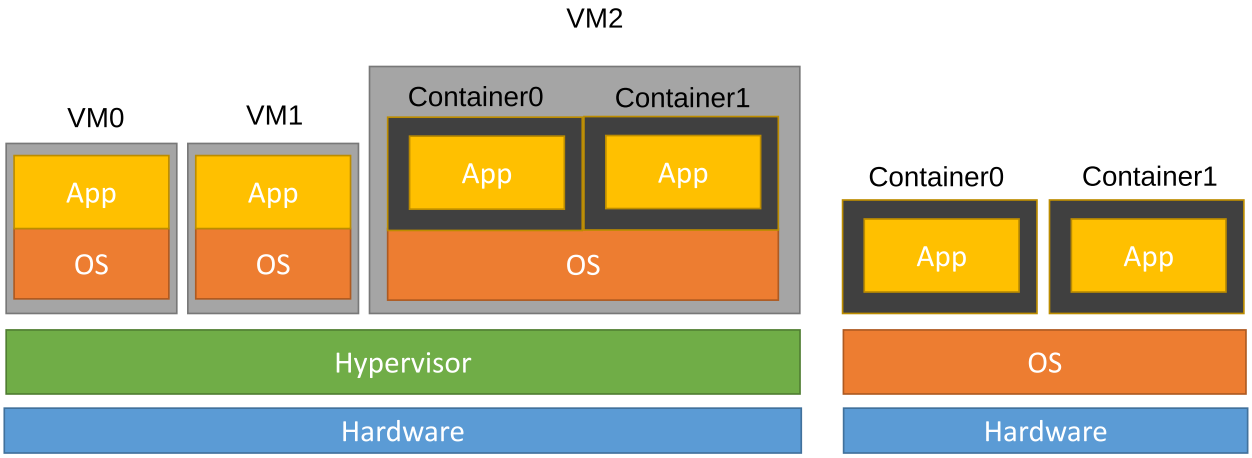


Figure 2.1: Virtualized systems. Left: a system based on hardware virtualization. Right: a system based on OS-level virtualization.

2.1 Virtualization

Figure 2.1 shows the overall architecture of a virtualized system. Two types of virtualization are shown in the figure. The left part of the figure shows hardware virtualization, that enables multiple VM instances to run on top of a single physical machine [RG05]. Each VM instance has its own operating system. A hypervisor is placed between the hardware and the VMs and is responsible for managing the VMs and virtualizing the underlying hardware resources for the VMs. The particular hypervisor discussed throughout this thesis is the Xen hypervisor [BDF03].

Another type of virtualization, as shown in the right part of the figure, is OS-level virtualization [Ber14]. OS-level virtualization enables multiple container instances to run on top of a single operating system. Each container instance has a separate namespace to provide the processes within it an illusion that they are the only processes running on the operating system. Furthermore, the operating system can be configured to limit the hardware resources allocated to and seen by the processes in a container. Throughout this thesis, the OS-level virtualization is provided by the Linux kernel.

Both a container and a VM can be used as an encapsulation unit that contains applications and the runtime environment to run the applications. This greatly simplifies the deployment and management of applications. Furthermore, both containers and VMs can be used to provide an isolation and multitenancy layer on top of a physical machine and thus to enable server consolidation [Ber14, Mer14, RG05]. Compared to VMs, containers have the advantages of lower virtualization overhead and shorter boot/shutdown time [Ber14, LKG15]. On the other hand, VMs provide better isolation and are thus potentially more secure. Sometimes, as shown in Figure 2.1, containers and VMs can be used together by running containers inside a VM. In this case, applications from the same party can be placed into multiple containers, and then these containers are placed in the VM. Thus, the VM provides strong isolation to protect these applications from other applications of untrustworthy parties co-located in the same physical machine. The containers allow applications deployed in them to still enjoy the benefits of encapsulation.

2.2 Fault/Failure Models

This thesis is focused on enhancing reliability for critical components in the system. This section presents the relevant background knowledge of fault models and failure models to facilitate the discussion for the rest of the thesis.

Faults can be classified as hardware faults and software faults [LBK90]. Based on the frequency of the occurrence, a hardware fault can be *permanent*, *intermittent*, and *transient*. A *permanent* hardware fault persists until the faulty component is fixed or replaced. An *intermittent* hardware fault occurs at random intervals. A *transient* hardware fault occurs once and then disappears forever.

Software faults are programming errors in the code. Examples of software faults are buffer overflow/underflow and use-after-free bugs. This thesis is focused on two important types of software faults: data race bugs [SBN97] and Heisenbugs [Gra86]. A data race is caused by two threads that simultaneously access the same memory location without proper synchronization and at least one of the accesses is a write. Old programming language standards allow such behavior and programmers sometimes intentionally develop code with data races to, for example, perform customized synchronization operations. However, it is often the case that data races are caused by unintentional sharing and thus they are good indicators for concurrency bugs. Modern language standards define all data races as bugs [ISO11].

Another type of software faults is Heisenbugs [Gra86], which occur only under particular timing and/or thread interleaving in the system. Similar to a transient hardware fault, a software Heisenbug is unlikely to recur when the program is executed again.

A fault may lead to a system failure. A failure model describes the behavior of the system as a result of a fault. One extreme case is a *fail-stop failure* [Sch84], where the fault halts the system before it causes externally visible erroneous behaviors, such as writing corrupted data to storage or sending incorrect messages to other components. Another extreme case is a *Byzantine failure*. In such a case, the fault eventually causes the system to behave arbitrarily.

2.3 Enhancing the Dependability of OSes/Hypervisors

A critical step in most fault tolerance mechanisms is that, after an error is detected, the system is restored to a valid state. This state restoration is referred to as *recovery*. In many cases, a simple yet effective recovery mechanism is to reboot the entire system since this brings the system to a consistent, frequently tested, and well-understood state. However, naively rebooting the entire system has two drawbacks: (1) it incurs a long service interruption time during recovery, and (2) it causes all the volatile state (e.g., those stored in memory) in the system to lose.

To overcome these drawbacks, Candea et al. propose microreboot [CKF04]. Contrary to a whole system reboot, microreboot only reboots the failed component of the system and then reintegrates it back to the system. There are three requirements from the target systems for microreboot to be effective. First, components in the system need to be loosely coupled. Specifically, the only way for one component to affect other components is through narrow, well-defined interfaces. In such a system, faults that occur in one component are highly unlikely to propagate to other components. Second, all the operations in the system must be transactional to prevent state inconsistencies upon recovery. For example, a failure might occur when a component is in the middle of handling a request from another component. If the component that handles the request fails and the request handler is not transactional, after recovery is attempted, state changes performed by the request handler prior to failure may leave the system in an inconsistent state, resulting in recovery failure. Last, failed requests can be retried transparently to hide failures from other components.

Otherworld [DS10] applies microreboot to the Linux kernel, where the kernel is viewed as one of the components in the entire system. The state of the running processes is preserved in place. After rebooting a new kernel instance, Otherworld rebuilds many kernel data structures associated with each process, such as the process descriptor, the file descriptor table, and signal handler descriptors. Restoration of kernel data structures requires traversing many complex data structures in a possibly corrupted kernel, increasing the chance of failed recoveries. In many cases, user-level processes require custom crash procedures in order to properly resume execution.

Similar to Otherworld, ReHype [LT11, LT14] applies microreboot to the Xen hypervisor [BDF03]. Upon the detection of a hypervisor failure, ReHype pauses all the VMs and boots a new hypervisor instance. Critical state in the old hypervisor is preserved and is reused in the new hypervisor to allow the hypervisor to continue managing existing VMs. After the reboot, ReHype resolves various inconsistencies between the new hypervisor and the preserved components in the system. For example, it acknowledges all pending interrupts in all processors to avoid blocked interrupts after recovery. Finally, it unpauses all the VMs to finish the recovery.

A hypervisor maintains less state for each VM compared to the state for each process maintained by the kernel. This makes the microreboot of a hypervisor simpler than of kernel and increases the chance of a successful recovery [LT11].

RootHammer [KC07] uses microreboot to rejuvenate virtualized systems based on Xen. It reduces the time for this rejuvenation by rebooting only the Xen hypervisor and the privileged VM (PrivVM, also known as Dom0) [BDF03]. The PrivVM is a VM that performs management operations, such as creating and destroying VMs. It may also host the device drivers for the I/O devices in the system and facilitate their sharing among the application VMs. RootHammer preserves in memory the states of VMs and their configurations while rebooting the hypervisor and the PrivVM. During rejuvenation, the PrivVM is properly shut down and the VMs suspend themselves cleanly. Hence, RootHammer operates within a healthy and functioning system. RootHammer is not designed to recover from failure and thus does not deal with possible arbitrary corruptions and inconsistencies in the system.

There are works focusing on ways to partition the kernel or the hypervisor, isolate the partitions (fault domains) from each other, and recover failed partitions without requiring a full system reboot [BVK16, DCC08, HBG06, LAK09, NB13]. In many cases this is facilitated by an underlying design, based on a small microkernel and a collection of drivers and servers isolated from each other by the memory-management hardware [BVK16, DCC08, HBG06].

VirtuOS [NB13] proposes *vertical slicing* of the Linux kernel to service domains. The encapsulation of the slices is performed using virtualization based on Xen. Requests from user processes interact directly with the appropriate service domain, which is isolated from the rest of the kernel. Since VirtuOS requires virtualization, applying the idea to a hypervisor would require nested virtualization with its associated overheads.

Akeso [LAK09] dynamically partitions the Linux kernel into request-oriented recovery domains. A recovery domain is formed by the execution thread that handles a request, such as a system call or interrupt. A modified compiler is used to instrument the code to track state changes caused by the domain as well as dependencies among domains. When an error is detected, the affected domain and dependent domains are rolled back. Due to the code instrumentation, the performance overhead of Akeso is between 8% and 560%.

Yoshimura et al. [YYK11b, YYK12] proposes the idea that it is possible to recover from some errors in the Linux kernel without reboot. Their recovery scheme always involves killing a running process. This achieves a recovery success rate of 60%.

TinyChecker [TXC12] proposes the use of nested virtualization to manage hardware enforced protection domains during hypervisor execution. TinyChecker is a small hypervisor that runs underneath the commodity hypervisor, monitors transitions between the commodity hypervisor and the VMs it hosts, limits the memory regions writable during hypervisor execution, and attempts to detect accesses that are possibly erroneous and take checkpoints to facilitate recovery. TinyChecker has never been fully implemented or evaluated. In a full implementation, the nested virtualization mechanism is expected to involve significant overhead.

There are works that use virtualization to provide resilience to device driver failures [JKJ10, LGT09] and the ability to recover from PrivVM failures [LT12]. The hypervisor, device drivers, and PrivVM together form the *virtualization infrastructure* (VI). The resilient VI can be combined with appropriate middleware to provide, on a virtualized system, a resilient platform for applications and services [LHT11, LT14].

2.4 Fault Tolerance Based on Duplication

Duplication is one of the foundational fault-tolerance mechanisms and has been in use for many decades. One way to achieve duplication is through active replicas. With active replicas, a primary replica and a backup active receive identical inputs in the same order. In one form of active replication, the outputs from two replicas are continuously compared and any mismatch signals an error. This can be used to detect Byzantine replica failures (§2.2). In such a case, an additional mechanism is needed for recovery. Another form of active replication relies on failed replicas to be *fail-stop*. In this case, there is no need to compare the outputs and recovery from a primary replica failure is achieved by the backup replica taking over. The duplication mechanisms developed in this thesis target fail-stop replicas.

A key requirement for active replication is that the replicas are deterministic so that execution on the backup precisely tracks the primary. Unfortunately, in reality, there are many sources of non-determinism in a system. This greatly complicates the use of active replications in many deployment scenarios. One approach for overcoming this challenge is to force the backup replica to mirror the execution of the primary replica through the use of deterministic replay (§2.5). Another approach replaces the active backup replica with a "warm spare" backup replica. In this case, the backup replica state is kept almost, but not fully, in sync with the primary replica through high-frequency asynchronous incremental checkpointing. In this case, the backup replica is active only if the primary backup fails and thus non-determinism is not a problem. This approach is proposed by Remus [CLM08] in the context of VM replication (§2.6). We discuss these two approaches in the following sections.

2.5 Deterministic Replay

Deterministic replay is the reproduction of some original execution in a subsequent execution [CZG15]. With deterministic replay, during the original execution, the results of non-deterministic events are recorded in a log. This log is used in the subsequent execution. There are two forms of deterministic replay: offline replay and online replay. With offline replay, the replay is performed after the original execution has completed. This can be used for debugging or forensics. With online replay, the replay proceeds concurrently with the original execution. An important use case for online replay is to provide fault-tolerance based on active replication (§2.4). Specifically, in such a case, the primary replica records non-deterministic events and sends them to the backup replica. The backup replica replays non-deterministic events based on the log [BS95, GHY14, MGT17, LWV10]. With this, the determinism requirement for active replica is satisfied.

With a uniprocessor, non-deterministic events include asynchronous events, such as interrupts; system calls, such as *gettimeofday*; and inputs from the external world. These kinds of non-deterministic events occur rarely. Thus, uniprocessor deterministic replay systems [DKC03, KDC05, SKA04, GWT08, OJF17] have a low overhead. Deterministic replay on uniprocessors is considered a solved problem [CZG15, OJF17, VLW11].

Shared-memory multiprocessors introduce a new type of non-deterministic events: the order of memory accesses to the same memory location performed by different processors. Such non-deterministic events occur at a high frequency. A straightforward approach that instruments memory accesses to record these non-deterministic events incurs prohibitive overhead. Various works have been proposed to address this challenge. Some of them require developing custom hardware [XBH03, HH08, MCT08, NPC05]. Others rely on utilizing existing commodity hardware, for example, the memory management unit [DLF08, LVN10, CC13] and the hardware virtualization extensions [RTL16]. Nonetheless, without custom hardware support, the performance overhead of deterministic replay on multiprocessor systems is prohibitive.

A common practical approach supporting deterministic replay on multiprocessors is to require the programs being replayed to be data-race-free [OAA09, RB99]. If this condition is met, as long as the results of synchronization operations are deterministically replayed, any ordering of shared memory accesses that can affect execution results is preserved. This incurs much lower overhead since the frequency of synchronization operations is much lower than normal memory accesses.

There are various works aimed at achieving full deterministic replay in the presence of data races even though only results of synchronization operations are recorded [AS09, PZX09, LWV10]. ODR [AS09] adopts an offline search algorithm to find the order of shared memory accesses that satisfies the recorded results of synchronization operations and produces the recorded outputs. PRES [PZX09] detects data races during replay. It then enumerates the order of memory accesses in the detected data races to find one possible order that leads to a successful replay.

Respec [LWV10] is an online deterministic replay system. With Respec, to handle replay failures caused by data races, the execution of the primary replica is divided into epochs and a checkpoint is taken before an epoch starts. A replay failure caused by data races is detected if the backup replica executes an unlogged system call or the state of the two replicas is different at the end of an epoch. In such a case, the two replicas are rolled back to the checkpoint of the previous epoch and then continue with a conservative thread scheduling where only one thread is allowed to execute at a time.

The recording of non-deterministic events can occur at different levels: hardware [XBH03, HH08], hypervisor [DKC03, KDC05, DLF08], operating system [GWT08, LVN10], or library [RB99, OAA09]. Without dedicated hardware support, the recording must be done in software. It is advantageous to record the events at the user level, thus avoiding the overhead for entering the kernel or hypervisor [LWV10].

The degree to which the replay must recover the details of the original execution depends on the use case [CZG15]. To support seamless failover with replication, it is sufficient to provide *externally deterministic replay* [LWV10]. This means that, with respect to what is visible to external clients, the replayed execution is identical to the original execution. Furthermore, the internal state at the end of replay must be a state that corresponds to a possible original execution that could result in the same external behavior. This latter requirement is needed so that the replayed execution can transition to consistent live execution at the end of the replay phase.

With deterministic replay, to support seamless failover with replication, outputs to the

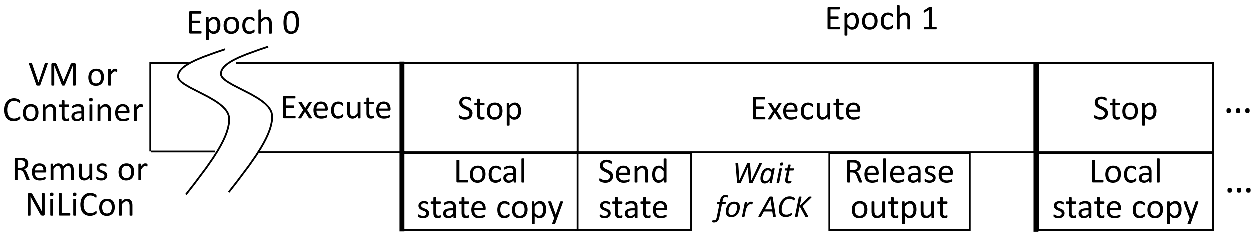


Figure 2.2: Workflow of Remus on the primary VM.

external world must be buffered for a short amount of time before they can be released. This is because, before releasing any output, the backup replica must ensure it can replay to a state consistent with the released output. Hence, outputs must be buffered until the backup replica has received the corresponding non-deterministic event log.

2.6 VM/Process Replication

This section presents prior works related to replication, with a specific focus on VM/process replication. As discussed in Section 2.4, another approach to achieve duplication is through high-frequency asynchronous incremental checkpointing to a warm spare. This approach is first proposed by Remus [CLM08], in the context of VM replication. In this section, we first present in detail Remus [CLM08]. We then present other related works on VM/process replication.

2.6.1 VM Replication with Remus

Figure 2.2 shows the workflow of Remus. With Remus, there is a primary VM that executes the applications and a backup VM that receives periodic checkpoints so that it can take over execution if the primary VM fails. As shown in Figure 2.2, processing on the primary VM consists of a sequence of epochs (intervals). Once per epoch, the VM is paused and changes in the VM state (incremental checkpoints) since the last epoch are copied to a staging buffer (*Local state copy* in Figure 2.2). The primary VM then resumes execution while the content of the staging buffer is concurrently transferred to the backup VM (*Send state*). In order to identify the changes in VM state since the last state transfer, during the *Pause* interval of each epoch all the pages within the VM are set to be read-only. Thus, an exception is generated the first time that a page is modified in an epoch, allowing the hypervisor to track modified pages.

A key issue addressed by Remus is the handling of the output of the primary VM to the external world. There are two key destinations of output from the VM: network and disk. For the network, incoming packets are processed normally. However, outgoing packets generated during the *Execute* phase are buffered. The outgoing packets buffered during an epoch, k, are released (*Release output*) during epoch $k + 1$, once the backup VM acknowledges the receipt of the primary VM's state changes produced during epoch k. The delay (buffering) of outputs is needed to ensure that, upon failover, the state of the backup is consistent with the most recent outputs observable by the external world. Due to this delay, in order to support client-server applications, the checkpointing interval is short — tens of milliseconds.

Remus handles disk output as changes to internal state. Specifically, the primary and backup VMs have separate disks, whose content are initially identical. During each epoch, reads from the disk are processed normally. Writes to the disk are directly applied to the primary VM's disk and asynchronously transmitted to the backup VM. The backup VM buffers the disk writes in memory. Disk writes from an epoch, k, are written to the backup disk during epoch $k + 1$, after the backup receives all the state changes performed by the primary VM during epoch k.

2.6.2 Related Work on VM/Process Replication

Early works on VM replication are based on deterministic replay (§2.5) and are limited to uniprocessor systems. Specifically, hypervisor-based fault-tolerance (HBFT) [BS95] leverages online deterministic replay to let the primary VM record non-deterministic events and send them to an active backup VM for replay. To simplify replaying interrupts, HBFT buffers interrupts and only delivers them until the end of an epoch. Another work [CSX16] similarly requires the primary VM to record non-deterministic events and send them to the backup. However, to reduce resource usage, instead of executing an active VM in the backup, the primary VM periodically performs checkpointing and sends the checkpoint image to the backup.

It is challenging to extend these works to multiprocessor systems with low overhead. This is due to the cost of recording non-deterministic events caused by shared memory accesses with multiprocessors. While the technique of recording the outcomes of synchronization operations instead of shared memory accesses (§2.5) can be leveraged to reduce the record overhead, this requires the application to be free of data races. More importantly, applying this technique to VMs would be complicated since there would be a need to track and replay non-deterministic events in the kernels of the primary and backup VMs, respectively.

Realizing the above challenge, Remus [CLM08] achieves VM replication on multiprocessor systems based on high frequency incremental checkpointing to a warm backup as discussed above. Many of the follow-up works of Remus are focused on reducing the performance overhead incurred by Remus.

Phantasy [RZP19] uses Intel's page-modification logging to reduce the overhead of tracking dirty pages. It further uses RDMA to speculatively transfer the content of dirty pages during execution to reduce the pause time due to checkpointing. Phantasy reduces the throughput overhead of Remus and because of this, Phantasy can sustain a shorter checkpointing interval to further reduce the latency overhead.

Tardigrade [LBV15] uses the Remus mechanism, but operates with *lightweight* VMs (LVMs). The use of LVMs significantly reduces the amount of state transferred between the primary and backup for each epoch, thus allowing higher frequency checkpointing and resulting in a shorter delays for outgoing packets. An LVM typically runs only the application and a library OS (LibOS), which translates and forwards system calls to the host OS. An LVM is similar to a container in that it also does not include a privileged software component, has a separate namespace, and only contains the target application. Compared to a container, an LVM has looser coupling with the host OS kernel since: (1) the LibOS maintains in the LVM state that for a container is maintained in the kernel, and (2) the interface (ABI) between the LibOS and the host kernel is narrower.

A disadvantage of Tardigrade is that it requires deploying a LibOS, not currently a common part of the software stack, potentially resulting in compatibility problems. A severe limitation of Tardigrade is that it breaks all the established TCP connections upon failover since the ABI does not provide a way to access the TCP stack in the host OS.

COLO [DYJ13] reduces the performance overhead by deploying active replication. Inputs to the primary VM are also transferred to the backup VM. Outputs from the primary VM and the backup VM are compared. If there is a match, one copy of the outputs is released immediately. If the outputs differ, state synchronization is performed. Compared to Remus, with some workloads COLO requires less data (state) to be transferred between the primary and backup. Furthermore, when the primary and backup outputs match, the only delay of outgoing packets is for the comparison, far less than the buffering delay with Remus. A key downside of COLO is that, for largely non-deterministic workloads, mismatches are frequent, resulting in prohibitive overhead.

PLOVER [WCJ18] optimizes Remus with active VM replication to reduce the size of the transferred state, which is mainly the dirty pages generated in an epoch. Specifically, PLOVER duplicates inputs, enforces a total order on the inputs, and sends them to both primary VM and the backup VM. The state of the two VMs are synchronized when they are both idle. Only the dirty pages whose content are different in the two VMs are sent from the primary VM to the backup VM.

Rex [GHY14] and Castor [MGT17] use online deterministic replay (§2.5) to perform active replication of multithreaded processes. To avoid prohibitive performance overhead, with Rex and Castor, the outcomes of synchronization operations, instead of shared memory accesses, are recorded. As a result, untracked non-deterministic events, such as data races, can lead to replay failure in the backup. Rex [GHY14] requires the application to be free of data races. Castor [MGT17] handles data races by buffering the output to the external world until the backup finishes replaying the associated log. If a divergence, due to a data race, prevents the backup from continuing replay, the primary's state is transferred to the backup and overwrites the backup's state.

2.7 Container/Process Live Migration

Container/process checkpointing mechanism can be used as a basic building block to implement migration. Specifically, migration requires the state of a container or a process to move from one physical machine to another. This can be achieved by checkpointing the container or the process state on the source physical machine, transferring the checkpoint image to the destination physical machine and finally restoring the container or the process state in the destination machine. Checkpointing can also be a basic building block in the implementation of replication. Specifically, as discussed in prior sections (§2.4, §2.6), replication can be implemented by high-frequency checkpointing.

This section first presents CRIU (Checkpoint/Restore In Userspace) [cria], a container checkpointing tool, which is the starting point for the implementation of the container replication mechanisms discussed in this thesis. We then briefly present related work on process checkpoint/migration.

CRIU is a tool that can checkpoint and restore complicated real-world container state on Linux. Obviously, user-level memory and register state of the container are checkpointed. Additionally, due to the tight coupling between the container and the underlying operating system, there is critical container state, such as opened file descriptors and sockets, within the kernel that must be checkpointed. For checkpointing and restoring in-kernel container state, CRIU relies on kernel interfaces, such as the *proc* and *sys* file systems, as well as system calls, such as *ptrace*, *getsockopt*, and *setsockopt*. This requires CRIU to run as a privileged process within the root namespace.

In order to obtain a consistent state, CRIU ensures that the container state does not change during checkpointing. This is done utilizing the kernel's *freezer* feature that sends virtual signals to all the threads in the container, forcing them to pause. For threads executing system calls, the virtual signal forces a return from the system calls, as if they are interrupted by a normal signal.

Without significant kernel modifications or prohibitive overhead, parts of container state can only be obtained from within the processes being checkpointed. This includes the timers, signal mask, and memory contents. Hence, CRIU maps (injects) a code segment (*parasite code*) into each of the processes being checkpointed (using *ptrace*). The parasite code communicates with the CRIU process via pipes and processes requests, such as to obtain the signal mask.

CRIU can checkpoint and restore established TCP connections using a socket *repair mode* supported by the Linux kernel [TCP]. When a process places a socket in *repair mode*, it can get/set critical state that cannot be otherwise accessed. This includes sequence numbers, acknowledgment numbers, packets in the write queue (transmitted but not acknowledged), and packets in the read queue (received but not read by the process).

CRIU supports incremental checkpointing of the user-space memory state. It identifies the memory pages modified since the last checkpoint using a kernel feature called soft-dirty pages [sof]. A process writes to a special file (*/proc/pid/clear_refs*) to cause the kernel to start the tracking of modified pages and reads from another file (*/proc/pid/pagemap*) to identify the pages modified since the beginning of the tracking.

There are other wide variety of works on process checkpointing/migration [BW89, AAC09, RR81, DO91, MDP00]. However, with these works, it is difficult to avoid potential resource naming conflicts upon process restore/migration as well as to identify and isolate all the necessary state components that need to be checkpointed. Zap [OSS02] addresses the limitations of process checkpointing/migration by allocating separate namespaces for groups of processes. This approach, in its essence, is the same as the namespace implementation for Linux containers.

2.8 Data Race Detection

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. The problem of precisely detecting data races is known as an NP-hard problem [NM92]. This is because a complete solution requires enumeration of all possible interleavings among threads.

One way to detect data races is based on static analysis of the application code. The benefits of this approach include no extra overhead during normal operation and the ability to find data races in rarely executed code paths. Static data race detectors without any false positive nor false negative can be developed for programming languages that are carefully designed to facilitate the detection of data races [Gro03]. However, it is hard for static analysis to deal with common programming language features such as dynamic memory allocation, pointer arithmetic, and syntactically unscoped synchronization operations. Some data race detectors require special annotations by the programmer and can only be applied to certain languages whose syntax is more restricted, such as Java [BLR02]. Static data race detectors for programming languages with less restricted syntax, such as C, suffer from high false positives and false negatives [EA03, PFH06, VJL07].

Another way to detect data races is to execute the program and then identify those racy memory accesses that occur during the execution. We refer to this type of data race detectors as dynamic data race detectors. Some of the dynamic data race detectors are based on happens-before analysis [Lam78]. Specifically, the detector instruments global memory accesses of the program. Happens-before analysis is performed on each memory access to identify racy memory accesses [FF09, SI09]. These data race detectors do not generate false alarms and can detect most of the manifested data races during the execution. However, they suffer from high performance and memory overhead during normal operation.

Another type of dynamic data race detectors replaces happens-before analysis with lockset analysis [SBN97, EQT07, CLL02]. Specifically, lockset analysis assumes that all shared variables are protected by a lock and the algorithm checks whether a thread is holding the correct lock when it accesses a shared variable. Lockset analysis is much simpler than happens-before analysis and can thus lead to lower performance overhead. However, a fundamental disadvantage of lockset analysis is that it incurs false positives. This is because, for example, with lockset analysis, memory accesses to a shared variable properly synchronized by a barrier are mistakenly flagged as data races. Some enhancements are proposed to reduce the false positives of the lockset analysis [SBN97] but do not eliminate them.

There are a large number of works aiming at reducing the performance overhead of dynamic data race detectors. Some of them adopt static analysis to eliminate unnecessary memory instrumentation [CLL02, EQT07]. FastTrack [FF09] proposes a new algorithm to simplify happens-before analysis. RaceTrack [YRC05] adaptively changes the granularity of detection during execution. MultiRace [PS03] uses page-level protection hardware to detect the first access to a shared page in each time frame. Aikido [OZK12] is a memory instrumentation framework that maintains a per-thread page table to monitor pages that are shared among threads and based on this, it only instruments memory accesses to these shared pages. As shown in the paper, Aikido can be used to speed up dynamic data race detectors. TxRace [ZLJ16] speeds up data race detection with hardware transactional memory in commodity Intel x86 processors. IFRit [ELC12] detects data races based on region overlapping.

Despite these optimizations, dynamic data race detectors still suffer from high performance overhead. Specifically, for IFRit [ELC12]; TxRace [ZLJ16]; FastTrack [FF09] and ThreadSanitizer [SI09, thr], with the data reported in [LLZ18], with eight threads, for applications from PARSEC benchmarks [Bie11]: *blackscholes*, *streamcluster*, *swaptions*, and *ferret*, these mechanisms incur performance overheads of 1.82x-79.2x. For *streamcluster* with 32 threads, the performance overhead for these mechanisms are 20x-400x [LLZ18].

Some dynamic data race detectors rely on the programmers to specify the sharing policies of the shared objects through annotations. The detectors then catch the memory accesses that violate the specified sharing policy to detect data races. Examples of the sharing policies are *read write accessible to a thread (private)* or *read-only among all the threads (read-only).* These works also provide a way for programmers to annotate the changes of the sharing policies during execution. Examples of these works are SharC [AGE08], Shoal [AGN09] and a work that we refer to as DCOP [MHC10] (Dynamically Checking Ownership Policies). Since checking whether a memory access violates the specified sharing policy is much faster than the happens-before or lockset analysis, the annotation-based approaches incur lower performance overhead at a cost of extra burdens on programmers due to annotation.

The annotation-based works are typically a combination of static analysis and dynamic memory instrumentation. DCOP [MHC10] instruments memory accesses to enforce the specified sharing policies in the runtime. Static analysis is used to eliminate unnecessary instrumentation of memory accesses.

To reduce the performance overhead, SharC uses static analysis to enforce the *private* and *read-only* policies. Other sharing polices are enforced during runtime by instrumenting memory accesses to those objects. To facilitate static analysis, the following requirements are imposed: (1) the sharing policy of an object can only be changed when its reference count is 1, and (2) after the policy change, the pointer pointing to an object whose sharing policy is changed is set to NULL. These requirements make SharC difficult to apply to programs containing data structures such as linked lists and trees.

Shoal [AGN09] partially mitigates the impact of SharC's restrictions by introducing the concept of *groups* of objects whose sharing policy can be atomically changed. Shoal also adds the *barrier* sharing policy, which allows an object to be either read only or write by one thread between two barrier operations. However, there remain restrictions on dynamic policy changes, which increase the burden on programmers. Furthermore, for both SharC and Shoal, there is a need for a runtime mechanism to track object reference counts and this incurs a performance overhead of up to 30% [AGN09].

Some dynamic data race detectors are aimed at performing data race detection during production runs. These works rely on sampling-based approaches. Specifically, with sampling-based approaches, only a small portion of the memory accesses performed by the application are used to detect data races [MMN09, EMB10, SVE11, ZJL17]. Obviously, this approach will miss some of the data races that occur during execution. The amount of missed data races depends on the average sampling frequency. However, to keep the performance overhead low, the average sampling frequency cannot be too high. For example, with RACEZ [SVE11], an average sampling frequency of every 20,000 retired instructions can incur an overhead of 30%.

Finally, there are also works developing custom hardware to perform data race detection. These works are mostly based on modifying memory caches. Specifically, these works store metadata in cache lines and piggyback information on coherence protocol messages of the cache to detect data races. HARD [ZTZ07] implements lockset analysis while ReEnchant [PT03], CORD [Prv06], and RADISH [DWS12] implement happens-before analysis. Lucia et al. propose conflict exceptions that detect data races in synchronization-free regions among threads [LCS10]. Pacman detects asymmetric data races [QON12]. Unlike the above works, SigRace does not modify the cache and detects data races by using hardware memory signatures [MSQ09]. Compared to software-based dynamic data race detectors, works based on custom hardware incur a much lower performance overhead.

CHAPTER 3

NiLiHype: Fast Hypervisor Recovery with Microreset

With hardware virtualization (§2.1), errors that occur during the execution of one of the virtual machines (VMs) due to a transient hardware fault or software fault are highly likely to be confined within that particular VM. Hence, other VMs are not affected. However, if such errors occur during the execution of hypervisor code, the resulting hypervisor failure leads to the failure of all the VMs on the host, and thus, potentially, to a significant impact on datacenter operation. Thus, as explained further below, there is strong motivation to develop fault tolerance mechanisms that allow the VMs to survive across hypervisor failure [LT11, LT14, TXC12]. A critical step in most fault tolerance mechanisms is that, after an error is detected, the system is recovered to a valid state. This chapter presents a hypervisor recovery mechanism: *NiLiHype* (Nine Lives Hypervisor).

VM replication [CLM08] (§2.6) is often used to provide fault tolerance with respect to VM failures, including with commercial products, such as VMware's vLockstep [VMw]. If the replicas are running on different hosts, failures during the execution of the hypervisor are also covered. Obviously, the resource overhead during normal operation is high; too high for many deployment scenarios. A decision *not* to use VM replication implies that the loss of any one specific VM is tolerable. For example, this might be the case when a VM is providing web service [ZZ03]. However, the loss of all the VMs on a host has more impact since it leads to the unavailability of a larger fraction of datacenter capacity, which is particularly significant in a small datacenter. Furthermore, even if VM replication is used, there are performance benefits as well as greater flexibility in VM placement, if the replicas are on the same host [RHK06]. Placing replicas on the same host can only be considered if an error during hypervisor execution is unlikely to lead to a hypervisor crash.

In order to prevent the hypervisor from being a single point of failure, the system must support recovery from hypervisor failure while preserving the hosted VMs. This has been done in *ReHype* [LT11, LT14]. With ReHype, upon detection of an error in the hypervisor, *microreboot* [CKF04] of the hypervisor is performed, while allowing other system components to maintain their states. The overhead during normal operation is small and essentially no work is lost when recovery is performed.

ReHype involves booting a new hypervisor instance. The state of the hypervisor is then updated to make it consistent with the states of the other system components (e.g., the guest VMs) [LT11, LT14]. This requires reusing a significant amount of state from the previous (failed) hypervisor instance. Obviously, this reused state is potentially corrupted. However, it has been shown that recovery success rates above 85% are achieved. Achieving such recovery rates despite reusing state from the failed instance is an indication that there is a relatively low probability of a fault corrupting state critical to the survival of the entire system.

Fault injection studies of the Linux kernel [YYK11b, YYK12] have shown that the errors caused by most faults affect "process local" state as opposed to state that can impact the kernel itself or other processes. Based on this motivation, recovery of the Linux kernel without reboot was proposed. For a subset of possible faults (those leading to kernel *oops*), a simple recovery scheme was proposed that involves aborting the faulting process. This yielded a recovery rate of approximately 60%. These results for the Linux kernel together with the ReHype results discussed in the previous paragraph raise the possibility that recovery from hypervisor failure may not require booting a new hypervisor instance.

For many important applications, the latency of the reboot step of microreboot results in unacceptably long service interruption. This chapter investigates component-level recovery of a hypervisor without reboot. Using a mechanism which we call *microreset*, a failed component is reset to a quiescent state that is highly likely to be valid and where the component is ready to handle new or retried interactions with the rest of the system. By avoiding the reboot step of microreboot, the recovery latency is dramatically reduced.

