USENIX Annual Technical Conference
Carlsbad, CA (July 2022).

RRC: Responsive Replicated Containers

Diyu Zhou
UCLA and EPFL

Yuval Tamir
UCLA

Abstract

Replication is the basic mechanism for providing application-transparent reliability through fault tolerance. The design and implementation of replication mechanisms is particularly challenging for general multithreaded services, where high latency overhead is not acceptable. Most of the existing replication mechanisms fail to meet this challenge.

RRC is a fully-operational fault tolerance mechanism for multiprocessor workloads, based on container replication. It minimizes the latency overhead during normal operation by addressing two key sources of this overhead: (1) it decouples the latency overhead from checkpointing frequency using a hybrid of checkpointing and replay, and (2) it minimizes the pause time for checkpointing by forking a clone of the container to be checkpointed, thus allowing execution to proceed in parallel with checkpointing. The fact that *RRC* is based on checkpointing makes it inherently less vulnerable to data races than active replication. In addition, *RRC* includes mechanisms that further reduce the vulnerability to data races, resulting in high recovery rates, as long as the rate of manifested data races is low. The evaluation includes measurement of the recovery rate and recovery latency based on thousands of fault injections. On average, *RRC* delays responses to clients by less than 400μs and recovers in less than 1s. The average pause latency is less than 3.3ms. For a set of eight real-world benchmarks, if data races are eliminated, the performance overhead of *RRC* is under 48%.

1 Introduction

For many applications hosted in data centers, high reliability is a key requirement, demanding fault tolerance. The key desirable properties of a fault tolerance mechanism, especially for server applications, are: A) Low throughput and latency overheads; B) Support for multithreaded applications; and C) Application transparency. Replication, has long been used to implement application-transparent fault tolerance, especially for server applications.

The two main approaches to replication, specifically, duplication, are: (1) high-frequency checkpointing of the primary replica state to a passive backup [33], and (2) active replication, where the primary and backup both execute the application [38, 47]. A key disadvantage of the first approach is that, for consistency between server applications and their clients after failover, outputs must be delayed before being released to the client, typically for tens of milliseconds. Such delays are unacceptable for many server applications.

To support active replication of multiprocessor workloads, where there are many sources of nondeterminism, active replication is implemented using a leader-follower algorithm. With this algorithm, the outcomes of identified nondeterministic events on the primary, namely synchronization operations and certain system calls, are recorded and sent to the backup. This allows the backup to deterministically replay their outcomes [38, 47]. A disadvantage of this approach is that it is vulnerable to even rare replay failures due to untracked nondeterministic events, such as those caused by data races. Another disadvantage is that, for application with a high rate of synchronization operations, the replay on the backup may be significantly slower than the execution on the primary, resulting in high throughput overhead [38]. This is due to the interaction between thread scheduling by the OS and the need to mirror on the backup the execution on the primary.

This paper presents a fault tolerance scheme, based on container replication, called *RRC* (Responsive Replicated Containers). *RRC* targets server applications and is thus optimized to minimize response latency overhead. *RRC* overcomes the disadvantages of existing approaches using a combination of periodic checkpointing [33, 62] and externally-deterministic replay [29]. The primary sends periodic checkpoints to the passive backup. While executing, the primary logs to the backup the outcomes of nondeterministic events. Upon failure, the backup restores the latest checkpoint and deterministically replays the execution up to the last external output. Hence, external outputs only need to be delayed by the short amount of time it takes to send and commit the relevant portion of the nondeterministic event log to the backup.

RRC minimizes request-reply latency overhead, not only for the average case, but also the tail latency overhead. To that end, *RRC* had to overcome a key challenge, namely, that while the state of the primary is collected for transmission to the backup, execution has to be paused. Even with various optimizations, this latency is often tens of milliseconds, largely due to the cost of retrieving in-kernel state associated with the container, such as the state of open file descriptors [62]. To meet this challenge, *RRC* introduces a new kernel primitive: container fork. For checkpointing, *RRC* pauses the primary container, forks a shadow container, and resumes execution. This results in pause times of less than 3.5ms. The checkpoint is obtained from the shadow container.

RRC decouples the response latency from the checkpointing duration. This enables high performance by allowing the tuning of epoch duration to trade off performance and resource overheads with recovery latency and vulnerability to untracked nondeterministic events. The latter is important since applications may contain data races (§3, §6.3). *RRC* is focused on dealing with data races that rarely manifest and are thus more likely to remain undetected (§3). Since *RRC* only requires replay during recovery and for the short interval since the last checkpoint, it is inherently more resilient to data races than active replication schemes that rely on replay of the entire execution [38]. Furthermore, *RRC* includes timing adjustment mechanisms that result in a high recovery rate even for applications that include data races, as long as their rate of unsynchronized writes is low (§4.7).

RRC also decouples the performance on the primary from the backup. Thus, unlike active replication schemes [38], the backup is not a performance bottleneck (§6.1).

Replication can be at the level of VMs [27,33,35,53,58], processes [32, 38, 47], or containers [62]. Containers have advantages over VMs due to smaller memory and storage footprints, faster startup, and avoiding the need to manage updates of multiple VMs [25,45]. Furthermore, containers are the best fit for mechanisms such as *RRC*. Applying *RRC*'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 (e.g., process IDs) upon failover. While such name conflicts can be solved, the existing container mechanism already solves them efficiently.

The implementation of *RRC* involved developing solutions to implementation challenges that have not been addressed by prior works. The most important of these is dealing with the integration of timer-triggered checkpointing, that is not synchronized with the application, and user-level recording of nondeterministic events (§4.2). *RRC* also efficiently handles the failover of TCP connections through checkpoint restoration, a replay phase, and finally resumption of live execution (§4.3, §4.4). *RRC* is application-transparent and does not require any changes to the application code.

