34th IEEE International Parallel and Distributed Processing Symposium
New Orleans, LA (May 2020)

Fault-Tolerant Containers Using NiLiCon

Diyu Zhou
Computer Science Department, UCLA
zhoudiyu@cs.ucla.edu

Yuval Tamir
Computer Science Department, UCLA
tamir@cs.ucla.edu

***Abstract*—Many services deployed in the cloud require high reliability and must thus survive machine failures. Providing such fault tolerance transparently, without requiring application modifications, has motivated extensive research on replicating virtual machines (VMs). Cloud computing typically relies on VMs or containers to provide an isolation and multitenancy layer. Containers have advantages over VMs in smaller size, faster startup, and avoiding the need to manage updates of multiple VMs. This paper reports on the design, implementation, and evaluation of *NiLiCon* — a transparent container replication mechanism for fault tolerance. To the best of our knowledge, *NiLiCon* is the first implementation of container replication, demonstrating that it can be used for transparent deployment of critical services in the cloud.**

***NiLiCon* is based on high-frequency asynchronous incremental checkpointing to a warm spare, as previously used for VMs. The challenge to accomplishing this is that, compared to VMs, there is much tighter coupling between the container state and the state of the underlying platform. *NiLiCon* meets this challenge, eliminating the need to deploy services in VMs, with performance overheads that are competitive with those of similar VM replication mechanisms. Specifically, with the seven benchmarks used in the evaluation, the performance overhead of *NiLiCon* is in the range of 19%-67%. For fail-stop faults, the recovery rate is 100%.**

***Keywords*-fault tolerance; replication;**

I. Introduction

Servers commonly host applications in virtual machines (VMs) and/or containers to facilitate efficient, flexible resource management [19], [30], [35]. 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 [19], [27].

Many of the applications and services hosted by datacenters require high availability and/or high reliability. Meeting these needs can be left to the application developers who can develop fully customized solutions or adapt their applications to be compatible with more general-purpose middleware. However, there are obvious benefits to avoiding this extra burden on developers and support legacy software without requiring extensive modifications. This has motivated the development of application-transparent mechanisms for high reliability.

The above considerations have led to the development of a plethora of techniques and products for tolerating VM failures using replication [21], [23], [24], [29], [36], [37]. Most of these techniques involve a primary VM and a "warm backup" (standby spare) VM [23], [36]. The applications run in the primary, which is periodically paused so that its state can be checkpointed to the backup. If the primary fails, the applications or services are started on the backup from the checkpointed state. In order for this failover to be transparent to the environment outside the VM (e.g., the service clients), these fault tolerance mechanisms ensure that the backup resumes from a state that is consistent with the final primary state visible to the clients.

Despite the advantages of containers, there has been very little work on high availability and fault tolerance techniques for containers [26]. In particular, there has been limited work on high availability techniques [27], [28]. 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* (Nine Lives Containers), as described in this paper, is such a mechanism.

The VM-level fault tolerance techniques discussed above do support stateful applications and provide application transparency as well as client transparency. Hence, *NiLiCon* uses the same basic approach [23]. 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) [3], 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 un-

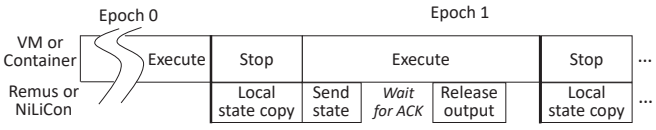

Figure 1. Workflow of Remus and *NiLiCon* on the primary host.

modified 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 faults, 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.

Section II presents Remus [23] and CRIU [3], which are the bases for *NiLiCon*. The key differences between *NiLiCon* and Remus are explained in §III. The design and operation of *NiLiCon* are discussed in §IV. Key implementation optimizations are presented in §V. The experimental setup and evaluation results are presented in §VI and §VII, respectively. Related work is discussed in §VIII.

II. Background

NiLiCon is built on Remus [23] and CRIU [3]. Algorithmically, *NiLiCon* operates on containers in the same way that Remus operates on VMs, as described in §II-A. A key part of this technique is the mechanism used to periodically checkpoint the primary state to the backup. CRIU, as described in §II-B, is the starting point for *NiLiCon*'s checkpointing mechanism.

A. Remus: Passive VM Replication

With Remus [23], 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 fails. As shown in Figure 1, processing on the primary 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 1). 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.