We implement and evaluate microreset for the Xen hypervisor [BDF03], using a mechanism which we call *NiLiHype*. We show that NiLiHype and ReHype have the same relatively small overhead during normal operation. NiLiHype achieves a successful recovery rate of over 88%, slightly lower than ReHype's rate of over 90%. However, NiLiHype performs the recovery more than 30 times faster than ReHype. NiLiHype's recovery latency is low enough (22ms) that service interruption is negligible in most deployment scenarios. Considering this recovery rate vs. recovery latency tradeoff, NiLiHype is an attractive point in the design space.

We make the following contributions: 1) present the design of microreset-based component-level recovery as an alternative to microreboot; 2) describe the implementation issues involved in converting the microreboot-based ReHype to the microreset-based NiLiHype; 3) use fault injection to evaluate the recovery rate of NiLiHype for different fault types and workloads; 4) evaluate the hypervisor processing overhead of NiLiHype during normal operation, NiLiHype's recovery latency, and its implementation complexity.

The next section presents microreset and compares it to microreboot as a technique for component-level recovery. The basic operation of Xen hypervisor recovery using ReHype and NiLiHype is described in Section 3.2. The starting point for our implementation was the ReHype source code [LT11, LT14]. Section 3.3 describes the porting of ReHype to a more modern platform and key enhancements that improve its recovery rate. The implementation of NiLiHype is described in Section 3.4. The experimental setup and evaluation results are presented in sections 3.5 and 3.6, respectively.

3.1 Component-Level Recovery

The key idea in component-level recovery (CLR) is to reduce recovery latency by limiting recovery to the failed component instead of involving the entire system [CKF04]. This section presents the key challenges in CLR as well as two alternative CLR mechanisms that involve low overhead during normal operation: microreboot and microreset, in Subsections 3.1.1 and 3.1.2, respectively.

Three main challenges must be overcome in any CLR implementation: (1) preventing the process of recovering the failed component from harming the rest of the system while also preventing the rest of the system from interfering with the recovery process; (2) restoring the failed component to a valid state; and (3) ensuring that the state of the recovered component is consistent with the state of the rest of the system.

One example of interference between CLR and the rest of the system is when the component being recovered is an OS kernel [DS10]. If an I/O interrupt is generated during recovery, when the kernel is not in a valid state to handle the interrupt, the result is likely to be recovery failure. The simple step of disabling interrupts during part of the recovery process is an obvious solution for this example.

As an example of challenge (2) above, consider the case where recovery involves stopping further execution of the failed component. At that point different parts of the component state may be inconsistent with each other. For instance, a lock may have been acquired by an execution thread within the component but that execution thread is abandoned [LT11, LT14]. Hence, component recovery must involve one, or a combination of, booting a new instance, rollback to a known valid checkpoint, and/or a roll forward procedure to fix the invalid state.

Ensuring that the state of the recovered component is consistent with the rest of the system is the most difficult challenge for CLR (challenge (3) above). Again we use an example where the component being recovered is an OS kernel [DS10]. If recovery involves booting a new kernel, the new kernel instance will not have the up-to-date state of the data structures that maintain information regarding the user processes that were running prior to the failure. Hence, there is no way to continue running these processes following recovery. To resolve this issue, the recovery mechanisms must reuse part of the state from the previous instance. This introduces an obvious vulnerability since, when recovery is triggered, the state of the previous instance is, by definition, invalid. Recovery can be successful only if the error is not propagated to the reused part of the state or the recovery mechanism can somehow fix corrupted reused state.

3.1.1 Microreboot: Component-Level Recovery with Reboot

Microreboot is a CLR that involves booting a new component instance. As discussed in connection with the third challenge above, this requires reuse of part of the state of the previous instance. Therefore microreboot needs to preserve part of the state of the failed instance across the reboot and then reintegrate it with the state of the newly booted instance [DS10, LT11, LT12, LT14]. The state of the component after recovery is thus a combination of the state of the initial boot and the reused state from the failed instance.

3.1.2 Microreset: Component-Level Recovery without Reboot

A significant drawback of microreboot is recovery latency. This latency includes the time to reboot plus the time to re-integrate the state from the previous instance. For large, complex components, such as kernels and hypervisors, this latency can be from multiple hundreds of milliseconds [LT14] to tens of seconds [DS10]. It is possible to reduce part of the reboot time by replacing the reboot with a rollback to a checkpoint saved right after a previous reboot [YYK11a]. However, even in this case, there would be significant latency for reintegrating state from the previous instance. For example, with a mechanism similar to ReHype, this latency would be multiple hundreds of milliseconds even for a very small workload of three simple application VMs [LT14].

With the goal of reducing the recovery latency, this chapter investigates an alternative to microreboot, which we call *microreset*. Microreset is suitable for large, complex components that process requests from the rest of the system. OS kernels and hypervisors are examples of such components. Specifically, a hypervisor receives requests in the form of hypercalls or traps from the VMs as well as interrupts from hardware timers and potentially other devices. Multiple requests may be processed simultaneously by separate execution threads.

With microreset, upon error detection, the processing of all current requests is abandoned. This resets the component to a quiescent state. At that point, the microreset mechanism must perform additional operations to deal with the last two CLR challenges discussed above. Specifically, there is a need to perform roll forward operations to fix any corruptions in the component state as well as inconsistencies among different parts of the component state. Next, inconsistencies between the recovering component state and the states of other components in the system must be resolved. For example, this may require retrying requests from other components that were abandoned when the error was detected.

With microreset, only a small fraction of the component state is discarded during recovery. Specifically, it is just the "local" states of the abandoned execution threads (e.g., variables on the stacks). The entire remaining state is kept in place and reused. On the other hand, with microreboot, only part of the global state from the failed instance is reused and the rest of the state is restored to its initial values by the reboot. Microreset's reuse of a larger fraction of the pre-recovery component state increases the probability that the post-recovery state is invalid. Hence, there is a reason to expect some reduction in recovery rate with microreset compared to microreboot.

3.2 Implementing CLR of a Hypervisor

This section discusses how component-level recovery (CLR) can be applied to the Xen [BDF03] hypervisor, based on microreboot and microreset. Subsection 3.2.1 is a brief overview of the

Xen virtualization platform and examples of CLR challenges (Section 3.1) applicable to this particular component. Subsection 3.2.2 provides a high-level description of how microreboot has been applied to Xen with *ReHype* [LT11, LT14]. Subsection 3.2.3 provides a high-level description of how we apply microreset to Xen with *NiLiHype*.

3.2.1 The Xen Virtualization Platform

The Xen virtualization platform consists of two components: the hypervisor and the privileged VM (PrivVM, also known as Dom0). The hypervisor provides the core functionality of the virtualization platform, such as memory management and scheduling of the VMs. The PrivVM performs management operations, such as creating, checkpointing, and destroying VMs. The PrivVM may also host the device drivers for the I/O devices in the system and facilitates their sharing among the application VMs (AppVMs) [BDF03].

There are three ways for control to transfer from the VMs to the hypervisor: hypercalls, exceptions and hardware interrupts. VMs issue hypercalls to the hypervisor to request service from the hypervisor. An example of a hypercall is a request by a VM for the hypervisor to update page table entries. This particular example is relevant for *paravirtualized* VMs, (PVMs) [RG05, BDF03] and it should be noted that the PrivVM is a PVM. Exceptions occur when VMs execute privileged or illegal instructions. An interrupt is triggered by hardware and causes a trap to a handler in the hypervisor, from which it is sometimes forwarded to some VM.

The ability to recover from a failure during the execution of any part of the virtualization platform requires dealing with the PrivVM as well as the hypervisor. This issue has been addressed in previous work [LT12, LT14] and is not discussed in this thesis.

The challenges faced by any CLR, discussed in Section 3.1, must, of course, be handled by CLR of the Xen hypervisor. For example (challenge (1)), if VMs continue to execute during recovery, a trap from a VM may cause the hypervisor to fail since it is not in a proper state for handling such events. An example of challenge (2) from Section 3.1 is that internal hypervisor data structures, such as timer heap, may be corrupted by the fault or left in an inconsistent state. An example of challenge (3) is that the hypervisor may be in the middle of handling a hypercall when an error is detected. The failure to complete this hypercall may cause the initiating VM to fail following hypervisor recovery.

3.2.2 ReHype: Microreboot of the Xen Hypervisor

ReHype [LT11, LT14] recovers from failures of the Xen hypervisor using microreboot. When an error is detected, a recovery handler is invoked. The first few steps cause all the CPUs to disable interrupts and all but one to halt. The CPU that does not halt handles most of the rest of the recovery process. These initial steps prevent interference between the recovery process and the rest of the system. The handler then saves a copy of the data in the static data segments to a memory location where it will not be overwritten by the new hypervisor instance. The next step is to boot a new hypervisor instance. During reboot, parts of the preserved static data segments are used to overwrite some of the values initialized earlier in the boot process. The non-free heap pages in the pre-recovery hypervisor instance are preserved and re-integrated in the new heap. Pre-recovery page tables are restored. The final step of the basic scheme is to wake up all the CPUs and resume normal operation.

Enhancements of the basic mechanism outlined above are used to achieve a high recovery rate [LT11, LT14]. These enhancements deal with the last two CLR challenges discussed in Section 3.1. For example, the reused state from the previous hypervisor instance includes locks. In order to make the new hypervisor state self-consistent, all of these locks are released. In order to resolve inconsistencies with respect to the VMs, for any partially executed hypercall, the VM state of the corresponding VM is set up so that the hypercall is retried once VM execution is resumed. To partially resolve inconsistencies with respect to the hardware, all pending and in-service interrupts are acknowledged.

3.2.3 NiLiHype: Microreset of the Xen Hypervisor

NiLiHype is essentially ReHype without reboot. Many of the basic operations as well as enhancements that are performed by NiLiHype are identical or similar to those in ReHype. For example, both have to ensure that partially executed hypercalls are retried following recovery. Similarly, as part of recovery, both have to release locks used by the hypervisor.

Since NiLiHype does not include reboot, some complex operations required by ReHype are not needed. An example of this is the ReHype step of rebuilding the heap to re-integrate the data reused from the previous instance. On the other hand, the reboot in ReHype does help in producing a hypervisor state that is self-consistent. Thus, NiLiHype requires additional enhancements to overcome some CLR challenges (challenge 2 in Section 3.1) that ReHype overcomes by performing a reboot.

When an error is detected, the recovery handler of NiLiHype is invoked on the CPU where the error is detected. The handler disables interrupts on its own CPU and interrupts all the other CPUs, which then disable interrupts. Each of the CPUs discards its execution thread within the hypervisor by discarding the hypervisor stack (resetting the stack pointer). All the CPUs, except for the one that detected the error, then enter busy waits. At this point, enhancements to increase the recovery rate should be performed by the CPU that detected the error (see below). The above initial steps prevent interference between the recovery process and the rest of the system. The final step of the basic scheme is to allow all the CPUs to exit their busy waits and resume normal operation.

With the basic mechanism described above, recovery always fails. Enhancements that deal with the last two CLR challenges discussed in Section 3.1 are necessary to achieve a high recovery rate. These enhancements are described in Subsection 3.4.1.

With NiLiHype, all threads of execution within the hypervisor are discarded. A possible alternative design choice would be to discard only the execution thread of the CPU that detects the error. The choice made in NiLiHype makes it more similar to ReHype, where the reboot effectively discards all the execution threads. It is expected that the alternative choice would be more complex to implement and result in lower recovery rate. The reasons for this are interactions among hypervisor threads of execution as well as interactions between these threads and the recovery process itself. An example of this is if CPU1 sends an interprocessor interrupt (IPI) to CPU2 and waits for a response. An error may be detected on CPU2 after receiving the IPI but before responding. Since CPU2 discards its execution thread, CPU1 may be blocked forever. A second example is related to the impact of changes to the global state by the recovery process. These changes may cause non-discarded threads to encounter unexpected state changes that lead to their failure.

3.3 Porting and Enhancing ReHype

Due to the similarity between NiLiHype and ReHype, the starting point for the NiLiHype implementation was the ReHype source code [LT11, LT14]. Our first step was to port this implementation to the x86-64 ISA (from x86-32), Xen version 4.3.2 (from 3.3.0), with all VMs running Linux 3.16.1 kernels. As described in the rest of this section, we then implemented enhancements to improve the recovery rate.

Our initial port mostly resolved the expected porting issues caused by the evolution of Xen code, such as changes in function/variable/macro names. We then used fault injection to evaluate the port and guide further enhancements. This was based on running a simple workload (one AppVM) and injecting fail-stop faults. After the initial port, the recovery rate was 65%. Three enhancements were required due to platform changes while the fourth would also have been useful for the older platform. Together, these enhancement increased the recovery rate to 96%.

Syscall retry. With the x86-32 ISA, system calls from the VM processes directly trap into the VM kernel. However, with the x86-64 ISA, system calls from the VM processes trap into the hypervisor which then forwards them to the appropriate kernel. In order to handle the possibility that an error is detected when the hypervisor is forwarding a system call, ReHype had to be enhanced to ensure that such system calls are retried following recovery. The implementation is similar to hypercall retry.

Fine-granularity batched hypercall retry. With the new Xen platform, in order to reduce the virtualization overhead, several hypercalls may be batched into one hypercall. In order to better handle such batched hypercalls, the hypervisor logs the completion of each hypercall within a batch as it completes. If, following recovery, the batched hypercall is retried, those component hypercalls that completed earlier are skipped.

Save FS/GS. Xen on x86-64 doesn't use the FS and GS registers and thus does not save them when the hypervisor is entered. Xen on x86-32 does save these registers. Hence, with the initial ReHype port, these registers are lost following recovery. The fix is for the hypervisor to save these registers when an error is detected.

Mechanisms to mitigate hypercall retry failure. With the above three enhancements, the recovery rate is 84%. The remaining recovery failures are largely caused by re-executing non-idempotent hypercalls. For example, several hypercalls increase or decrease a reference counter in the page frame descriptor by one. If an error is detected after a hypercall updates the counter but before completion, the re-execution results in an inconsistent state.

A comprehensive solution to the above problem would be to transactionalize all the non-idempotent hypercalls. Doing that would require major changes to the code and/or significant overhead. Instead, we used fault injection to identify problem cases and resolve them using lightweight logging and code reordering. A downside of our approach is that we have not tested all hypercall handlers. Thus, there are likely to be several infrequently-used non-idempotent hypercall handlers that we have not properly enhanced. Furthermore, even for the handlers that have been modified, the changes do not resolve 100% of the problem. However, the changes do significantly reduce the window of vulnerability and minimize the probability of recovery failure.

Table 3.1: Mechanisms to enhance ReHype with the corresponding successful recovery rate after the enhanced mechanisms are deployed as well as the 95% confidence interval of the successful recovery rate.

Mechanism	Successful Recovery Rate
ReHype after initial port	65.4% ± 2.9%
+ Mechanisms to deal with platform differences	83.8% ± 2.3%
+ Mechanisms to mitigate hypercall retry failure	96.4% ± 1.2%

Logging enables undoing changes performed by partially executed hypercalls. Changes to critical variables are logged and the changes are undone following recovery, before a retried hypercall reads or modifies these variables. For some hypercalls, it was possible to reduce the window of vulnerability by simply reordering the code, without changing the functionality or incurring overhead. An example of this is moving modifications of critical variables to the end of the hypercall so that there is minimal code to execute between the state changes and the completion of the hypercall. Altogether, these changes for handling non-idempotent hypercalls increase the recovery rate from 84% to 96%.

After adding the above mechanism, the successful recovery rate of ReHype boosts to 96.4%. Table 3.1 summarizes all the mechanisms to enhance ReHype with the corresponding successful recovery rate after the enhanced mechanisms are deployed as well as the 95% confidence interval of the successful recovery rate. To further validate the correctness of our implementation, we have changed the benchmark in the AppVM to *BlkBench* and ReHype achieves a similar successful recovery rate of 96.8%.

3.4 NiLiHype Implementation

The starting point for the NiLiHype implementation is the enhanced ReHype implementation described in Section 3.3. Some of the features that are common to NiLiHype and ReHype are: 1) VMs are suspended and interrupts are disabled during recovery; 2) almost all the ReHype enhancements described in [LT11, LT14]; 3) all the enhancements described in Section 3.3.

As mentioned in Subsection 3.2.3, since NiLiHype does not involve reboot, some of the most complex and time-consuming operations required by ReHype are not needed. Specifically, these include hardware initialization as well as operations to preserve and later reintegrate state from the pre-recovery hypervisor instance. These operations are described in detail in Section 3 of [LT11] and their latencies are presented in Section 10 of [LT14]. Similar latency measurements, for our enhanced ReHype implementation, are presented in Subsection 3.6.3.

This section focuses on additional enhancements needed by NiLiHype to overcome CLR challenges that are resolved by the reboot in ReHype. Subsection 3.4.1 presents the enhancements. Section 3.4.2 presents the measurement-based incremental development of the NiLiHype-specific enhancements.

3.4.1 Enhancements Required by NiLiHype

With ReHype, a very low recovery rate (5.6%) is achieved without any enhancements [LT11]. That recovery is even possible with the basic scheme is due to the operations performed by the reboot, that include re-initializing the hardware and initializing a new, valid hypervisor memory state. As mentioned in Subsection 3.2.3, with just the basic NiLiHype mechanism (discard all hypervisor threads of execution), recovery never succeeds. Hence, the enhancements of the basic scheme, that resolve the CLR challenges (Section 3.1), are even more critical in NiLiHype.

One of the enhancements developed for NiLiHype is needed to bring the hardware to a consistent state: *reprogram hardware timer*. Four additional enhancements deal with hypervisor memory state. All of these enhancements are described below.

Reprogram hardware timer. Xen relies on the hardware timer in the interrupt controller (APIC) to trigger the examination of the software timer heap. The handler reprograms the APIC timer to fire again at a time determined by the top node of the timer heap. If the fault occurs after the APIC timer has fired but before Xen reprograms it, without additional mechanisms, the APIC timer will never fire again after recovery. NiLiHype handles this issue by ensuring that each CPU reprograms its APIC timer before resuming normal operation.

Clear IRQ count. In Xen, each CPU maintains a per-CPU variable named *local_irq_count* that records the nesting level of interrupts. When the CPU enters or leaves an interrupt handler, *local_irq_count* is, respectively, incremented or decremented. The *local_irq_count* value is used in hypervisor assertions to check whether the CPU is currently servicing an interrupt. As NiLiHype discards all the execution threads in the hypervisor, the *local_irq_count* variables of all the CPUs are set to zero during the recovery.

Ensure consistency within scheduling metadata. Xen maintains scheduling metadata that includes: (1) the *runqueue* of each CPU, which is a linked list of vCPUs; (2) per-CPU variables indicating the current executing vCPU; and (3) per-vCPU variables representing the execution states of the vCPUs. Hypervisor failure followed by recovery can easily leave this scheduling metadata in an inconsistent state. Inconsistencies within the scheduling metadata can cause the hypervisor to incorrectly restore the register context of one vCPU when another vCPU is scheduled to run. Such inconsistencies can also result in the failure of assertions in the scheduling routine, leading to hypervisor failure.

Resolving potential scheduling metadata inconsistencies is done based on two key ideas: 1) where possible, initialize the data to a fixed valid value instead of relying on the existing value; 2) if it is necessary to use existing data, pick the most reliable source and make the rest of the metadata consistent with that.

With NiLiHype, we encountered a particularly critical problem related to scheduling metadata inconsistencies. The information regarding which vCPU is currently running on each CPU is stored redundantly in multiple places. Specifically, it is stored in the per-CPU structures as well as two different locations in the per-vCPU structures. To resolve the inconsistencies, the information in the per-CPU structures is used to set the information in all the per-vCPU structures.

Unlock static locks. Since both ReHype and NiLiHype effectively discard all threads of execution in the hypervisor, all locks should be in their unlocked state following recovery. ReHype includes a mechanism to release all the locks stored in the heap. NiLiHype uses the same mechanism. With ReHype, locks in the static data segment ("static locks") are initialized to their unlocked state during boot. NiLiHype requires an additional mechanism to release all such locks.

NiLiHype avoids a complex mechanism for tracking the static locks. Instead, NiLiHype takes advantage of the fact that, in Xen, all the static locks are defined using a macro. We modified the linker script used to build the Xen image and the macro defining locks to put all the static locks in a separate segment in the Xen image, effectively placing them all in one array. During the recovery process, before multiple CPUs are allowed to execute, the CPU that detects the error iterates over all the locks in the segment and unlocks any locked locks.

Reactivate recurring timer events. Xen uses several recurring timer events. These include events to synchronize system time among CPUs and to update the execution time of vCPUs for the scheduler. For each of these timers, the handler re-activates the timer each time it is fired. If a fault occurs while a CPU is executing one of these timer handlers, before the handler successfully re-activates the timer, the recurring timer is lost.

This problem is very similar to the problem described above that motivates the *reprogram hardware timer* enhancement. To solve the problem discussed here, for each relevant timer event, there is a flag that is set on entry to the handler and cleared upon exit. As part of the

Table 3.2: Summary of Mechanisms to enhance NiLiHype

Mechanism	Successful Recovery Rate
Basic	0%
+ Clear IRQ count	16.0% ± 2.3%
+ Enhanced with ReHype mechanisms	51.8% ± 3.1%
+ Ensure consistency within scheduling metadata	82.2% ± 2.4%
+ Reprogram hardware timer	95.0% ± 1.4%
+ Unlock static locks + Reactivate recurring timer	96.1% ± 1.2%

recovery process, for any of these flags that is set, the corresponding timer is re-activated.

3.4.2 Incremental Development of NiLiHype Enhancements

To identify the need for the enhancements discussed in Subsection 3.4.1, we use the same incremental procedure, based on fault injection, used in multiple previous works [NC99, LT11]. Results from fault injections are analyzed to identify the cause of the plurality of recovery failures. A mechanism is developed to handle that particular problem. The process is repeated with new fault injections in each iteration.

Table 3.2 summarizes all the mechanisms we use to enhance NiLiHype with their impact on the recovery rate. The fault injection setup is the same one used in Section 3.3. Specifically, we use the 1AppVM workload with the *UnixBench* benchmark (Subsection 3.5.1). 1000 fail-stop faults are injected in each iteration.

Table 3.2 shows that the *Clear IRQ count* enhancement is mandatory in order for NiLiHype recovery to succeed. This is because the CPU that detects the error sends IPIs to other CPUs to initiate the recovery. Each of the CPUs receiving the IPI increments its *local_irq_count*. Since the CPU then discards its thread of execution (discards its stack), it never returns from the IPI. As a result, *local_irq_count* ends up with an inconsistent value (not 0). This later causes hypervisor failure as a result of the failure of assertions in several critical routines that check whether the CPU is in interrupt context.

Table 3.2 also shows that, with all the enhancements, for this particular setup, NiLiHype achieves the same recovery rate as ReHype (Section 3.3).

3.5 Experimental Setup

This section presents the experimental setup used to evaluate NiLiHype and ReHype. This setup is very similar to the one used in [LT11, LT14]. The key difference is the use of a more modern platform (ISA and Xen/Linux versions). The configurations of the systems evaluated (target systems) are described in Subsection 3.5.1. Subsection 3.5.2 presents the error detection mechanisms used in the target systems. Details regarding the fault injection are described in Subsection 3.5.3.

3.5.1 Target System Configurations

We evaluate virtualized systems running synthetic benchmarks designed to stress different aspects of the system. The hypervisor is Xen 4.2.3. The system includes the privileged VM (PrivVM) and either one application VM (AppVM), in the *1AppVM* configuration, or three AppVMs, in the *3AppVM* configurations. Each VM consists of one vCPU (virtual CPU) and each of the vCPU is pinned to a different physical CPU. The physical machines used for all the experiments are 8-core systems based on Intel Nehalem CPUs.

All the AppVMs are paravirtualized VMs (PVMs). It should be noted that previous work has shown that fault injection results obtained with AppVM supported by full hardware virtualization (HVMs) are very similar to those obtained with paravirtualized AppVMs [LT14].

Table 3.3: Benchmarks used to evaluate *NiLiHype*

Benchmark	Description
BlkBench	a program that creates, copies, reads, writes and removes multiple 1MB files containing random content.
UnixBench [Uni]	a collection of programs designed to stress different aspects of the system.
NetBench	a user-level network ping program.

As summarized in Table 3.3, three synthetic benchmarks are used: *BlkBench*, *UnixBench*, and *NetBench*. *BlkBench* focuses on the interface to block devices (disk). To ensure that the device is actually accessed, requiring hypervisor activity, caching of block and file system data in the AppVM is turned off. Without this setting, caching within the AppVM would minimize the chances for exposing recovery failure. We use a subset of the programs in the original UnixBench. The programs were selected for their ability to stress the hypervisor's handling of hypercalls, especially those related to virtual memory management. *NetBench* involves two processes: the *receiver* runs in an AppVM in the target system, and the *sender* runs on a separate physical host. The *sender* sends a UDP packet to the *receiver* every 1ms. Upon receiving a packet, the *receiver* sends a reply back to the *sender*.It is used to exercise the interface to the network as well as to evaluate the recovery latency.

For *BlkBench* and *UnixBench*, the execution is considered as failed if 1) one or more files produced by the benchmark are different from the ones in a golden copy, or 2) logging messages from the benchmarks indicate that one or more than one system calls to the OS of the AppVM failed. *NetBench* execution is considered as failed if the packet reception rate of the *sender* drops by more than 10% compared to its reception rate during normal execution in any one-second interval.

In the *1AppVM* setup, the AppVM runs either *BlkBench* or *UnixBench*. Each one of the benchmarks is configured to run for around 10 seconds. This setup is mostly used to guide the measurement-based incremental development of recovery enhancement mechanisms.

In the *3AppVM* setup, the system initially runs two AppVM: one with *UnixBench* and the other with *NetBench*. Following recovery, a third AppVM is created and it runs *BlkBench*. The third AppVM is used to check whether the hypervisor still maintains its ability to create and host newly created VMs after recovery. The first two AppVMs are configured to run for approximately 24 seconds.

3.5.2 Error Detection

The focus of this chapter is on error recovery, not error detection. However, an error detection mechanism is necessary for the experimental evaluation since such a mechanism is responsible for initiating recovery.

We rely on the built-in panic and hang detectors in Xen to detect errors. A panic is detected when a fatal hardware exception occurs or a software assertion fails. The hang detector is based on a watchdog timer. It is implemented based on hardware performance counters and software timer events. A performance counter on each CPU is used to generate an NMI every 100ms of unhalted CPU cycles. There is also a recurring software timer event that increments a counter every 100ms. The handler of the performance counter checks for changes in the counter. If the counter is not incremented for three consecutive invocations of the performance counter handler, a hang is detected.

3.5.3 Fault Injection

Software-implemented fault injection is used to determine the recovery rate of NiLiHype and ReHype. A fault injection *run* consists of booting the target system, starting the benchmarks in the AppVMs, injecting one fault, and collecting logs that allow the results to be analyzed. For each fault type and system configuration there is an injection *campaign* that consists of

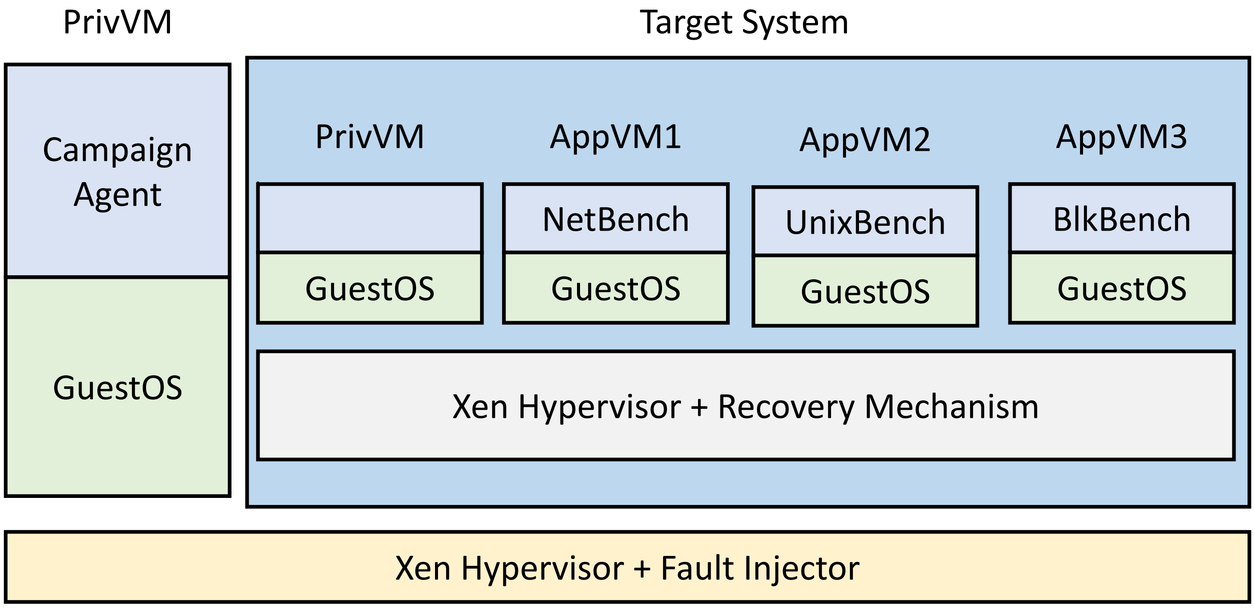


Figure 3.1: Fault injection setup with the 3AppVM workload. The entire target system is in a VM. The injector is in the outside hypervisor

multiple *runs*.

We ported the Gigan fault injector [LT15] to a modern platform (ISA and Xen/Linux version). To minimize intrusion by the injector and simplify campaign setups, we use Gigan in a configuration based on two-level nested virtualization. Specifically, the entire virtualized target system runs in a VM supported by full hardware virtualization (i.e., an HVM). It has been shown that fault injection and recovery results obtained with this setup are similar to those obtained when the target system runs on bare hardware [LT14, LT15].

The injector runs outside the target system, in the hypervisor (the “outside” hypervisor) that hosts the HVM with the target system. A user-level campaign script, the *Campaign Agent*, runs in the PrivVM of the outside hypervisor. The *Campaign Agent* creates the VM with the target system, configures the fault injector, and collects logs and output from each run. The fault injection setup with the 3AppVM workload is shown in Figure 3.1.

We inject three types of faults: *Failstop*, transient *Register* faults, and *Code* faults.

Failstop faults are injected by changing the value of the program counter to 0. *Register* faults are injected by flipping a random bit in a random register selected from the 16 general-purpose registers, the stack pointer, the flag register, and the program counter. *Code* faults are injected by flipping a random bit in a random byte chosen within the 15-byte range (the maximum length of an x86-64 instruction) starting from the current value of the program counter. When/if the *Code* fault causes an error that is detected, the injector "repairs" the fault. Thus, the effects of a *Code* fault do not persist during the recovery process. Hence, the effects of a *Code* fault are almost the same as if it was a transient fault.

The injected faults do not cover all possible faults. NiLiHype is designed to recover from transient hardware faults as well as rare software bugs (Heisenbugs) [Gra86], that occur only under particular timing and ordering of asynchronous events in the system. Transient hardware faults in the CPU datapath are likely to be manifested as erroneous values in registers. Hence, register bit flips can be expected to be reasonably representative of such faults. Injection of bit flips in the code attempts to partially represent faults in the instruction fetch and decode hardware. Similar injection campaigns have been widely used in prior works [DCC08, JF12, BVK16]. As discussed in Subsection 3.6.1, results from the injection of failstop faults, together with other results, help with the understanding of the tradeoffs between microreboot and microreset.

All the faults are injected by using a two-level chained trigger. When the first-level trigger fires, it triggers the second-level, which, when fired, triggers the fault injection. The first-level trigger is a timer that fires after a specified amount of time has elapsed. It is configured differently in the 1AppVM and 3AppVM setups. With the 1AppVM setup, it is set to fire at a random time after the initial 10% and before the final 10% of the benchmark execution time. With the 3AppVM setup, the first-level trigger is configured to fire at a random time between 500ms and 6 seconds. This is well past the start of the *UnixBench* and *NetBench* AppVMs while leaving most of their 24 seconds execution to occur after recovery.

The second-level trigger fires after a random number of instructions between 0 to 20000 have been executed in the target hypervisor. This trigger ensures that faults are injected only while the CPU is executing code of the target hypervisor.

3.6 Evaluation

This section presents an evaluation of NiLiHype using the experimental setup described in Section 3.5. Along every axis, NiLiHype is compared to ReHype. The recovery rate, recovery latency, hypervisor processing overhead during normal operation, and implementation complexity are presented in Subsections 3.6.1, 3.6.2, 3.6.3, and 3.6.4, respectively.

3.6.1 Successful Recovery Rate

It has been shown that, for typical deployments of virtualization, less than 5% of CPU cycles are spent executing hypervisor code [BHH13, BDD10]. Hence, a random transient fault is much more likely to occur when executing code in a VM than when executing hypervisor code. Thus, a random transient fault is highly likely to affect *one* of the VMs, possibly causing it to fail, even if the virtualization platform is completely immune to all faults. Whether a fault in the hypervisor can affect a single VM becomes relevant only if mechanisms implemented strictly within the VM itself allow it to mask or recover from the overwhelming majority of faults that may occur during the execution of the VM. Taking this into account, it is not meaningful to evaluate any hypervisor resilience mechanism based on a criterion that a manifested fault in the hypervisor should not affect even *one* VM.

Without any resilience mechanisms, a single transient fault can cause the hypervisor to fail, taking down all the VMs it hosts. Based on the discussion above, we can define a reasonable goal for a hypervisor resilience mechanism. Specifically, taking into account only transient faults (including Heisenbugs [Gra86]), running multiple VMs on a single host should not be worse than running them without virtualization on separate physical machines [LHT11]. In practice, this goal cannot really be met due to practical issues, such as power supplies, network connections, etc. However, this forms the basis for our definition of "successful recovery" from hypervisor failure.

We define recovery from hypervisor failure to be "successful" if no more than one AppVM is affected by the fault and, after recovery, the hypervisor continues to operate correctly (new VMs can be created, etc). The 3AppVM setup is designed to allow evaluation based on this definition. Specifically, the setup includes creating a new AppVM (*BlkBench*) after recovery, and an ability to verify that *BlkBench* runs correctly to completion. Note that, for the 1AppVM setup, we define "recovery success" to mean that no VM is affected.

The recovery rate is evaluated with the 3AppVM setup. Separate campaigns are run with the three fault types: *Failstop*, *Register*, and *Code*. For each fault injection run, the outcome can be classified into three categories: non-manifested, silent data corruption (SDC), and detected. Non-manifested means that the injected fault does not cause any observable abnormal behavior: the benchmarks finish successfully (produce the correct outputs) and the detection mechanisms are not triggered. SDC means that the detection mechanisms are not triggered but at least one of the benchmarks fails to produce the expected outputs. Detected means that one of the detection mechanisms is triggered. Obviously, the recovery mechanism is triggered only for fault outcomes in the last category.

The breakdown of injection outcomes varies with fault type. Obviously, all *Failstop* faults are detected. For *Register* faults, the breakdown in our campaign is: 74.8% non-manifested, 5.6% SDC, and 19.6% detected. For *Code* faults, the breakdown is: 35.0% non-manifested, 12.1% SDC, and 52.9% detected.

The fault injection campaigns for evaluating the recovery rate include 1000 *Failstop* faults, 5000 *Register* faults, and 2000 *Code* faults. In each case, the number of the injected faults was chosen so that, for both NiLiHype and ReHype, the 95% confidence interval for the recovery rate was within ±2%.