We have implemented a prototype of *RRC* and evaluated its performance and reliability. With 1s epochs, *RRC*'s throughput and average latency overheads were less than 49% and 230μs, respectively, for all eight benchmarks. With 100ms epochs, the corresponding overheads were less than 53% and 291μs for seven benchmarks, 86% and 264μs for the eighth. *RRC* is designed to recover from fail-stop faults. We used thousands of fault injections to validate and evaluate *RRC*'s recovery mechanism. For all eight benchmarks, after data races identified by ThreadSanitizer [6] were resolved, *RRC*'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 *RRC*'s timing adjustments, the recovery rate was over 99.1%.

RRC achieves both low response latency overhead and resilience to infrequently-manifested data races. This combination provides a fundamental advance over both Remus-based techniques [33] and active replication [38, 47], respectively. Specifically, we make the following contributions: 1) a fault tolerance scheme based on container replication, using a unique combination of periodic checkpointing, deterministic replay, and an optimized scheme for failover of network connections; 2) a new system call, container fork, used to minimize tail latency overhead; 3) a replication mechanism with inherent resilience to untracked nondeterministic events, further enhanced by mechanisms that increase recovery success rate in the presence of data races; 4) a thorough evaluation of *RRC* with respect to performance overhead, resource overhead, and recovery rate, demonstrating the lowest reported external output delay compared to competitive mechanisms.

Section 2 presents two key building blocks for *RRC*: NiLiCon [62] and deterministic replay [21, 29, 44, 50, 57]. An overview of *RRC* is presented in §3. *RRC*'s implementation is described in §4, with a focus on key challenges. The experimental setup and evaluation are presented in §5, and §6, respectively. Limitation of *RRC* and of our prototype implementation are described in §7. §8 provides a brief overview of related work.

2 Background

RRC integrates container replication based on periodic checkpointing [33, 62], described in §2.1, and deterministic replay of multithreaded applications, described in §2.2.

2.1 NiLiCon

Remus [33] introduced a practical application-transparent fault tolerance scheme based on VM replication using high-frequency checkpointing. NiLiCon [62] is an implementation of the Remus mechanism for containers. A key challenge faced by NiLiCon 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, based on a tool called CRIU (Checkpoint/Restore in User Space) [4], with novel optimizations that significantly reduce overhead. CRIU checkpoints and restores the user-level and kernel-level state of a container, except for disk state. NiLiCon handles disk state by adding system calls to checkpoint and restore the page cache and a modified version of the DRBD module [8]. NiLiCon relies on CRIU to preserve established TCP connections across failover, using a special repair mode of the socket provided by the Linux kernel [18].

2.2 Deterministic Replay on Multiprocessors

Deterministic replay is the reproduction of some original execution in a subsequent execution. During the original execution, the results of nondeterministic events/actions are recorded in a log. This log is used in the subsequent execution [29]. With a uniprocessor, nondeterministic events include: asynchronous events, such as interrupts; system calls, such as *gettimeofday()*; and inputs from the external world.

With shared-memory multiprocessors, there is a higher frequency of nondeterministic events related to the order of

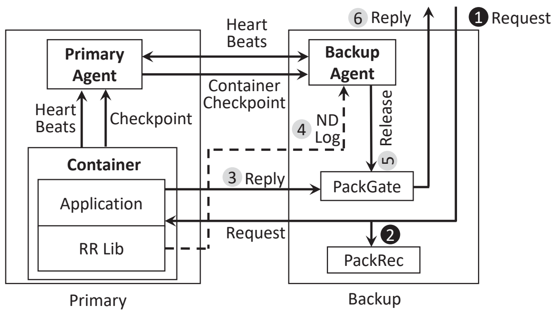

Figure 1: Architecture and workflow of *RRC*.

accesses to the same memory location by different processors. For such systems, a common approach is to support deterministic replay only for programs that are data-race-free [49]. For such programs, as long as the results of synchronization operations are deterministically replayed, the ordering of shared memory accesses are preserved. The recording of nondeterministic events can occur at different levels: hardware [40,59], hypervisor [36,37,41], OS [39,43], or library [49,54]. Without dedicated hardware support, it is advantageous to record the events at the user level, thus avoiding the overhead for entering the kernel or hypervisor [44].

To support seamless failover with replication, it is sufficient to provide *externally deterministic replay* [44]. 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.

3 Overview of *RRC*

RRC provides fault tolerance by maintaining a primary-backup pair with an inactive backup that takes over when the primary fails. Execution on the primary is divided into epochs and the primary state is checkpointed to an inactive backup at the end of each epoch [33,62]. 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, *RRC* 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.

Figure 1 shows the overall architecture of *RRC*. 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 §4.1.

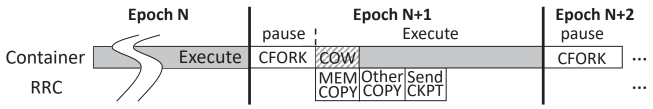

Figure 2: Timeline of an epoch on the primary replica.

Figure 1 shows the processing of requests and replies for server applications. (1) Client requests are sent to the backup. (2) To support TCP failover, the backup records incoming packets and forwards them to the primary. (3) Replies from the primary are forwarded to the backup and blocked by the *PackGate* queueing discipline kernel module. (4) The primary sends the nondeterministic event log to the backup. (5) Upon receiving the log, *PackGate* releases the corresponding replies.

Figure 2 shows a timeline of each epoch on the primary replica. First, the container is paused and a container fork is performed. Execution is then resumed. The first write to a page results in a Copy On Write (COW) so that the state of the forked shadow container is maintained. Concurrently, the pages modified since the last checkpoint are copied to a staging buffer (*MEM COPY*). Once this copy is completed, the original container ceases to perform the COW operations. A container checkpoint includes in-kernel state associated with the container, such as the state of open file descriptors [62]. This state is obtained from the shadow container and written to the staging buffer (*Other COPY*). The entire checkpoint is then sent from the primary to the backup.