B. CRIU: Container Live Migration Mechanism

CRIU (Checkpoint/Restore In Userspace) [3] is a tool that can checkpoint and restore complicated real-world container state on Linux. CRIU can be used to perform live migration of a container. This involves obtaining the container state (checkpointing) on one host and restoring it on another. This requires migrating the user-level memory and register state of the container. 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 migrated. 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. Threads executing user-level code pause immediately. For threads executing system calls, the virtual signal forces

a return from the system calls, as if they are interrupted by a normal signal. Some container state, such as TCP socket state, can continue to be changed by the underlying kernel and receive special handling (§III).

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 migrate established TCP connections using a socket *repair mode* supported by the Linux kernel [16]. 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 [14]. 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.

III. *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 (§II-B).

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.

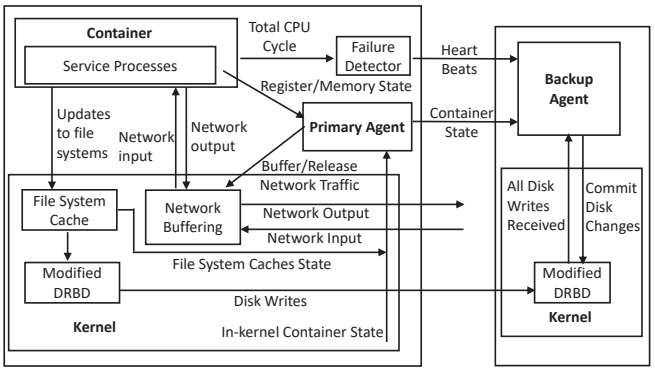

Figure 2. The architecture of *NiLiCon*.

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.

IV. *NiLiCon* Basics

This section presents the basic design and operation of *NiLiCon*, without the optimizations described in §V. The overall architecture of *NiLiCon* is shown in Figure 2. 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 initiate the recovery. As with most VM replication works [21], [23], [24], [29], [37], we assume a fail-stop fault 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 1). With the implementation of *NiLiCon* used in this paper, 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 (§II-B) 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) [12]. 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 [6] (a distributed software RAID implementation), as described in §II-A. 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 (§II-A). File system updates are tracked using the DNC state for page and inode cache entries (§III). 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 (§II-B) 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 (§II-B), 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 (§III). After the container has been restored, the backup agent reconnects the container network namespace to the bridge, thus reestablishing the network connection.

V. Optimizations

The basic implementation of *NiLiCon* presented in §IV 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 [7]. 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 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

Table I
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%

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 I lists *NiLiCon*'s optimizations, described in the rest of this section, and their impact on the overhead for the *streamcluster* benchmark (§VI). 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 (§III). 1093 LOC are in two kernel modules: changes to DRBD (§IV) and a module for tracking some in-kernel state changes (§V-B). In addition, we applied an 1140 LOC kernel patch from the CRIU developers [4] that speeds up obtaining VMA information from the kernel (§V-D).

A. 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 (§II-B), 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.

B. 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.

C. Optimizing Blocking Network Input

As explained earlier (§III), 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 (§IV). 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.

D. 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 [7] 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 (§ II-A)).

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.

E. 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 (§II-B), *NiLiCon* sets the retransmission timeout to be the minimum value: 200ms.

VI. 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 [9], 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 [5]. VMs were fully virtual with paravirtual drivers. Each VM or container was hosted on a dedicate core and allocated 4GB physical memory.

The evaluation was based on five server benchmarks: *Redis* [11], *SSDB* [15], *Node* [10], *Lighttpd* [1], and *DJCMS* [2]; as well as two non-interactive CPU/memory-intensive PARSEC [20] benchmarks: *streamcluster* and *swaptions*. Unless otherwise mentioned below, all benchmarks were configured with the default characteristics.

Redis and *SSDB* are NoSQL databases. *Redis* was configured to stress memory by storing all data in memory (persistence: None). *SSDB* was configured to stress disk I/O by using full persistence. As in [37], each request to *Redis*/*SSDB* was a batch of 1K requests consisting of 50% reads and 50% writes. YCSB [22] generated 2M requests with 100K 1KB records. A custom client with the *hiredis* library [8] batched and sent these requests to *Redis*/*SSDB*.

DJCMS, *Lighttpd*, and *Node* are web-based servers. The SIEGE [13] client was used to send to each of them concurrent requests. *DJCMS* is a content management system platform that uses Nginx, Python, and MySQL. It was evaluated with requests on the administrator dashboard page. The *Lighttpd* web server was evaluated with requests to a PHP script that watermarks an image. *Node*, written in Node.js, searches through a database for a keyword and generates a response consisting of text and figures. While the original *Node* benchmark sends a response as a Facebook chat message, we modified it to reply with a static web page.