Figure 3.2 presents the successful recovery rate of NiLiHype and ReHype with the 3Ap-

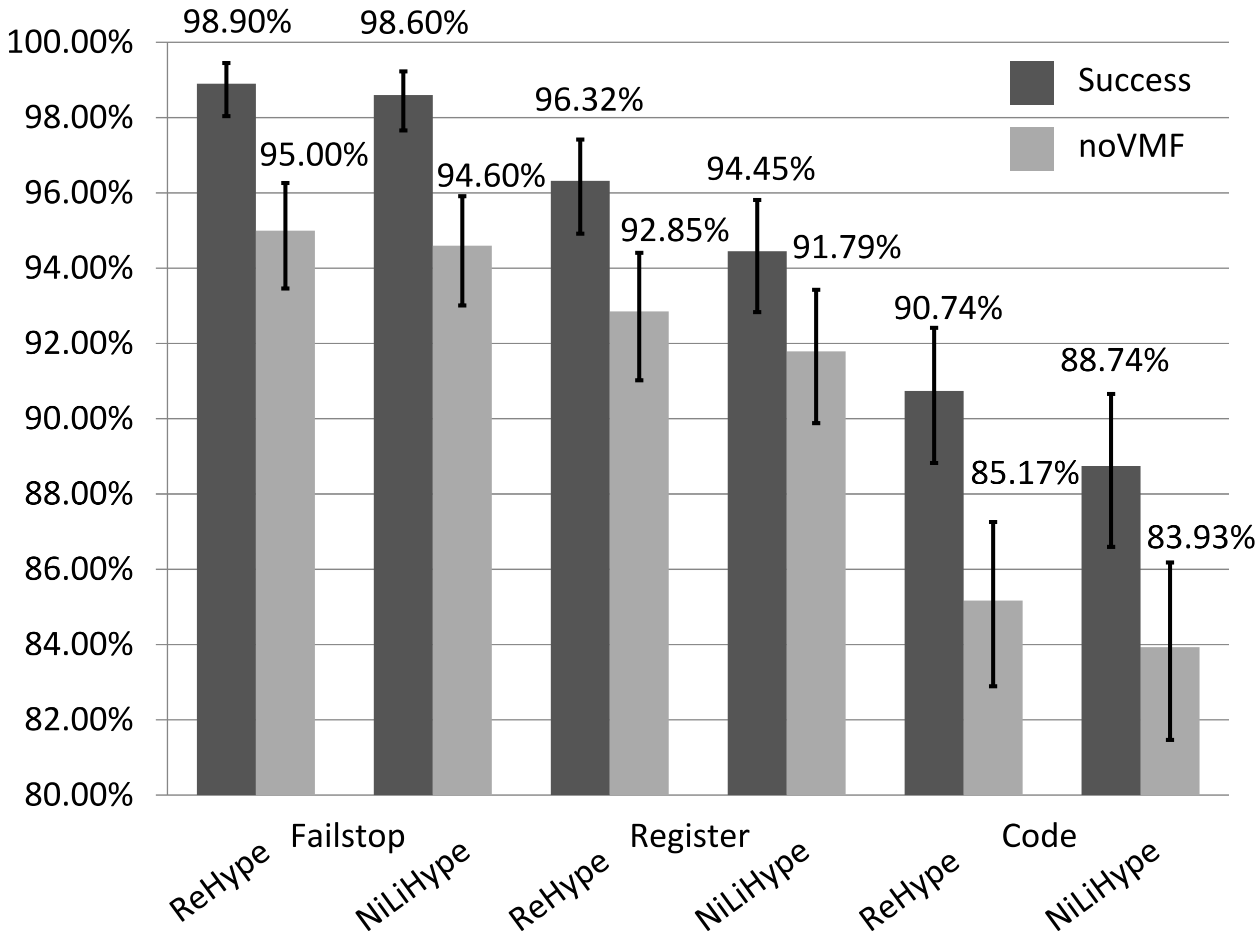


Figure 3.2: Successful recovery rates of NiLiHype and ReHype for different fault types with the 3AppVM setup. Error bars show the 95% confidence intervals. noVMF stands for no AppVM failure cases.

pVM setup. For the completeness, we also report the portion of detected errors (recovery initiations) that resulted in no AppVM failures (*noVMF* — no VM failures). The differences between *Success* and *noVMF* are due to injection runs where the only impact of the fault is to cause one of the first two initial AppVMs to fail.

Figure 3.2 shows that NiLiHype and ReHype achieve essentially identical recovery rates for *Failstop* faults, but not for the other fault types. *Failstop* faults can only result in inconsistencies within the hypervisor state or between the hypervisor state and the states of other hardware and software components. The other fault types can result in the same inconsistencies but also in state corruption. It is likely that, for *Register* and *Code* faults, ReHype's small advantage is due to cases where the fault corrupted part of the hypervisor state that is discarded and re-initialized by the reboot.

To understand the reasons for recovery failures, we analyzed the recovery failure cases when *Register* faults are injected. ReHype resulted in 35 such cases and NiLiHype in 54. The reasons for recovery failures with ReHype and NiLiHype are similar. The top three reasons are: 1) the recovery routine fails to be invoked due to the corrupted hypervisor state, 2) the PrivVM fails, and 3) the error causes a data structure in the hypervisor, typically a linked list or the heap, to be corrupted or left in an inconsistent state.

Figure 3.2 shows that *Code* faults result in the lowest recovery rate. This is likely due to the significantly longer detection latency of these faults [LT15], providing more time for errors to propagate and cause greater state corruption.

3.6.2 Recovery Latency

When hypervisor recovery is in progress, all the AppVMs are paused. Hence, it is easy to measure the recovery latency by measuring the service interruption of a service executing in an AppVM in the target system. Latency measurements may be distorted when the system is deployed in a nested virtualization configuration. Hence, in order to obtain accurate latency results, the target system runs on bare hardware. As a service, we use *NetBench* in the 1AppVM setup. The service interruption is measured at the *sender*, that runs on a separate physical host (Subsection 3.5.1).

For NiLiHype, we measured a recovery latency of 22ms. For ReHype, with all the recovery latency optimizations discussed in [LT14], we measured a recovery latency of 713ms. Repeating each experiment five times, the latencies varied by no more than 1ms for NiLiHype and 10ms for ReHype.

Table 3.4: Recovery Latency breakdown of ReHype

Operations	Time
Hardware initialization:	412ms
- Early initialize of the boot CPU	12ms
- Initialize and wait for other CPUs to come online	150ms
- Verify, connect and setup local APIC and setup IO ACPI	200ms
- Initialize and calibrate TSC timer	50ms
Memory initialization	266ms
- Record allocated pages of old heap (Use to preserve content of old heap)	21ms
- Restore and check consistency of page frame entries	21ms
- Re-initialize the page frame descriptor for un-preserved pages	13ms
- Recreate the new heap	211ms
Misc	35ms
- SMP initialization	20ms
- Identify valid page frame, relocate boot up modules	2ms
- Others	13ms
Total	713ms

To determine how much time the different operations involved in recovery contribute to the overall recovery latency, we added code in the recovery code of NiLiHype and ReHype to record the value of the time stamp counter (TSC) after each major recovery step is completed. The results for ReHype and NiLiHype are shown in Table 3.4 and Table 3.5,

Table 3.5: Recovery Latency breakdown of NiLiHype

Operations	Time
- Restore and check consistency of page frame entries	21ms
- Others	1ms
Total	22ms

respectively. These tables list every step that takes more than 1ms.

Most of NiLiHype's recovery latency is due to an operation done to ensure consistency of the page frame descriptors. This is an operation that ReHype also performs [LT11, LT14]. The problem that this operation solves is that, following recovery, there are two components in each page frame descriptor that may be left in inconsistent states: the *validation bit* and the *page use counter*. This can cause the hypervisor to hang following recovery. To fix this issue, the recovery routine iterates over all the page frame descriptors in the hypervisor to check for the inconsistency and update as required to restore consistency.

The latency of the operation described above is proportional to the size of the host memory (and thus the number of page frame descriptors). In our system, 8GB of physical memory results in a latency of 21ms. Obviously, this would be a problem in a large system with tens or hundreds of GB of memory. The problem could be mitigated by exploiting parallelism. For example, use multiple cores to perform the operation. Another option is to not perform this recovery step. This option has the disadvantage that it results in a reduction of 4% in the recovery rate [LT11].

3.6.3 Hypervisor Processing Overhead in Normal Operation

A key question regarding any resilience mechanism is how much performance overhead during normal operation it incurs. With NiLiHype, this translates to the extent to which, for a fixed workload, the count of CPU cycles spent executing hypervisor code is higher with the NiLiHype modifications compared to with stock Xen.

To get accurate results, the hypervisor processing overhead is measured with the target system running on bare hardware. In each one of the CPUs (including the one running the PrivVM), a hardware performance counter is used to count the unhalted cycles spent in the hypervisor. For repeatable results, the precise points in time for starting and ending the measurement are carefully controlled. All the benchmarks running in all the AppVMs are synchronized. Measurement starts when all the benchmarks are ready to begin executing. Measurement ends when all the benchmarks complete. The benchmarks "inform" the measurement code when they begin and end using traps (the CPUID instruction). All the benchmarks execute for approximately the same amount of time (21s).

We define the hypervisor processing overhead as the percent increase in the unhalted cycle count in the hypervisor with NiLiHype relative to that same count with stock Xen.

We used four target system configurations. The first three use the 1AppVM setup with the three benchmarks: *BlkBench*, *UnixBench*, and *NetBench*. Obviously, for these three configurations, synchronizing the execution of the benchmarks is not relevant. The fourth configuration is a slightly modified version of the 3AppVM setup. Since recovery is not actually done in these measurements, all three of the AppVMs are created at the same time and they all run their benchmarks throughout the experiment. For each configuration, we repeated the measurements five times and found that the differences in the measurement results were all less than 1%.

Figure 3.3 shows the hypervisor processing overhead for NiLiHype for all the configurations. Most of this overhead is due to logging used to mitigate recovery failures due to retries of non-idempotent hypercalls. To show that, the figure also shows the overhead of NiLiHype without this logging (NiLiHype*).

We have also measured the hypervisor processing overhead of ReHype and found it to be

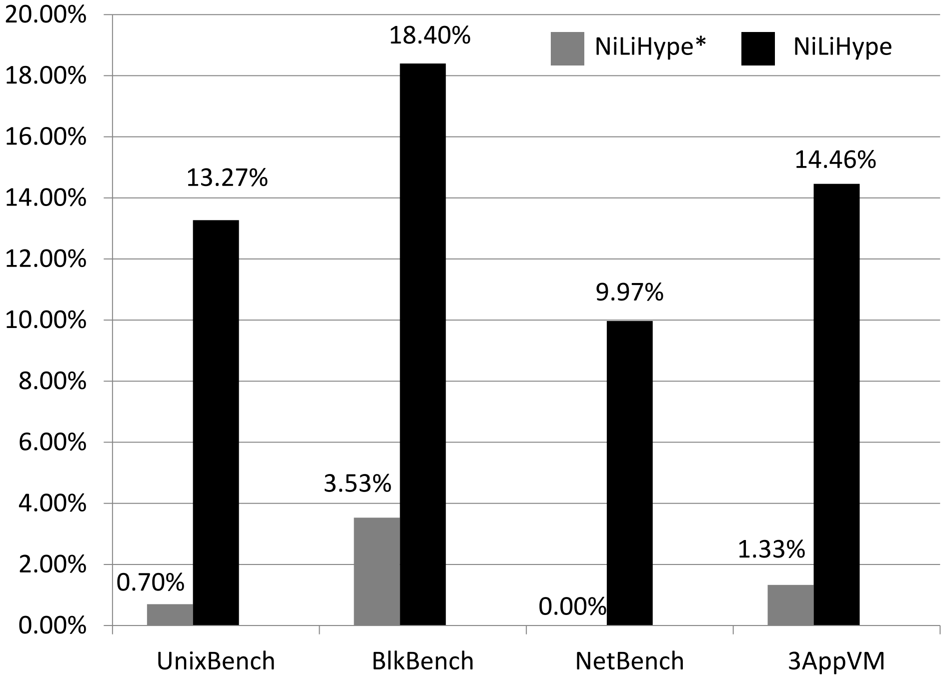


Figure 3.3: Hypervisor processing overhead (based on CPU cycles) of NiLiHype during normal execution. NiLiHype* stands for NiLiHype without logging to mitigate hypercall retry failure.

the same as for NiLiHype. This is not surprising since the logging in NiLiHype and ReHype are almost identical,

As mentioned earlier (Subsection 3.6.1), it has been shown that, for typical deployments of virtualization, less than 5% of CPU cycles are spent executing hypervisor code [BHH13, BDD10]. Hence, even with logging, the actual impact of the hypervisor processing overhead is negligible. Specifically, even in the worst case (*BlkBench*), the overhead in terms of total CPU cycles can be expected to be less than 1%. If this overhead is not acceptable, there is the option of turning the logging off. As discussed in Section 3.3, this will reduce the recovery rate by approximately 12%.

Table 3.6: Implementation Complexity of NiLiHype and ReHype

Type	Mechanisms	NiLiHype	ReHype
Normal operation	Mitigating hypercall retry failure	991	991
	Other logging	532	594
Recovery routine	Shared recovery mechanism	543	543
	Specific recovery mechanism	113	642
Total	Total	2179	2770

3.6.4 Implementation Complexity

As a measure of the implementation complexity of NiLiHype and ReHype, we use the total number of lines of code (LOC) added and modified starting with the source code of the stock Xen hypervisor. We use the code line count tool CLOC [CLO] to measure the LOC.

We partition the LOC added and modified into two categories: (1) code that executes during normal operation to enable or enhance NiLiHype/ReHype functionality, and (2) code that executes only during recovery. Table 3.6 presents the results.

For both NiLiHype and ReHype, most of the added/modified code in category (1) is related to mitigating hypercall retry failure (Section 3.3). For all the code in category (1), NiLiHype requires slightly less code. This is due to two types of logging that are not needed for NiLiHype. Firstly, ReHype needs to log changes to the I/O APIC registers during the normal execution. This is because ReHype recovery re-initializes these registers as part of the boot process. However, for recovery to succeed, these registers must be restored to their pre-recovery values. Secondly, ReHype needs to log the values of the boot line options during the normal boot up. These values are used by the hypervisor to correctly initialize the system during boot. Hence, ReHype recovery needs to reuse the previously logged boot line options to correctly boot up the hypervisor.

The main difference in the implementation complexity between NiLiHype and ReHype is in code that executes during recovery. ReHype has significantly more code needed to preserve and re-integrate the failed hypervisor state back to the new hypervisor instance.

3.7 Summary

There are compelling reasons to enhance hypervisors with the ability to recover from failures while allowing the VMs they host to resume normal operation without loss of work. With such capability, the fraction of datacenter capacity unavailable due to a single fault is reduced, there is greater flexibility in assigning VMs to hosts, and VM replication with both replicas running on a single host becomes an attractive point in the design space in some deployments.

ReHype, which is based on microreboot, has been previously presented as a mechanism for providing the above hypervisor recovery capability. This chapter investigated an alternative to microreboot, which we call microreset, that allows component-level recovery without reboot. Instead of rebooting a new instance, microreset resets the component to a quiescent state that is highly likely to be valid. Microreset is suitable for large, complex components that process requests from the rest of the system. The state reset it performs involves discarding all threads of execution within the component. By avoiding component reboot, microreset has the potential to achieve significantly lower recovery latencies than microreboot.

NiLiHype utilizes microreset to implement recovery from hypervisor failures. To achieve a high recovery rate, NiLiHype includes numerous enhancements needed to restore the hypervisor to a valid consistent state. Thus, the idea of microreset, by itself, is not sufficient for building an effective recovery scheme.

We have implemented NiLiHype and evaluated it in terms of recovery rate, recovery latency, hypervisor processing overhead during normal operation, and implementation complexity. We have shown that NiLiHype achieves a recovery rate of over 88%, only 2% lower than the rate achieved by ReHype. In return, NiLiHype's recovery latency is over a factor of 30 lower, at 22ms. With this low recovery latency, for many important application, service interruption would not be noticeable. Based on our measurements, the performance overhead of NiLiHype during normal operation is expected to be under 1%. NiLiHype's implementation required adding or modifying less than 2200 lines in the Xen hypervisor.

CHAPTER 4

NiLiCon: Fault-Tolerant Replicated Containers

NiLiHype, presented in the last chapter, prevents a single fault that occurs during the execution of hypervisor code from taking down all the VMs hosted by the hypervisor, including the applications running on these VMs. However, *NiLiHype* does not protect an individual critical application from failures. This chapter presents a container replication mechanism: *NiLiCon* (Nine Lives Containers) that achieves this goal.

Servers commonly host applications in virtual machines (VMs) and/or containers to facilitate efficient, flexible resource management [Ber14, Mer14, RG05]. In some environments containers and VMs are used together. However, in others containers alone have become the preferred choice since their storage and deployment consume fewer resources, allowing for greater agility and elasticity [Ber14, LKG15].

The strong isolation and encapsulation provided by virtual machines or containers make them a natural framework for implementing application-transparent fault-tolerance mechanisms. Despite the advantages of containers, there has been very little work on high availability and fault tolerance techniques for containers [LK15]. In particular, there has been limited work on high availability techniques [LKG15, Liu16]. However, to the best of our knowledge, there are are no prior works that report on application-transparent, client-transparent, container-based fault tolerance mechanisms that support stateful applications. *NiLiCon*, as described in this chapter, is such a mechanism.

The VM-level fault tolerance techniques discussed in §2.6, in particular, Remus [CLM08], do support stateful applications and provide application transparency as well as client transparency. Hence, *NiLiCon* uses the same basic approach. Since container state does not include the entire kernel, the size of the periodic checkpoints can be expected to be smaller, potentially resulting in lower overhead when the technique is applied at the container level. On the other hand, there is a much tighter coupling between a container and the underlying kernel than between a VM and the underlying hypervisor. In particular, there is much more container state in the kernel (e.g., the list of open file descriptors of the applications in the container) than there is VM state in the hypervisor. Thus, implementing the warm backup replication scheme with containers is a more challenging task. *NiLiCon* is an existence proof that this challenge can be met.

The starting point of *NiLiCon*'s implementation is based on a tool called CRIU (Checkpoint/Restore in User Space) [cria], which is able to checkpoint a container under Linux. However, the existing implementation of CRIU and the kernel interface provided by Linux kernel incur high overhead for some of CRIU's operations. For example, our measurements show that collecting container namespace information may take up to 100ms. Hence, using the unmodified CRIU and Linux, it is not feasible to support the short checkpointing intervals (tens of milliseconds) required for client-server applications. An important contribution of our work is the identification and mitigation of all major performance bottlenecks in the current implementation of container checkpointing with CRIU.

The implementation of *NiLiCon* has involved significant modifications to CRIU and a few small kernel changes. We have validated the operation of *NiLiCon* and evaluated its overhead using seven benchmarks, five of which are server applications. For fail-stop failures, the recovery rate with *NiLiCon* was 100%. The performance overhead was in the range of 19%-67%. During normal operation, the CPU utilization on the backup was in the range of 6.8%-40%.

We make the following contributions: 1) Demonstrate a working implementation of the first container replication mechanisms that is client-transparent, application-transparent, and supports stateful applications; 2) Identify the major performance bottlenecks in the current CRIU container checkpoint implementation and the Linux kernel interface and propose mechanisms to resolve them.

The rest of the chapter is organized as follows. The key differences between *NiLiCon* and Remus are explained in §4.1. The design and operation of *NiLiCon* are discussed in §4.2. Key implementation optimizations are presented in §4.3. The experimental setup and evaluation results are presented in §4.4 and §4.5, respectively.

4.1 ***NiLiCon*** vs Remus

This section presents the key differences between *NiLiCon* and Remus. These differences are due to the fact that, unlike VMs with respect to the hypervisor, a significant part of the container state is in the kernel. This in-kernel state is a combination of processes state: file descriptors, virtual memory area (VMA), sockets, signals, process trees; and container state: control groups, namespaces, mount points, and file system caches. For checkpointing and restoring most of these state components, *NiLiCon* relies on existing CRIU code (§2.7).

NiLiCon does not rely on CRIU code for handling file system caching. CRIU expects the container to use a NAS (network attached storage), accessible from the original container host and the host on which the container checkpoint is restored. CRIU flushes the file system cache to the NAS after the checkpoint completes, thus committing that part of the state. With *NiLiCon*, flushing the file system cache at every epoch (tens of milliseconds) is not practical since, for disk-intensive applications, it may introduce prohibitive overhead of up to hundreds of milliseconds per epoch.

NiLiCon includes kernel changes to efficiently deal with file system caching. These changes add a new state that is maintained for pages in the page cache and inode cache entries: "Dirty but Not Checkpointed" (*DNC*). For checkpointing, a new system call, *fgetfc*, obtains all the *DNC* inode and page cache entries and clears the *DNC* state. For restoring a file system cache checkpoint, existing system calls are used, such as *chown* for the inode cache and *pwrite* for the page cache.

NiLiCon and Remus handle the backup differently. Remus maintains a ready-to-go backup VM — it commits state changes directly to the backup VM during each epoch. Thus, if the primary VM fails, Remus can resume the backup VM with minimal delay. With *NiLiCon*, maintaining a ready-to-go backup container is impractical. This is due to the latency of the large number of system calls that must be invoked in order to apply the in-kernel container state changes to the kernel — potentially adding up to hundreds of milliseconds. Hence, at the backup, *NiLiCon* maintains all the in-kernel container state in buffers, applying this state to the kernel only upon a failover.

Both *NiLiCon* and Remus pause the primary while state changes are saved in order to prevent inconsistent state changes. With Remus, once a VM is paused, its state can no longer be affected by packets incoming from the network. However, with *NiLiCon*, pausing the primary is insufficient since incoming packets can modify the primary state while the primary is paused. Thus, *NiLiCon* blocks incoming packets at the primary during checkpointing. With *NiLiCon*, incoming packets are also blocked during recovery at the backup. During recovery, the network namespace must be restored before restoring the sockets. If incoming packets are not blocked, an incoming TCP packet arriving after the network namespace is restored but before the relevant socket is restored would cause the kernel to send an RST (reset) packet to the client, breaking the connection.

4.2 *NiLiCon* Basics

This section presents the basic design and operation of *NiLiCon*, without the optimizations described in §4.3. The overall architecture of *NiLiCon* is shown in Figure 4.1. The core components are the primary and backup agents, that coordinate with the rest of the components and also perform the main task of checkpointing/restoring the container state.

While *NiLiCon* is focused on error recovery, an error detection mechanism is needed to

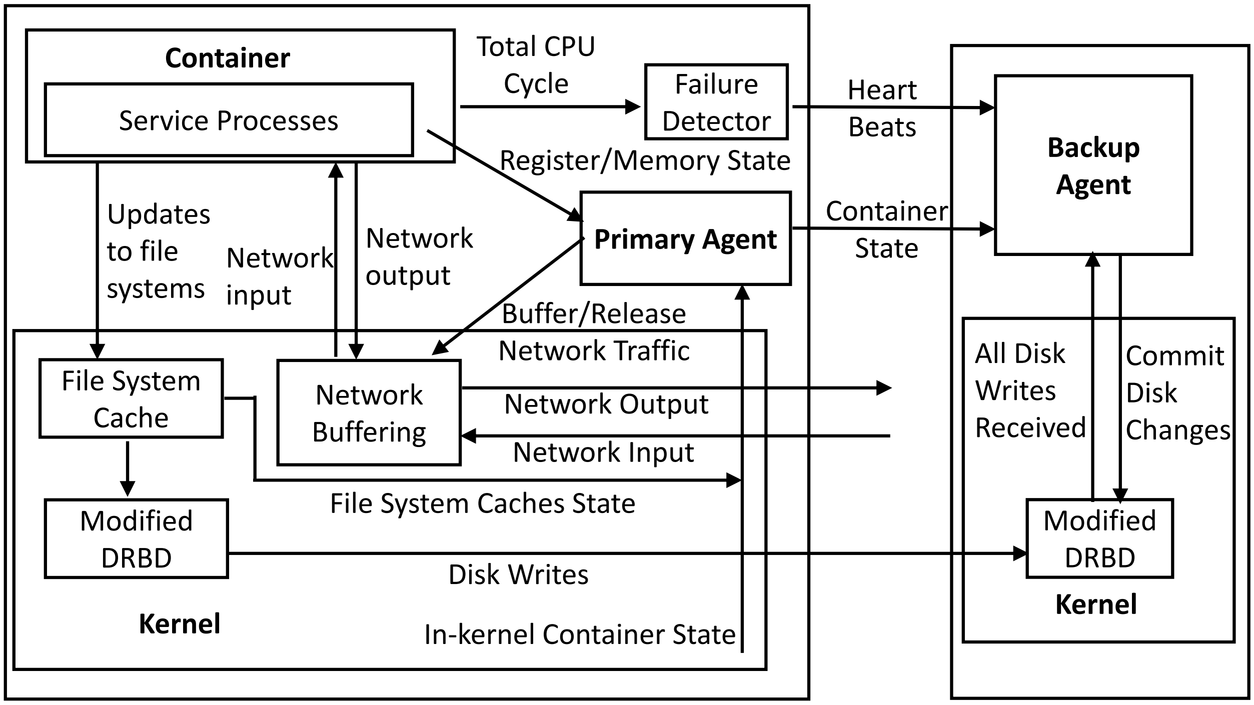


Figure 4.1: The architecture of *NiLiCon*.

initiate the recovery. As with most VM replication works [CLM08, WCJ18, DYJ13, LBV15, BS95], we assume a fail-stop failure model. *NiLiCon* includes a simple error detector that sends a heart beat from the primary agent to the backup agent every 30ms, as long as there is an increase in the CPU cycles usage of the container. The detector obtains the CPU cycles usage from the *cpuacct.usage* file of the container's control group. To prevent false alarms when the container is idle, the container includes a simple keep-alive process that wakes up every 30ms and executes 1000 instructions. If the backup agent fails to receive the heart beat in three consecutive 30ms intervals, recovery is initiated.

As with Remus, execution on the primary consists of a sequence of epochs (Figure 2.2). With the implementation of *NiLiCon* used in this chapter, the duration of the execution phase of each epoch is 30ms. *NiLiCon* uses the CRIU code to checkpoint/restore the container state. After the execution phase of an epoch completes, the primary agent forks a CRIU process to take an incremental checkpoint (§2.7) and send it to the backup agent.

When recovery is initiated, the backup agent forks a CRIU process to restore the container.

Most of *NiLiCon*'s code for handling network and disk I/O is from the Remus implementation in Xen (RemusXen) [rem]. For network I/O, the implementation is unchanged. The buffering and releasing is performed using the *sch_plug* kernel module in the mainstream kernel. For disk I/O, RemusXen uses a modified version of the DRBD module [drb] (a distributed software RAID implementation), as described in §2.6. We ported the RemusXen DRBD modifications to the latest version of DRBD, which is currently in the mainstream kernel.

During each epoch's execution phase, network output is buffered (§2.6). File system updates are tracked using the DNC state for page and inode cache entries (§4.1). Disk writes are sent asynchronously to the backup by the primary's DRBD module. The backup's DRBD module receives and buffers disk writes in memory. When the primary's execution phase completes, the primary agent stops the container using virtual signals (§2.7) and directs the network buffering module to block network input. The primary agent also directs the DRBD module to send to the backup a "barrier" to mark the end of this epoch's disk writes. The primary agent obtains the register and memory state from the parasite code (§2.7), as well as in-kernel container state, including the file system cache, from the kernel. The primary agent sends this state to the backup agent. The backup agent receives and buffers the state in memory. The primary agent then unblocks the network input and resumes the container to execute another epoch. Once the backup agent has received both the disk writes and container state, it sends an acknowledgment to the primary agent, which then releases the buffered outgoing packets. The backup then commits all the buffered disk, register and memory state, thus completing one checkpoint iteration.

If the backup agent detects a failure, recovery is initiated. The backup agent discards any uncommitted state and uses the committed state to create image files in a format that CRIU expects. It then forks a CRIU process to restore the container state. The container network namespace connects to the external network via a virtual bridge. During recovery, the backup agent disconnects the container network namespace from the virtual bridge, thus blocking network input (§4.1). After the container has been restored, the backup agent reconnects the container network namespace to the bridge, thus reestablishing the network connection.

4.3 Optimizations

The basic implementation of *NiLiCon* presented in §4.2 imposes a prohibitive performance overhead. This overhead is due to the checkpointing time, which is often hundreds of milliseconds. While such latency is acceptable for container migration, it is not for replication, that requires repeated high-frequency checkpointing. This section presents optimizations in the implementation of *NiLiCon* that brought the overhead down to a level competitive with the overhead of similar VM replication schemes. In addition, it presents a key optimization that reduced the recovery latency.

The CRIU developers have indicated that checkpointing is slow due to the prohibitive latencies to obtain in-kernel container state using existing kernel interfaces [tas]. The causes are: (1) a large number of system calls are needed; (2) some of the kernel interfaces provide extra information that is not needed for checkpointing, but is expensive to generate; and (3) some of the kernel interfaces provide information in a format that is expensive to generate and parse. In addition, the kernel lacks mechanisms for identifying modified in-kernel container state, requiring all the in-kernel container state to be checkpointed at every epoch.

An example of cause (1) above is that to obtain the state of memory-mapped files, the *stat* system call needs to be invoked for each one. Since memory-mapped files are used for dynamically-linked libraries, applications often have a large number of such files, resulting in high overhead. An example of cause (2) is related to obtaining the process memory state, stored in the VMAs (Virtual Memory Areas) maintained by the kernel. CRIU reads this information from */proc/pid/smaps*. However, *smaps* also provides a large number of

Table 4.1: Impact of *NiLiCon*'s performance optimizations.

Optimization	Overhead
Basic implementation	1940%
+ Optimize CRIU	619%
+ Cache infrequently-modified state	84%
+ Optimize blocking network input	65%
+ Obtain VMAs from *netlink*	53%
+ Add memory staging buffer	37%
+ Transfer dirty pages via shared memory	31%

page statistics, such as the number of clean/dirty pages in the VMA, that is not needed for container checkpointing. An example of cause (3) is that the *proc* and *sys* file systems provide the information as formatted text, instead of the binary format in which they exist in the kernel.

An ideal solution for the problems above would be to develop new kernel interfaces optimized for fast container checkpointing. However, this would require substantial modification to the underlying kernel. Instead, this section shows that, with only minor changes to the kernel, it is possible to achieve for container replication performance overhead that is competitive with that of VM replication.

Table 4.1 lists *NiLiCon*'s optimizations, described in the rest of this section, and their impact on the overhead for the *streamcluster* benchmark (§4.4). Altogether, these optimizations reduced the overhead from 1940% to 31%.

In total, *NiLiCon*'s implementation consists of 7494 lines of code (LOC). 257 LOC are in the main part of the kernel, mostly for dealing with the file system cache (§4.1). 1093 LOC are in two kernel modules: changes to DRBD (§4.2) and a module for tracking some in-kernel state changes (§4.3.2). In addition, we applied an 1140 LOC kernel patch from the CRIU developers [crib] that speeds up obtaining VMA information from the kernel (§4.3.4).

4.3.1 Optimizing CRIU

NiLiCon implements three optimizations to CRIU. The most important of these is a change in the way the pages from each incremental checkpoint are stored in the backup. CRIU uses a linked list of directories in the file system, each containing files containing an incremental checkpoint. For each page received in an incremental checkpoint, CRIU iterates through this linked list to possibly find and remove a previous copy of the page. *NiLiCon*'s optimization is to store the committed pages in a four-level radix tree, mimicking the implementation of the hardware page tables. The time to process each received page is thus short and independent of the number of previous checkpoints.

With the original CRIU implementation, the primary agent issues virtual signals to pause all the threads (§2.7), sleeps for 100ms, and then checks whether all the threads are paused. The goal is to avoid busy waiting (CPU usage) but this increases checkpointing time. *NiLiCon*'s optimization is to eliminate the sleep and implement continuous polling of the thread state. Even with our most system call intensive benchmarks, the average busy looping time is less than 1ms, resulting in negligible additional CPU usage.

The stock CRIU uses proxy processes at the primary and backup hosts that serve as intermediaries in the state transfer from the primary agent to the backup agent. This adds extra copies to the state transfer and complicates the overall structure. *NiLiCon*'s third optimization of CRIU is the removal of the proxy processes, allowing the primary agent to directly transfer state to the backup agent.

4.3.2 Caching Infrequently-Modified In-Kernel Container State

The most effective optimization in *NiLiCon* is based on the observation that part of the in-kernel container state is rarely modified and thus rarely changes between checkpoints. Hence, it is wasteful to retrieve all of this state for every checkpoint. We identified the following infrequently-modified in-kernel state components: control groups, namespaces, mount points, device files, and memory mapped files. As an example, the time to obtain these state components while executing *streamcluster* is approximately 160ms.

NiLiCon's optimization is to avoid retrieving infrequently-modified in-kernel state components if they have not changed since the last checkpoint. Instead, the values of these state components are cached and the cached values are included in the checkpoint sent to the backup.

The optimization described above requires the ability to detect when infrequently-modified state components are modified and retrieve their new values. We developed a kernel module that utilizes the *ftrace* debugging functionality provided by the kernel to detect these state change and inform the checkpointing agent. *Ftrace* has negligible overhead and allows kernel modules to add a "hook function" to any target kernel function such that, whenever the target function is called, the hook function is invoked.

NiLiCon uses *ftrace* to add hooks to target kernel functions that potentially modify the infrequently-modified state components listed above. Each hook function invokes the actual target function and then performs additional checks based on the argument and the return value of the target function as well as the identity of the calling thread. These checks determine whether there may have been a change in the infrequently-modified state of a thread in the container. If the check result is positive, a signal is sent to the primary agent. The hook function then returns with the return value of the target function. In our research prototype, we did not attempt to find and instrument *all possible* code paths that change the infrequently-modified state components. Instead, our implementation only covers the most common paths and that was sufficient for all of our benchmarks.

4.3.3 Optimizing Blocking Network Input

As explained earlier (§4.1), network input must be blocked and then unblocked during every epoch. CRIU does this using the firewall. However, setting up and removing firewall rules adds a 7ms delay during each epoch. Furthermore, if the dropped packets are part of a TCP connection establishment, this can introduce delays of up to three seconds.

NiLiCon's optimization is to use for network input the same mechanism used to buffer and release network output (§4.2). Input network packets arriving during checkpointing are buffered by a kernel module instead of being dropped. These packets are released to the container once the checkpoint completes. This implementation avoids long delays for TCP connection establishment and introduces a delay of only 43μs during checkpointing.

4.3.4 Optimizing Memory Checkpointing

Three optimizations that reduce the overhead of checkpointing memory address three corresponding deficiencies of the basic implementation: (1) VMA information of the processes is obtained using */proc/pid/smaps*, which is slow; (2) containers do not resume execution until all dirty memory pages have been transferred to the backup; and (3) the content of the dirty pages are transferred by the parasite code via a pipe, involving multiple system calls.

The developers of CRIU are aware of deficiency (1) and have proposed a kernel patch [tas] that uses the *netlink* functionality to transfer the memory mapping information. *NiLiCon* utilizes this patch to resolve deficiency (1).

NiLiCon resolves deficiency (2) using a staging buffer. During checkpointing dirty pages are first copied to a local staging buffer and later transferred to the backup after the container has resumed execution (as with Remus (§ 2.6)).

Deficiency (3) is resolved by using shared memory to transfer the dirty pages. Specifically, *NiLiCon* creates a shared memory region between the parasite code and the primary agent to allow for the parasite code to directly transfer dirty pages to the primary agent.

4.3.5 Reducing Recovery Latency

After recovery, the backup container needs to retransmit the unacknowledged packets to the client. At that time, TCP sockets in the backup are new and for new TCP sockets the default retransmission timeout is long — at least one second. This leads to an unnecessarily long recovery latency. *NiLiCon* resolves this problem with a small modification to the kernel TCP stack (two LOC). Specifically, if the socket is in the special repair mode (§2.7), *NiLiCon* sets the retransmission timeout to be the minimum value: 200ms.

4.4 Experimental Setup

This section presents the experimental setup used to evaluate *NiLiCon*, including the description of the experimental platform and the benchmarks.