RRC is based on having the ability to identify all sources of nondeterminism that are potentially externally visible, record their outcomes, and replay them when needed. Thus, 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. This implies that applications are expected to be free of data races. However, not all multithreaded programs meet this expectation. Furthermore, precise race detection is NP-hard [48]. Hence, it is not possible to ensure that all data races are detected and eliminated. Fortunately, frequently-manifested data races are detectable using tools such as ThreadSanitizer [6]. Hence, only rarely-manifested data races are likely to remain in applications.

Since *RRC* 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 [38]. As an addition to this inherent advantage of *RRC*, *RRC* includes optional mechanisms that significantly increase the probability of correct recovery despite data races, as long as the manifestation rate is low (§4.7). During execution on the primary, these mechanisms record the order and timing of returns from nondeterministic system calls by *all* the threads. During replay, the recorded order and relative timing are enforced.

If the primary fails, network connections must be maintained and migrated to the backup [19, 20, 22, 60]. Like CoRAL [19, 20], requests are routed through backup by advertising the service IP address in the backup. Unlike FT-TCP [22, 60] or CoRAL, replies are also routed through the backup, resulting in lower latency (§4.3).

As with most other state replication work [33, 53, 58, 62], *RRC* assumes fail-stop faults. Either the primary or the backup may fail. Heartbeats are exchanged between the primary and backup so failures are detected as missing heartbeats. Thus, *RRC* relies in the synchrony assumption [34] with respect to both the hosts and the network. If the backup fails, the primary configures its network, advertises the service IP address, and communicates with the clients directly. To maintain redundancy, a new backup needs to be instantiated and take over the service IP address.

4 Implementation

This section presents the implementation of *RRC*, focusing on the mechanisms used to overcome key challenges. *RRC* 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 (§4.1); support for a few variables shared between the kernel and RR library, used to coordinate checkpointing with record and replay (§4.2); a new queueing discipline kernel module used to pause and release network traffic (§4.3); and container fork (§4.6).

In the rest of this section, §4.1 presents the basic record and replay scheme. §4.2 deals with the challenge of integrating checkpointing with record and replay. §4.3 presents the handling of network traffic. The transition from replay to live execution is discussed in §4.4. The performance-critical operation of transmitting the nondeterministic event log to the backup is explained in §4.5. Container fork is presented in §4.6. §4.7 presents our best-effort mechanism for increasing the probability of correct replay in the presence of infrequently-manifested data races.

4.1 Nondeterministic Events Record/Replay

To minimize overhead and implementation complexity, *RRC* records synchronization operations and system calls at the user level. This is done by code added in glibc before (*before hook*) and after (*after hook*) 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 are recorded to handle the *trylock* variants as well as errors. During replay, synchronization operations must actually be performed in order to properly enforce the correct semantics. For each lock, the ordering of successful lock acquires is enforced. Since there is no need to enforce ordering among different locks, it is sufficient to maintain a separate log for each lock.

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 §4.4, functions, such as *open()*, that involve the manipulation of kernel state, are actually executed during replay.

There can be dependencies among system calls, even if they are invoked by different threads. For example, this is the case for system calls whose execution involve writes and reads from kernel data structures, such as the file descriptor table. Hence, simply maintaining a separate log for each thread is not sufficient. To handle such cases, the kernel was modified to maintain an access sequence number for each such shared kernel resource. 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.

4.2 Integrating Checkpointing with Record/Replay

Checkpointing is triggered by a timer external to the container [62], and is thus not synchronized with the recording of nondeterministic events on the primary. This has the potential of resulting in a checkpoint and log contents on the backup from which correct replay cannot proceed. One example is that the checkpoint may include a thread in the middle of executing code in the RR library, resulting in the backup, during replay, attempting to send the nondeterministic event log to the backup. A second example is that there may be ambiguity at the backup as to whether a particular system call, such as *open()*, was executed after the checkpoint and thus needs to be reexecuted during replay, or executed before the checkpoint and thus should not be reexecuted during replay.

A naive solution to the above problem would be to delay the checkpointing of a thread if it is in execution anywhere between the beginning of the before hook and the end of the after hook. However, this could delay checkpointing for arbitrarily long time if a thread is blocked on a system call, such as *read()*.

The actual solution in *RRC* has two properties: (I) checkpointing of a thread is delayed if the thread is *within* the before hook or *within* the after hook, and (II) checkpointing of a thread can occur even if the thread is between the end of the before hook and the beginning of the after hook.

To enforce property (I), 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* when it respectively enters/leaves the hook function. An addition to the kernel code prevents the thread from being paused if the thread's *in_rr* flag is set.

To deal with property (II), *RRC* includes mechanisms to: (A) detect that this scenario has occurred, and (B) eliminate the potential ambiguities, such as the one mentioned above and take appropriate actions during replay. To implement the required mechanisms, *RRC* uses three variables: two per-thread flags – *in_hook* and *syscall_skipped*, as well as a global *current_phase* variable [63]. These variables are shared between the user level and the kernel. In the record phase, *in_hook* is set in the before hook and cleared in the after hook – this is mechanism (A) above.

For mechanism (B), *syscall_skipped* is used, during the replay phase, to determine whether, during the record phase, the checkpoint was taken before or after executing the system call. During the record phase, this flag is cleared during initialization and is not otherwise read or written. With CRIU (§2.1), if a checkpoint is triggered while a thread is executing a system call, before that call performs any state changes, the system call is retried after the checkpoint is restored. In the replay phase, at an early point in the kernel code executing a system call, if *in_hook* is set, the system call is skipped and *syscall_skipped* is set. Thus, if the system call was not executed before the checkpoint, it will be initially skipped during replay. During replay, if the after hook finds that *in_hook* and *syscall_skipped* are set, it passes control back to the before hook and the system call is then replayed or re-executed.

The handling of lock operations is similar to the handling of system calls. In the after hook, if *in_hook* is set, the lock is released and control is passed to the before hook, thus allowing enforcement of the order of lock acquires.