VII. Evaluation

This section presents the validation (§VII-A), recovery latency (§VII-B), and the performance overhead (§VII-C) of *NiLiCon*.

A. 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 §VI. 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

Table II
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

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 fault 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.

B. 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* (§IV), 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.

Each experiment is executed ten times and the variations in the measured service interruption latencies are within 10% of the mean. Table II 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 (§V-E) 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*.

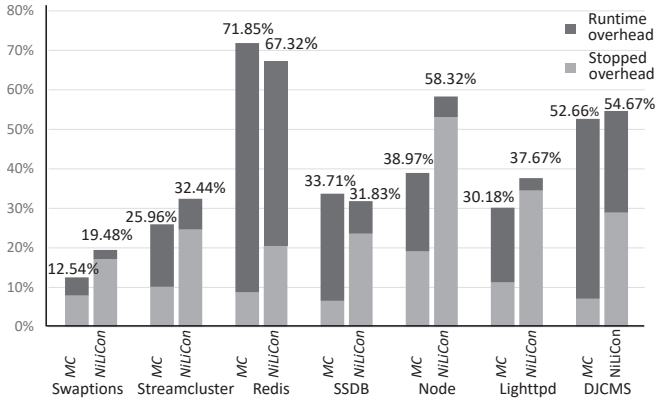

Figure 3. Performance overhead comparison between MC and *NiLiCon* with breakdown of sources of overhead.

C. 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 [5]. 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 (§IV) 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 [20] 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 execution time. For server applications, the performance overhead is the relative reduction in maximum throughput.

Figure 3 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 stop 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.

Table III
AVERAGE STOP TIME & #DIRTY PAGES PER EPOCH, MC AND *NiLiCon*.

		SwapT	StreamC	Redis	SSDB	Node	Lhttpd	DJCMS
Stop	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

Table IV
STOP TIME AND TRANSFERRED STATE SIZE FOR *NiLiCon*.

		SwapT	StreamC	Redis	SSDB	Node	Lhttpd	DJCMS
Stop	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

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 tracking modified pages.

Table III presents the average stop time and number of dirty pages per epoch. For *NiLiCon*, the stop time is higher since *NiLiCon* needs to obtain container in-kernel state using the slow kernel interface (§V). This is the main reason for *NiLiCon*'s higher overhead for most of the benchmarks. For example, the *Node* benchmark has the highest stop 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.

Stop Time and Transferred State Variations. The stop 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 IV 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 stop time for *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 (§VIII) is lower utilization of the CPU on the backup host. Table V 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.

Table V
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

Table VI
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

As shown in Table V, 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.

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 (§II-A). Table VI 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, *streamcluser* 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.

VIII. Related Work

NiLiCon is closely related to works on VM replication and process checkpointing/migration. The rest of the section discusses works in these two areas.

VM replication. Bressoud and Schneider introduced active VM replication [21]. A primary VM and backup VM execute on different hosts in lock-step at epoch granularity. The pair of hosts must follow a deterministic execution path, requiring overhead for coordination that would grow to prohibitive levels for multiprocessor systems [23].

Remus [23] introduced passive VM replication based on high frequency incremental checkpointing to a warm backup (§II-A). Many of the follow-on works focused on reducing the performance overhead. For example, Phantasy [34] reduces the overhead during normal execution and latency of state transfer using the hardware features of, respectively, Intel's page-modification logging (PML) and RDMA.

COLO [24] reduces the performance overhead by deploying active replication. Inputs to the primary VM are also transferred to the backup VM. Outputs from the primary and the backup 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 and *NiLiCon*. A key downside of COLO is that, for largely non-deterministic workloads, mismatches are frequent, resulting in prohibitive overhead.

PLOVER [37] uses active replication coupled with the Remus [23] mechanism. Specifically, as with Remus, execution is partitioned into epochs and outgoing packets are buffered and released only after the backup is confirmed to have the corresponding state. The purpose of the active replica is to reduce the difference between the states of the primary and backup states and thus reduce the state that needs to be transferred per epoch. As with all active replication schemes, the resource overheads (CPU cycles and memory) of COLO [24] and PLOVER is more than 100%.

Tardigrade [29] also uses the Remus [23] 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 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. *NiLiCon* does not have these disadvantages.