For logistical reasons, two pairs of hosts were used in the evaluation. The first pair was used to measure the recovery rate and the recovery latency. Each of these hosts had 8GB of memory and dual quad-core Intel Xeon-E5520 CPU chips. The second pair was used to measure the performance overhead during normal operation. Each of these hosts was more modern, with at least 32GB memory and dual 18-core Intel Xeon CPU chips (one with E5-2695v4 chips and the other with Xeon Gold 6140 chips). Each pair of hosts were connected to each other via a dedicated 10Gb Ethernet link and connected to the client host via 1Gb Ethernet.

We used Fedora 29 Linux with kernel version 4.18.16. Containers were hosted by runC version 1.0.1 [run], a container runtime used in Docker to host and manage containers. The *NiLiCon* implementation was based on CRIU version 3.11. Experiments with VMs, were hosted by KVM with QEMU version 2.3.50, the latest version that supports micro-checkpointing (MC), KVM's implementation of Remus [qem]. VMs were fully virtual with paravirtual drivers. Each VM or container was hosted on a dedicate core and allocated 4GB physical memory.

Table 4.2: Benchmarks used to evaluate *NiLiCon*.

Benchmark	Description
Redis	key-value store, no persistence, 100K 1KB records, evaluated with 200K requests batch, each consists of 1K requests consisting of 50% reads and 50% writes.
SSDB	key-value store, full persistence, 100K 1KB records, evaluated with 200K requests batch, each consists of 1K requests consisting of 50% reads and 50% writes.
DJCMS	content management system platform that uses Nginx, Python, and MySQL, evaluated with concurrent requests on the administrator dashboard page.
Lighttpd	web server, evaluated with concurrent requests to a PHP script that watermarks an image.
Node	a Node.js program that searches through a database for a keyword and generates a static web page consisting of text and figures, evaluated with concurrent requests with random keywords.
Streamcluster	a kernel that solves online clustering problems, evaluated with native input suite.
Swaptions	a kernel that uses HJM framework to price a portfolio of swaption, evaluated with native input suite.

As summarized in Table 4.2, the evaluation was based on five server benchmarks: *Redis* [red], *SSDB* [ssd], *Node* [nod], *Lighttpd* [lig], and *DJCMS* [djc]; as well as two non-interactive CPU/memory-intensive PARSEC [Bie11] benchmarks: *streamcluster* and *swaptions*. Unless otherwise mentioned below, all benchmarks were configured with the default characteristics.

For *Redis* and *SSDB*, we use YCSB [CST10] to generate 2M requests with 100K 1KB records. And then we use a custom client with the *hiredis* library [hir] to batch and send these requests to *Redis*/*SSDB*. For *DJCMS*, *Lighttpd*, and *Node*, we use the SIEGE [sie] client to send to each of them concurrent requests. The original *Node* benchmark sends a response as a Facebook chat message, we modified it to reply with a static web page.

4.5 Evaluation

This section presents the validation (§4.5.1), recovery latency (§4.5.2), and the performance overhead (§4.5.3) of *NiLiCon*.

4.5.1 Validation

Fault injection is used to test *NiLiCon*'s ability to recover from container failures. Two microbenchmarks are used in addition to the benchmarks described in §4.4. The first microbenchmark stresses the handling of the disk, file system cache, and heap memory. It performs a mix of writes and read of random size (1-8192 bytes) to random locations in a file. An error is flagged if the data returned by a read differs from the data written to that location earlier. The second microbenchmark stresses the handling of the kernel's network stack as well as a server application's stack in memory. A client sends a message of random size (1B-2MB) to the server, the server saves it on its stack and then sends it back to the client. The client flags an error if the message it receives is different or if the TCP connection is broken.

All the benchmarks are configured to run for at least 60 seconds. A fault is injected at a random time during the middle 80% of the benchmark's execution time, thus triggering recovery on the backup host. A fail-stop failure is emulated, using the *sch_plug* module, by blocking incoming and outgoing traffic on all the primary container's network interfaces. For the microbenchmarks, recovery is considered successful if no errors are flagged. For *Redis* and *SSDB*, the client program records the value it stores with each key, compares that value with the value returned by the corresponding *get* operation, flagging an error if there is a mismatch. Recovery is considered successful if no errors are flagged. For all the other benchmarks, the container output is validated by comparison with a golden copy.

Each benchmark is executed 50 times. We find that in all the executions *NiLiCon* is able to detect and recover from the container failure with no broken network connections! We also manually unplug the network cable a few times for each benchmark, and verify that *NiLiCon* is able to recover from these failures as well.

4.5.2 Recovery Latency

We measure the service interruption duration due to a fault using server applications. This duration is the sum of the detection and recovery latencies, which increase the response time at the client. With the detection mechanism used by *NiLiCon* (§4.2), the detection latency is, on average, 90ms. Hence, the recovery latency is obtained by subtracting 90ms from the average increase in response time.

Two benchmarks are used: *Net* and *Redis*. *Net* is a microbenchmark, where the client sends 10 bytes to the server and the server responds with the same 10 bytes. For *Redis*, a client uploads (sets) approximately 100MB of data to the server. Next, one client keeps sending batched requests to stress the server, resulting in approximately 30% CPU usage. Each member of another set of four clients continuously sends a single get or set request at a time. The service interruption latency measurements are based on the responses to these latter requests.

Table 4.3: Recovery latency breakdown.

	Restore	ARP	TCP	Others	Total
Net	218ms (71%)	28ms (9%)	54ms (18%)	7ms (2%)	307ms
Redis	314ms (84%)	28ms (8%)	23ms (6%)	7ms (2%)	372ms

Each experiment is executed ten times and the variations in the measured service interruption latencies are within 10% of the mean. Table 4.3 shows the key components of the recovery latency. *Restore* is the time to restore the container state. *ARP* is the time to broadcast a gratuitous ARP reply to advertise the new MAC address. *TCP* is the portion of the delay for packet retransmission (§4.3.5) that is not overlapped with other recovery actions. The recovery latency difference between *Net* and *Redis* is due to the additional time to restore the 100MB memory state of *Redis*.

4.5.3 Performance Overhead During Normal Operation

This subsection presents the performance overhead of *NiLiCon* during normal operation as well as additional measurements to help explain the high-level results. The results include comparisons with MC, the Remus implementation on KVM [qem]. MC only supports disk-IO over a networked file systems. Thus, due to network buffering, MC has a significant extra delay for each file access, giving *NiLiCon* an unfair performance advantage in many comparisons. We have verified that the disk state replication we use (§4.2) has no performance impact with our benchmarks. Hence, with MC, we use a local disk, without disk state replication, even though this does not provide correct handling of disk state.

Unless otherwise specified, the non-interactive benchmarks use the *native* input set [Bie11] and are set to utilize four worker threads. For the server benchmarks, clients are configured to "saturate" the server to reach its maximum request processing rate.

Overhead with Maximum CPU Utilization. With non-interactive applications, such as *streamcluster* and *swaptions*, the performance overhead is the relative increase in

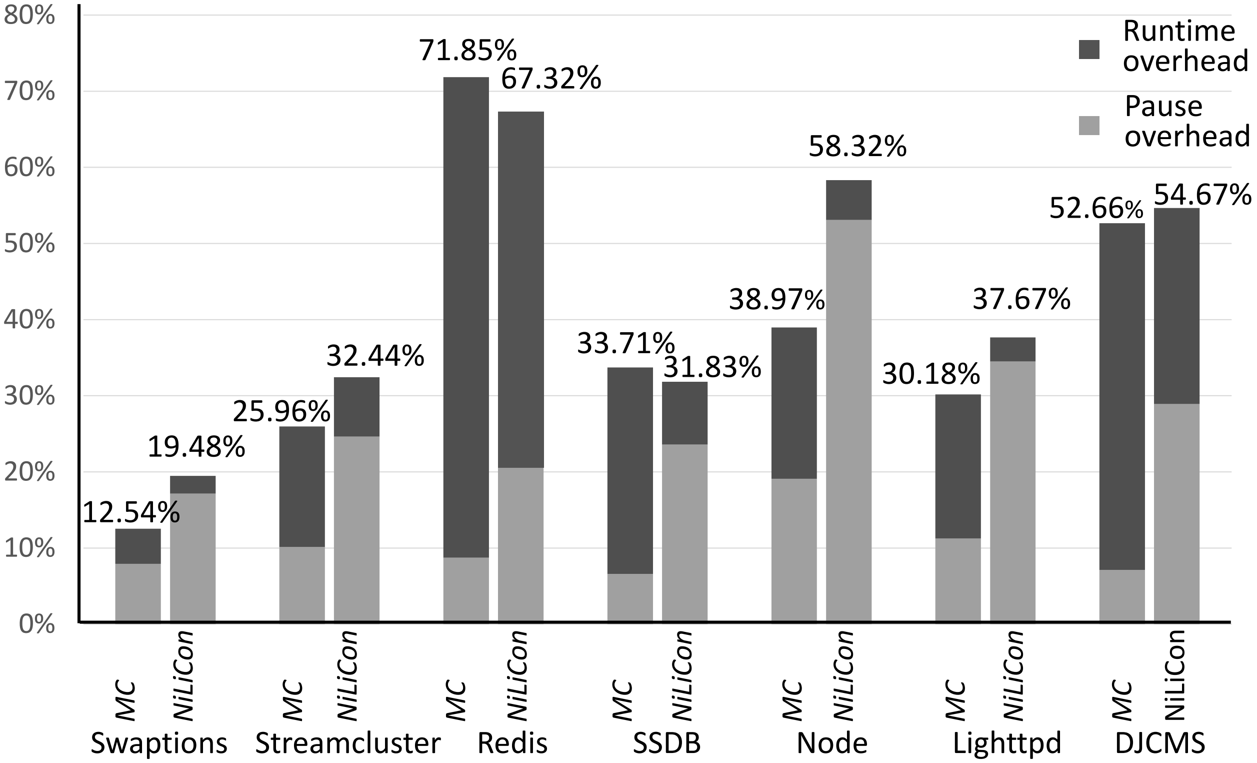


Figure 4.2: Performance overhead comparison between MC and *NiLiCon* with breakdown of sources of overhead.

execution time. For server applications, the performance overhead is the relative reduction in maximum throughput.

Figure 4.2 presents the performance overhead of *NiLiCon*, with a comparison against MC. Each of the benchmarks is executed for 100 times. The results have a coefficient of variation of less than 2%. The overheads of *NiLiCon* and MC are comparable and are due to two sources: the pause time of the container/VM for checkpointing and the overhead during normal operation for tracking the pages that are modified during each epoch. For MC, most of the overhead is the runtime overhead while for *NiLiCon*, except *Redis* and *DJCMS*, most of the overhead is the stop overhead.

NiLiCon's runtime overhead component is lower than MC's for all the benchmarks. This is mainly due to the high overhead of VM exit and entry operations needed in MC for

Table 4.4: Average pause time & #dirty pages per epoch, MC and *NiLiCon*.

		SwapT	StreamC	Redis	SSDB	Node	Lhttpd	DJCMS
Pause	MC	2.4ms	3.0ms	9.3ms	3.0ms	9.4ms	4.8ms	4.5ms
	NiLiCon	5.1ms	7.4ms	18.9ms	10.4ms	38.2ms	25.0ms	19.1ms
DPage	MC	212	462	6.2K	1107	6.4K	2.9K	2.8K
	NiLiCon	46	303	6.3K	590	5.4K	1.6K	3.0K

tracking modified pages.

Table 4.4 presents the average pause time and number of dirty pages per epoch. For *NiLiCon*, the pause time is higher since *NiLiCon* needs to obtain container in-kernel state using the slow kernel interface (§4.3). This is the main reason for *NiLiCon*'s higher overhead for most of the benchmarks. For example, the *Node* benchmark has the highest pause time since 128 clients are required to reach saturation. The result is that the container has a large number of sockets and *NiLiCon* spends around 13ms collecting the socket states.

Since the pause time is the main source of overhead for *NiLiCon*, Table 4.5 provides a detailed breakdown of the pause time. The largest factor is shown on the first row – the time to retrieve various components of the in-kernel container state. These state components have not been individually discussed in this chapter since, out of these, we have not identified any one component or a small subset of components responsible for most of the overhead. The second largest factor, as shown in the second row, is the time to copy dirty pages to the staging buffer.

Pause Time and Transferred State Variations. The pause time and size of the state transferred per epoch vary among the epochs of an application as well as from one application to another. Table 4.6 shows these variations, providing the 10, 50, and 90 percentile values of these metrics. The results indicate that the impact of *NiLiCon* on an application's performance can vary significantly over time (e.g., due to pause time for

Table 4.5: Breakdown of the pause time. Sorted by the average portion across all benchmarks. Reset dirty pages is the time to change all the tracked dirty pages back to clean. Mprotect is the time to modify/restore the permission of read-only memory regions.

	SwapT	StreamC	Redis	SSDB	Node	Lhttpd	DJCMS
Collect Other State	66%	48%	25%	42%	25%	34%	25%
Copy Dirty Pages	2%	8%	50%	8%	24%	10%	23%
Find Dirty Pages	18%	15%	4%	15%	7%	13%	18%
Reset Dirty Pages	1%	17%	10%	21%	9%	8%	14%
Collect Sockets	3%	2%	7%	9%	32%	13%	4%
Collect VMAs	4%	4%	1%	2%	1%	13%	8%
Inject ParaCode	5%	4%	1%	2%	1%	5%	3%
Mprotect	1%	2%	2%	1%	1%	4%	5%

Table 4.6: Pause time and transferred state size for *NiLiCon*.

		SwapT	StreamC	Redis	SSDB	Node	Lhttpd	DJCMS
Pause	10%	5.1ms	6.3ms	15ms	9ms	38ms	20ms	16ms
	50%	5.1ms	6.4ms	18ms	10ms	41ms	25ms	18ms
	90%	5.2ms	13.1ms	20ms	11ms	46ms	35ms	21ms
State	10%	189K	257K	17.9M	1.43M	22.7M	2.05M	53.1K
	50%	193K	269K	24.2M	2.88M	24.2M	7.17M	9.5M
	90%	201K	306K	30.0M	3.41M	25.2M	14.65M	13.3M

Table 4.7: Core utilization on active and backup hosts.

	SwapT	StreamC	Redis	SSDB	Node	Lighttpd	DJCMS
Active	3.96	3.91	0.98	1.70	1.01	3.95	1.41
Backup	0.07	0.08	0.28	0.12	0.40	0.18	0.26

streamcluster, state size for *DJCMS*).

The main components of the transferred state are the dirty pages and the read/write queues of TCP sockets. For our benchmarks, the dirty pages portion is in the range of 85% to over 95%.

Backup CPU Utilization. A benefit of schemes like Remus and *NiLiCon* over active replication (§2.6) is lower utilization of the CPU on the backup host. Table 4.7 shows an approximation of the CPU utilization on the backup host with *NiLiCon*. These measurements are done by pinning all of *NiLiCon* activities on the backup to a single core and running it with high priority (*niceness* -20). A simple program that continuously increments a counter is pinned to the same core, running with low priority (*niceness* 19). The core utilization is derived by comparing the rate at which the counter increments on the backup to the rate at which it increments on an otherwise idle dedicated core. For comparison, similar core utilization measurements were done on a host executing the benchmarks *without* replication.

As shown in Table 4.7, with *NiLiCon* the core utilization on the backup host is indeed significantly lower than on an active host. On an active host, the utilization is evenly divided among the container's four cores. Most of the backup CPU cycles are spent reading the state transferred from the primary. This processing increases when the granularity at which this state arrives is finer, since more read system calls must be invoked. For example, a significant portion of *Node*'s transferred state is the state TCP sockets, which arrives at the backup in small chunks. Thus, *Node*'s backup CPU utilization is higher than *Redis*'s even though they have similar sizes of transferred state.

Table 4.8: Response latency with a single client.

	Redis	SSDB	Node	Lighttpd	DJCMS
Stock	3.1ms	93ms	2.4ms	285ms	89ms
NiLiCon	36.9ms	143ms	39.4ms	542ms	245ms

Request Response Latency. For server applications, one of the impacts of schemes like Remus and *NiLiCon* is to increase the latency of responding to requests. There are two causes for this increase: (1) more time is spent processing each request due to checkpointing and runtime overhead; and (2) the response outgoing packets are delayed due to buffering (§2.6). Table 4.8 compares the response times with *NiLiCon* and with the stock server application (no replication). In all cases there is only one client. For benchmarks with short processing time, such as *Node* and *Redis*, overhead (2) dominates. For the other benchmarks, overhead (1) dominates.

Scalability. We present the scalability of *NiLiCon* with respect to the number of threads or processes in the container as well as the number of clients sending requests to a server application executing in the container.

To evaluate the impact of varying the number of threads, *streamcluster* is executed with 1 to 32 threads, with another core allocated to the container for each additional thread. *NiLiCon*'s performance overhead increases from 23% to 52%, respectively. This is caused by: (1) the average time to retrieve the per-thread states (e.g., registers, signal mask, scheduling policies) increases from 148μs to 4ms; (2) the process' memory footprint increase from 49K to 111K pages, increasing the time to identify dirty pages from 1441μs to 2887μs; and (3) due to the increased number of cores, more processing is performed per epoch, causing the number of dirty pages to increase from 121 to 495, resulting in increased runtime overhead for tracking dirty pages and increased memory copying time, from 263μs to 1099μs.

Lighttpd is used to evaluate the impact of: (1) varying the number of clients, and (2) varying the number of processes in the container. For (1), the number of *Lighttpd* processes is fixed at 4 and the number of clients is varied from 2 to 128. With 32 or fewer clients, the overhead is approximately 34%. This overhead increases to 45% with 128 clients. This overhead increase is almost entirely caused by the increased time to checkpoint socket states: from 1.2ms with 2 clients to 13ms with 128 clients.

The number of *Lighttpd* processes is varied from 1 to 8, with another core allocated to the container for each additional process. The overhead increases from 23% to 63%, respectively. This is caused by: (1) the average time to retrieve the per-process states increased from 6.5ms to 28.7ms; (2) with more cores, more clients are needed (from 2 to 8) to saturate the server, requiring more sockets, increasing the time to retrieve socket states from 1.2ms to 3.8ms; (3) with more cores, more processing is performed per epoch, causing the number of dirty pages to increase from 391 to 2062, resulting in increased runtime overhead for tracking dirty pages and increased memory copying time, from 519μs to 3.5ms.

4.6 Summary

The ability to provide application-transparent fault tolerance can be highly beneficial in data centers and cloud computing environments in order to provide high reliability for legacy applications as well as to avoid burdening application developers with the need to develop customized fault tolerance solutions for their particular applications. VM replication was developed and has been continuously enhanced in order to provide such application-transparent fault tolerance. This has led to not only many publications but also several commercial products. Due to their lower resource requirements and reduced management costs, in many situations there are compelling reasons to deploy containers instead of VMs to provide an isolation and multitenancy layer. Hence, there is strong motivation to explore the possibility of using container replication to provide transparent fault tolerance.

We have presented *NiLiCon*, which, to the best of our knowledge, is the first working prototype of container replication that provides fault tolerance in a way that is transparent to both the protected applications and external clients. *NiLiCon* does this by providing seamless failover from a failed container to a backup container on a different host. *NiLiCon* uses the basic mechanism developed for VM replication. However, in its implementation it overcomes unique challenges due to the tight coupling between containers and the underlying kernel. *NiLiCon*'s implementation is based on CRIU, an open source container checkpointing and migration tool. However, the overhead of the available CRIU is too high for use in replication. *NiLiCon* implements critical optimizations that reduce this overhead, resulting in a tool with performance overhead that is competitive with the overhead of VM replication schemes.

CHAPTER 5

HyCoR: Fault-Tolerant Replicated Containers based on Checkpoint and Deterministic Replay

As discussed in Chapter 2, replication has long been used to implement application-transparent fault-tolerance mechanisms. The two approaches for replication are: (1) high-frequency transfer of the primary replica state (checkpoint), at the end of every execution *epoch*, to an inactive backup, so that the backup can take over if the primary fails [CLM08]; and (2) active replication, where the backup mirrors the execution on the primary so that it is ready to take over. *NiLiCon* discussed in the last chapter is based on the first approach. The second approach is challenging for multiprocessor workloads, where there are many sources of nondeterminism. Hence, it is implemented in a leader-follower setup, where the outcomes of identified nondeterministic events, namely synchronization operations and certain system calls, on the primary are recorded and sent to the backup, allowing the backup to deterministically replay their outcomes [GHY14, MGT17].

A key disadvantage of the first approach above is that, for consistency between server applications and their clients after failover, outputs must be delayed and released only after the checkpoint of the corresponding epoch is committed at the backup (§2.6). Since checkpointing is an expensive operation, for acceptable overhead, the epoch duration is typically set to tens of milliseconds. Since, on average, outputs are delayed by half an epoch, this results in delays of tens of milliseconds. A key disadvantage of the second approach is that it is vulnerable to even rare replay failures due to untracked nondeterministic events, such as those caused by data races.

This chapter presents a novel fault tolerance scheme, based on container replication, called *HyCoR* (Hybrid Container Replication). *HyCoR* overcomes the two disadvantages above using a unique combination of periodic checkpointing [CLM08], externally-deterministic replay [CZG15, VLW11, LWV10, AS09, PZX09], user-level recording of nondeterministic events [LWV10, MGT17], and failover of network connections [ABE01, AT01]. A critical feature of *HyCoR* is that the checkpointing epoch duration does not affect the response latency, enabling *HyCoR* to achieve sub-millisecond added delay (§5.4.2). This allows adjusting the epoch duration to trade off performance and resource overheads with recovery latency and vulnerability to untracked nondeterministic events. The latter is important since, especially legacy applications, may contain data races (§5.4.4). *HyCoR* is focused on dealing with data races that rarely manifest and are thus more likely to remain undetected. Since *HyCoR* only requires replay during recovery and for the short interval since the last checkpoint, it is inherently more resilient to data races than schemes that rely on replay of the entire execution [GHY14]. Furthermore, *HyCoR* includes a simple timing adjustment mechanism that results in a high recovery rate even for applications that include data races, as long as their rate of unsynchronized writes is low.

Replication can be at the level of VMs [BS95, CLM08, RZP19, WCJ18, DYJ13], processes [GHY14, MGT17], or containers (§4). We believe that containers are the best choice for mechanisms such as *HyCoR*. Applying *HyCoR*'s approach to VMs would be complicated since there would be a need to track and replay nondeterministic events in the kernel. On the other hand, with processes, it is difficult to avoid potential name conflicts upon failover. A simple example is that the process ID used on the primary may not be available on the backup. While such name conflicts can be solved, the existing container mechanism already solves them efficiently.

With *HyCoR*, execution on the primary is divided into epochs and the primary state is checkpointed to an inactive backup at the end of each epoch [CLM08]. Upon failure of the primary, the backup begins execution from the last primary checkpoint and then deterministically replays the primary's execution of its last partial epoch, up to the last external output. The backup then proceeds with live execution. To support the backup's deterministic replay, *HyCoR* ensures that, before an external output is released, the backup has the log of nondeterministic events on the primary since the last checkpoint. Thus, external outputs are delayed only by the time it takes to commit the relevant last portion of the log to the backup.

There are several other works combining checkpointing with externally-deterministic replay for replication [LWV10, GHY14, CSX16, KDC05, MGT17]. However, Respec [LWV10] requires an average external output delay greater than half an epoch and is based on *active* replication. [CSX16, KDC05] do not provide support for execution on multiprocessors. See §2.6 for additional discussion. Furthermore, these prior works do not provide an evaluation of recovery rates and are not designed or evaluated for containers.

We have implemented a prototype of *HyCoR* and evaluated its performance and reliability using eight benchmarks. *NiLiCon* (§4) is the basis for the implementation of checkpointing and restore. The rest of the implementation is new, involving instrumenting standard library calls at the user level, user-level agents, and small kernel modifications. With 1s epochs, *HyCoR*'s performance overhead was less than 59% for all eight benchmarks. With more conservative 100ms epochs, the overhead was less than 68% for seven of the benchmarks and 145% for the eighth. *HyCoR* is designed to recover from fail-stop failures. We used fault injection to evaluate *HyCoR*'s recovery mechanism. For all eight benchmarks, after data races identified by ThreadSanitizer [thr] were resolved, *HyCoR*'s recovery rate was 100% for 100ms and 1s epochs. Three of the benchmarks originally included data races. For two of these, without any modifications, with 100ms epochs and *HyCoR*'s timing adjustments, the recovery rate was over 99.4%.

We make the following contributions: 1) A novel fault tolerance scheme based on container replication, using a unique combination of periodic checkpointing, deterministic replay of multiprocessor workloads, user-level recording of non-deterministic events, and an optimized scheme for failover of network connections. 2) A practical "best effort" mechanism that enhances the success rate of deterministic replay in the presence of data races 3) A thorough evaluation of *HyCoR* with respect to performance overhead, resource overhead, and recovery rate, demonstrating the lowest reported external output delay compared to competitive mechanisms.

An overview of *HyCoR* is presented in §5.1. *HyCoR*'s implementation is described in §5.2, with a focus on key challenges. The experimental setup and evaluation are presented in §5.3, and §5.4, respectively. Limitation of *HyCoR* and of our prototype implementation are described in §5.5.

5.1 Overview of *HyCoR*

HyCoR provides fault tolerance by maintaining a primary-backup pair with an inactive backup that takes over when the primary fails. As discussed earlier, this is done using a hybrid of checkpointing and deterministic replay [CSX16]. The basic checkpointing mechanism is based on *NiLiCon* (§4).

Figure 5.1 shows the overall architecture of *HyCoR*. The primary records nondeterministic events: operations on locks and nondeterministic system calls. The record and replay are done at the user level, by instrumentation of glibc source code. When the primary executes, the instrumented code invokes functions in a dedicated RR (Record and Replay) library that create logs used for replay. There is a separate log for each lock. For each thread, there is a log of the nondeterministic system calls it invoked, with their arguments and return values. Details are presented in §5.2.1.

When the container sends a reply to a client, the RR library collects the latest entries (since the last transmission) of the nondeterministic event logs and sends them to the backup. To ensure consistency upon failover, the reply is not released until the backup receives the relevant logs.

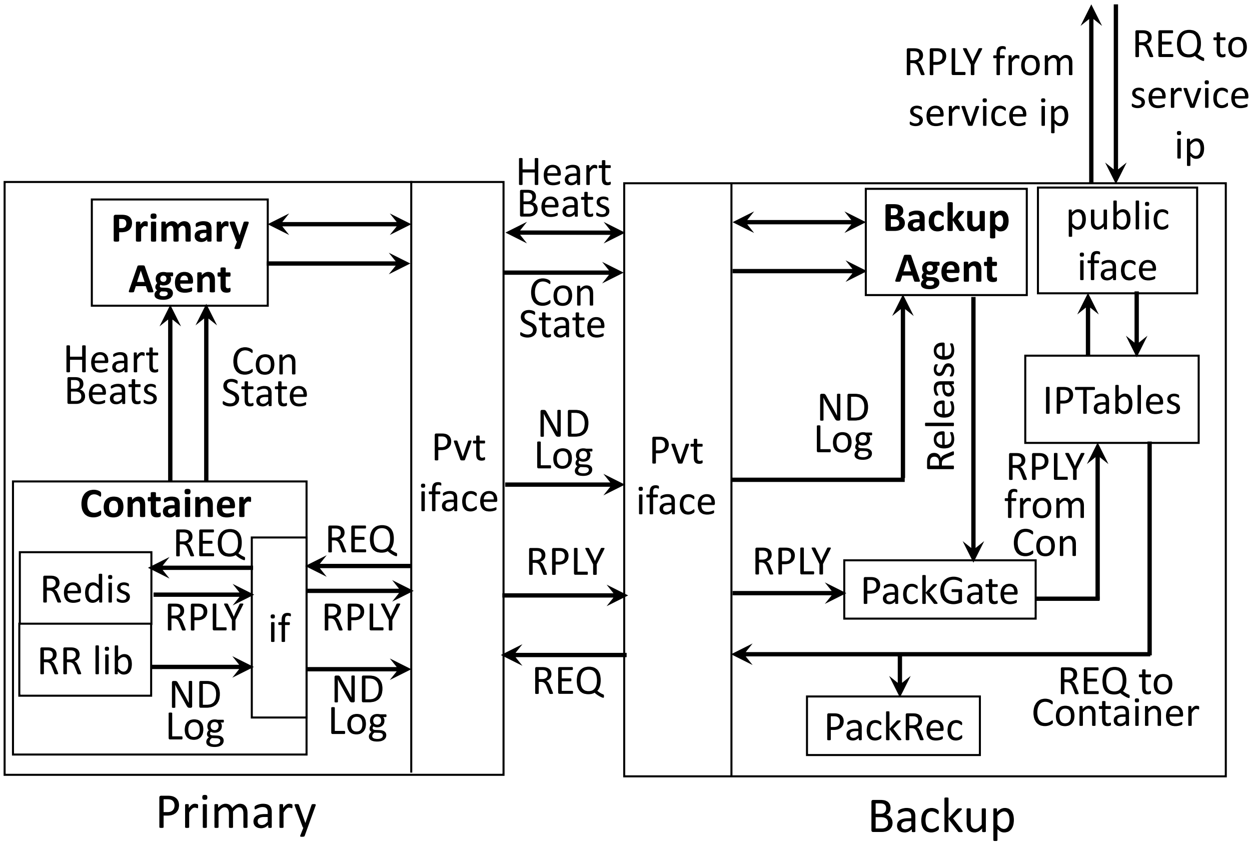


Figure 5.1: Architecture of *HyCoR*. (ND log: non-deterministic event log).

HyCoR does not guarantee recovery in the presence of data races. Specifically, unsynchronized accesses to shared memory during the epoch in which the primary fails may cause replay on the backup to fail to correctly reproduce the primary's execution, leading the backup to proactively terminate. However, *HyCoR* includes a simple "best-effort" mechanism that increases the probability of success in such circumstances for application with a low rate of unsynchronized accesses to shared memory (§5.2.6). With this mechanism, the order and timing of returns from nondeterministic system calls by *all* the threads is recorded during execution on the primary. During replay, the recorded order and relative timing are enforced.

If the primary fails, network connections must be maintained and migrated to the backup [ABE01, ZMA09, AT01, AT09]. Like CoRAL [AT01, AT09], requests are routed through backup by advertising the service IP address in the backup. Unlike FT-TCP [ABE01, ZMA09] or CoRAL, replies are also routed through the backup, resulting in lower latency (§5.2.3).

As with most other state replication work [CLM08, WCJ18, RZP19], *HyCoR* assumes fail-stop failures. Either the primary or the backup may fail. Heartbeats are exchanged between the primary and backup so failures are detected as missing heartbeats. Handling of primary failures have already been discussed. If the backup fails, the primary configures its network, advertises the service IP address, and communicates with the clients directly.

5.2 Implementation

This section presents the implementation of *HyCoR*, focusing on the mechanisms used to overcome key challenges. *HyCoR* is implemented mostly at the user level but also includes small modifications to the kernel. At the user level, the implementation includes: agent processes on the primary and backup hosts that run outside the replicated container; a special version of the glibc library (that includes Pthreads), where some of the functions are instrumented (wrapped), used by the application in the container; and a dedicated RR (record and replay) library, that provides functions that actually perform the record and replay of nondeterministic events, used by the application in the container.

The kernel modifications include: an ability to record and enforce the order of access to key data structures (§5.2.1); support for a few variables shared between the kernel and RR library, used to coordinate checkpointing with record and replay (§5.2.2); and a new queueing discipline kernel module used to pause and release network traffic (§5.2.3).

In the rest of this section, §5.2.1 presents the basic record and replay scheme. §5.2.2 deals with the challenge of integrating checkpointing with record and replay. §5.2.3 presents the handling of network traffic. The transition from replay to live execution is discussed in §5.2.4. The performance-critical operation of transmitting the nondeterministic event log to

```
 1 int read (fd, buf, nbytes):
 2 before:
 3     replayed, ret, err =
 4        __read_before(fd, buf, nbytes);
 5
 6     if (replayed):
 7        if (ret == -1):
 8           set_errno(err)
 9        return ret;
10
11     ret = __real_read(fd, buf, nbytes);
12
13     goto_before = __read_after(fd,
14           buf, nbytes, ret, errno);
15
16     if (goto_before):
17        goto before;
18
19     return ret;
```

Figure 5.2: Pseudo Code for recording and replaying read. L3 - L14 is explained in §5.2.1, L16 - L17 is explained in §5.2.2.

```
 1 int lock(lock):
 2 before:
 3     replayed, ret = __lock_before(lock);
 4
 5     if (replayed && ret != 0)
 6        return ret;
 7
 8     ret = __real_lock(mutex);
 9
10     goto_before = __lock_after(mutex, ret);
11
12     if (goto_before):
13        unlock(mutex);
14        goto before;
15
16     return ret;
```

Figure 5.3: Pseudo Code for recording and replaying acquiring locks. L3 - L10 is explained in §5.2.1, L12 - L14 is explained in §5.2.2.

the backup is explained in §5.2.5. §5.2.6 presents our best-effort mechanism for increasing the probability of correct replay in the presence of infrequently-manifested data races.

5.2.1 Nondeterministic Events Record/Replay

To minimize overhead and implementation complexity, *HyCoR* records synchronization operations and system calls at the user level. This is done by code added in glibc before (*before hook*, Fig 5.2: L4 and Fig 5.3: L3) and after (*after hook*, Fig 5.2: L13 and Fig 5.3: L10) the original code. Recording is done in the after hook, replay is in the before hook.

For each lock there is a log of lock operations in the order of returns from those operations. The log entry includes the ID of the invoking thread and the return value. The return values is recorded to handle the trylock variants as well as errors. During replay, in most cases synchronization operations must actually be performed in order to properly enforce the correct semantics. However, if the recorded return value indicates that the thread failed to acquire the lock, the thread directly returns instead of trying to acquire the lock (Fig 5.3: L5 - L6). For each lock, the total ordering of returns from each operation is enforced. This is not really necessary for reader-writer locks and trylock operations. However, it simplifies the implementation and there are minimal negative consequence since these events are relatively rare.

For each thread, there is a log of invoked system calls. The log entry includes the parameters and return values. During replay, the recorded parameters are used to detect divergence (replay failure). For some functions, such as gettimeofday(), replay does not involve the execution of the function and the recorded return values are returned. However, as discussed in §5.2.4, functions, such as open(), that involve the manipulation of kernel state, are actually executed during replay.

A key challenge is the replay of system calls that are causally dependent. These functions interact with the kernel and *HyCoR* does not replay synchronization within the kernel [LVN10]. Thus, for example, if two threads invoke open() at approximately the same time, without user-level synchronization, the access within the kernel to the file descriptor table may occur in a different order during record and replay. As a result, during replay, each thread would not obtain the same file descriptor as it did during the original execution.

To meet the above challenge, *HyCoR* uses a modified version of the Rendezvous mechanism in Scribe [LVN10]. Specifically, the kernel is modified to maintain an access sequence number for each shared kernel resource, such as the file descriptor table. Each thread registers the address of a per-thread variable with the kernel. When the thread executes a system call accessing a shared resource, the kernel increments the sequence number and copies its value to the registered address. At the user level, this sequence number is attached to the corresponding system call log entry. During replay, the before and after hooks enforces the recorded execution order.

5.2.2 Integrating Checkpointing with Record/Replay

Two aspects of *HyCoR* complicate the integration of checkpointing with record/replay: I) the RR library and the data structures it maintains are in the user level and are thus part of the state that is checkpointed and restored with the rest of the container state; and II) checkpointing is triggered by a timer in the agent, external to the container, and is thus not synchronized with the recording of nondeterministic events on the primary.