4.3 Handling Network Traffic

The current *RRC* 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 [20, 60]. With *RRC*, this is done by routing incoming and outgoing packets through the backup (§3). Incoming packets are recorded by the PackRec thread in the agent. Outgoing packets 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 (*RSN*) for each TCP stream. When the primary container sends an outgoing message, the nondeterministic event log it sends to the backup (§3) includes a release request that updates the stream's *RSN*. The outgoing packets with sequence numbers lower than the *RSN* are then released.

PackGate is implemented in the kernel since it operates frequently and must thus be efficient. PackGate maintains fairness among the TCP streams using a FIFO queue of release requests ordered by the order of sends.

4.4 Transition to Live Execution

As with [38, 43] and unlike the deterministic replay tools for debugging [44, 55–57], *RRC* needs to transition from replay mode to live mode. This 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 atomically 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 *RRC*, these are reads and writes on sockets associated with a connection to an external client. As discussed in §4.1, 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, *RRC* records the state changes on the sockets based on the nondeterministic event logs. When the replay phase completes, *RRC* 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, *RRC* cannot rely on the event log since some packets received and acknowledged by the kernel may not have been read by the application. Instead, *RRC* uses information obtained from PackRec (§4.3).

With respect to incoming packets, once the container transitions to live execution, *RRC* 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. PackRec obtains the last acknowledged sequence number of each stream from PackGate and stops recording when it has all the required (acknowledged) incoming packets. Before the container is restored on the backup, PackRec copies the recorded incoming packets to a log. Using the information from the nondeterministic event log and PackRec, before the transition to live execution, the packet repair mode (§2.1) is used to restore the socket state so that it is consistent with the state of the container and the external world.

4.5 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 (§4.1) and send them to the backup (§3). Hence, the collection and sending of the log is a frequent activity, which is thus performance critical. Specifically, with our initial implementation, with the *Memcached* benchmark under maximum load, the throughput overhead was approximately 300%.

To address the performance challenge above, *RRC* 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 (as in [47]). With available CPU cycles, such as additional cores, this minimizes the overhead for 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 maximize performance, *RRC* allows concurrent access to different logs. One application thread may log a lock operation concurrently with another 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, there is the potential for the log transferred to the backup for a particular outgoing message to be incomplete – missing an entry for an event on which the outgoing message depends. This can lead to replay failure.

There are two key properties of *RRC* 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 the two properties above, the RR library maintains two corresponding global sequence numbers: *primary batch sequence number* (PBSN) and *backup batch sequence number* (BBSN) in the primary and backup, respectively. They are both initialized to 0. Application threads attach the PBSN to the entries they log for nondeterministic events. When the logging thread picks up an entry from the aforementioned ring buffer, that is a request to collect and send the current event log. Before taking any other action, the logging thread scans the ring buffer to determine the number of pending requests. It then increments the PBSN by that number. Thus, every event log entry that is created after the logging thread begins collecting the log, has a higher PBSN tag. After the logging thread sends the log, it sends to the backup a message that directs the backup to increment the BBSN by the most recent increment of the PBSN. 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, replay terminates and live execution commences.

4.6 Container Fork

The new container fork (*cfork*) system call is based on the existing process *fork*. Given a process ID in a container, *cfork* duplicates the container state shared among its processes and threads: namespaces (e.g., mount points, network interfaces) and control groups. *Cfork* then duplicates all the processes and their threads in the container and assigns them to the new container. *Fork* duplicates the file descriptor state, but does not duplicate the underlying state, such as socket state or pipe state. However, *cfork* does duplicate this underlying state.

The implementation of *cfork* for *RRC* includes optimizations to minimize the container fork time. We identified two major sources of overhead: (1) duplicating the namespaces and control groups, and (2) page table copy. To minimize (1), *RRC* exploits the fact that most namespace and control group state rarely changes after initialization [62]. Thus, at the first checkpoint, *RRC* creates a staging container with an idle process. *Cfork* assigns the forked container to the namespace and control group of the staging container instead of creating new ones. To ensure correctness, *RRC* detects state changes of the namespaces and control groups of the service container using hooks, added with *ftrace* to the kernel functions that can change the namespace and cgroup state. *Cfork* updates those changes to the staging container. *Ftrace* only incurs overhead if a hooked function is invoked. Since the namespace and cgroup states rarely change, such functions are rarely invoked and *ftrace* does not incur high overhead.

To minimize the latency of the page table copy, *RRC* avoids copying the page table of the data region of the RR library, whose size can be up to several gigabytes and thus takes tens of milliseconds to copy. Specifically, the RR library tags the VMA of the data region with a new special flag and thus informs the *cfork* to skip copying its page table. This optimization is correct because *RRC* does not need to checkpoint the data region of the RR library; its state is initialized upon replay by reading the saved nondeterministic logs in the backup.

4.7 Mitigating the Impact of Data Races

As discussed in §3, *RRC* includes mechanisms that significantly increase the probability of successful recovery in the presence of rarely-manifested data races. Specifically, *RRC* 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-replayed 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 that last-replayed 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 §6.3.

5 Experimental Setup

All the experiments were hosted on Fedora 29 with the 4.18.16 Linux kernel. The containers were hosted using runC [12] (version 1.0.1), a popular container runtime used in Docker. The primary and backup replicas were hosted on different 36-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.

Five benchmarks were in-memory databases handling short requests: *Redis* [13], *Memcached* [10], *SSDB* [15], *Tarantool* [16] and *Aerospike* [2]. These benchmarks were evaluated with 50% read and 50% write requests to 100,000 100B records, driven by *YCSB* [31] clients. The number of user client threads ranged from 60 to 480. The evaluation also included a web server, *Lighttpd* [7], and two batch PARSEC [26] benchmarks: *Swaptions* and *Streamcluster*. *Lighttpd* was evaluated using 20-40 clients retrieving a 1KB static page. For *Lighttpd*, benchmarking tools SIEGE [14], ab [1] and wget [5] were used to evaluate, respectively, the performance overhead, response latency, and recovery rate. *Swaptions* and *Streamcluster* were evaluated using the native input test suites. We evaluated only two benchmarks from the PARSEC suite since *RRC* targets server applications and its design is thus focused on low latency overhead. Low latency overhead is not relevant for the batch applications, such as those included in the PARSEC suite. Nonetheless, we show that such applications can be handled by *RRC* with very low throughput overhead.