Process checkpoint/migration. There are a wide variety of works on process checkpointing/migration [17], [18], [25], [31], [33]. However, with process replication it is difficult to avoid potential resource naming conflicts upon failover as well as to identify and isolate all the necessary state components that need to be replicated. Zap [32], addresses the limitations of process migration by allocating separate namespaces for groups of processes. This approach, in its essence, is the same as the namespace implementation for Linux containers.

IX. Conclusion

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.

REFERENCES

[1] "Adding watermarks to images using alpha channels," https://www.php.net/manual/en/image.examples-watermark.php, accessed: 2019-10-02.

[2] "An improved django-admin-tools dashboard for Django projects," https://github.com/django-fluent/django-fluent-dashboard, accessed: 2019-10-02.

[3] "CRIU: Checkpoint/Restore In Userspace," https://criu.org/Main_Page, accessed: 2019-09-25.

[4] "CRIU Task-diag," https://criu.org/Task-diag, accessed: 2019-10-02.

[5] "Features/MicroCheckpointing," https://wiki.qemu.org/Features/MicroCheckpointing, accessed: 2019-10-02.

[6] "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.

[7] "Linux-task-diag," https://github.com/avagin/linux-task-diag, accessed: 2019-10-02.

[8] "Minimalistic C client for Redis," https://github.com/redis/hiredis, accessed: 2019-10-02.

[9] "opencontainers/runc," https://github.com/opencontainers/runc, accessed: 2019-10-02.

[10] "Pokedex Messenger Bot for Pokemon GO," https://github.com/zwacky/pokedex-go, accessed: 2019-10-02.

[11] "Redis," https://redis.io, accessed: 2019-10-02.

[12] "Remus - Xen," https://wiki.xenproject.org/wiki/Remus, accessed: 2019-10-02.

[13] "Siege Home," https://wiki.qemu.org/Features/MicroCheckpointin://www.joedog.org/siege-home/, accessed: 2019-10-02.

[14] "SOFT-DIRTY PTEs," https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt, accessed: 2019-09-27.

[15] "SSDB - A fast NoSQL database, an alternative to Redis," https://github.com/ideawu/ssdb, accessed: 2019-10-02.

[16] "TCP connection repair," https://lwn.net/Articles/495304/, accessed: 2019-09-27.

[17] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *2009 IEEE International Symposium on Parallel and Distributed Processing*, Rome, Italy, May 2009.

[18] A. Barak and R. Wheeler, "MOSIX: An Integrated Multiprocessor UNIX," in *USENIX Winter 1989 Technical Conference*, San Diego, CA, USA, Feb. 1989.

[19] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[20] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[21] T. C. Bressoud and F. B. Schneider, "Hypervisor-based Fault Tolerance," in *15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, Dec. 1995.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *1st ACM Symposium on Cloud Computing*, Jun. 2010, pp. 143–154.

[23] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *5th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2008, pp. 161–174.

[24] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service," in *4th ACM Annual Symposium on Cloud Computing*, Santa Clara, CA, Oct. 2013.

[25] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software – Practice and Experience*, vol. 21, no. 8, pp. 757–785, Aug. 1991.

[26] W. Li and A. Kanso, "Comparing Containers versus Virtual Machines for Achieving High Availability," in *IEEE International Conference on Cloud Engineering*, Tempe, AZ, Mar. 2015, pp. 353–358.

[27] W. Li, A. Kanso, and A. Gherbi, "Leveraging Linux Containers to Achieve High Availability for Cloud Services," in *IEEE International Conference on Cloud Engineering*, Mar. 2015, pp. 76–83.

[28] Y. Liu, "High Availability of Network Service on Docker Container," in *5th International Conference on Measurement, Instrumentation and Automation*, Nov. 2016.

[29] J. R. Lorch, A. Baumann, L. Vlendenning, D. Meyer, and A. 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.

[30] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[31] D. S. Milojicic, F. Douglis, Y. Paindaveine, and R. W. S. Zhou, "Process Migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, Sep. 2000.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *5th Operating Systems Design and Implementation*, Boston, MA, USA, Dec. 2002.

[33] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," in *8th ACM Symposium on Operating Systems Principles*, Dec. 1981, pp. 64–75.

[34] S. Ren, Y. Zhang, L. Pan, and Z. Xiao, "Phantasy: Low-Latency Virtualization-based Fault Tolerance via Asynchronous Prefetching," *IEEE Transactions on Computers*, vol. 68, no. 2, pp. 225–238, Feb. 2019.

[35] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[36] 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.

[37] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui, "PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance," in *15th USENIX Symposium on Networked Systems Design and Implementation*, Renton, WA, Apr. 2018, pp. 483–499.