Based on the implementation described so far, the above complications can lead to the failure of *HyCoR* in two key scenarios: (1) a checkpoint may be triggered while the RR library is executing code that must not be executed in the replay mode, such as sending the nondeterministic event log to the backup; (2) a checkpoint may be triggered while a thread's execution falls between the beginning of a before hook and the end of an after hook ((Fig 5.2: L5 - L12, Fig 5.3: L4 - L9), potentially resulting in a state from which replay cannot properly proceed;

To handle Scenario (1), *HyCoR* prevents the checkpoint from occurring while any application thread is executing RR library code. Each thread registers with the kernel the address of a per-thread *in_rr* variable. In user mode, the RR library sets/clears the *in_rr*

```
1  void enter_rr(void):
2     if (mode is record):
3        if (checkpoint flag is set):
4           dummy_syscall();
5        in_rr = 1;
6
7  void leave_rr(void):
8     if (mode is record):
9        in_rr = 0;
10       if (checkpoint flag is set):
11          dummy_syscall();
```

Figure 5.4: Pseudo Code for Entering and Leaving RR library.

when it respectively enters/leaves the hook function (Fig 5.4: L4, L9). An addition to the kernel code that handles the *freezer* virtual signal (§2.7) prevents the thread from being paused if the thread's *in_rr* flag is set. However, the virtual signal remains pending. To prevent checkpointing from being unnecessarily delayed, after checkpointing is requested by the agent, threads are paused immediately before entering or after returning from RR library code. A *checkpointing* flag, shared between the agent that controls checkpointing and the RR library code, is used by the agent to indicate that checkpointing is requested, causing the RR library code to invoke a *do nothing* system call ((Fig 5.4: L3 - L4, L10 - L11)), thus allowing the virtual signal to pause the thread.

Scenario (2) cannot be handled as Scenario (1) since preventing checkpointing from occurring while a thread is between the before hook and after hook could delay checkpointing for a long time if the thread is blocked on a system call, such as read(). To handle this problem, *HyCoR* uses three variables: two per-thread flags – *in_hook* and *syscall_skipped*, as well as a global *current_phase* variable. The addresses of these variables are registered with the kernel and are accessed by kernel modifications required by *HyCoR*. The *current_phase* variable is in memory shared between the agent and the applications in the container (the RR library code). It indicates the current execution phase of the container and is thus set to *record*, *replay*, or *live*. In the record phase, *in_hook* is set in the before hook and cleared in the after hook. Flag *syscall_skipped* is used to indicate whether, during the record phase, the checkpoint was taken before or after executing the system call. This flag is cleared in the before hook. In kernel code executing a system call, if *current_phase* is set to *replay* and *in_hook* is set, the system call is skipped and *syscall_skipped* is set.

Replay is performed in the before hook (§5.2.1). During replay, if the after hook finds that *in_hook* is set, that indicates that checkpointing occurred between the before and after hooks. Thus, if *current_phase* is *replay* and *in_hook* is set, the after hook passes control back to the before hook (Fig 5.2: L13 - L17). This allows the system call to be correctly replayed. For system calls that are actually executed during replay (§5.2.1), there is a need to determine whether the system call was actually invoked during the record phase. If it was, the system call must not be invoked again during replay. This required determination is accomplished based on the *syscall_skipped* flag.

The key problem in Scenario (2) is relevant for lock operations as well as for system calls. The solution described above for system call is thus also used, in a simplified form, for lock operations. In this case, the *syscall_skipped* flag is obviously not used. In the after hook, if *in_hook* is found to be set, the lock is released and control is passed to the before hook, thus allowing enforcement of the order of lock acquires (Fig 5.3: L10 - L14).

5.2.3 Handling Network Traffic

The current *HyCoR* implementation assumes that all network traffic is via TCP. To ensure failure transparency with respect to clients, there are three requirements that must be met: (1) client packets that have been acknowledged must not be lost; (2) packets to the clients that have not been acknowledged may need to be resent; (3) packets to the clients must not be released until the backup is able to recover the primary state past the point of sending those packets.

Requirements (1) and (2) have been handled in connection with other mechanisms, such as [ABE01, ZMA09, AT01, AT09]. With *HyCoR*, this is done by mapping the advertised service IP address to the backup. Incoming packets are routed through the backup, where they are recorded by the PackRec thread in the agent, using the *libpcap* library. Outgoing packets are also routed through the backup. To meet requirement (2), copies of the outgoing traffic are sent to the backup as part of the nondeterministic event log.

The PackGate kernel module on the backup is used to meet requirement (3). PackGate maintains a *release sequence number* for each TCP stream. When the primary container sends an outgoing message, the nondeterministic event log it sends to the backup (§5.1) includes a release request that updates the stream's release sequence number.

PackGate operates frequently and must thus be efficient. Hence, it is critical that it is implemented in the kernel. Furthermore, it must maintain fairness among the TCP streams. These goals are met by maintaining a FIFO queue of release requests that is scanned by PackGate. Thus, PackGate avoids iterating through the streams looking for packets to release and releases packets based on the order of sends.

5.2.4 Transition to Live Execution

As with [LVN10, GHY14] and unlike the deterministic replay tools for debugging [SKA04, Sai05, LWV10, VLW11], *HyCoR* needs to transition from the replay mode to the live mode. The switch occurs when the backup replica finishes replaying the nondeterministic event log, specifically, when the last system call that generated an external output during the original execution is replayed. To identify this last call, after the checkpoint is restored, the RR library scans the nondeterministic event log and counts the number of system calls that generated an external output. Once replay starts, this count is decremented and the transition to live execution is triggered when the count reaches 0.

To support live execution, after replay, the kernel state must be consistent with the state of the container and with the state of the external world. For most kernel state, this is achieved by actually executing during replay system calls that change kernel state. For example, this is done for system calls that change the file descriptor table, such as open(), or change the memory allocation, such as mmap(). However, this approach does not work for system calls that interact with the external world. Specifically, in the context of *HyCoR*, these are reads and writes on sockets associated with a connection to an external client. As discussed in §5.2.1, such calls are replayed from the nondeterministic event log. However, there is still a requirement of ensuring that, before the transition to live execution, the state of the socket, e.g., sequence numbers, must be consistent with the state of the container and with the state of external clients.

To overcome the above challenge, when replaying system calls that affect socket state, *HyCoR* records the state changes on the sockets based on the nondeterministic event logs. When the replay phase completes, *HyCoR* updates all the sockets based on the recorded state. Specifically, the relevant components of socket state are: the last sent sequence number, the last acknowledged (by the client) sequence number, the last received (from the client) sequence number, the receive queue, and the write queue. The initial socket state is obtained from the checkpoint. The updates to the sent sequence number and the write queue contents are determined based on writes and sends in the nondeterministic event log. For the rest of the socket state, *HyCoR* cannot rely on the event log since some packets received and acknowledged by the kernel may not have been read by the application. Instead, *HyCoR* uses information obtained from PackRec (§5.2.3).

With respect to incoming packets, once the container transitions to live execution, *HyCoR* must provide to the container all the packets that were acknowledged by the primary but were not read by applications. During normal operation, on the backup host, PackRec keeps copies of incoming packets while PackGate extracts the acknowledgment numbers on each outgoing stream. If the primary fails, PackGate stops releasing outgoing packets and it thus has the last acknowledged sequence number of each incoming stream. Before the container is restored on the backup, PackRec copies the recorded incoming packets to a log. PackRec uses the information collected by PackGate to determine when it has all the required (acknowledged) incoming packets. Using the information from the nondeterministic event log and PackRec, before the transition to live execution, the packet repair mode (§2.7) is used to restore the socket state so that it is consist with the state of the container and the external world.

5.2.5 Transferring the Event Logs

Whenever the container on the primary sends a message to an external client, it must collect the corresponding entries from the multiple nondeterministic event logs (§5.2.1) and send them to the backup (§5.1). Hence, the collection and sending of the log is a frequent activity, which is thus performance critical. To optimize performance, *HyCoR* includes performance optimizations, such as a specialized heap allocator for the logs and maintaining a list of logs that have been modified since the last time log entries were collected. However, such optimizations proved to be insufficient. Specifically, with one of our benchmarks, *Memcached*, under saturation, the performance overhead was approximately 300%.

To address the performance challenge above, *HyCoR* offloads the transfer of the nondeterministic event log from the application threads to a dedicated *logging thread* added by the RR library to the application process. With available CPU cycles, such as additional cores, this minimizes interruptions in the operation of the application threads. Furthermore, if multiple application threads generate external messages at approximately the same time, the corresponding multiple transfers of the logs are batched together, further reducing the overhead. When an application thread sends an external message, it notifies the logging thread via a shared ring buffer. The logging thread continuously collects all the notifications in the ring buffer and then collects and sends the nondeterministic logs to the backup. To reduce CPU usage and enable more batching, the logging thread sleeps for the minimum time allowed by the kernel between scans of the buffer.

To minimize the performance overhead, *HyCoR* allows concurrent access to different logs. Thus, one application thread may log a lock operation concurrently with another application thread that is logging a system call, while the logging thread is collecting log entries from a third log for transfer to the backup. This enables the logging thread to collect entries from different logs out of execution order. Thus, the collected log transferred to the backup for a particular outgoing message may be missing log entries on which some included log entries depend. For example, for a particular application thread, a log entry for a system call may be included but the entry for a preceding lock operation may be missing. This can result in an incomplete log, leading to replay failure.

There are two key properties of *HyCoR* that help address the correctness challenge above: A) there is no need to replay the nondeterministic event log beyond the last system call that outputs to the external world, and B) when an application thread logs a system call that outputs to the external world, all nondeterministic events on which this system call may depend are already logged in nondeterministic event logs. To exploit these properties, the RR library maintains on the primary a global sequence number that is accessible to the application threads and the logging thread. We'll refer to this sequence number as the *primary batch sequence number* (PBSN). A corresponding sequence number is maintained on the backup, which we'll refer to as *backup batch sequence number* (BBSN).

When an application thread logs a system call that outputs to the external world, it attaches the PBSN to the log entry. When the logging thread receives a request to collect and send the current event log, it increments the PBSN before taking any other action. Thus, any log entry corresponding to a system call that outputs to the external world that is created after the logging thread begins collecting the log, has a higher sequence number. When the backup receives the event log, it increments the BBSN. If the primary fails, before replay is initiated on the backup, all the nondeterministic event logs collected during the current epoch are scanned and the entries for system calls that output to the external world are counted *if* their attached sequence number is not greater than the BBSN. During replay, this count is decremented for each such system call replayed. When it reaches 0, relay terminates and live execution commences.

5.2.6 Mitigating the Impact of Data Races

Fundamentally, *HyCoR* is based on being able to identify all sources of non-determinism that are potentially externally visible, record their outcomes, and replay them when needed. This implies that applications are expected to be free of data races. However, since *HyCoR* only requires replay of short intervals (up to one epoch), it is inherently more tolerant to rarely manifested data races than schemes that rely on accurate replay of the entire execution [GHY14]. As an addition to this inherent advantage of *HyCoR*, this section describes an optional mechanism in *HyCoR* that significantly increases the probability of correct recovery despite data races, as long as the manifestation rate is low.

HyCoR mitigates the impact of data races by adjusting the relative timing of the application threads during replay to approximately match the timing during the original execution. As a first step, in the record phase, the RR library records the order and the TSC (time stamp counter) value when a thread leaves the after hook of a system call. In the replay phase, the RR library enforces the recorded order on threads before they leave the after hook. As a second step, during replay, the RR library maintains the TSC value corresponding to the time when the after hook of the last-executed system call was exited. When a thread is about to leave a system call after hook, the RR library delays the thread until the difference between the current TSC and the TSC of the last system call is larger than the corresponding difference in the original execution. System calls are used as the basis for the timing adjustments since they are replayed (not executed) and are thus likely to cause the timing difference. This mechanism is evaluated in §5.4.4.

5.3 Experimental Setup

All the experiments were hosted on Fedora 29 with the 4.18.16 Linux kernel. The containers were hosted using runC [run] (version 1.0.1), a popular container runtime used in Docker.

Three hosts were used in the evaluation. The primary and backup were hosted on 36-

Table 5.1: Benchmarks used to evaluate *HyCoR*.

Benchmark	Description
Redis	a key-value store, evaluated with requests consisting of 50% reads and 50% writes driven by YCSB on 100K 100 byte records.
Memcached	a key-value store, evaluated with requests consisting of 50% reads and 50% writes driven by YCSB on 100K 100 byte records.
SSDB	a key-value store, evaluated with requests consisting of 50% reads and 50% writes driven by YCSB on 100K 100 byte records.
Tarantool	a key-value store, evaluated with requests consisting of 50% reads and 50% writes driven by YCSB on 100K 100 byte records.
Aerospike	a key-value store, evaluated with requests consisting of 50% reads and 50% writes driven by YCSB on 100K 100 byte records.
Lighttpd	a web server, evaluated with concurrent requests to a 1KB static page.
Streamcluster	a kernel that solves online clustering problems, evaluated with the native input suite.
Swaptions	a kernel that uses HJM framework to price a portfolio of swaption, evaluated with the native input suite.

core servers, using modern Xeon chips. These hosts were connected to each other through a dedicated 10Gb Ethernet link. The clients were hosted on a 10-core server, based on a similar Xeon chip. The client host was in a different building, interconnected through a Cisco switch, using 1Gb Ethernet.

Table 5.1 summarizes the benchmarks used to evaluate *HyCoR*. Mechanisms like *HyCoR* are most useful for server applications. The mechanism is stressed by applications that manage significant state, execute frequent system calls and synchronization operations, and interact with clients at a high rate through many TCP connections. Hence, five of the benchmarks used were in-memory databases handling short requests: *Redis* [red], *Memcached* [mem], *SSDB* [ssd], *Tarantool* [tar] and *Aerospike* [aer].

The evaluation also included a web server, *Lighttpd* [lig], and two batch PARSEC [Bie11] benchmarks: *swaptions* and *streamcluster*. For *Lighttpd*, benchmarking tools SIEGE [sie], ab [ab] and wget [wge] were used to evaluate, respectively, the performance overhead, response latency, and recovery rate. *Swaptions* and *streamclusters* were evaluated using the native input test suites.

Redis, *SSDB*, *Lighttpd*, *Streamcluster* and *Swaptions* are the same benchmarks as the ones presented in §4.4. *Streamcluster* and *Swaptions* use the same workload as the ones in §4.4 while others use a different workload. Specifically, for *Redis* and *SSDB*, requests to the servers are not batched so that the clients can interact with the server much more frequently to stress the system. For *Lighttpd*, a static webpage, instead of the PHP program that watermarks an image, is replied to the clients. As further discussed in §5.5, our prototype implementation of *HyCoR* currently only supports C programs and containers with a single process.

We used fault injection to evaluate *HyCoR*'s recovery mechanism. Since fail-stop failures are assumed, a simple failure detector was sufficient. Failures were detected based on heart beats exchanged every 30ms between the primary and backup hosts. The side not receiving heart beats for 90ms identified the failure of the other side and initiates recovery.

For *swaptions* and *streamcluster*, recovery was considered successful if the output was identical to the golden copy. For *Lighttpd*, we used multiple wget instances that concurrently fetched a static page. Recovery was considered successful if all the fetched pages were identical to the golden copy. For the in-memory database benchmarks, the *YCSB* clients could not be used since they do not verify the contents of the replies and thus could not truly validate correct operation. Instead, we developed customized clients, using existing client libraries [hir, libb, libc, liba], that spawns multiple threads and let each thread work on separate set of database records. Each thread records the value it stores with each key, compares that value with the value returned by the corresponding get operation and flags an error if there is a mismatch. Recovery was considered successful if no errors were reported.

A possible concern with the customized client programs is that, due to threads working on separate sets of database records, lock contention is reduced and this could skew the results. We compared the recovery rate and recovery latency results of the customized clients with the *YCSB* clients. For the *YCSB* clients, recovery was considered successful if replay succeeded and the clients finished without reporting errors. The results were similar: the recovery rate difference was less than 2% and the recovery latency difference was less than 5%. In §5.4.4, we report the more robust results obtained with the customized client programs.

For the fault injection experiments, for server programs, the clients were configured to run for at least 30 seconds and drive the server program to consume around 50% of the CPU cycles. A fail stop failure was injected at a random time within the middle 80% of the execution time, using the *sch_plug* module to block network traffic on all the interfaces of a host. To emulate a real world cloud computing environments, while also stressing the recovery mechanism, we used a *perturb* program to compete for CPU resources on the primary host. The *perturb* program busy loops for a random time between 20 to 80 ms and sleeps for a random time between 20 to 120ms. During fault injection, a *perturb* program instance was pinned to each core executing the benchmark.

5.4 Evaluation

This section presents *HyCoR*'s performance overhead and CPU usage overhead (§5.4.1), the added latency for server responses (§5.4.2), service interruption time during normal operation (§5.4.3), as well as the recovery rate and recovery latency (§5.4.4). Two configurations of *HyCoR* are evaluated: *HyCoR-SE* (short epoch) and *HyCoR-LE* (long epoch), with epoch durations of 100ms and 1s, respectively. Setting the epoch duration is a tradeoff between the lower overhead with long epochs and the lower susceptibility to data races and lower recovery time with short epochs. Hence, *HyCoR-LE* may be used if there is high confidence that the applications are free of data races. Thus, with the *HyCoR-SE* configuration, the data race mitigation mechanism described in §5.2.6 is turned on, while it is turned off for *HyCoR-LE*.

HyCoR is compared to NiLiCon (§4) with respect to the performance overhead under maximum CPU utilization and the server response latency. NiLiCon is configured to run with an epoch interval of 30ms. The short epochs of NiLiCon are required since, unlike *HyCoR*, the epoch duration with NiLiCon determines the added latency in replying to client requests (§4). In all cases, the "stock setup" is the application running in an unreplicated container.

5.4.1 Overheads: Performance, CPU Utilization

Two measures of the overhead of *HyCoR* are, for a fixed amount of work, the increase in execution time and the increase in the utilization of CPU cycles. These measures are distinct since many of the actions of *HyCoR* are in parallel with the main computation threads.

For the six server benchmarks, the measurements reported in this section were done with workloads that resulted in maximum CPU utilization for the cores running the application worker threads[1] with the stock setup. To determine the required workloads, the number of client threads was increased until an increase by 20% resulted in an increase of less than 2% in throughput. This led to CPU utilization of above 97% for all the worker threads except with *SSDB*. With *SSDB* a bottleneck thread resulted in utilization of 98%, while the rest resulted in utilization of approximately 48%. Additional measurements were done to verify the network bandwidth was not the bottleneck.

With four of the server benchmarks, the number of the worker threads cannot be configured (*Lighttpd*, *Redis*: 1, *Tarantool*: 2, *SSDB*: 12). *Memcached*, *Aerospike* were configured

[1] Some application "helper threads" are mostly blocked sleeping.

to run with four worker threads. For these applications, the number of the worker threads is set to four because, with our experimental setup, it was not possible to generate enough client traffic from YCSB to saturate more than four worker threads. For consistency, the two non-interactive benchmarks were also configured to run with four threads.

For each benchmark, the workload that saturates the cores in the stock setup was used for the stock, *HyCoR*, and NiLiCon setups. With NiLiCon, due to the long latencies it normally incurs for server responses (§5.4.2), it is impossible to saturate the server with this setup. Hence, for the NiLiCon measurements in this subsection, the buffering of the server responses was removed. This is not a valid NiLiCon configuration, but it provides a comparison of the overheads excluding buffering of external outputs.

Performance Overhead. The performance overhead is reported as the percentage increase in the execution time for a fixed amount of work compared to the stock setup. Figure 5.5 shows the performance overheads of NiLiCon, *HyCoR-SE*, and *HyCoR-LE*, with the breakdown of the sources of overhead. Each benchmark was executed 50 times. The margin of error of the 95% confidence interval was less than 2%.

The record overhead is caused by the RR library recording non-deterministic events. The pause overhead is due to the time the container is paused for checkpointing. The page fault overhead is caused by the page fault exceptions that track the memory state changes of each epoch (§4).

As shown by a comparison of the results for *HyCoR-SE* and *HyCoR-LE*, due to locality in accesses to pages, the pause and page fault overheads decrease as the epoch duration is increased. This comparison also shows that the fact that the data race mitigation mechanism is on with *HyCoR-SE* and of with *HyCoR-LE*, has no significant impact on the record overhead.

Table 5.2 can be used to explain the sources of the performance overhead shown in Figure 5.5. With *HyCoR-SE*, the average incremental checkpoint size per epoch was 0.2MB for

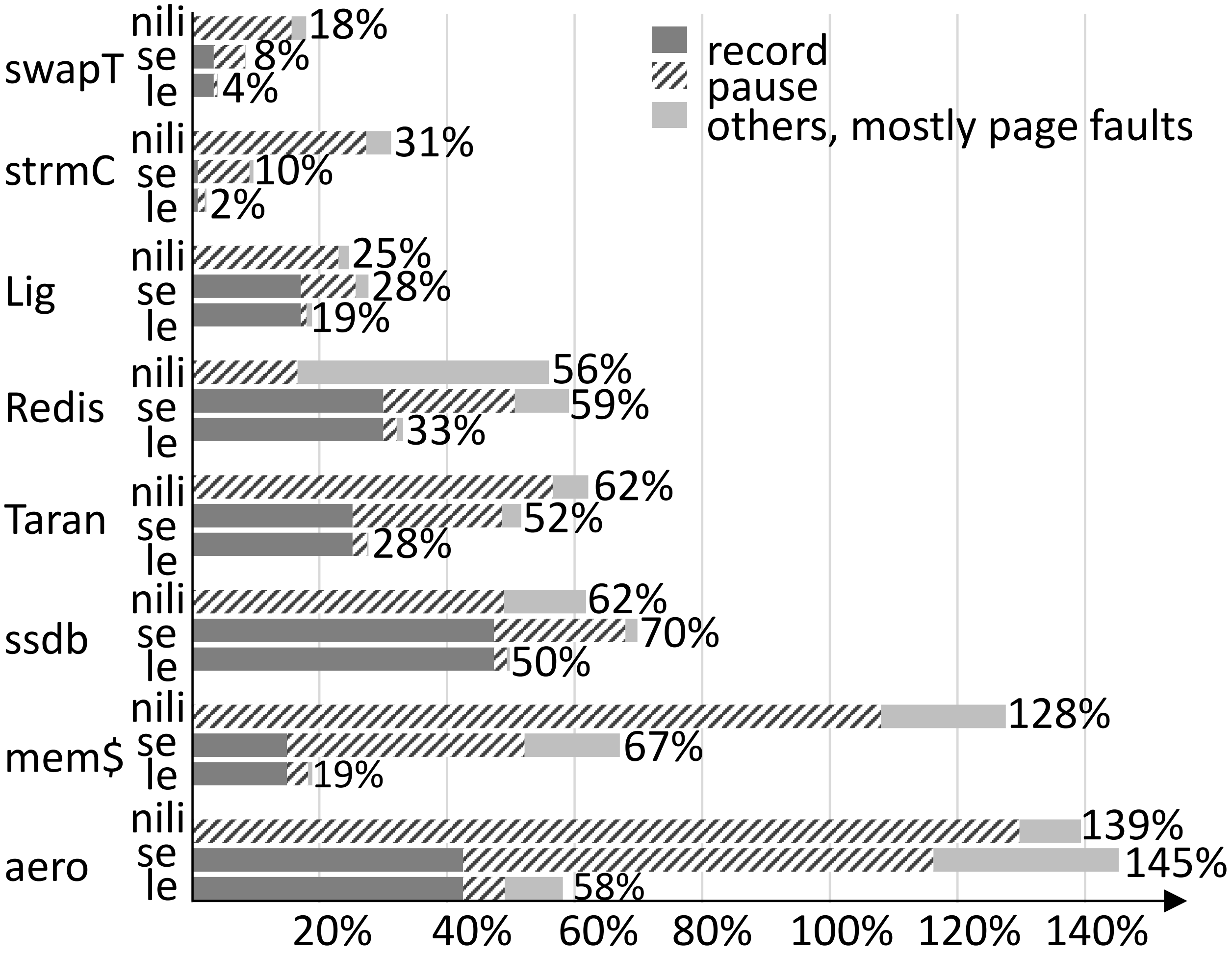


Figure 5.5: Performance overheads: NiLiCon, *HyCoR-SE*, *HyCoR-LE*.

Swaptions, 15.6MB for *Redis*, and 41.2MB for *Aerospike*, partially explaining the differences in pause overhead, which is also affected by the time to obtain required kernel state. With *HyCoR-SE*, the average number of logged lock operations plus system calls per epoch was 9 with *streamcluster*, 907 with *Tarantool*, and 2137 with *Aerospike*, partially explaining the differences in record overhead. However, the overhead of logging system calls is much higher than for lock operations. *Memcached* is comparable to *Aerospike* in terms of the rate of logged system calls plus lock operations, but has 341 compared to 881 logged system calls per epoch and thus lower record overhead.

CPU utilization overhead. The CPU utilization is the product of the average numbers

Table 5.2: Average checkpoint size in MB (CP), average number of page faults caused by tracking memory state changes per epoch (PF) and non-deterministic system call (SYS) and lock acquisition (LK) rate per milliseconds for *HyCoR-SE*.

	ST	SC	Lig	Redis	Taran	SSDB	Mem$	Aero
CP	0.2	1.3	0.3	15.6	15.4	3.3	24.3	41.2
PF	61	326	88	3986	3936	823	6195	10508
LK	942	9	101	8	618	849	1711	1256
SYS	∼0	∼0	167	438	289	1243	341	881

Table 5.3: CPU utilization overhead for *HyCoR-SE*. LogTH: logging thread. KerNet: kernel's handling of network packets. P: primary host and B: backup host.

		ST	SC	Lig	Redis	Taran	SSDB	Mem$	Aero
P	LogTH	∼0	∼0	17%	11%	12%	5%	12%	20%
	others	6%	3%	27%	63%	45%	55%	36%	87%
B	KerNet	∼0	∼0	43%	70%	43%	18%	31%	45%
	PKRec	∼0	∼0	17%	15%	10%	4%	9%	13%
	others	1%	2%	41%	54%	35%	15%	13%	20%
	total	7%	5%	145%	213%	145%	97%	101%	185%

of CPUs (cores) used and the total execution time. The CPU utilization overhead is the percentage increase in utilization with *HyCoR* compared to with the stock setup. The measurement is done by pinning each *HyCoR* component to a dedicated set of cores. All user threads are configured to run at high priority using the SCHED_FIFO real-type scheduling policy. Instances of a simple program that continuously increments a counter run at low priority on the different cores. The CPU utilization is determined by comparing the values of counts from those instances to the values obtained over the same period on an idle core. The experiment is repeated 50 times, resulting in a 95% confidence interval margin of error of less than 1%.

Table 5.3 shows a breakdown of CPU utilization overhead with *HyCoR-SE*. The "others" row for the primary is the overhead for recording the nondeterministic events, handling the page fault exceptions for tracking memory changes, and collecting and sending the incremental checkpoints. The "others" row for the backup is the overhead for receiving and storing the nondeterministic event logs and the checkpoints. The KerNet row for the backup is the overhead for packet handling in the kernel, that includes the routing of requests and responses to/from the primary and the PackGate module.

In terms of CPU utilization overhead, the worst case is with *Redis*. A significant factor is the overhead for packet handling in the backup kernel (KerNet). We have found that this overhead is mostly due to routing, not PackGate. *Redis* involves only one worker thread and it receives and sends a large number of small packets, leading to this overhead. Techniques for optimizing software routing [KMC00] can be used to reduce this overhead.

With *HyCoR-LE*, the CPU utilization overhead is 2% to 169%, with *Redis* still being the worst case. The CPU utilization on the primary is 1% to 69% – significantly less than that with *HyCoR-SE* due to the reduction in CPU time to handle checkpointing and page faults. The CPU utilization overhead on the backup is only slightly lower than with *HyCoR-SE*, due to the reduction in CPU time to receive checkpoints.

5.4.2 Response Latency

A key advantage of *HyCoR* compared to schemes based on checkpointing alone, such as Remus [CLM08] and *NiLiCon* is significantly lower response latency. Table 5.4 shows the response latencies with the stock setup, *HyCoR-SE* and NiLiCon. The number of client threads for stock and *HyCoR-SE* is separately adjusted so that the CPU load on the cores running application worker threads is approximately 50%. For NiLiCon, due to its long response latencies, it is not possible to reach 50% CPU usage. Instead, NiLiCon is evaluated with the same number of client threads as *HyCoR-SE*, resulting in CPU utilization of less than 5%, thus favoring NiLiCon. To evaluate the impact of response size, *Lighttpd* is eval-

Table 5.4: Response Latency in μs. S: Stock, H: *HyCoR-SE*, N: NiLiCon

		Lig1K	Lig100K	Redis	Taran	SSDB	Mem$	Aero
S	avg	549	2059	406	393	388	643	373
	99%	<1ms	<3ms	734	617	622	2982	711
H	avg	740	2215	637	651	709	1092	945
	99%	<1ms	<9ms	1105	1191	1087	5901	1724
N	avg	38ms	38ms	42ms	42ms	45ms	45ms	51ms
	99%	<39ms	<39ms	44ms	42ms	47ms	53ms	63ms

uated serving both 1KB as well as 100KB files. Each benchmark is executed 50 times. We report the average of the mean and the 99th percentile latencies of the different runs. For the average response latencies, the 95% confidence interval has a margin of error of less than 5%. For the 99th percentile latencies, it is less than 15%.

With *HyCoR*, there are three potential sources for the increase in response latency: forwarding packets through the backup, the need to delay packet release until the corresponding event log is received by the backup, and increased request processing time on the primary. With *HyCoR-SE*, the increase in average latency is only 156μs to 581μs. The worst case is with *Aerospike*, which has the highest processing overhead (Fig. 5.5) and a high rate of nondeterministic events and thus long logs that have to be transferred to the backup. The increase in 99th percentile latency is 371μs to 6ms. The worst case is with *Lighttpd* serving a 100KB file. This is because the request service time is much longer than with the other benchmarks and thus a checkpoint is more likely to happen in the middle of this time. The pause time for checkpoint of this benchmark is approximately 6ms. It should be noted that, in terms of increase in response latency, *NiLiCon* is not competitive, as also indicated by the results in Chapter 4.

With *HyCoR-LE*, the increase in the average response latency is from 40μs to only 343μs, due to the lower processing overhead. The increase in the 99th percentile latency is under

Table 5.5: The pause time of *HyCoR-SE* in ms

	ST	SC	Lhttpd	Redis	Taran	SSDB	Mem$	Aero
avg	5.9	7.6	7.2	14.9	18.4	13.9	28.7	42.9
median	5.9	7.7	7.3	16.1	19.5	14.1	31.1	45.1
90%	5.9	8.0	7.4	16.7	20.2	14.8	33.7	45.8

534μs since checkpoint are much less frequent and thus less likely to interrupt the processing of a request.

5.4.3 Service Interruption Time During Normal Operation

The container needs to frequently pause to take a checkpoint. If the container is paused, it cannot process any existing or new requests. Hence, the pause time of the container shows the service interruption time during normal operation. Table 5.5 shows the pause time of *HyCoR-SE*. In the worst case, the pause time is around 45.8ms with *Aerospike*. The results for *HyCoR-LE* are similar with only a slight increase (<2ms) in the mean, median and 90 percentile of the pause time.

5.4.4 Recovery Rate and Latency

This subsection presents an evaluation of the recovery mechanism and the data race mitigation mechanism. The service interruption time is obtained by measuring, at the client, the increase in response latency when a fault occurs. The service interruption time is the sum of the recovery latency plus the detection time. With *HyCoR*, the average detection time is 90ms (§5.3). Hence, since our focus is not on detection mechanisms, the average recovery latency reported is the average service interruption time minus 90ms.

Backup Failure. 50 fault injection runs are performed for each benchmark. Recovery is always successful. The service interruption duration is dominated by by the Linux TCP

Table 5.6: Recovery rate and replay time (in ms). *HyCoR* with different levels of mitigation of data race impact.

		Recovery Rate		Replay Time	
		Mem$	Aero	Mem$	Aero
100ms	stock	94.1%	83.4%	23	33
	+ Total order of syscalls	93.9%	92.8%	128	289
	+ Timing adjustment	99.5%	99.8%	234	377
1s	stock	50.2%	35.3%	245	370
	+ Total order of syscalls	50.6%	78.1%	1129	1342
	+ Timing adjustment	98.7%	99.2%	1218	1474

retransmission timeout, which is 200ms. The other recovery events, such as detector timeout and broadcasting the ARP requests to update the service IP address, occur concurrently with this 200ms. Thus, the measured service interruption duration is between 203ms and 208ms. The 95% confidence interval margin of error is less than 0.1%.

Primary Failure Recovery Rate. Three of our benchmarks contain data races that may cause recovery failure: *Memcached*, *Aerospike* and *Tarantool*. Running *Tarantool* with *HyCoR-SE*, through 50 runs of fault injection in the primary, we find that, due to data races, in all cases replay fails and thus recovery fails. Due to the high rate of data race manifestation, this is the case even with the mechanism described in §5.2.6. Thus, we use a modified version of *Tarantool* in which the data races are eliminated by manually adding locks.

We divide the benchmarks into two sets. The first set consists of the five data-race-free benchmarks and a modified version of *Tarantool*. For these, 50 fault injections are performed for each benchmark. Recovery is always successful.

The second set of benchmarks consists of *Memcached* and *Aerospike*, used to evaluate the data race mitigation mechanisms (§5.2.6). For these, to ensure statistically significant results, 1000 fault injection runs are performed with each benchmark with each setup. The results are presented in Table 5.6. For both the recovery rate and replay time, the 95% confidence interval is less than 1%. Without the §5.2.6 mechanism, the recovery rate for *HyCoR-LE* is much lower than with *HyCoR-SE*, demonstrating the benefit of short epochs and thus shorter replay times. Enforcing a total order of the recorded system calls in the after hook is not effective for *Memcached* but increases the recovery rate of *Aerospike* for both *HyCoR* setups. However, with the timing adjustments, both benchmarks achieve high recovery rates, even with *HyCoR-LE*. The total order of the system calls is the main factor that increase the replay time. Thus, there is no reason to not also enable the timing adjustments.

To explain the results above, we measured the rate of racy memory accesses in *Tarantool*, *Memcached* and *Aerospike*. To identify "racy memory accesses", we first fixed all the identified data races by protecting certain memory access with locks. We then removed the added locks and added instrumentation to count the corresponding memory accesses. For *Tarantool*, the rates of racy memory writes and reads are, respectively, 328,000 and 274,000 per second. For *Memcached* the respective rates are 1 and 131,000 per second and for *Aerospike* they are 250 and 372,000 per second. These results demonstrate that when the rate of accesses potentially affected by data races is high our mitigation scheme is not effective. Fortunately, in such cases, data races are unlikely to remain undetected.

As an additional validation of *HyCoR*, the three benchmarks mentioned above were modified to eliminate the data races. With the *HyCoR-LE* setup, 200 fault injection runs are executed with *Memcached* and *Aerospike*. 50 fault injection runs are executed with the remaining six benchmarks. Recovery is successful in all cases.