We used fault injection to evaluate *RRC*'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 "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 "successful" if all the fetched pages were identical to the golden copy. For the in-memory database benchmarks, we developed customized clients, using existing client libraries [3, 9, 11, 17], that spawn 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.

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

	TP Overhead			Avg. Latency(μs)		
	Redis	Taran	Aero	Redis	Taran	Aero
Custom	49%	31%	153%	574	471	456
RRC-LE	31%	26%	47%	543	564	602

Table 1: Throughput overhead and average latency. *RRC* vs. custom replication mechanisms.

was pinned to each core executing the benchmark.

6 Evaluation

This section presents *RRC*'s performance overhead and CPU usage overhead (§6.1), the added latency for server responses (§6.2), as well as the recovery rate and recovery latency (§6.3). Two configurations of *RRC* are evaluated: *RRC-SE* (short epoch) and *RRC-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, *RRC-LE* may be used if there is high confidence that the applications are free of data races. Thus, with the *RRC-SE* configuration, the data race mitigation mechanism described in §4.7 is turned on, while it is turned off for *RRC-LE*.

RRC is compared to NiLiCon (§2.1) 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, as in [62]. The short epochs of NiLiCon are required since, unlike *RRC*, the epoch duration with NiLiCon determines the added latency in replying to client requests (§2.1). Thus, for many server applications, even with 30ms epochs, NiLiCon provides unacceptably long response latencies. In all cases, the "stock setup" is the application running in an unreplicated container.

Some server applications can be configured to enable their own custom fault tolerance mechanisms. However, developing and validating such mechanisms is time consuming and error prone. Hence, mechanisms, such as *RRC*, that can be deployed for many applications, are likely to be of higher quality (reliability) and incur lower total development cost. Table 1 compares the overhead of *RRC* with the custom mechanisms of three of our benchmarks (§5). The custom mechanisms are all configured to provide *strong consistency* (outputs are not released until the changes are reflected in the backup), which *RRC* also provides. The results show that *RRC-LE* actually has lower throughput overhead. On average, the custom mechanisms do result in lower response latency. This is mainly due to their ability to release the outputs of read requests without waiting for acknowledgments from the backup. However, on average, the overall results are comparable.

6.1 Overheads: Performance, CPU Utilization

Two key overhead measures of *RRC* 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 *RRC* are in parallel with the main computation threads.

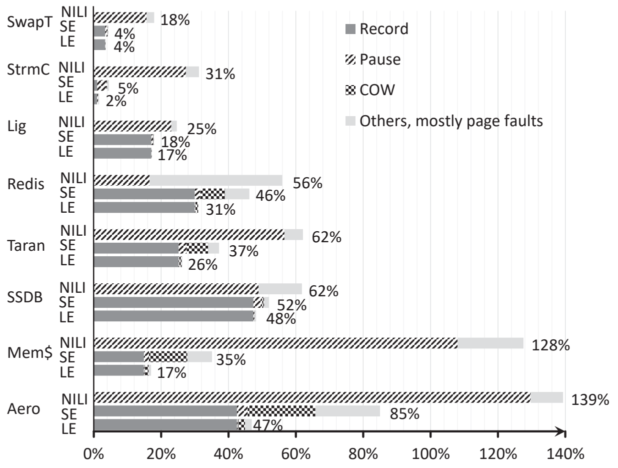

Figure 3: Performance overheads: NiLiCon, *RRC-SE*, *RRC-LE*.

For the six server benchmarks, the measurements reported in this subsection were done with workloads that resulted in maximum CPU utilization for the cores running the application worker threads[1] with the stock setup.

With four of the server benchmarks, the number of the worker threads cannot be configured (*Lighttpd*, *Redis*: 1, *Tarantool*: 2, *SSDB*: 12). The remaining four benchmarks were configured to run with four worker threads.

For each benchmark, the workload that saturates the cores in the stock setup was used for the stock, *RRC*, and NiLiCon setups. With NiLiCon, due to its large latency overhead (§6.2), 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 3 shows the performance overheads of NiLiCon, *RRC-SE*, and *RRC-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 nondeterministic events. The pause overhead is due to the time the container is paused during the container fork. The COW overhead is caused by the time to copy the pages after the container fork. The page fault overhead is caused by the page fault exceptions that track the memory state changes of each epoch (§2.1).

With *RRC-SE*, the average incremental checkpoint size per epoch was 0.2MB for *Swaptions*, 15.6MB for *Redis*, and 41.2MB for *Aerospike*. With *RRC-SE*, the average number of

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

WWSS	Redis	Taran	SSDB	Mem$
1x	47% (1.00)	37% (1.00)	53% (1.00)	36% (1.00)
2x	56% (1.19)	51% (1.38)	57% (1.08)	58% (1.61)
3x	73% (1.55)	63% (1.70)	62% (1.17)	73% (2.03)

Table 2: The impact of the write working set size (WWSS), relative to the WWSS used in Figure 3, on the performance overhead with *RRC-SE*. The overheads relative to the 1x case are in prentheses.

		ST	SC	Lig	Redis	Taran	SSDB	Mem$	Aero
CP	Primary	3%	5%	8%	30%	16%	8%	13%	31%
	Backup	1%	2%	5%	26%	15%	4%	13%	18%
RR	Primary	4%	1%	34%	47%	46%	55%	36%	77%
	Backup	∼0	∼0	33%	29%	20%	11%	15%	19%
TCP	Primary	∼0	∼0	∼0	∼0	∼0	∼0	∼0	∼0
	Backup	∼0	∼0	59%	86%	54%	23%	40%	43%
	total	8%	8%	139%	218%	151%	101%	117%	188%

Table 3: CPU utilization overhead for *RRC-SE*. **CP**: checkpointing. **RR**: recording nondeterministic events. **TCP**: handling TCP failover.

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.

The number of pages modified during an epoch determines the rate of page faults and COW operations, as well as the size of the incremental checkpoint that is transferred to the backup (§3). Hence, this write working set size (WWSS) impacts the performance overhead. Table 2 shows the performance overhead of *RRC-SE* with four of our benchmarks as the WWSS is increased to 2x and 3x of the WWSS used in Figure 3. These measurements were obtained by increasing the number of records and then, with a fixed number of records, varying the ratio of writes to reads to obtain the different WWSS values. As expected, the checkpointing component of the overhead ("COW" plus "Others" in Figure 3) increases approximately linearly with the WWSS. As shown in Figure 3, as the epoch duration is increased, the checkpointing component of the overhead is decreased and thus the impact of the WWSS becomes less significant. It should be noted that the number of pages read during an epoch has no impact on the performance overhead. The footprint of the application has only negligible impact that is due to the time to scan the page table to identify the modified pages.

CPU utilization overhead. The CPU utilization (Table 3) is the product of the average numbers of CPUs (cores) used and the total execution time. The CPU utilization overhead is the percentage increase in utilization with *RRC* compared to with the stock setup. The breakdown of the overhead into its components was obtained by incrementally enabling each component and measuring the corresponding increase in CPU

		Lig1K	Lig100K	Redis	Taran	SSDB	Mem$	Aero
S	avg	549	2059	406	393	388	643	373
	99%	<1ms	<3ms	734	617	622	2982	711
R	avg	694	2203	604	604	651	812	663
	99%	<1ms	<3ms	969	992	988	3941	1273
N	avg	38ms	38ms	42ms	42ms	45ms	45ms	51ms
	99%	<39ms	<39ms	44ms	42ms	47ms	53ms	63ms

Table 4: Response latency in μs. S: Stock, R: *RRC-SE*, N: NiLiCon

overhead. A significant factor in the CPU utilization overhead is for packet handling in the backup kernel needed to support TCP failover. This overhead is mostly due to routing. Techniques for optimizing software routing [42] can be used to reduce this overhead.

The overheads shown in Table 3 should be evaluated in the context of comparable alternative techniques. The only alternatives that can achieve low latency overheads necessary for many server applications are based on active replication [38, 47]. Such techniques have CPU overheads comparable to *RRC*'s for recording nondeterministic events and handling TCP failover. They do not have the overhead for checkpointing but instead have 100% overhead for execution on the backup. Table 3 shows the with *RRC-SE* the overhead for checkpointing is 4%-56%. Hence, *RRC*'s CPU overhead is significantly smaller than the comparable alternatives'.

Performance decoupling. An important property of *RRC* is that, unlike active replication, it decouples the performance of the application on the primary host from the performance on the backup. To illustrate the impact of this, we selected two representative benchmarks: *Redis* and *Aerospike*, which incur a significant CPU usage on the backup host, and ran them with *RRC-SE*. We ran the perturb program (§5), which consumes 40% of a CPU, first on all the cores of the primary and then the backup. When the perturb program runs on the primary, the performance overhead increases from 46% to 71% and 85% to 116% for *Redis* and *Aerospike*, respectively. However, when the perturb program runs on the backup, the execution time remains the same.

6.2 Response Latency

Table 4 shows the response latencies with the stock setup, *RRC-SE* and NiLiCon. The numbers of client threads for stock and *RRC-SE* are adjusted so that the CPU load on the cores running application worker threads is 50%. For NiLiCon, the number of client threads is the same as with *RRC-SE*, resulting in CPU utilization of less than 5%, thus favoring NiLiCon. To evaluate the impact of response size, *Lighttpd* is evaluated serving both 1KB as well as 100KB files.

With *RRC*, 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 *RRC-SE*, the increase

		ST	SC	Lhttpd	Redis	Taran	SSDB	Mem$	Aero
CF	avg	0.7	2.7	0.5	1.6	2.4	2.6	1.5	3.2
	90%	0.7	3.1	0.6	1.9	2.7	2.9	1.7	3.5
NCF	avg	5.9	7.6	7.2	14.9	18.4	13.9	28.7	42.9
	90%	5.9	8.0	7.4	16.7	20.2	14.8	33.7	45.8

Table 5: The pause time of *RRC* with container fork (**CF**) and without container fork (**NCF**) in millisecond.

	ST	SC	Lhttpd	Redis	Taran	SSDB	Mem$	Aero
avg	3.1	3.9	2.5	9.1	11.5	6.5	15.6	27.4

Table 6: The average time (ms) between resuming container execution and the stop of COW.

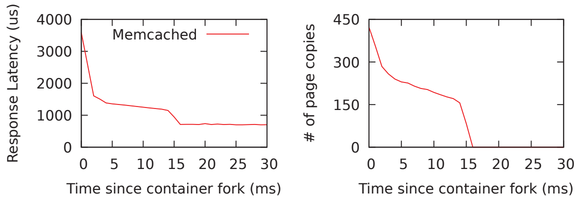

Figure 4: Average response latency and the number of COW since container fork.

in average latency is only 144μs to 290μs. The worst case is with *Aerospike*, which has the highest checkpointing overhead (COW+Others in Figure 3) 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 235μs to 959μs. The worst case is with *Memcached*. As shown in Table 4, in terms of increase in response latency, NiLiCon is not competitive, as also indicated by the results in [62].

With *RRC-LE*, the increase in the average response latency is from 42μs to only 229μs, due to the the lower checkpointing overhead. The increase in the 99th percentile latency is under 510μs since the container fork are much less frequent and thus less likely to interrupt the processing of a request.