Primary Failure Recovery Latency. Figure 5.6 shows a breakdown of the factors that make up the recovery latency with *HyCoR-SE* and *HyCoR-LE*. The batch benchmarks, *swaptions* and *streamcluster*, are not included since their execution times are above 60s, so

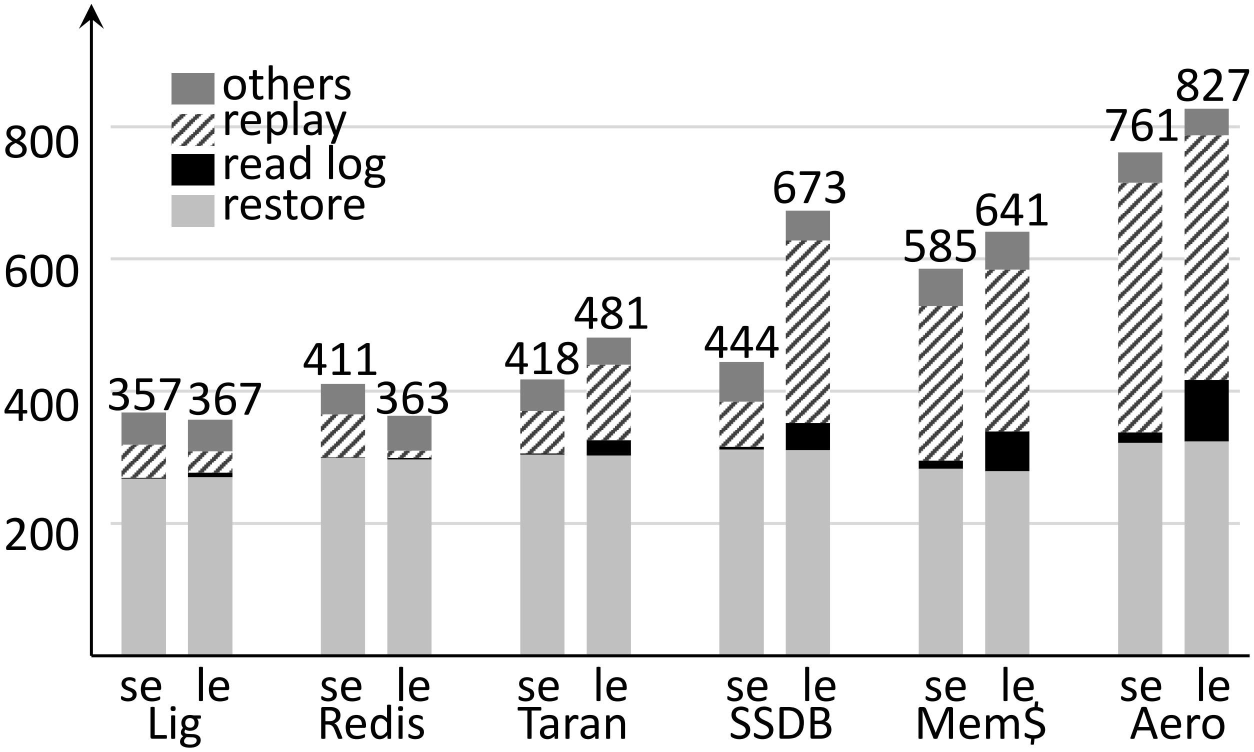


Figure 5.6: Recovery Latency (ms) breakdown with *HyCoR-SE* and *HyCoR-LE*.

their recovery latency is insignificant. With *HyCoR-SE*, the data race mitigation scheme is enabled, while with *HyCoR-LE* it is disabled. The 95% confidence interval margin of error is less than 5%. *Restore* is the time to restore the checkpoint, mostly for restoring the in-kernel states of the container (e.g., mount points and namespaces). *Read log* is the time to process the stored logs in preparation for replay. *Others* include the time to send ARP requests and connect the backup container network interface to the bridge.

The recovery latency differences among the benchmarks are due mainly to the replay time. It might be expected that the average replay time would be approximately half an epoch duration. However, replay time is increased due to different thread scheduling by the kernel that causes some threads to wait to match the order of the original execution. This increase is more likely when the data race impact mitigation mechanism is enabled since it enforces more strict adherence to the original execution. A second factor that impact the replay time is a decrease due to system calls that are replayed from the log and not executed.

5.5 Limitations

We have identified one inherent limitation of *HyCoR* and four limitations of the current research prototype implementation. An inherent limitation is that the mechanism used for mitigating the impact of data races (§5.2.6) is incapable of handling a high rate of racy accesses (§5.4.4). However, as discussed in §5.4.4, such data races are easily detectable and are thus easy to eliminate, even in legacy applications.

HyCoR does not currently support multiple processes. To overcome this limitation, the RR library would need significant enhancements, such as support for inter-process communications via shared memory. Techniques presented in [BHC10] may be applicable. *HyCoR* also does not handle asynchronous signals. This may be resolved by techniques used in [LVN10], that delay signal delivery until a system call or certain page faults.

HyCoR does not handle C atomic types, functions, intrinsics and inline assembly code that performs atomic operations transparently. In this work, such cases were handled by protecting such operations with locks. Specifically, this was done for *Aerospike* and *glibc*. Compiler support [MGT17] is needed to overcome this limitation.

Recovery currently fails if a socket is created via accept() or connect() during replay. Resolving this limitation would require recording and restoring during replay various socket state components, such as the timestamp and window scale. This can be done by enhancing the RR library with code used in *NiLiCon* to checkpoint and restore socket state.

5.6 Summary

HyCoR is a unique point in the design space of application-transparent fault tolerance schemes for multiprocessor workloads. By combining checkpointing with externally deterministic replay, it facilitates trading off performance and resource overheads with vulnerability to data races and recovery latency. Critically, the response latency is not determined by the frequency of checkpointing, and sub-millisecond added delay is achieved with all our server applications. As we have found (§5.4.4), legacy applications may still have data races. *HyCoR* targets data races that are most likely to remain undetected and uncorrected, namely, rarely-manifested data races. Unlike mechanism based strictly on active replication and deterministic replay [GHY14], *HyCoR* is not affected by data races that manifest during normal operation, long before failure. For handling data races that manifest right before failure, *HyCoR* introduces a simple best effort mechanism that significantly reduces of the probability of the data races causing recovery failure. *HyCoR* is a full fault tolerance mechanism. It can recover from primary or backup host failure and includes transparent failover of TCP connections.

This chapter describes key implementation challenges encountered in the development of *HyCoR* and outlines their resolution. The extensive evaluation of *HyCoR*, based on eight benchmarks, included performance and resource overheads, impact on response latency, as well as recovery rate and latency. The recovery rate evaluation, based on fault injection, subjected *HyCoR* to particularly harsh conditions by intentionally perturbing the scheduling on the primary, thus challenging the deterministic replay mechanism (§5.3). With high checkpointing frequency (*HyCoR-SE*), *HyCoR*'s throughput overhead is less than 68% for seven of our benchmarks and 145% for the eighth. If the applications are known to be data race free, with a lower checkpointing frequency (*HyCoR-LE*), the overhead is less than 59% for all benchmarks, significantly outperforming *NiLiCon*. With data race free applications, *HyCoR* recovered from all fail-stop failures. With two applications with infrequently-manifested data races, the recovery rate was over 99.4% with *HyCoR-SE*.

CHAPTER 6

PUSh: Data Race Detection Based on Hardware-Supported Prevention of Unintended Sharing

Prior chapters present several fault-tolerance mechanisms. This chapter presents a debugging tool called *PUSh* (Prevention of Unintended Sharing). *PUSh* helps to detect an important type of programming errors: data races in parallel applications. Thus, *PUSh* enhances the dependability of the system by reducing design faults in the applications (Chapter 1).

Data races are a major cause of concurrency bugs. Data races are caused by unintended sharing. Hence, motivated by limitations of the various alternatives, one approach to detecting data races is to require the programmer to specify the intended sharing, using special annotations, and detect accesses that violate these intentions [FF00, AGE08, AGN09, MHC10, BLR02, Gro03, HBS14]. Especially for languages like C and C++, this approach can simplify and reduce the overhead of data race detection since tools based on this approach do not need to autonomously differentiate between accesses that correspond to intended sharing and those that violate these intentions.

detection tools that require program annotations, *PUSh* are: *SharC* [AGE08], *Shoal* [AGN09], and a tool that we refer to as *DCOP* (Dynamically Checking Ownership Policies) [MHC10]. All three tools use a combination of static analysis and software instrumentation of memory accesses. *PUSh* requires programs to be similarly annotated. However, with *PUSh*, the detection of unintended sharing is implemented using off-the-shelf hardware. Thus, *PUSh* does not rely on static analysis and does not require high-overhead software instrumentation of *any* normal memory accesses.

With *PUSh*, when an object is allocated (statically or dynamically), annotation indicates the *sharing policy* of the object, such as *private* – read/write accessible by one thread, or *read-shared* – potentially readable by all the threads (§6.1.1). Subsequently, *change policy* annotations can be used to change the sharing policy of objects. Unannotated objects are accessible by only a single thread. While annotating programs is an extra burden on the programmer, prior work has provided positive indications that many programmers are willing to annotate their programs for this purpose [SY14].

Logically, *PUSh* maps sharing policies to per-thread read/write access permissions (§6.1.3) and enforces these permissions without instrumenting any normal (read/write) memory accesses. Ideally, this enforcement would be performed by specialized hardware that efficiently supports fine-grained memory protection for variable size objects [WCA02, ZKD08]. A key contribution of *PUSh* is an efficient implementation of the mechanism with off-the-shelf hardware, utilizing page-level protection. ISOLATOR [RRR09] also uses page-level protection for data race detection. However, ISOLATOR only addresses one particular type of data race, where one thread acquires a lock for an object and another thread accesses the object without acquiring a lock. ISOLATOR cannot detect other types of data races. Furthermore, ISOLATOR incurs high overhead if an object protected by a lock is repeatedly accessed by different threads.

If two threads perform conflicting change policy operations on the same object in unordered (concurrent) *vector time frames* [PS07] (*epochs* [FF09]), unordered conflicting accesses to the object (data races) may not be detected by simply checking sharing policy violations (§6.1.2). To deal with this problem, *PUSh* identifies conflicting unordered change policy operations utilizing the FastTrack algorithm [FF09] to perform happens-before tracking of those operations. The number of change policy operations in a program is much lower than the number of normal object accesses. Hence, the performance overhead of the tracking performed by *PUSh* is dramatically lower than mechanisms, such as FastTrack, that track every object access. This idea has the potential to enhance *any* dynamic race detector that relies on explicit sharing policy changes.

PUSh could be implemented in a straightforward manner by placing each object in a separate page and using a separate page table for each thread. Such an implementation would require significant changes to the OS kernel and involve memory overhead for multiple page tables. More importantly, such an implementation would involve high performance overhead since every sharing policy change would require a system call which, at times, would require synchronously updating all the page tables. Critically, this would have to be done every time a lock is acquired or released. *PUSh* avoids most of these disadvantages with a novel use of memory protection keys (MPKs), recently added to the x86 ISA [Int18]. Multiple additional optimizations further reduce the performance and memory overheads of the straightforward implementation.

One way to evaluate *PUSh* is by comparing it to ThreadSanitizer (TSan) [SI09, thr]. TSan is a widely-used and well-maintained data race detector, which is included as part of gcc. A clear advantage that TSan has over *PUSh* is that it does not require annotation. Our comparison was based on eleven C benchmarks. Running with eight threads, in terms of performance overhead (additional execution time relative to the stock applications), for TSan the range was 384% to 12,820%, while for *PUSh* it was 0% to 67%. For four applications, the TSan slowdown exceeded 1600%, while the maximum overhead of *PUSh* for those applications was negligible. In terms of race detection, neither TSan nor *PUSh* had any false positives (identified false races). Excluding races due to standard library calls (see §6.4.1), *PUSh* detected all the data races detected by TSan. These results indicate that, in many deployment scenarios, *PUSh* can be used in production runs, while TSan is restricted to use in offline debugging.

We have evaluated *PUSh* using the eleven C benchmarks mentioned above running with up to 32 threads. *PUSh*'s memory overhead is negligible for all benchmarks: <5.8%. The worst performance overhead was 67% for one benchmark, due to a high rate of policy changes. For nine of the benchmarks the performance overhead was under 35% for all thread counts. *PUSh* detected a total of ten data races. As mentioned above, these were also the races detected by TSan.

We make the following contributions: 1) a novel technique for using current MMUs with MPKs for efficient data race detection by the prevention of unintended sharing coupled with happens-before tracking of *only* sharing policy changes; 2) several optimizations for increasing detection accuracy as well as reducing memory and performance overheads, including enhanced annotations, kernel changes, software caching, and the use of a universal family of hash functions; 3) analysis of the sources of overhead of *PUSh* and the effectiveness of the different optimizations; 4) comparison of *PUSh* to the most closely-related annotation-based race detectors schemes [AGE08, AGN09, MHC10] as well as TSan.

Section 6.1 is an overview of the sharing policy annotation framework of *PUSh* and the basic approach of preventing unintended sharing using access permissions of protection domains. §6.2 is a detailed description of the implementation, including the various optimizations and their potential impact. The experimental setup and evaluation results are presented in §6.3 and §6.4, respectively. §6.5 is a summary of the limitations and disadvantages of *PUSh* and of its current implementation.

6.1 Overview of *PUSh*

For any data race detection mechanism based on detection of violations of explicitly-specified intended sharing, there are two key issues: how the intended sharing is specified and how are violations of these intentions detected. The core sharing policies of *PUSh* and the permitted sharing policy changes are described in §6.1.1. In order to avoid hiding some data races, some combinations of sharing policy change operations on the same object performed by different threads must not be *concurrent*. The use of happens-before tracking [PS07, FF09] of these operations to detect these situations is described in §6.1.2.

PUSh relies on hardware-enforced protection domains to detect unintended sharing. To facilitate efficient implementation, *PUSh* supports two additions to the core sharing policies: *sticky-read* (§6.1.2) and *locked* (§6.1.3). As explained in §6.1.3, in some specific scenarios, the use of the *locked* policy can result in false negatives (undetected races). §6.1.3 also provides a high-level overview of *PUSh*'s mapping of sharing policies to read/write access permissions in "ideal" protection domains. The implementation using off-the-shelf hardware is described in §6.2.

For a single execution, *PUSh*, as all dynamic data race detectors, can only detect races in code that is actually executed. However, *PUSh* also has the "pseudo-completeness" property, as defined in [ELC12]: with a sufficient number of *different* executions, *PUSh* will *eventually* detect all the data races. As other tools, *PUSh* requires synchronization operations to be explicitly identified. Only Pthreads operations are currently supported.

6.1.1 Core Sharing Policies and Policy Changes

The core sharing policies of *PUSh* are essentially identical to those of DCOP [MHC10]. Since every global object is potentially shared, *PUSh* must associate sharing policies with all such objects, which we will henceforth refer to as *tracked objects*. There are five core sharing policies: *private*, *read-shared*, *racy*, *inaccessible*, and *untouched*. We use the term "*private* object" to refer to an object whose current sharing policy is *private*. Similarly for "*read-shared* object," etc. Two additional sharing policies: *sticky-read* and *locked*, are discussed in §6.1.2 and §6.1.3, respectively.

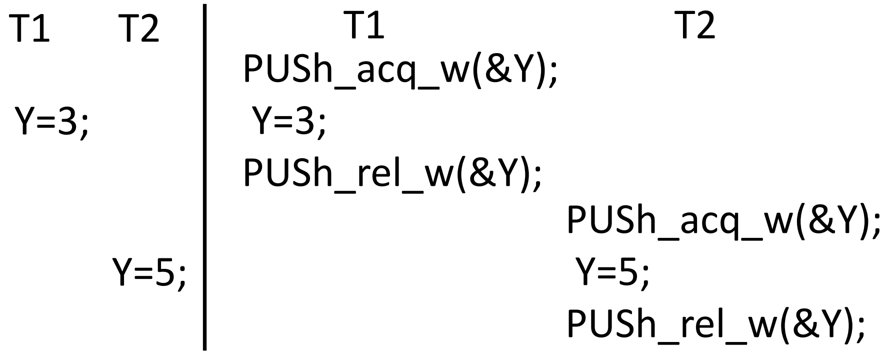
A *private* object is read/write accessible by one thread. A *read-shared* object is potentially readable by all the threads. *Racy* objects are read/write accessible by all the threads. This sharing policy is used for objects that are intentionally racy, such as synchronization objects (e.g. lock variables). An *inaccessible* object is not accessible by any thread. An *untouched* object becomes *private* to the first thread that accesses it. The initial sharing policy for an object is specified when a static object is declared or when an object is dynamically allocated. Static objects that are not annotated are *untouched.* Objects allocated dynamically with the standard API (e.g., *malloc*) are *private* to the invoking thread.

The *racy* policy allows incorrect annotation to hide races. However, with the C11 Standard [ISO11], this policy can be removed (used only *internally* for synchronization objects). Without *racy*, incorrect annotation cannot result in false negatives. Incorrect annotation can cause false positives. However, those are eliminated during the annotation process (§6.4.2). C11 atomics are currently not supported.

During execution, the intended accessibility of an object may change. For example, a *private* object may later become *read-shared* by multiple threads. Hence, *PUSh* supports runtime *change policy* operations. *Acquire/release write* operations by a thread make an object private to the thread or release the association between the object and the thread. Similarly, *acquire/release read* operations allow or disallow the thread from reading and object. Since one thread cannot force another to relinquish its access, a thread can *acquire write* only if the object is *untouched*, *inaccessible*, *locked*, or is the only thread for which the object is *read-shared.* Similarly, a thread can *acquire read* only if the object is *untouched*, *inaccessible*, *locked*, *private* to this thread, or *read-shared* by other threads. An object that is released by all the threads becomes *inaccessible.* An object that is *untouched* or *inaccessible* can be changed to *racy.*

6.1.2 Ensuring Ordering of Policy Changes

Simply enforcing the sharing policies presented in §6.1.1 does not guarantee that all data races will be detected. For example, as shown in Fig. 6.1, a thread may write to a *private* object, release the write permission, and later, without an intervening synchronization operation, another thread can *acquire write* and write to the object. Allowing this kind of scenario is a limitation of prior data race detectors based on the detection of sharing policy



```
T1    T2   |     T1                T2
           | PUSh_acq_w(&Y);
 Y=3;      |  Y=3;
           | PUSh_rel_w(&Y);
           |                   PUSh_acq_w(&Y);
      Y=5; |                    Y=5;
           |                   PUSh_rel_w(&Y);
```

Figure 6.1: *Left:* original code with a data race. *Right:* sharing policy correctly annotated, but possibility of detection failure under some execution interleavings.

violations [AGE08, AGN09, MHC10].

PUSh detects conflicting concurrent policy changes on the same object using the FastTrack algorithm [FF09] for happens-before tracking. Specifically, since a thread must have exclusive access in order to write to an object, in *PUSh*'s deployment of FastTrack, *acquire write* and *release write* are processed as *writes*, while *acquire read* and *release read* are processed as *reads* from the object. Thus, the happens-before tracking is performed *only* for sharing policy changes. By combining this idea with hardware enforcement of sharing policies, it is possible to significantly reduce the performance overhead compared to conventional mechanisms based on happens-before tracking, without incurring false negatives.

With the happens-before tracking of policy changes, an ideal implementation of *PUSh* would be sound and precise (no false negatives or positives), without requiring instrumentation of normal read/write accesses.

A drawback of all happens-before tracking is the memory overhead for each tracked object. This overhead is particularly large for *read-shared* objects [FF09]. *PUSh* mitigates this overhead by introducing the *sticky-read* sharing policy. All the threads can read *sticky-read* objects. Once an object is changed to *sticky-read*, its sharing policy can never be changed. Hence, following the change, there is no need to perform any tracking of the object and this does *not* compromise soundness. To avoid false negatives, a policy change to *sticky-read* is allowed only when happens-before tracking guarantees that there is only one thread in the process. *PUSh* enforces this constraint and flags violations.

6.1.3 Enforced Protection Domains

PUSh prevents unintended sharing by associating each global object with a Logical Protection Domain (LPD). "LPD" denotes a set of addresses for which each thread has the same read/write access permissions, but the access permissions for different threads may be different. Sharing policies are enforced by assigning the proper access permissions (§6.1.1) to each of the LPDs. While LPDs could be implemented using software instrumentation of normal memory accesses, this would result in excessive performance overhead. Thus, ideally, the LPDs should be implemented using hardware-enforced fine granularity, variable size protection domains [WCA02, ZKD08].

In general, the overhead associated with protection domains increases as the number of domains increases. This motivates *PUSh* to map the LPDs to a smaller number of Enforced Protection Domains (EPDs). This requires including multiple objects in the same EPD, regardless of their physical location. For each thread, *PUSh* maps all the LPDs that are *private* to that thread to the same EPD. For each of the sharing policies *untouched*, *inaccessible*, *racy*, and *sticky-read*, all the LPDs containing objects with that policy are mapped to a single EPD. The above consolidations of LPDs into fewer EPDs do not impact the soundness of *PUSh*.

PUSh places all the *read-shared* objects in a single EPD. This consolidation does impact soundness since it is not possible to strictly enforce the requirement for a thread to *acquire read* before it can read the object. Once one thread performs an *acquire read*, all the threads are able to read. Thus, *PUSh* would not detect a data race where a thread writes to a *private* object, later it, or another thread, changes the object to *read-shared*, and finally a third thread reads the object but there is no intervening synchronization between the original write and this read. Fortunately, in many cases, all the reading threads are executing the same code. In such cases, either none of the threads or all of the threads perform the *acquire read*. With either possibility, the data race will be detected.

With DCOP [MHC10], and thus also with the core annotations in §6.1.1, every critical section requires changing the sharing policy of all the accessed objects to *private* and then back to *inaccessible*. If policy changes to/from *private* are slow (e.g., require a system call §6.2.2), this may incur prohibitive overhead. Hence, *PUSh* adds the *locked* sharing policy [AGE08], that associates an object with a specific lock. Multiple objects can be associated with the same lock. A *locked* object is read/write accessible by a thread that holds the lock associated with that object. All the LPDs containing *locked* objects protected by the same lock are mapped to the same EPD. Policy changes to/from *locked* are processed as *writes* (§6.1.2).

Once an object's sharing policy is changed to *locked*, any thread holding the associated lock can access the object even if that access is *unordered* with respect to a prior access to the object before its sharing policy was changed to *locked*. Thus, when a thread performs a change policy operation to *locked*, *PUSh* actually maps the object to the *inaccessible* EPD. Hence, the first access to the object is trapped. *PUSh* then verifies that the access is performed while the thread holds the lock and the access is ordered (happens-after) with respect to the sharing policy change. If this verification succeeds, the object is moved to the EPD of the associated lock. Subsequent accesses to the object under the lock are data race free, as such accesses are ordered by the lock.

Since a *locked* object may be accessed by any thread holding the associated lock, a change policy from *locked* should be ordered with respect to accesses under the lock. Verifying this would involve happens-before tracking of every access to an object under a lock. For performance reasons, *PUSh* avoids tracking normal read/write accesses to objects. Hence, in some executions, the current implementation of *PUSh* can fail to detect accesses under lock which are unordered with respect to a change policy from *locked*, leading to the possibility of failing to detect a data race.

6.2 Implementation

PUSh is usable with current widely-available off-the-shelf hardware. In fact, the development of *PUSh* was motivated by the introduction of memory protection keys (MPKs) to the Intel x86 ISA [Int18]. This section describes the implementation of *PUSh*, including motivating and evaluating key optimizations.

Subsection 6.2.1 describes a straightforward implementation based on off-the-shelf hardware with only one modification to the OS: a separate page table for each thread. This subsection also lists the deficiencies of the straightforward implementation. *PUSh*'s optimizations that mitigate the impact of these deficiencies are discussed in the remaining subsections.

6.2.1 Basic Implementation

PUSh implements EPDs using page-level protections. Specifically, each EPD is implemented as a set of pages with the same access permissions. These access permissions are set as described in §6.1.3. A straightforward implementation of this idea requires a separate page table for each thread. Since with standard OS kernels all the threads of a process share the same page table, this implementation requires a change in the kernel's memory subsystem.

With the above implementation, access permissions are enforced at the granularity of pages. Hence, tracked objects that have different sharing policies must be in separate virtual pages [DA06]. If several tracked objects are in the same virtual page, a *change policy* for one of them would require a memory copy. Thus, tracked objects that *may* have different sharing policies in the future should also be in separate virtual pages.

PUSh places each tracked object in separate pages upon creation. For statically-allocated objects, this is done by a script that modifies the alignment of global objects in the assembly file. For dynamically-allocated objects, this is done by modified functionality of the calls to functions such as *malloc*. Due to the way the *glibc* heap allocator handles its metadata, it is not suitable for use with *PUSh*. Instead, *PUSh* links to the application the tcmalloc [Ghe] heap allocator, which uses a dedicated memory region for its metadata.

Ideally, local objects on the stack should also be tracked objects and placed in separate pages upon creation. However, this would require compiler supports and the current *PUSh* implementation does not implement it. Instead, *PUSh* treats all the local objects allocated on the stack as a single private object. A challenge to achieve this is under Linux, objects that are shared with other threads, such as environmental variables, are also placed on the stack and such objects must not be private. *PUSh* overcomes this by using a script to modify the assembly file to direct the entrance point of the program to a specialized main function. The specialized main function first adds a page between the local variable and the shared variable on the stack, by declaring a local array of page size, which isolates the local variables from the shared variable and also places these local variables on a separate page. It then makes the all the local variables a private tracked objects and invoke the actual main function. An option is provided for the programmers to disable the local variable protection in case the program shares the local variables among threads.

Lock acquire and release operations change the access permissions of pages containing the protected object. This is necessary so that locked objects are only accessible to threads that acquire the appropriate locks. *PUSh* utilizes the *--wrap* option in the Linux linker to intercept Pthreads synchronization operations and add the required functionality. The alignment requirement for dynamic memory allocation is enforced in the same way.

PUSh incurs storage overhead for metadata for each tracked object, each synchronization object, and each executing thread. For each tracked object, this metadata consists of at least 48 bytes, out of which 24 bytes are for the happens-before tracking. The storage overhead for synchronization objects and executing threads is mostly for vector clocks. Since the number of synchronization objects and executing threads is typically small, the related storage overhead is insignificant.

The rest of this section addresses the following deficiencies with the straightforward implementation:

(1) Each thread needs a separate page table, which requires significant changes to the OS kernel's memory subsystem. Keeping the page tables properly synchronized incurs significant overhead.

(2) Several operations performed by *PUSh* require system calls to change page table permissions. The resulting performance overhead is exacerbated by item (1) above. The relevant operations are: (2.1) acquiring and releasing locks, (2.2) allocating and freeing dynamically-allocated objects, and (2.3) sharing policy changes.

(3) Allocating each tracked object on separate pages incurs substantial memory overhead due to *internal fragmentation*: the remaining space in the page is wasted.

6.2.2 Permission Management Using MPKs

Starting with the straightforward implementation of *PUSh* (§6.2.1), deficiencies (1) and (2.1) can be largely alleviated using memory protection keys (MPKs), recently added to the x86 ISA [Int18]. This optimization is the focus of this subsection.

With MPKs, each virtual page is tagged, in its page table entry, with a single protection domain number. There are 16 domains. A per-thread user-accessible register, PKRU, controls the access permissions to each protection domain for the current thread. The possible access permissions are: no access, read-only access, and read-write access. An access to the memory only succeeds when both the page table entry permission bit and the control bits in PKRU for the corresponding protection domain permit the access.

MPKs enable a user-level program to modify the memory access permissions of its threads without the overhead of a system call (*mprotect* on Linux). Specifically, such changes are performed by simply changing the contents of the PKRU register. With our experimental platform (§6.3), changing the PKRU register takes approximately 13ns, while the latency of an *mprotect* call is between 913ns and 12μs, for 1 and 32 threads, respectively.

For *PUSh*, with a small number of EPDs, MPKs eliminate the need for separate page tables. Specifically, the access permissions of each thread to the different EPDs are determined, in part, based on the contents of its PKRU register.

MPKs can also reduce the overhead for acquiring and releasing locks. Specifically, if an object protected by the lock is in a separate MPK domain, only a thread with the appropriate value in its PKRU register can access the object. By default, the PKRU registers of all the threads are set to prevent any access to the object. The lock acquire operation is augmented to modify the PKRU register to allow the thread to access the protected object *after* the thread acquires the lock. *Before* actually releasing the lock, the lock release operation changes the PKRU register to restore the accessibility of the object to what it was prior to the lock acquire. Thus lock acquire and release operations can be performed without the slow *mprotect* system calls.

We use the *ctrace* benchmark (§6.3) to demonstrate the value of MPKs for *PUSh*. In *ctrace* a lock protects a hash table, which is accessed frequently by multiple threads. In a simple setup, this lock protects objects that are stored in six pages. Without MPKs, a *PUSh* implementation would have to invoke *mprotect* a total of twelve times during critical section entry and exit. We approximate the execution time of *ctrace* with an implementation of *PUSh* without the MPK optimization by delaying the thread acquiring or releasing a lock by the measured latency of the *mprotect* call multiplied by the required number of invocations. For one execution thread, with *PUSh* the execution time compared to the stock version increased by only 18%, while for the version without the MPK optimization, the execution time increased by a factor of 32.2.

PUSh maps the four EPDs for *read-shared*, *racy*, *inaccessible*, and *untouched* objects to a single domain: domain 1. Normal page table permissions are used to provide the access permissions appropriate for each of these sharing policies. For example, read-only access for the *read-shared* EPD. All the objects *private* to a particular thread are in a single EPD. All the *locked* objects protected by the same lock are in a single EPD. Each EPD is mapped to a different MPK domain, chosen from domain 2 to domain 15. MPK domain 0 is used for memory that is not tracked. For *private* and *locked* EPDs, *PUSh* maps each thread IDs and each lock addresses, respectively, to an MPK domain number between 2 and 15.

6.2.3 Dealing with the Limited Number of MPKs

The sum of the number of threads and number of locks often exceeds 14, while only 14 MPK domains are available for the *private* and *locked* EPDs. *PUSh* uses hashing to map thread IDs and lock addresses to domains. Obviously, multiple EPDs may map to the same MPK domain. These hashing collisions can hide data races. For example, a lock address may map to the same domain as a particular thread ID, allowing that thread to access the protected object without acquiring the lock. Similar problems can occur if the addresses of two different locks map to the same MPK domain or the thread IDs of two threads map to the same domain. *PUSh*'s mechanism for mitigating this problem is presented in this subsection.

If the sum of the number of threads and number of locks is much larger than 14, when a single instance of a race occurs, there is a probability of 1/14 that the race will not be detected. If the hash function is changed and the same race occurs again, there is a probability of 1/14 that the second instance of the race will not be detected. However, the probability that *both* instances of the race will be missed is $1/14^2$. This is the basis for *PUSh*'s mechanism for mitigating the problem of hashing collisions.

PUSh periodically changes the hash function. For this, it uses a universal family of hash functions, based on multiply-mod-prime [Tho18]. Based on linearity of expectation, it can be shown that, with n inputs (thread IDs and lock addresses), b MPK domains, and k different hash functions, for any pair of inputs, an upper bound on the probability of a collision in all k hash functions is $(n(n-1))/(2b^k)$. This bound is useful only when it is small. For example, if $n = 40$, $b = 14$ (as it is with *PUSh*), and $k = 6$, the upper bound is 0.01%. Thus, if, due to a hashing collision, a race is missed the first time it is encountered, as long as it occurs multiple times in the program, it will be detected later, when another hash function is in use. Since a different starting hash function is chosen every time the program executes, executing the program multiple times will ensure detection even if the race occurs only once in the program.

To implement the above, a timer periodically interrupts the main thread, causing it to generate a new hash function and send signals to all the other threads, causing them to stop. The main thread then iterates over the metadata of all the tracked objects. The MPK domain of every *private* or *locked* object, is reset, based on the new hash function. The main thread then signals all the other threads to resume execution. Based on the new hash function, each one of the application's threads re-initialize its PKRU register to "open" the protection domain for its current private domain and the domain whose corresponding locks are currently held by the thread. To facilitate this operation, the *PUSh* runtime maintains a list of all the locks currently held by threads. Finally, all threads return back to the user application. This entire rehashing procedure is effectively atomic since all the threads are blocked for its duration.

The effectiveness of the rehashing mechanism in practice is demonstrated in §6.4.3. The results presented in §6.4.5 show that, if the hash function is changed every few seconds, the performance overhead of rehashing is negligible.

6.2.4 Reducing the Memory Overhead

PUSh's memory overhead is directly related to the number of tracked objects, due to fragmentation (deficiency (3)) and the required metadata (§6.2.1). This is especially a severe limitations for programs that have large arrays. Specifically, if different array elements have different sharing policies, they must all be tracked objects allocated in separate pages. With

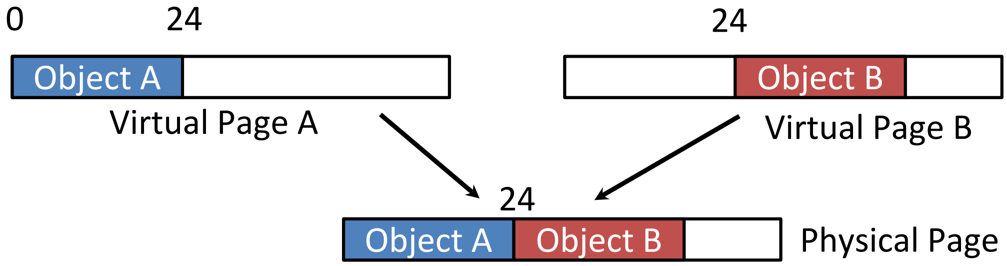


Figure 6.2: Mapping objects in different virtual pages into the same physical page.

the implementation of *PUSh* discussed so far, this would incur prohibitive memory overhead. This section is focused on how the memory overhead of *PUSh* can be reduced. Three ideas are presented: (1) reducing the overhead for metadata of arrays by incrementally increasing the size of the metadata on demand; (2) reducing the impact of fragmentation using memory mapping; and (3) using annotation to specify array slices, where all the elements within a slice have the same sharing policy.

An object's metadata includes its starting address and size. With optimization (1), all the elements of the array initially share one metadata structure. If the sharing policy of a slice of an array is modified to be different from its neighbors, a new metadata structure is allocated for this slice.

Optimization (2) reduces the memory overhead caused by internal fragmentations. Each tracked objects, including elements of array and structs, may have a different sharing policy and must thus be placed in a separate EPD. The optimization is based on mapping multiple virtual pages to a single physical page [DA06, AHM09, LZW17]. Specifically, as shown in Figure 6.2, each element is allocated in a different virtual page and at a different offset from the beginning of the page and they are mapped to the same physical page. No physical memory is wasted – consecutive elements are mapped to consecutive addresses in physical memory. Following the Linux's terminology, we refer to this type of mapping as *nonlinear* mapping. We refer to an array or struct to which this optimization is applied as a *split array* or *split struct*, respectively.

Unfortunately, with the current memory subsystem implementation in Linux, a large number of *nonlinear mappings* would incur prohibitive memory and performance overheads. Specifically, in terms of memory overhead, each *nonlinear mapping* would create 80 byte metadata. The metadata is collectively referred to as a VMA (Virtual Memory Area). There are typically multiple VMAs for each process in the system. Each VMA stores information for a contiguous memory region of the process, where each page in the region has the same VMA flags (e.g., same access permission). The VMA is used in the implementation of various memory-related functionalities, such as demand paging and page swapping. For each memory-related system call, such as *mmap* or *pkey_mprotect*, the kernel would need to find the corresponding VMA. Thus, in addition to the memory overhead, a large number of VMAs would also incur high performance overhead.

We overcome the above challenge of adopting optimization (2) by modifying the Linux kernel. Given that the VMA is a basic building block of Linux's memory subsystem, a key implementation challenge is to minimize the changes to the Linux kernel and also avoid affecting processes that do not use *PUSh*. The key goal of our kernel modifications is to avoid adding a VMA for each *nonlinear mapping*. This is done by introducing a new special type of VMA, which we call NM-VMA (Nonlinear Mapping VMA). Only pages within the NM-VMA are used for the *nonlinear mapping*. For memory pages within the NM-VMA, the kernel does not support features that rely on metadata stored in the VMA, such as on-demand mapping, page swapping and automatic page migration for NUMA machines. As a result of disabling those features, VMAs are not needed for each *nonlinear mapping*.

We currently only apply this optimization to dynamically-created objects, using an approach that is similar to that used in [DA06]. To simplify the implementation, we implement a layer above the heap allocator that still uses the heap allocator to manage the underlying physical memory. Specifically, upon process creation, *PUSh* creates an NM-VMA with a large number of virtual pages (100 million in the current implementation). When the application invokes *malloc* to create a tracked object, *PUSh* intercepts malloc (§6.2.1) and first invokes the real *malloc* to obtain a memory region for the object. We refer to this memory region: *real virtual memory region* and its address: *real virtual memory address*. *PUSh* then accesses each page of the *real virtual memory region* to force the kernel to allocate a physical page for each virtual page in it. *PUSh* then finds unused virtual pages in the NM-VMA and invokes an added system call that maps these virtual pages (*shadow virtual memory region*) in the NM-VMA to the same physical page mapped from the *real virtual memory region*. The *real virtual address* is stored in the metadata of the object and the *shadow virtual address* is returned to the application. When the application invokes *free*, *PUSh* first invokes an added system call to unmap the virtual pages in the *shadow virtual memory region*, finds the *real virtual address* in the metadata and then invokes the real *free* on the *real virtual address* to return the *real virtual memory region* to the heap allocator.

A downside of optimization (2) is that a system call is invoked by each *nonlinear mapping*. For arrays with a large number of elements that do not share the same sharing policy, creating such an array with optimization (2) will incur a long time. Optimization (3) facilitates efficient handling of such situation. It is suitable for applications where groups of array elements (array *slices*) have the same sharing policy. A runtime function allows the programmer to specify the number and sizes of the slices when the array is allocated. Each slice is handled as a separate tracked object. An array allocated and managed in this way is a *sliced array*.

For *split struct*, *split array* and *sliced array*, elements/members must be placed at different virtual addresses from those generated by the standard compiler. In our prototype implementation, this is done by modifying application source code. For *split struct*, padding is manually inserted in the struct definition. For *split array* and *sliced array* we use a script to replace the reference to the array with a macro that computes the correct memory address.

The metadata for each *sliced array* includes an array that stores the starting and ending index of each slice. In order to access a *sliced array* element, the code that maps an index to the correct memory address needs to walk through a metadata array. To reduce the performance overhead for such accesses, a very simple per-thread software cache is maintained for each *sliced array*. This cache stores the single most-recently-accessed slice.

To demonstrate the potential benefit of the software cache, we used a micro-benchmark that iterates over the one million integer elements of an array that consists of 32 slices. The software cache reduces the execution time overhead of *PUSh* from 4.7x to 2.2x.

6.2.5 Reducing Permission Changes Overhead

This section is focused on optimizations that mitigate the performance overhead of deficiencies (2.2) and (2.3) discussed in §6.2.1: (1) recycling recently-freed pages tagged with the same domain number, and (2) eliminating unnecessary remote TLB shootdowns. (3) Avoiding the serialization on policy changes incurred by kernel.

Upon every allocation of a dynamically-allocated object, the object needs to be placed in an EPD, requiring setting the protection domain tag in the corresponding page table entries (PTEs). When the object is freed, domain tags in the corresponding PTEs must be restored to 0 (§6.2.2), so that the freed pages may be reused by application components, such as standard libraries, that have not been annotated and instrumented for *PUSh*.

Optimization (1) reduces the number of PTE changes associated with allocate and free operations. For each protection domain number, *PUSh* maintains a simple software cache of up to 64 pages that are in that domain. The cache holds contiguous blocks of pages consisting of up to 32 pages. When possible, objects are allocated using pages from the cache and pages of freed objects are placed in the cache. Allocating objects using pages from the cache or freeing pages to the cache are done without requiring system calls. We call this cache a *domain-tagged page cache* (DTPC).

The DTPC is effective in reducing the overall overhead of *swaptions* (See §6.3). Without the DTPC, *PUSh* increases the execution time a factor of 2, 147 with 2, 32 threads respectively. With the optimization, the overhead caused by *PUSh* is less than 2%.

Optimization (2) reduces the performance overhead for TLB shootdowns. Following every change to PTEs in the page table, those PTEs must be flushed from any TLB that may cache them. In multithreaded programs, PTEs of pages containing shared objects may be cached in TLBs of multiple CPUs. TLB consistency is ensured by broadcasting IPIs (interprocessor interrupts) to all cores running the same process, causing each of them to flush the stale PTEs. The IPIs incur a significant overhead as the number of threads increase and thus limits the scalability of *PUSh*.

Optimization (2) utilizes the information provided by the *PUSh* annotations to identify whether a stale PTE can possibly be cached in remote TLBs. Specifically, *private* objects and *inaccessible* objects cannot be accessed by any remote CPUs and thus the corresponding PTEs cannot be cached in remote TLBs. Thus, when a *change policy* operation is applied on these objects, no IPI broadcasting is needed.

For Optimization (2), we implemented a small kernel modification, adding a local counterpart (*pkey_mprotect_l*) of the system call that changes the page table entry (*pkey_mprotect*). The local version does not broadcast IPIs for TLB flush. When a *change policy* is invoked on *private* or *inaccessible* objects, *pkey_mprotect_l* is invoked. It is critical to note that this works correctly *only* if threads are not allowed to migrate among cores. Hence, in all our experiments each application thread is pinned to a specific core.

We use *pfscan* (§6.3) to illustrate the benefit of Optimization (2). Originally, for 8 and 32 threads, the execution time increases by 31% and 80%, respectively, due to the 570 thousand calls to *pkey_mprotect*, where, on average, each call takes 15μs and 51μs, respectively. With the optimization, all but one *pkey_mprotect* is replaced by *pkey_mprotect_l*. The average latency of *pkey_mprotect_l* for 8 and 32 threads is 6μs and 34μs, respectively, resulting in application execution time increases of only 6.5% and 33%, respectively. The dependency of *pkey_mprotect_l* latency on the number of threads is due to contention for the kernel's memory subsystem lock.

Optimization (3) deals with the serialization of policy changes imposed by Linux kernel.

Specifically, in the Linux kernel, a single read-write lock (*mmap_sem*) protects all the VMAs of a process. A single lock is used in Linux because to minimize the overhead, VMAs will merge with adjacent VMAs if their VMA flags are the same, and also one VMA might split into several VMAs, if their VMA flags are no longer the same. As *pkey_mprotect* needs to modify the VMAs and thus possibly cause merge or split, concurrent *pkey_mprotect* needs to acquire a write lock of *mmap_sem* and thus serialized by this lock, limiting the scalability of *PUSh* with respect to increasing number of process threads. In fact, the Linux kernel developer is aware of the issue imposed by *mmap_sem* but unfortunately, due to the wide usage and the complex locking pattern of this lock, proposals to mitigate this problem, such as replacing *mmap_sem* with several finer granularity locks appear to be an extremely difficult task [lwna, lwnb].

We found that with the optimization (2) proposed in §6.2.4, the serialization can be easily resolved. Specifically, as the pages in the *NM-VMA* does not need to support VMA-related memory functionalities, for *pkey_mprotect* performed on the *NM-VMA*, no VMA changes are needed nor performed and hence, only a read lock needs to be acquired on *mmap_sem*. This enables *pkey_mprotect* applied on *NM-VMA* to be performed concurrently. We have found that the remaining shared resources, such as page tables, are protected by the fine granularity locks and thus does not cause a scalability problem. As almost all tracked objects are dynamically allocated and thus will be placed on the *NM-VMA*, allowing concurrent *pkey_mprotect* on this significantly reduces the performance overhead of *PUSh* (§6.4.5).

6.3 Experimental Setup

The experiments are performed on a machine running Fedora 27 with Linux Kernel version 4.15.13. The machine is equipped with 192GB memory and two 2.3GHz Intel Xeon Gold 6140 processor chips. Each chip contains 18 cores and a 24.75MB L3 cache.

Table 6.1 summarizes the 11 benchmarks used for evaluation. *Streamcluster* and *Swap-*

Table 6.1: Benchmarks used to evaluate *PUSh*.

Benchmark	Description
Ctrace [Ctr]	a multithreaded debugging library, evaluated by printing 32,000,000 debug records.
Pfscan-1.0 [Eri]	a parallel file scanner, evaluated by finding ‘hello’ in 500 copies of DSN proceedings.
Pbzip2-0.9.4 [Jie, YN09]	a parallel version of bzip2 file compressor, evaluated by compressing a 2GB file with random content.
nullhttpd-0.5.1 [nul]	a simple multithreaded web server evaluated with multiple ab [ab2] clients to retrieve a 10KB file 10K times.
Memcached-1.2.2 [mem]	key-value store, evaluated with Workload A (50% read, 50% write) and B (95% read and 5% write) of YCSB [CST10], with 100K 1KB records and 2M requests.
Streamcluster	a kernel that solves online clustering problems, evaluated with native input suite.
Blackscholes	a program that calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation, evaluated with the native input suite.
Swaptions	a kernel that uses HJM framework to price a portfolio of swaption, evaluated with the native input suite.
Ferret	Content-based similarity search application, evaluated with the native input suite.
Dedup	a kernel that compresses a data stream with a combination of global and local compression, evaluated with the native input suite.
Fft	a program that performs fast Fourier transform, evaluated with the native input suite.

tions are the same benchmarks as the ones used in prior sections: 3.5 and 5.3 with the same workload. We use an earlier version of the *Memcached* compared to the one used in the prior sections. This version of the *Memcached* is its first version that supports multithreading. We expect this version is likely to contain some data races so that we can evaluate *PUSh*'s ability to detect them.

Ctrace, *pbzip2*, *pfscan*, and *nullhttpd* are selected to facilitate direct comparisons with SharC [AGE08] and DCOP [MHC10]. For all benchmarks inputs are set so that the execution time is at least 10 seconds with the maximum number of threads. For all benchmarks, to minimize the performance skew caused by disk I/O, all the input/output files are placed in ramdisk. All the threads are pinned to dedicate cores to eliminate the intrusion caused by thread migration. To eliminate variable network latency effects, for *nullhttpd* and *memcached*, the client programs run on the same machine as the *server* program on a disjoint set of cores.

Since *PUSh* currently only supports C, C++ benchmarks: *pbzip2*, *streamcluster*, *swaptions* and *fft* were ported to C. We have verified that performance was not affected. To remove operations that can distort performance measurements, for *ctrace* we removed all *printf*s (which create extensive I/O activity) and replaced *localtime* (which serializes the threads) with a customized scalable version. We removed unnecessary *sleeps* in *nullhttpd*.

6.4 Evaluation

This section presents the discovered data races(§6.4.1), information regarding the annotations and changes to the source code (§6.4.2), a validation of the rehash mechanism (§6.4.3), the memory overhead (§6.4.4), and the performance overhead (§6.4.5). The section includes comparisons of *PUSh* with SharC [AGE08], DCOP [MHC10], and TSan [SI09, thr], in terms of the required annotations and code changes as well as memory and performance overheads.

An important optimization of *PUSh* is the use of the *NM-VMA*, which significantly reduces both the memory overhead (§6.2.4) and the performance overhead (§6.2.5). To illustrate this, we report both the memory and performance overhead of *PUSh* without this optimization (*PUSh-oo*: One to One mapping) and with this optimization (*PUSh-mo*: Multiple to One mapping) in §6.4.4 and §6.4.5.

6.4.1 Discovered Data Races

PUSh detected ten data races: eight were violations of the intended sharing policies and two due to detection of conflicting concurrent sharing policy changes (§6.1.2). The data races were: one each in *ferret* and *nullhttpd*, two each in *streamcluster* and *memcached*, and four in *pbzip2*. Out of the eight sharing policy violations, in four one thread was attempting to read an object *private* to another thread. In the remaining cases, the relevant object was *locked* and a thread attempted to access the object without acquiring the lock.

The two races, one in *nullhttpd* and the other in *pbzip2*, identified due to detection of conflicting concurrent sharing policy changes were related to customized synchronization code, where concurrent memory accesses are used. Since the C11 C standard prohibits such accesses to normal variables, these are data races. In these cases, inserting into the code the change policy operations necessary to avoid *PUSh* detecting violations of intended sharing policies, resulted in *PUSh* detecting conflicting concurrent policy changes.

To validate *PUSh*'s results, we ran our benchmarks with TSan [SI09, thr]. Initially, TSan detected three races that *PUSh* did not. However, these were all related to calls to standard library functions, for which TSan uses versions which are instrumented for data race detection. As a test, we also instrumented these functions. For example, to handle a call to *free()* of an object as an *acquire_write*. Once this was done, *PUSh* also detected all three races.

Table 6.2: Annotation overhead & code changes for *PUSh*.

Benchmark	*LOC*	Annotations		Changes	
ctrace	859	13	1.5%	7	0.8%
pfscan	753	19	2.5%	8	0.0%
pbzip2	7410	31	0.4%	0	0.0%
fft	723	33	4.6%	0	0.0%
streamcluster	1097	159	14.5%	0	0.0%
blackscholes	294	18	6.1%	0	0.0%
swaptions	1099	14	1.3%	0	0.0%
ferret	9663	80	0.8%	9	0.1%
nullhttpd	1348	3	0.2%	0	0.0%
memcached	3552	43	1.2%	3	0.1%
dedup	3416	33	1.0%	3	0.1%

6.4.2 Annotation and Code Changes

PUSh and related data race detectors [AGE08, AGN09, MHC10] require the programmer to annotate their code. This subsection quantifies this burden, based on our benchmarks (§6.3). Table 6.2 shows the annotations overhead of *PUSh*. The LOC column shows the count of the lines of code in the stock benchmarks, without blank lines or comments, as measured by CLOC [CLO].

The Changes column shows the source code modifications, which were all due to adding the padding space for *split struct* (§6.2.4). In addition to the changes reported in Table 6.2, a script was used to replace all the element references for *split array* and *sliced array* with macros (§6.2.4). The number of lines changes by the script were 15 for *fft*, 72 for *streamcluster*, 5 for *blackscholes*, 30 for *swaptions* and 4 for *dedup*.

Similarly to [AGE08, AGN09, MHC10], annotations are added using a trial-and-error approach. Initially, we run the benchmark with *PUSh*, without any annotations, and the benchmark aborts on the first shared access. We then analyze the code and add the proper annotations. With this methodology, annotations were added to most of the benchmarks within a few hours. The one exception was *streamcluster*, which took around 35 hours, due to complicated object sharing patterns. It should be noted that almost all of the time spent on code annotation was on understanding the code, with which we were not familiar. Hence, the task would have been *much* easier for the developers or maintainers of the code.

Annotation Burden Comparison. We compare the annotation burden of *PUSh* to DCOP [MHC10] and SharC [AGE08] based on the sum of annotations and other code changes, referred to as *mods*. For *pfscan*, *pbzip2*, *ctrace*, and *nullhttpd*, the numbers of *mods* required for *PUSh* are 27, 31, 20, 3. DCOP [MHC10] requires 62 (2.3x), 103 (3.3x) and 41 (2.1x), 13 (4.3x). DCOP's burden is higher since: (1) DCOP lacks the *locked* policy and thus requires insertion of two annotations for every object accessed in every critical sections, and (2) with DCOP, for statically-allocated objects, the default sharing policy is *inaccessible*, as opposed to *untouched* with *PUSh* (§6.2).

For *pfscan* and *pbzip2* the numbers of *mods* required for *PUSh* are 27 and 31. The SharC [AGE08] *mods* counts are comparable: 19 and 46. In general, compared to *PUSh*, the SharC (and Shoal [AGN09]) type system requires additional annotations on, for example, function arguments, local variables, and function return values. On the other hand, unlike *PUSh*, SharC does not require annotations before and after accessing *read-shared* objects. The *barrier* annotations supported in Shoal (§2.8), can significantly reduce the annotation overhead for programs with many barriers, such as *streamcluster*.

6.4.3 Validation and Effectiveness of Rehashing

This section demonstrates the effectiveness of the mechanism that periodically changes the hash function that maps thread IDs and lock addresses to MPK domains (§6.2.3). This is done using the eight data races reported in §6.4.1 detected as violations of intended sharing policies. These races were detected in *ferret*, *memcached*, *pbzip2*, and *streamcluster*.

In one test, the rehashing interval was set to five seconds and the thread count to 7-8. Each one of the four benchmarks was executed 50 times. The execution times were in the range of 15 to 222 seconds. For four of the benchmarks, all the races reported in §6.4.1 were detected in *every* execution. For *pbzip2*, out of the 50 executions, 43 detected all three data races. In each of the remaining seven runs, one race was missed. The missed race was always one of the two races that occurs only once, when the program exits. As discussed in §6.2.3, for such races, rehashing during a single execution obviously does not help.

In a second test, to increase the potential impact of domain collisions, thread IDs and lock addresses were mapped to only *two* MPK domains. Furthermore, the applications were run with the minimal number of threads that can trigger the race. With a fixed hash function, six out of the eight data races have a 50% probability to escape detection. The two remaining races are in *Streamcluster*, that performs multiple iterations on the input data. Each iteration creates new threads, with new thread IDs, that are terminated at the end of the iteration. Consequently, even with this setup, the probability of these races escaping detection is much smaller than 50%. Thus, in our experiments, they were always detected.

Changing the hash function once per second resulted in the detection of four out of the six problem data races in *every* execution. In those cases, the relevant code executed multiple times during a single execution of the program. As in the first test above, periodically changing the hash function did not help with two of the data races in *pbzip2* that only occur when *pbzip2* finishes processing.

6.4.4 Memory Overhead

This section presents *PUSh*'s memory overhead, including an evaluation of the effectiveness of the optimization mechanisms in §6.2.4.

We measured the memory usage of *PUSh* by measuring the *maximum* virtual memory

Table 6.3: Memory overhead. V: vsize, R: rss, oo: *PUSh-oo*, mo: *PUSh-mo*, max #threads.

Type	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$	ferret	dedup
V-oo	∼0	∼0	∼0	∼0	∼0	2.3%	∼0	∼0	260%	127%	NA
R-oo	∼0	∼0	∼0	∼0	∼0	64%	∼0	∼0	309%	444%	NA
R-mo	∼0	∼0	∼0	∼0	∼0	2.1%	∼0	∼0	3.7%	5.8%	2.4%

size (vsize) and resident set size (rss) during the execution of each of the benchmarks with the maximum number of threads. For *PUSh-oo*, the measurement of vsize is not meaningful, due to the pre-allocation of the *NM-VMA* (§6.2.4) and hence is not reported. Also, *dedup* cannot run with *PUSh-oo* (§6.2.4). Each of the benchmarks was executed twenty times, and the average results are reported. The measurement variations were less than 2%.

We compare the results for the stock benchmarks, linked with tcmalloc [Ghe] (§6.2.1) to the results with the full overhead of *PUSh*, which includes additional libraries, memory fragmentation, the metadata. These measurement are not completely accurate due to, for example, the granularity at which tcmalloc allocates virtual memory to the application.

As shown in Table 6.3, seven of the benchmarks do not have a measurable overhead in terms of both vsize and rss. With *PUSh-oo*, for *Swaptions*, both *Memcached* workloads, and *Ferret*, the increases are 7MB, 452MB, and 553MB, respectively, in both vsize and rss. With *PUSh-mo*, for *Swaptions*, both *Memcached* workloads, *Ferret* and *dedup*, the increases in rss are 260.0KB, 5.1MB, 7.2MB, 42.2MB respectively. Since the memory overhead increases negligibly with the number of threads, we report the results with just the maximum number of threads (31 or 32). As discussed in §6.2.4, *PUSh-mo* almost eliminates the fragmentation caused by placing objects on page boundaries and thus explains the memory overhead advantage.

As explained in §6.2.1, *PUSh*'s memory overhead is directly related to the number of tracked objects, since it is due to the metadata associated with each object, and for *PUSh-*

Table 6.4: Maximum number of tracked objects in each application with different thread counts. *Memcached* results are the same with both workloads.

#Threads	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$	**#Threads**	ferret	**#Threads**	dedup
2	19	53	20	37	39	1810	NA	38	100K	**7**	141K	**9**	682K
8	25	71	26	49	75	1834	78	44	100K	**23**	141K	**21**	754K
16	33	95	34	65	113	1852	141	52	100K	**27**	141K	**27**	767K
32	49	143	50	97	219	1861	268	68	NA	**31**	142K	**33**	869K

oo, fragmentation. Thus, the memory overhead results can be explained using Table 6.4, which shows the maximum number of tracked objects in each application. For most of the benchmarks, the number of tracked objects is so small that the memory overhead caused by them is below what our measurement procedure can detect. There are a few other sources of memory overhead, such as the DTPC (§6.2.5). However, with our benchmarks, the overhead due to these other sources is negligible.

The memory overhead due to metadata can be calculated based on the number of tracked objects shown in Table 6.4. In the best case, with no *read-shared* objects, the size of the metadata for each object would be 48 byte (§6.2.1). Thus, for *swaptions*, *memcached*, *ferret* and *dedup*, the required storage for metadata would be 86KB (2% of rss), 4.8MB (5% of rss), and 6.6MB (6% of rss) and 39.8MB (2.2% of rss) respectively. This overhead is negligible in terms of vsize. Half of this overhead is due to happens-before tracking. This best-case metadata overhead is not a significant portion of the total overhead reported above.

In the worst case for metadata overhead, every object is *read-shared*, requiring a full vector clock for happens-before tracking (§6.1.2). However, with our benchmarks, due to the *sticky-read* sharing policy (§6.1.2), the number of *read-shared* objects was less than 100. *Ferret* benefits most from the *sticky-read* policy since 99% of its objects are read shared by multiple threads. With the *sticky-read* sharing policy, the metadata overhead in *Ferret* is only 6% of rss.

For *PUSh-oo*, in most cases, the major source of memory overhead is fragmentation (§6.2.1). Several memory overhead optimization techniques are presented in §6.2.4. For our benchmarks, there are few opportunities to benefit from the *split struct* and *split array* optimizations. The most common situation for *split struct* was where one of struct members is a lock protecting the struct and must thus be a separate tracked object. The largest benefits of *split struct* was with *ferret*, that has five instances for which *split struct* was useful. For most of the arrays, the number of elements was well beyond the limit of 10000 that can be handled by the *split array* optimization. The largest reductions in memory overhead were 512KB with *swaptions* and 2MB with *streamcluster*. *PUSh-mo* eliminates almost all the memory overhead caused by the fragmentation and thus almost all of the *PUSh-mo*'s memory overhead is caused by the metadata.

For our benchmarks, the most effective memory overhead reduction technique was *sliced array*. Several data-parallel programs, such as *fft* and *blackscholes*, have a simple sharing pattern: each thread works on a disjoint set of the array elements. Such patterns are a good match for *sliced array*. Since these arrays are large, without *sliced array*, the memory overhead would have been prohibitive. The largest memory overhead reduction was with *fft*. Without *sliced array*, a total of 4TB of memory is needed for two large arrays. However, with *sliced array*, the memory overhead is negligible.

Memory Overhead Comparison. The most closely-related annotation-based race detectors [AGE08, MHC10], were not available to us for evaluation with our configuration. Hence, we compared the published results for several benchmarks that we also evaluated with *PUSh*. With SharC [AGE08], benchmarks in common are *pbzip2* and *pfscan*. For these benchmarks, the memory overheads of *PUSh* (Table 6.3) and the reported memory overhead of SharC were negligible. With DCOP [MHC10], benchmarks in common are *ctrace*, *nullhttpd*, *pbzip2*, and *pfscan*. For all these benchmarks, the memory overheads of *PUSh* were negligible, while the reported overheads for DCOP were in the range of 6.5% to

Table 6.5: Memory (rss) overhead comparison: additional memory use as percentage of use by stock application. P-oo: *PUSh-oo* vs. P-mo: *PUSh-mo* vs. T: TSan. Max #threads.

Type	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$	ferret	dedup
P-oo	∼0	∼0	∼0	∼0	∼0	64	∼0	∼0	309	444	NA
P-mo	∼0	∼0	∼0	∼0	∼0	2.1	∼0	∼0	3.7	5.8	2.4
T	94	391	406	*11K*	408	170	54	203	412	386	376

14%.

If [MHC10, AGE08] were enhanced to flag unordered sharing policy changes (§6.1.2), their memory overhead for every dynamically-checked 16/8-byte block would increase from one byte by at least 16 more bytes. *PUSh* requires the extra tracking metadata per object, as opposed to per fixed-size block. Thus, at least for the benchmarks evaluated, *PUSh* would have a much greater memory overhead advantage.

To compare *PUSh*'s memory overhead with that of TSan, we evaluated TSan on our experimental platform. The results in Table 6.5 are based only on rss since, due to TSan's implementation, vsize measurements for it are meaningless [thr]. The results are that, with *PUSh-oo* for nine out of the ten benchmarks, *PUSh*'s memory overhead is lower, *much* lower for eight of them. With *ferret*, *PUSh-oo*'s memory overhead is a little higher, but this must be weighed against *PUSh-oo*'s dramatically lower (factor of 145) performance overhead 6.5. With *PUSh-mo*, its memory overhead is always two orders of magnitude lower than that of TSan.

As in Table 6.3, the results in Table 6.5 are with the maximum number of threads. With TSan, for eight of the benchmarks, the number of threads does not affect the memory overhead. For *pbzip2* and *pfscan*, TSan's performance overhead is very high (§6.4.5) and this affect the execution characteristics with TSan so that the *maximum* memory use does decrease with decreasing thread counts. The memory overhead for *pfscan* is still much higher

Table 6.6: Performance overhead (in percentage) and policy change rates with different thread counts. The policy change rates are the number of changes per second. Left:*PUSh-oo*, Right:*PUSh-mo*

#Threads	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$A	mem$B	ferret	**#Threads**
	Performance Overhead (%)											
1	18	1.9	NA	NA	NA	NA	NA	~0	37	10	5.4	**7**
2	14	1.9	1.9	1.7	1.2	~0	NA	~0	59	19	6.0	**11**
4	10	2.1	1.9	2.0	7.1	1.0	0.8	~0	79	21	7.3	**19**
8	7	3.0	2.0	6.5	13	1.1	1.9	~0	99	39	8.2	**23**
16	7	3.2	2.1	14	21	1.4	2.3	~0	82	23	9.1	**27**
32	12	3.3	2.1	33	99	1.4	3.9	~0	NA	NA	11	**31**

#Threads	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$A	mem$B	**#Threads**	ferret	**#Threads**	dedup
	Performance Overhead (%)													
1	19	1.8	NA	NA	NA	NA	NA	~0	37	9	**7**	0.4	**6**	30
2	14	1.9	1.8	~0	~0	~0	NA	~0	48	18	**11**	0.5	**9**	30
4	10	2.1	1.9	~0	~0	~0	~0	~0	51	19	**19**	0.5	**15**	31
8	7	3.0	2.0	~0	~0	~0	~0	~0	67	35	**23**	0.5	**21**	36
16	7	3.2	2.1	~0	~0	~0	~0	~0	59	21	**27**	0.6	**27**	55
32	12	3.2	2.2	7.0	29	0.3	~0	~0	NA	NA	**31**	0.6	**33**	54
Policy Change Rate														
4	0.9	3.0	0.5	7K	2K	6.4	78	0.3	41K	6K	**7**	1K	**15**	205K
16	0.8	12.2	3.3	14K	16K	27.8	648	0.3	35K	5K	**27**	8K	**27**	276K
32	0.7	23.1	7.6	18K	55K	48.6	1361	0.2	NA	NA	**31**	9K	**33**	261K

than *PUSh*'s —at least 300% for all thread counts. For *pbzip2* the maximum memory use is significantly lower than that of the stock application for low thread counts.

6.4.5 Performance Overhead

This section presents the performance overhead of *PUSh* with the performance optimization discussed in §6.2.

Table 6.6 presents the performance overhead of *PUSh*, varying the number of threads from 1 to 32. For all the benchmarks, the performance measure is execution time. For *memcached* and *nullhttpd*, it is the execution time under maximum throughput. The reported overheads are averages over 20 executions. The measurements varied less than 2% over 20 runs. An *NA* in a cell of the table indicates that the benchmark could not be run with the corresponding number of threads.

For *PUSh-oo*, as shown in Table 6.6, for 7 out of 10 benchmarks, the performance overhead is under 19% for all thread counts. *Memcached* with YCSB Workload A is the worst case, reaching 59% overhead with two threads and 99% overhead with eight threads. For *PUSh-mo*, only two benchmarks: *Memcached* with YCSB workload A and *dedup* has a overhead greater than 35%, with the worst case being only 67%. The performance advantage of *PUSh-mo* over *PUSh-oo* is because *PUSh-mo* eliminates the serialization of policy changes caused by *mmap_sem* (§6.2.5).

For *ctrace* and *memcached-B*, due to frequent lock operations, most of the performance overhead is caused by modifying PKRU registers (13ns) and happens-before tracking. For *fft* and *blackscholes*, the main overhead is due to additional time spent accessing the array elements for *sliced array* (§6.2.4). For all the other benchmarks, most of the overhead is due to page table permission changes caused by sharing policy changes. The lower part of Table 6.6 presents the policy change rate for the different benchmarks. In all the cases where the performance overhead of *PUSh* is over 19%, that overhead is highly correlated with the policy change rate.

For most of the benchmarks, the performance overhead of the happens-before tracking is insignificant. The exceptions are *ctrace* and *memcached-B*. For *ctrace*, with one thread, happens-before tracking is responsible for around 33% out of the 18% or 19% overhead. When the number of threads is 16 or higher, this tracking is responsible for nearly all the overhead. For *memcached-B*, happens-before tracking it responsible for 17% to 51% of the overhead.

With *PUSh-oo*, for *streamcluster* and *pfscan*, the overhead of *PUSh* is highly dependent on the number of threads. This is due to the fact that, in the Linux kernel, a single lock (*mmap_sem*) serializes all page table changes invoked by the threads of a process. To directly quantify the impact of the serialization caused by the *mmap_sem*, we measured the average time spent in page table changes by *streamcluster*. With two threads, only one of which is active when the page table change is invoked, the average latency of the operation was 1μs. With 32 threads, this latency was 89μs. We repeat the measurements for *PUSh-mo* and found that the latency was 4μs with 32 threads. The latency is still higher than the one thread case for two reasons: (1) the serialization caused by other kernel locks such as the lock for page table entry and (2) the overhead of broadcasting IPIs to perform TLB shootdown (§6.2.5) increases as the number of threads increase.

Table 6.7: The average latency, in μs, of changing the hash function with different thread counts. The results for *memcached* are the same with the two workloads. Benchmarks for which the average latency is less than 1ms are not shown.

#Threads	fft	blksch	strmcl	swaptions	mem$	**#Threads**	ferret	**#Threads**	dedup
1	$9K$	NA	NA	NA	$135K$	**7**	$24K$	**6**	$226K$
2	$10K$	$1.4K$	200	$3.0K$	$136K$	**11**	$25K$	**9**	$230K$
4	$10K$	$1.8K$	282	$3.4K$	$136K$	**19**	$25K$	**15**	$256K$
8	$11K$	$2.3K$	430	$3.6K$	$137K$	**23**	$25K$	**21**	$273K$
16	$12K$	$2.6K$	737	$4.0K$	$137K$	**27**	$26K$	**27**	$282K$
32	$13K$	$2.8K$	1547	$4.5K$	NA	**31**	$26K$	**33**	$304K$

As discussed in §6.2.3, *PUSh* must periodically change the hash function used to hash thread IDs and lock addresses to MPK domain numbers. The latency of this operation contributes to the performance overhead of *PUSh*. We measured this latency for all the benchmarks for all thread counts as shown in Table 6.7. The highest latency for changing the hash function was 304ms for *dedup* with 33 threads. The second highest latency is 137ms for *memcached* with 8 or 16 threads. For all the other benchmarks, the highest latency was 26ms. These results indicate that, for most benchmarks, if the hash function is changed every few seconds, the associated overhead is negligible.

The most significant factors that determine the latency of changing the hash function are the number of *private* or *locked* objects and the total size (number of pages) of those objects. Both of these factors affect the time spent resetting the protection domains. *Dedup* and *Memcached* are extreme cases since they have a large number of objects and a significant portion of them are *locked* objects.

Performance Overhead Comparison. With SharC [AGE08], benchmarks in common, *pbzip2* and *pfscan*, were executed with thread counts of 5 and 3, respectively, with reported performance overheads of 11% and 12%, respectively. For *PUSh-oo*, its corresponding overheads were 2% or less, with overheads of 6.5% or less for 8 or fewer threads (Table 6.6). For *PUSh-mo*, its corresponding overheads were all negligible.

With DCOP [MHC10], benchmarks in common *ctrace*, *nullhttpd*, *pbzip2*, and *pfscan* were executed with thread counts of 2, 50, 5, and 3, respectively, with reported performance overheads of 27%, 0, 49% and 37.2%, respectively. *PUSh*'s closest corresponding overheads, as presented in Table 6.6, were significantly lower. For *PUSh-oo*, they are 14%, 0, 1.9%, and 2% and for *PUSh-mo*, they are 14%, 0, 0%, 0%. *PUSh* had lower overheads even with 32 threads.

As explained in §6.3, we modified *ctrace* and *nullhttpd* in order to be able to obtain meaningful performance results. However, for this comparison with DCOP, we also evaluated the original versions of *ctrace* and *nullhttpd*. With the unmodified *ctrace*, *PUSh* does not incur any performance overhead. For the original *nullhttpd* with 50 threads, DCOP reports and overhead of 24% in CPU cycles, while *PUSh* does not incur any overhead in CPU cycles, throughput, or latency.

If [MHC10, AGE08, AGN09] were enhanced to flag unordered sharing policy changes (§6.1.2), *PUSh*'s performance overhead advantage would be even greater. Specifically, for each object larger than 16/8-bytes, these operations would require checking and modifying tracking metadata for multiple blocks, as opposed to *PUSh*, where the metadata is per object.

Table 6.8: Performance overhead comparison: additional execution time as percentage of the stock application execution time. P-oo: *PUSh-oo* vs. P-mo: *PUSh-mo* vs. T: TSan.

Type	ctrace	fft	blksch	pfscan	strmcl	swaptn	pbzip2	nhttpd	mem$A	mem$B	ferret	dedup
8 threads												
P-oo	7	3.0	2.0	6.5	13	1.1	1.9	∼0	99	39	8.2	NA
P-mo	7	3.0	2.0	∼0	∼0	∼0	∼0	∼0	67	35	0.5	36
T	423	507	473	*13K*	2567	746	4245	529	780	384	1612	1303
Max #threads												
P-oo	12	3.3	2.1	33	99	1.4	3.9	∼0	82	23	11	NA
P-mo	12	3.2	2.2	7.0	29	0.3	∼0	∼0	59	21	0.6	54
T	304	392	1420	*36K*	6588	693	3940	2485	873	356	1595	1397

Table 6.8 presents a performance overhead comparison between *PUSh* and TSan. For *PUSh-oo*, its overhead is at least a factor of 7.8 lower and for *PUSh-mo*, its overhead is at least a factor of 11.6 lower. In several cases, compared to TSan, *PUSh*'s overhead is three orders of magnitude lower. These results reinforce the argument that *PUSh* can be used in production runs while TSan is restricted to offline debugging.

6.5 Limitations and Disadvantages

Like all data race detectors, *PUSh* has limitations and disadvantages. Some of these are inherent (§6.1), some are associated with tradeoffs made for efficient implementation. This section summarizes these limitations and disadvantages. Despite these, as shown in other sections of this chapter, *PUSh*, as currently implemented, is a useful tool with important advantages over other existing race detectors.

A key disadvantage of *PUSh* is that it requires annotation of the code. Furthermore, since *PUSh* relies on happens-before tracking [PS07, FF09] (§6.1.2), races that, in a single execution, can escape detection by full happens-before tracking [SBN97], can also escape detection by *PUSh*.

Implementation considerations motivate the mapping of LPDs to a smaller number of EPDs (§6.1.3). A consequence of that is that all *read-shared* objects are placed in a single EPD and this introduces the possibility of failing to detect races in limited specific scenarios. The *locked* sharing policy is also introduced as a performance optimization (§6.1.3). Associated with this is the fact that, under certain scenarios, a change policy from *locked* can result in missed races.

PUSh's implementation relies on MPKs (§6.2.2). The problem of potential missed races due to the limited number of MPKs is mitigated by periodically changing the hash function (§6.2.3). While this rehashing mechanism is highly effective (§6.4.3), there is, still, a small probability of missing races due to hash collisions, especially for races that occur only once during the execution of the program. The *pkey_mprotect_l* optimization (§6.2.5) requires threads to be pinned to cores. If this cannot be done, in some cases, the performance overhead may increase significantly.