The impact of container fork. The tail response time latency overhead is determined by the time the primary is paused for checkpointing. Table 5 shows *RRC*'s pause time with and without the container fork. Without the container fork, the container has to be paused during the entire checkpointing process, leading to a pause time between 5.9ms to 45.8ms. The pause time with the container fork is only from 0.5ms to 3.5ms. Most of the container fork time is spent on copying page tables and thus can be further reduced with recent techniques on optimizing *fork()* [61].

Due to the reduction in the pause time, with the SE setup, the container fork reduces the average response latency overhead from 156μs-581μs to 144μs-290μs, and the worst-case 99% response latency overhead from 6ms to 959μs. The throughput overhead is reduced from 8%-145% to 4%-85%.

Immediately after the container fork there is a period during which there is additional overhead due to COW of pages on the primary (§3). Table 6 shows that this period terminates at an early stage of each epoch. To evaluate the impact of the COW on response latency, we obtained fine grained measurements with *Memcached*. Figure 4 shows the results.

		Recovery Rate		Replay Time	
		Mem$	Aero	Mem$	Aero
100ms	stock	94.3%	84.5%	20	28
	+ Total order of syscalls	94.3%	92.7%	131	299
	+ Timing adjustment	99.2%	99.8%	234	383
1s	stock	51.4%	34.8%	249	373
	+ Total order of syscalls	51.6%	76.5%	1122	1345
	+ Timing adjustment	99.0%	99.4%	1230	1460

Table 7: Recovery rate and replay time (in ms). *RRC* with different levels of mitigation of data race impact.

Immediately after the container fork, due to the pause and a high rate of page copies, the response latency is around 3.5ms. However, the response latency almost immediately drops to around 1.5ms and then to 700μs, where it remains for the rest of the epoch.

6.3 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 *RRC*, the average detection time is 90ms (§5). 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 the Linux TCP 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.

Primary failure recovery rate. Three of our benchmarks contain data races that may cause recovery failure: *Memcached*, *Aerospike*, and *Tarantool*. Running *Tarantool* with *RRC-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 §4.7. Thus, we use a 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, *Memcached* and *Aerospike*, is used to evaluate the the data race mitigation mechanisms (§4.7). 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 7. For both the recovery rate and replay time, the 95% confidence interval

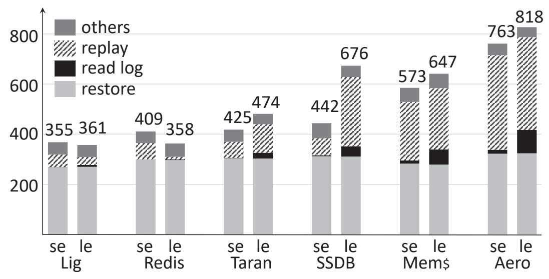

Figure 5: Recovery latency (ms) breakdown with *RRC-SE* and *RRC-LE*.

is less than 1%. Without the §4.7 mechanism, the recovery rate for *RRC-LE* is much lower than with *RRC-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 *RRC* setups. However, with the timing adjustments, both benchmarks achieve high recovery rates, even with *RRC-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.

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.

Primary failure recovery latency. Figure 5 shows a breakdown of the factors that make up the recovery latency for the server benchmarks with *RRC-SE* and *RRC-LE*. With *RRC-SE*, the data race mitigation scheme is enabled, while with *RRC-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.

footprint	Redis	Taran	SSDB	Mem$	Aero
1x	409 (1.00)	425 (1.00)	442 (1.00)	573 (1.00)	763 (1.00)
2x	424 (1.04)	460 (1.08)	479 (1.08)	583 (1.02)	836 (1.10)
3x	463 (1.13)	493 (1.16)	524 (1.18)	609 (1.06)	917 (1.20)

Table 8: The impact of the footprint size, relative to the footprint size used in Figure 5, on the primary recovery latency (in ms) with *RRC-SE*. The latencies relative to the 1x case are in parentheses.

With the current *RRC* implementation, the total memory occupancy of the application, i.e., its footprint, has an impact on the recovery latency. Specifically, during recovery on the backup host, all the pages are copied from the memory area where they are saved during prior checkpointing to new locations. Hence, as shown in Table 8, as the footprint is increased, there is a small increase in the recovery latency. In these measurements, the footprint was determined by the final checkpoint size. It should be noted that the impact of the footprint on recovery latency is a limitation of the current implementation. An optimization with kernel support would avoid copying the pages from one memory location to another by simply updating the page table.

7 Limitations

An inherent limitation is that the mechanism used for mitigating the impact of data races (§4.7) is incapable of handling a high rate of racy accesses (§6.3). However, as discussed in §3, such data races are easily detectable and are thus easy to eliminate, even in legacy applications.

The prototype implementation of *RRC* is restricted to single-process containers. This is not a major restriction since, in most cases, containers are used to run only a single process. Cito et al. [30] analyzed 38,079 Docker projects on Github and concluded that only 4% of the projects involved multi-process containers. This is reinforced by Internet searches regarding this issue that yield numerous hits on pages, such as [23], that suggest that running single-process containers is best practice. To overcome this limitation, the RR library would need to support inter-process communications via shared memory. Techniques presented in [24] may be applicable.

RRC also does not handle asynchronous signals. This can be resolved by techniques used in [43], that delay signal delivery until a system call or certain page faults.

The current implementation of *RRC* only supports C/C++ applications. Adding support for non-C/C++ applications would require instrumenting their runtimes to track nondeterministic events. *RRC* 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.

8 Related Work

RRC is related to prior fault-tolerance works on replication based on high-frequency checkpointing, replication based on deterministic replay, and network connection failover.