6.6 Summary

Decades of development of date race detectors have resulted in a rich design space of techniques and tools that vary widely in terms of precision, soundness, types of races detected, performance overhead, memory overhead, scalability, burden imposed on programmers, and applicability to various programming languages. A useful subspace of this design space covers race detectors that require explicit specification of intended sharing and then identify violations of these intentions. Since such tools don't have to infer the intended sharing, they have the potential of reduced complexity, increased precision and soundness, and reduced overhead.

PUSh advances the state-of-the-art in the design subspace described above by introducing the use of happens-before tracking to check for conflicting concurrent sharing policy changes. A second key novel feature of *PUSh* is the way it uses off-the-shelf hardware to reduce the performance overhead for detecting unintended sharing. A key aspect of the design and implementation of *PUSh* is the use of memory protection keys (MPKs). *PUSh* uses a universal family of hash functions to overcome the limitations of the widely-available implementation of MPKs. Additional optimizations that range from enhanced annotations to OS kernel changes further reduce memory and performance overheads. In many cases, the performance overhead of *PUSh* is at a level that allows its use during production runs.

We evaluated *PUSh* with eleven benchmarks and up to 32 threads. *PUSh* detected ten races. Comparison with results from ThreadSanitizer [SI09, thr] shows that, excluding races due to standard library calls, no data races were missed. *PUSh*'s memory overhead was under 5.8% for all the benchmarks. *PUSh*'s performance overhead exceeded 35% for only two of the benchmarks, reaching only 67% overhead in the worst case. Our work included targeted evaluations of the various optimization introduced in this work. For example, we have shown that, in an extreme case, *PUSh*'s novel use of MPKs reduces the performance overhead from a factor of 32.2 to just 18%. The kernel support added to *PUSh* significantly reduces the memory and performance overhead. We have also shown how additional annotations can dramatically reduce *PUSh*'s memory overhead.

CHAPTER 7

Conclusion and Future Work

Critical services require dependability mechanisms to prevent service failures due to hardware and/or software faults. Building a dependable system involves a tradeoff between soundness and overhead. Dependable systems for mainstream deployment are built upon commodity hardware with mechanisms that enhance resilience implemented in software. The goal of such dependable systems is to provide commercially viable, best-effort dependability cost-effectively. The focus of this thesis is on dependability mechanisms that can be deployed in this type of systems.

This thesis presents several practical, low-overhead dependability mechanisms for critical components in the system: hypervisors, containers, and parallel applications. The approaches towards building such dependability mechanisms are first, identifying sweet spots in the design space that balance soundness and overhead. Second, novel use of hardware to optimize critical operations in the dependability mechanisms. Third, dedicated optimizations of the internals of operating systems/hypervisors. Each of the proposed mechanisms in this thesis reduces some aspects of overhead by orders of magnitude while still maintaining nearly the same level of soundness compared to directly-comparable prior works.

Specifically, this thesis proposes four dependability mechanisms: *NiLiHype*, *NiLiCon*, *HyCoR*, and *PUSh*. We have implemented and evaluated the prototypes of these four dependability mechanisms on widely used software platforms: the Xen hypervisor [BDF03] and the Linux kernel.

NiLiHype is a recovery mechanism that recovers hypervisors from failures due to transient hardware and software faults. *NiLiHype* advances the prior start-of-the-art technique for hypervisor fault tolerance: *ReHype* [LT11, LT14], which similarly recovers hypervisors from failures due to these faults. *ReHype* is based on recovering the hypervisor from failure by rebooting the hypervisor. This thesis proposes a novel component-level recovery mechanism called microreset and *NiLiCon* is an implementation of microreset on the Xen hypervisor [BDF03]. Specifically, upon hypervisor failure, instead of rebooting a new hypervisor instance, *NiLiCon* resets the hypervisor to a quiescent state that is highly likely to be valid and where the hypervisor is ready to handle new or retried requests from the rest of the system. *NiLiCon* then resolves state inconsistencies within the restored system. It finally retries requests interrupted by the failure transparently and thus finishes the recovery. *NiLiHype* trades a small reduction in recovery rate (<2%) for a significant reduction in recovery latency. *NiLiHype* reduces recovery latency from 713ms (with *ReHype*) to 22ms, a factor of over 30x.

Microreset is suitable for large, complex components that process requests from the rest of the system. One direction of future work related to *NiLiHype* is to investigate whether it is possible to apply microreset to microkernels. With the increasing demand for high reliability and security, microkernels are likely to become widely used in the near future. However, there is very little work on fault tolerance mechanisms for microkernels. With a microkernel, each component is isolated from other components by page-level protection, and communication among components is through well-defined interfaces. Hence, a fault that occurs in one component is unlikely to propagate to others and thus makes microkernels suitable targets for component-level recovery mechanisms, such as microreset.

Data centers often rely on VMs and containers to provide an isolation and multitenancy layer [Ber14, Mer14, RG05]. Due to their lower resource requirements and reduced management costs [Ber14, LKG15], in many situations, there are compelling reasons to deploy containers alone instead of VMs. However, there has been very little work on the fault tolerance of containers. This thesis proposes *NiLiCon*, which, to the best of our knowledge, is the first container fault-tolerance mechanism that is transparent to applications and clients and supports stateful applications. *NiLiCon* is a mechanism for running duplicated containers, thus providing the ability to tolerate container fail-stop failures. *NiLiCon* applies the algorithm of a widely used VM replication mechanism: Remus [CLM08] to containers. The starting point for the implementation of *NiLiCon* is CRIU [cria], a container checkpoint/restore tool. Due to the tight state coupling between a container and the underlying kernel, a key challenge in implementing *NiLiCon* is to efficiently checkpoint and restore parts of the container state that are in the kernel. To overcome this challenge, *NiLiCon* adopts various critical optimizations to the Linux kernel and CRIU. *NiLiCon* achieves performance that is competitive with *Remus*. Specifically, for a set of seven benchmarks, the overhead with *NiLiCon* is from 19%-67% versus 13%-72% with *Remus*.

HyCoR builds upon *NiLiCon* and similarly provides the ability to tolerate container fail-stop failures. *HyCoR* enhances *NiLiCon* with deterministic replay. *HyCoR* leverages deterministic replay to decouple the checkpointing interval from output delay and thus resolves a fundamental disadvantage of all *Remus*-based schemes: long delay of outputs to clients. This also enables configurations with low throughput overhead by using longer checkpointing intervals. *HyCoR* includes a simple timing adjustment technique to effectively replay applications containing data races, as long as their rate of unsynchronized write operations is low. *HyCoR* operates in two modes: a long epoch mode that is suitable for applications that are known to be data-race-free and a short epoch mode that is suitable for applications that may include data races. We evaluate *HyCoR* with a set of eight benchmarks. *HyCoR* incurs only a small extra delay on outputs to the clients. Specifically, the latency overhead of *HyCoR* is 150μs-572μs versus 36ms-51ms with *NiLiCon*. In short epoch mode, *HyCoR*'s performance is comparable to *NiLiCon*'s (8%-145% versus 18%-139% with *NiLiCon*). The long epoch mode has much better performance: in all cases, the overhead is <58%. For applications without data races, *HyCoR*'s recovery rate is 100%. For applications with a low rate of unsynchronized write operations, *HyCoR*'s recovery rate is >99.5%.

One direction of future work related to *HyCoR* is to investigate the use of other hardware features to further optimize *HyCoR*. For example, Intel PT is a hardware feature that records the control flow of program executions with fine-granularity timing information. Intel PT can be used in *HyCoR* to record the outcomes of synchronization operations and also identify the results of data races. As another example, RDMA can be used to efficiently transfer non-deterministic event logs and checkpoints to the backup to further reduce the latency overhead. Together, these enhancements may enable *HyCoR* to provide microsecond-scale fault tolerance for the emerging microsecond-scale applications in data centers.

PUSh is a dynamic data race detector with extremely low overhead. *PUSh* requires programmers to annotate each global object with its intended sharing policy. It then uses existing memory protection hardware to enforce the annotated sharing policies. *PUSh* contributes an efficient algorithm to detect incorrect annotations that can hide data races. This algorithm can be applied to other annotation-based data race detectors. Another key contribution of *PUSh* is to show that it can be implemented on top of commodity hardware with low overhead. This is achieved with a key optimization that exploits Intel MPK, a hardware feature recently added to the x86 ISA. *PUSh* also develops novel enhancements to the memory management subsystem in the Linux kernel to eliminate the memory overhead due to placing each object in a separate page. Comparing *PUSh* to conventional data race detectors, such as the widely used ThreadSanitizer(TSan) [thr], a clear advantage of these conventional detectors is that they do not require annotations. Furthermore, in rare cases, due to the limited number of protection domains supported by MPK, *PUSh* might miss a manifested data race. Thus, *PUSh* may require several executions to eventually detect all the data races in the application. However, in return, the memory and performance overhead of *PUSh* is often orders of magnitude smaller. Specifically, for a set of eight benchmarks, with TSan, the memory overhead is 54%-11000% and the performance overhead is 304%-36000%. While with *PUSh*, the memory overhead is only 0%-5.8% and the performance overhead is only 0%-54%. Such low overhead allows *PUSh* to be used in production runs or at least during beta testing.

One direction of future work based on *PUSh* is to further reduce the performance overhead. As discussed in §6.4.5, with *PUSh*, most of the overhead is associated with policy changes. Specifically, each of the policy changes involves a system call to change the page table and sometimes requires broadcasting an IPI to flush the TLB entry. Hence, *PUSh* is not suitable for applications that have small objects with very frequent policy changes. It may be worthwhile to incorporate memory instrumentation into *PUSh*. Specifically, page-level protection could be used to enforce sharing policies for large objects with low policy change rates. The memory instrumentation could be used to enforce sharing policies for small objects with high policy change rates. Such a combination could lead to a practical default-on data race detectors for parallel applications.

REFERENCES

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop.” In *2009 IEEE International Symposium on Parallel and Distributed Processing*, Rome, Italy, May 2009.

[ab] “ab - Apache HTTP server benchmarking tool.” `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 2020-08-07.

[ab2] “ab-Apache HTTP server benchmarking tool.” `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 2018-12-31.

[ABE01] Lorenzo Alvisi, Thomas C. Bressoud, Ayman El-Khashab, Keith Marzullo, and Dmitrii Zagorodnov. “Wrapping Server-Side TCP to Mask Connection Failures.” In *IEEE INFOCOM*, pp. 329–337, Anchorage, AK, April 2001.

[aer] “Aerospike.” `https://https://www.aerospike.com/`. Accessed: 2020-08-07.

[AGE08] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. “SharC: Checking Data Sharing Strategies for Multithreaded C.” In *29th ACM Conference on Programming Language Design and Implementation*, pp. 149–158, Tucson, AZ, June 2008.

[AGN09] Zachary Anderson, David Gay, and Mayur Naik. “Lightweight Annotations for Controlling Sharing in Concurrent Data Structures.” In *30th ACM Conference on Programming Language Design and Implementation*, pp. 98–109, Dublin, Ireland, June 2009.

[AHM09] Martín Abadi, Tim Harris, and Mojtaba Mehrara. “Transactional Memory with Strong Atomicity Using Off-the-shelf Memory Protection Hardware.” In *Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 185–196, Raleigh, NC, USA, February 2009.

[AS09] Gautam Altekar and Ion Stoica. “ODR: Output-Deterministic Replay for Multicore Debugging.” In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, p. 193–206, Big Sky, Montana, USA, October 2009.

[AT01] Navid Aghdaie and Yuval Tamir. “Client-Transparent Fault-Tolerant Web Service.” In *20th IEEE International Performance, Computing, and Communications Conference*, pp. 209–216, Phoenix, AZ, April 2001.

[AT09] Navid Aghdaie and Yuval Tamir. “CoRAL: A Transparent Fault-Tolerant Web Service.” *Journal of Systems and Software*, **82**(1):131–143, January 2009.

[BDD10] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization." In *9th USENIX Conference on Operating Systems Design and Implementation*, pp. 423–436, Vancouver, BC, Canada, October 2010.

[BDF03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." In *19th ACM Symposium on Operating Systems Principles*, pp. 164–177, Bolton Landing, NY, October 2003.

[Ber14] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes." *IEEE Cloud Computing*, **1**(3):81–84, September 2014.

[BHC10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. "Deterministic Process Groups in DOS." In *9th USENIX Conference on Operating Systems Design and Implementation*, p. 177–191, Vancouver, BC, Canada, October 2010.

[BHH13] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and H. Reza Taheri. "Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications." *VMware Technical Journal*, **2**(1):19–28, June 2013.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks." In *17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 211–230, Seattle, Washington, USA, November 2002.

[BS95] Thomas C. Bressoud and Fred B. Schneider. "Hypervisor-based Fault Tolerance." In *15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, December 1995.

[BVK16] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. "OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems." In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 25–36, Toulouse, France, June 2016.

[BW89] Amnon Barak and Richard Wheeler. "MOSIX: An Integrated Multiprocessor UNIX." In *USENIX Winter 1989 Technical Conference*, San Diego, CA, USA, February 1989.

[CC13] Yufei Chen and Haibo Chen. "Scalable Deterministic Replay in a Parallel Full-System Emulator." In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 207–218, Shenzhen, China, February 2013.

[CKF04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. "Microreboot — A Technique for Cheap Recovery." In *6th Symposium on Operating Systems Design and Implementation*, pp. 31–44, San Francisco, CA, December 2004.

[CLL02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. "Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs." In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 258–269, Berlin, Germany, June 2002.

[CLM08] 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*, pp. 161–174, April 2008.

[CLO] "Cloc – count lines of code." `http://cloc.sourceforge.net/`. Accessed: 2018-12-04.

[cria] "CRIU: Checkpoint/Restore In Userspace." `https://criu.org/Main_Page`. Accessed: 2020-08-07.

[crib] "CRIU Task-diag." `https://criu.org/Task-diag`. Accessed: 2019-10-02.

[CST10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB." In *1st ACM Symposium on Cloud Computing*, pp. 143–154, June 2010.

[CSX16] Peter M. Chen, Daniel J. Scales, Min Xu, and Matthew D. Ginzton. "Low Overhead Fault Tolerance Through Hybrid Checkponting and Replay.", August 2016. Patent No. 9,417,965 B2.

[Ctr] "Ctrace." `http://ctrace.sourceforge.net/index.php`. Accessed: 2018-12-24.

[CZG15] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. "Deterministic Replay: A Survey." *ACM Computing Surveys*, **48**(2), September 2015.

[DA06] Dinakar Dhurjati and Vikram Adve. "Efficiently Detecting All Dangling Pointer Uses in Production Servers." In *36th IEEE International Conference on Dependable Systems and Networks*, pp. 269–280, Philadelphia, PA, June 2006.

[DCC08] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. "CuriOS: Improving Reliability Through Operating System Structure." In *8th USENIX Conference on Operating Systems Design and Implementation*, pp. 59–72, San Diego, California, December 2008.

[djc] "An improved django-admin-tools dashboard for Django projects." `https://github.com/django-fluent/django-fluent-dashboard`. Accessed: 2019-10-02.

[DKC03] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay." In *5th Symposium on Operating Systems Design and Implementation*, pp. 211–224, Boston, MA, USA, December 2003.

[DLF08] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. "Execution Replay of Multiprocessor Virtual Machines." In *Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, p. 121–130, Seattle, WA, USA, March 2008.

[DO91] Fred Douglis and John Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software – Practice and Experience*, **21**(8):757–785, August 1991.

[drb] "Install Xen 4.2.1 with Remus and DRBD on Ubuntu 12.10." `https://wiki.xenproject.org/wiki/Install_Xen_4.2.1_with_Remus_and_DRBD_on_Ubuntu_12.10`. Accessed: 2019-10-02.

[DS10] Alex Depoutovitch and Michael Stumm. "Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes." In *5th European conference on Computer systems*, pp. 181–194, Paris, France, April 2010.

[DWS12] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. "RADISH: Always-on Sound and Complete Ra Detection in Software and Hardware." In *39th Annual International Symposium on Computer Architecture*, pp. 201–212, Portland, Oregon, USA, June 2012.

[DYJ13] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service." In *4th ACM Annual Symposium on Cloud Computing*, Santa Clara, CA, October 2013.

[EA03] Dawson Engler and Ken Ashcraft. "RacerX: Effective, Static Detection of Race Conditions and Deadlocks." In *Nineteenth ACM Symposium on Operating Systems Principles*, pp. 237–252, Bolton Landing, NY, USA, October 2003.

[ELC12] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. "IFRit: Interference-free Regions for Dynamic Data-race Detection." In *2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 467–484, Tucson, Arizona, USA, October 2012.

[EMB10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. "Effective Data-race Detection for the Kernel." In *Nineth USENIX Conference on Operating Systems Design and Implementation*, pp. 151–162, Vancouver, BC, Canada, October 2010.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "Goldilocks: A Race and Transaction-aware Java Runtime." In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 245–255, San Diego, California, USA, June 2007.

[Eri] "Pfscan." `ftp://ftp.lysator.liu.se/pub/unix/pfscan`. Accessed: 2018-12-24.

[FF00] Cormac Flanagan and Stephen N. Freund. "Type-based Race Detection for Java." In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 219–232, Vancouver, British Columbia, Canada, June 2000.

[FF09] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and Precise Dynamic Race Detection." In *30th ACM Conference on Programming Language Design and Implementation*, pp. 121–133, Dublin, Ireland, June 2009.

[Ghe] "TCMalloc : Thread-Caching Malloc." `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`. Accessed: 2018-12-13.

[GHY14] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. "Rex: Replication at the Speed of Multi-Core." In *9th European Conference on Computer Systems*, pp. 161–174, Amsterdam, The Netherlands, April 2014.

[goo] "Failure Rates in Google Data Centers." `https://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/`. Accessed: 2020-12-21.

[Gra86] Jim Gray. "Why Do computers Stop and What Can Be Done About It?" In *5th Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, January 1986.

[Gro03] Dan Grossman. "Type-safe Multithreading in Cyclone." In *2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pp. 13–25, New Orleans, Louisiana, USA, January 2003.

[GWT08] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. "R2: An Application-Level Kernel for Record and Replay." In *8th USENIX Conference on Operating Systems Design and Implementation*, p. 193–208, San Diego, CA, USA, December 2008.

[HBG06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "Construction of a Highly Dependable Operating System." In *Sixth European Dependable Computing Conference*, Coimbra, Portugal, October 2006.

[HBS14] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. "C/C++ Thread Safety Analysis." In *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 41–46, Victoria, BC, Canada, September 2014.

[HH08] Derek R. Hower and Mark D. Hill. "Rerun: Exploiting Episodes for Lightweight Memory Race Recording." In *35th Annual International Symposium on Computer Architecture*, p. 265–276, Beijing, China, June 2008.

[hir] "Minimalistic C client for Redis." `https://github.com/redis/hiredis`. Accessed: 2020-08-07.

[Int18] Intel Corporation. "Protection Keys." In *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A*, pp. 4–31. May 2018.

[ISO11] ISO/IEC. *Information technology — Programming languages — C. International standard 9899:2011.* December 2011.

[JF12] Casey M. Jeffery and Renato J.O. Figueiredo. "A Flexible Approach to Improving System Reliability with Virtual Lockstep." *IEEE Transactions on Dependable and Secure Computing*, **9**(1):2–15, January 2012.

[Jie] "pbzip2-0.9.4." `https://github.com/jieyu/concurrency-bugs/tree/master/pbzip2-0.9.4`. Accessed: 2018-12-24.

[JKJ10] Heeseung Jo, Hwanju Kim, Jae-Wan Jang, Joonwon Lee, and Seungryoul Maeng. "Transparent Fault Tolerance of Device Drivers for Virtual Machines." *IEEE Transactions on Computers*, **59**(11):1466–1479, November 2010.

[KC07] Kenichi Kourai and Shigeru Chiba. "A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines." In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 245–255, Edinburgh, UK, June 2007.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. "Debugging Operating Systems with Time-Traveling Virtual Machines." In *2005 USENIX Annual Technical Conference*, pp. 1–15, Anaheim, CA, USA, April 2005.

[KMC00] Eddie Kohler, Robert Morris, Benjie Chen, and John Jannottiand Frans M. Kaashoek. “The Click Modular Router.” *ACM Transactions on Computer Systems*, **18**(3):263–297, August 2000.

[LAK09] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. “Recovery Domains: An Organizing Principle for Recoverable Operating Systems.” In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 49–60, Washington, DC, USA, March 2009.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, **21**(7):558–565, July 1978.

[LBK90] Jean-Claude Laprie, Christian Beounes, and Karama Kanoun. “Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures.” *Computer*, **23**(7):39–51, July 1990.

[LBV15] Jacob R. Lorch, Andrew Baumann, Lisa Vlendenning, Dutch Meyer, and Andrew Warfield. “Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services.” In *12th USENIX Symposium on Networked Systems Design and Implementation*, Oakland, CA, May 2015.

[LCS10] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. “Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-races.” In *37th Annual International Symposium on Computer Architecture*, pp. 210–221, Saint-Malo, France, June 2010.

[LGT09] Michael Le, Andrew Gallagher, Yuval Tamir, and Yoshio Turner. “Maintaining Network QoS Across NIC Device Driver Failures Using Virtualization.” In *8th IEEE International Symposium on Network Computing and Applications*, pp. 195–202, Cambridge, MA, July 2009.

[LHT11] Michael Le, Israel Hsu, and Yuval Tamir. “Resilient Virtual Clusters.” In *17th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 214–223, Pasadena, CA, December 2011.

[liba] “Aerospike C Client.” `https://www.aerospike.com/apidocs/c/`. Accessed: 2020-08-07.

[libb] “libMemcached.” `https://libmemcached.org/libMemcached.html`. Accessed: 2020-08-07.

[libc] “Tarantool C client libraries.” `https://github.com/tarantool/tarantool-c`. Accessed: 2020-08-07.

[lig] “Home - Lighttpd.” `https://www.lighttpd.net/`. Accessed: 2020-08-07.

[Liu16] Yikang Liu. "High Availability of Network Service on Docker Container." In *5th International Conference on Measurement, Instrumentation and Automation*, November 2016.

[LK15] Wubin Li and Ali Kanso. "Comparing Containers versus Virtual Machines for Achieving High Availability." In *IEEE International Conference on Cloud Engineering*, pp. 353–358, Tempe, AZ, March 2015.

[LKG15] Wubin Li, Ali Kanso, and Abdelouahed Gherbi. "Leveraging Linux Containers to Achieve High Availability for Cloud Services." In *IEEE International Conference on Cloud Engineering*, pp. 76–83, March 2015.

[LLZ18] Xiaofei Liao, Minhao Lin, Long Zheng, Hai Jin, and Zhiyuan Shao. "Scalable Data Race Detection for Lock-Intensive Programs with Pending Period Representation." *IEEE Transactions on Parallel and Distributed Systems*, **29**(11):2599–2612, November 2018.

[LT11] Michael Le and Yuval Tamir. "ReHype: Enabling VM Survival Across Hypervisor Failures." In *7th ACM International Conference on Virtual Execution Environments*, pp. 63–74, Newport Beach, CA, March 2011.

[LT12] Michael Le and Yuval Tamir. "Applying Microreboot to System Software." In *IEEE International Conference on Software Security and Reliability*, pp. 11–20, Washington, D.C., June 2012.

[LT14] Michael Le and Yuval Tamir. "Resilient Virtualized Systems Using ReHype." UCLA Computer Science Department Technical Report #140019, October 2014.

[LT15] Michael Le and Yuval Tamir. "Fault Injection in Virtualized Systems – Challenges and Applications." *IEEE Transactions on Dependable and Secure Computing*, **12**(3):284–297, May 2015.

[LVN10] Oren Laadan, Nicolas Viennot, and Jason Nieh. "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems." In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, p. 155–166, New York, New York, USA, June 2010.

[lwna] "Memory Management Locking." `https://lwn.net/Articles/591978/`. Accessed: 2018-12-29.

[lwnb] "Replace mmap_sem by a range lock." `https://lwn.net/Articles/723648/`. Accessed: 2018-12-31.

[LWV10] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. "Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism." In *Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 77–90, Pittsburgh, Pennsylvania, USA, March 2010.

[LZW17] Kai Lu, Wenzhe Zhang, Xiaoping Wang, Mikel Luján, and Andy Nisbet. "Flexible Page-level Memory Access Monitoring Based on Virtualization Hardware." In *Thirteenth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 201–213, Xi'an, China, April 2017.

[MCT08] Pablo Montesinos, Luis Ceze, and Josep Torrellas. "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently." In *35th Annual International Symposium on Computer Architecture*, p. 289–300, Beijing, China, June 2008.

[MDP00] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, and Richard Wheelerand Songnian Zhou. "Process Migration." *ACM Computing Surveys*, **32**(3):241–299, September 2000.

[mem] "memcached." `https://memcached.org`. Accessed: 2020-08-07.

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." *Linux Journal*, **2014**(239), March 2014.

[MGT17] Ali Jose Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. "Towards Practical Default-On Multi-Core Record/Replay." In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 693–708, Xi'an, China, April 2017.

[MHC10] Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. "Dynamically Checking Ownership Policies in Concurrent C/C++ Programs." In *37th ACM Symposium on Principles of Programming Languages*, pp. 457–470, Madrid, Spain, January 2010.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. "LiteRace: Effective Sampling for Lightweight Data-race Detection." In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 134–143, Dublin, Ireland, June 2009.

[MSQ09] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. "SigRace: Signature-based Data Race Detection." In *36th Annual International Symposium on Computer Architecture*, pp. 337–348, Austin, TX, USA, June 2009.

[NB13] Ruslan Nikolaev and Godmar Back. “VirtuOS: An Operating System with Kernel Virtualization.” In *24th ACM Symposium on Operating Systems Principles*, pp. 116–132, Farmington, PA, November 2013.

[NC99] Wee Teck Ng and Peter M. Chen. “The Systematic Improvement of Fault Tolerance in the Rio File Cache.” In *29th Annual International Symposium on Fault-Tolerant Computing*, pp. 76–83, Madison, WI, June 1999.

[NM92] Robert H.B. Netzer and Barton P. Miller. “What Are Race Conditions?: Some Issues and Formalizations.” *ACM Letters on Programming Languages and Systems*, **1**(1):74–88, March 1992.

[nod] “Pokedex Messenger Bot for Pokemon GO.” `https://github.com/zwacky/pokedex-go`. Accessed: 2019-10-02.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. “BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging.” In *32nd Annual International Symposium on Computer Architecture*, p. 284–295, Madison, Wisconsin, USA, May 2005.

[nul] “Nullhttpd.” `http://nulllogic.ca/httpd/`. Accessed: 2018-12-31.

[OAA09] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. “Kendo: Efficient Deterministic Multithreading in Software.” In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 97–108, Washington, DC, USA, March 2009.

[OJF17] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. “Engineering Record and Replay for Deployability.” In *2017 USENIX Conference on Usenix Annual Technical Conference*, p. 377–389, Santa Clara, CA, USA, July 2017.

[OSS02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. “The Design and Implementation of Zap: A System for Migrating Computing Environments.” In *5th Operating Systems Design and Implementation*, Boston, MA, USA, December 2002.

[OZK12] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. “Aikido: Accelerating Shared Data Dynamic Analyses.” In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 173–184, London, England, UK, March 2012.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. “LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection.” In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 320–331, Ottawa, Ontario, Canada, June 2006.

[Prv06] Milos Prvulovic. "CORD: cost-effective (and nearly overhead-free) order-recording and data race detection." In *The Twelfth International Symposium on High-Performance Computer Architecture*, pp. 232–243, Austin, TX, USA, February 2006.

[PS03] Eli Pozniansky and Assaf Schuster. "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs." In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 179–190, San Diego, California, USA, June 2003.

[PS07] Eli Pozniansky and Assaf Schuster. "MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs." *Concurrency and Computation: Practice and Experience - Parallel and Distributed Systems: Testing and Debugging*, **19**(3):327–340, March 2007.

[PT03] Milos Prvulovic and Josep Torrellas. "ReEnact: Using Thread-level Speculation Mechanisms to Debug Data Races in Multithreaded Codes." In *30th Annual International Symposium on Computer Architecture*, pp. 110–121, San Diego, California, June 2003.

[PZX09] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors." In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, p. 177–192, Big Sky, Montana, USA, October 2009.

[qem] "Features/MicroCheckpointing." `https://wiki.qemu.org/Features/MicroCheckpointing`. Accessed: 2019-10-02.

[QON12] Shanxiang Qi, Norimasa Otsuki, Lois Orosa Nogueira, Abdullah Muzahid, and Josep Torrellas. "Pacman: Tolerating asymmetric data races with unintrusive hardware." In *IEEE International Symposium on High-Performance Computer Architecture*, New Orleans, LA, USA, February 2012.

[RB99] Michiel Ronsse and Koen De Bosschere. "RecPlay: A Fully Integrated Practical Record/Replay System." *ACM Transactions on Computer Systems*, **17**(2):133–152, May 1999.

[red] "Redis." `https://redis.io`. Accessed: 2020-08-07.

[rem] "Remus - Xen." `https://wiki.xenproject.org/wiki/Remus`. Accessed: 2019-10-02.

[RG05] Mendel Rosenblum and Tal Garfinkel. "Virtual Machine Monitors: Current Technology and Future Trends." *IEEE Computer*, **38**(5):39–47, May 2005.

[RHK06] Hans P. Reiser, Franz J. Hauck, Rudiger Kapitza, and Wolfgang Schroder-Preikschat. "Hypervisor-Based Redundant Execution on a Single Physical Host." In *6th European Dependable Computing Conference, Supplemental Volume*, pp. 67–68, Coimbra, Portugal, October 2006.

[RR81] Richard F. Rashid and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel." In *8th ACM Symposium on Operating Systems Principles*, pp. 64–75, December 1981.

[RRR09] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. "ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs." In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 181–192, Washington, DC, March 2009.

[RTL16] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. "Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions." In *2016 USENIX Conference on Usenix Annual Technical Conference*, p. 551–564, Denver, CO, USA, June 2016.

[run] "opencontainers/runc." `https://github.com/opencontainers/runc`. Accessed: 2020-08-07.

[RZP19] Shiru Ren, Yunqi Zhang, Lichen Pan, and Zhen Xiao. "Phantasy: Low-Latency Virtualization-based Fault Tolerance via Asynchronous Prefetching." *IEEE Transactions on Computers*, **68**(2):225–238, February 2019.

[Sai05] Yasushi Saito. "Jockey: A User-Space Library for Record-Replay Debugging." In *Sixth International Symposium on Automated Analysis-Driven Debugging*, p. 69–76, Monterey, California, USA, September 2005.

[SBN97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs." In *Sixteenth ACM Symposium on Operating Systems Principles*, pp. 27–37, Saint Malo, France, October 1997.

[Sch84] Fred B. Schneider. "Byzantine Generals in Action: Implementing Fail-Stop Processors." *ACM Transactions on Computer Systems*, **2**(2):145–154, May 1984.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer – data race detection in practice." In *Workshop on Binary Instrumentation and Applications*, pp. 62–71, New York, NY, December 2009.

[sie] "Siege Home." `https://www.joedog.org/siege-home/`. Accessed: 2020-08-07.

[SKA04] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. “Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging.” In *2004 USENIX Annual Technical Conference*, pp. 29–44, Boston, MA, USA, June 2004.

[sof] “SOFT-DIRTY PTEs.” `https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt`. Accessed: 2020-08-07.

[ssd] “SSDB - A fast NoSQL database, an alternative to Redis.” `https://github.com/ideawu/ssdb`. Accessed: 2020-08-07.

[SVE11] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. “RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications.” In *the 33rd International Conference on Software Engineering*, pp. 401–410, Waikiki, Honolulu, HI, USA, May 2011.

[SY14] Caitlin Sadowski and Jaeheon Yi. “How Developers Use Data Race Detection Tools.” In *Fifth Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 43–51, Portland, Oregon, USA, October 2014.

[tar] “Tarantool - In-memory DataBase.” `https://tarantool.io`. Accessed: 2020-08-07.

[tas] “Linux-task-diag.” `https://github.com/avagin/linux-task-diag`. Accessed: 2019-10-02.

[TCP] “TCP connection repair.” `https://lwn.net/Articles/495304/`. Accessed: 2019-09-27.

[Tho18] Mikkel Thorup. “High Speed Hashing for Integers and Strings.” *Computing Research Repository*, **arXiv:1504.06804v4 [cs.DS]**, April 2018.

[thr] “Google ThreadSanitizer.” `https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual`. Accessed: 2019-01-01.

[TXC12] Cheng Tan, Yubin Xia, Haibo Chen, and Binyu Zang. “TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization.” In *2nd International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*, Boston, MA, June 2012.

[Uni] “UnixBench.” `https://github.com/kdlucas/byte-unixbench`. Accessed: 2017-10-12.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. “RELAY: Static Race Detection on Millions of Lines of Code.” In *Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 205–214, Dubrovnik, Croatia, September 2007.

[VLW11] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. "DoublePlay: Parallelizing Sequential Logging and Replay." In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 15–26, Newport Beach, California, USA, March 2011.

[VMw] VMware. "Providing Fault Tolerance for Virtual Machines." `https://pubs.vmware.com/vsphere-4-esx-vcenter/topic/com.vmware.vsphere.availability.doc_41/c_ft.html`. Accessed: 2017-12-01.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. "Mondrian Memory Protection." In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 304–316, San Jose, California, USA, October 2002.

[WCJ18] Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui. "PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance." In *15th USENIX Symposium on Networked Systems Design and Implementation*, pp. 483–499, Renton, WA, April 2018.

[wge] "GNU Wget." `https://www.gnu.org/software/wget/`. Accessed: 2020-08-07.

[XBH03] Min Xu, Rastislav Bodik, and Mark D. Hill. "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay." In *30th Annual International Symposium on Computer Architecture*, p. 122–135, San Diego, California, USA, May 2003.

[YN09] Jie Yu and Satish Narayanasamy. "A Case for an Interleaving Constrained Shared-memory Multi-processor." In *36th Annual International Symposium on Computer Architecture*, pp. 325–336, Austin, TX, USA, June 2009.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking." In *Twentieth ACM Symposium on Operating Systems Principles*, pp. 221–234, Brighton, UK, October 2005.

[YYK11a] Kazuya Yamakita, Hiroshi Yamada, and Kenji Kono. "Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery." In *41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 169–180, Hong Kong, China, June 2011.

[YYK11b] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. "Can Linux be Rejuvenated without Reboots?" In *IEEE Third International Workshop on Software Aging and Rejuvenation*, pp. 50–55, Hiroshima, Japan, November 2011.

[YYK12] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. "Is Linux Kernel Oops Useful or Not?" In *Eighth USENIX Workshop on Hot Topics in System Dependability*, pp. 1–6, Hollywood, CA, October 2012.

[ZJL17] Tong Zhang, Changhee Jung, and Dongyoon Lee. "ProRace: Practical Data Race Detection for Production Use." In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 149–162, Xi'an, China, April 2017.

[ZKD08] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Hardware Enforcement of Application Security Policies Using Tagged Memory." In *Eighth USENIX Conference on Operating Systems Design and Implementation*, pp. 225–240, San Diego, California, USA, December 2008.

[ZLJ16] Tong Zhang, Dongyoon Lee, and Changhee Jung. "TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory." In *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 159–173, Atlanta, Georgia, USA, April 2016.

[ZMA09] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. "Practical and Low-Overhead Masking of Failures of TCP-Based Servers." *ACM Transactions on Computer Systems*, **27**(2):4:1–4:39, May 2009.

[ZTZ07] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. "HARD: Hardware-Assisted Lockset-based Race Detection." In *IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 121–132, Scottsdale, AZ, USA, February 2007.

[ZZ03] Wensong Zhang and Wenzhuo Zhang. "Linux Virtual Server Clusters." *Linux Magazine*, **5**(11), November 2003.