Early work on VM replication is based on leader-follower active replication using deterministic replay [27]. This is combined with periodic checkpointing in [28], based on use of this technique for debugging [41]. These works focused on uniprocessor systems. Extending them to multiprocessors is impractical, due to the overhead of recording shared memory access order for a VM [37, 52]. Remus [33] (§2.1) and its follow-on works [46, 53, 62] focus on multiprocessor workloads and implement replication using high-frequency checkpointing. Plover [58] optimizes Remus by using an active replica to reduce the size of transferred state and by performing state synchronization adaptively, when VMs are idle. All the Remus-based mechanisms release outputs only after the primary and backup synchronize their states, Hence, outputs are delayed by multiple (often, tens of) milliseconds. COLO [35] compares outputs from two active VM replicas and synchronizes their states on a mismatch, resulting in high throughput and latency overheads for applications with significant nondeterminism.

For process-level checkpointing, libckpt [51] implements "forked checkpointing," where the unmodified *fork()* system call is used to minimize the pause time for checkpointing.

To handle nondeterminism in parallel applications, as with *RRC*, some works rely on replaying the order of synchronization operations [32, 38, 47]. Rex [38] and Crane [32] cannot handle state divergences caused by data races and require manual modifications of the application source code. Castor [47] handles data races by buffering outputs until the backup finishes replaying the associated logs. If divergence due to data races occurs, the two replicas synchronize their state.

Comparing *RRC* with Rex, Crane, and Castor, for data-race-free applications, *RRC* is likely to have a smaller throughput overhead. Specifically, Rex reports that under heavy load, replay may be slower than the original execution and thus the active replica is a performance bottleneck. With a data-race-free setup, both Rex and *RRC* are evaluated with *Memcached*, and the performance overheads are 40% vs. 17%.

For applications that have data races, the only relevant comparison is with Castor. Castor is likely to have higher response delays since outputs cannot be released until the backup finishes replaying the associated log. Additionally, a data race can also cause Castor to fail. Specifically, if the primary fails in the middle of state synchronization caused by a data race, the system fails. Hence, for an application with a high rate of racy memory accesses, such as *Tarantool* (§6.3), Castor would be frequently synchronizing the state and thus have low recovery rate (like *RRC*) and also high performance overhead. For applications with a lower rate of racy memory accesses, such as *Memcached* and *Aerospike*, Castor also has lower recovery rate. For example, for *Memcached*, based on Table 7, the probability of execution divergence in 50ms is 0.059. Hence, execution diverges approximately every 0.85s. With our setup, the time it takes to create and transfer the checkpoint for *Memcached* is 48ms. Hence, an upper bound on the recovery rate with Castor is expected be 94.7% versus 99.2% with *RRC* (Table 7). A similar calculation for *Aerospike*, taking into account 76ms to create and transfer the checkpoint, results in a recovery rate for Castor of 79.8% versus 99.8% for *RRC*.

9 Conclusion

RRC is a unique point in the design space of application-transparent fault tolerance schemes for multiprocessor workloads. By combining checkpointing, with externally deterministic replay, and container fork, it provides all the desirable properties of a fault tolerance scheme listed in §1, with specific emphasis on low latency overhead, which is critical for server applications. *RRC* facilitates trading off performance and resource overheads with vulnerability to data races and recovery latency. Critically, the response latency is decoupled from the frequency of checkpointing, and sub-millisecond added delay is achieved with all our server applications. *RRC* is a full fault tolerance mechanism. It can recover from primary or backup host failure and includes transparent failover of TCP connections.

As we have found (§6.3), legacy applications may have data races. *RRC* 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 [38], *RRC* is not affected by data races that manifest during normal operation, long before failure. For data races that manifest right before failure, *RRC* introduces simple mechanisms that significantly reduce the probability of the data races causing recovery failure.

This paper describes key implementation challenges encountered in the development of *RRC* and outlines their resolution. The extensive evaluation of *RRC*, 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 *RRC* to particularly harsh conditions by intentionally perturbing the scheduling on the primary, thus challenging the deterministic replay mechanism (§5). With high checkpointing frequency (*RRC-SE*), *RRC*'s throughput overhead is less than 53% for seven of our benchmarks and 85% for the eighth. If the applications are known to be data-race-free, with a lower checkpointing frequency (*RRC-LE*), the overhead is less than 49% for all benchmarks, significantly outperforming NiLiCon [62]. With data-race-free applications, *RRC* recovers from all fail-stop failures. With two applications with infrequently-manifested data races, the recovery rate is over 99% with *RRC-SE*.

Acknowledgments

We thank our reviewers, especially our shepherd, for constructive feedback that significantly improved this paper.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Aerospike. https://https://www.aerospike.com/.

[3] Aerospike C Client. https://www.aerospike.com/apidocs/c/.

[4] CRIU: Checkpoint/Restore In Userspace. https://criu.org/Main_Page.

[5] GNU Wget. https://www.gnu.org/software/wget/.

[6] Google threadsanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[7] Home - Lighttpd. https://www.lighttpd.net/.

[8] 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.

[9] libMemcached. https://libmemcached.org/libMemcached.html.

[10] memcached. https://memcached.org.

[11] Minimalistic C client for Redis. https://github.com/redis/hiredis.

[12] opencontainers/runc. https://github.com/opencontainers/runc.

[13] Redis. https://redis.io.

[14] Siege Home. https://www.joedog.org/siege-home/.

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

[16] Tarantool - In-memory DataBase. https://tarantool.io.

[17] Tarantool C client libraries. https://github.com/tarantool/tarantool-c.

[18] TCP connection repair. https://lwn.net/Articles/495304/.

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

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

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

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

[23] Rafael Benevides. 10 Things to Avoid in Docker Containers. https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers, February 2016. Accessed: 2022-04-25.

[24] 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*, page 177–191, Vancouver, BC, Canada, October 2010.

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

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

[27] 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.

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

[29] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic Replay: A Survey. *ACM Computing Surveys*, 48(2):17:1–17:47, September 2015.

[30] Jurgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 323–333, Buenos Aires, Argentina, May 2017.

[31] 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*, pages 143–154, June 2010.

[32] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made Transparent. In *25th Symposium on Operating Systems Principles*, pages 105–120, Monterey, California, October 2015.

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

[34] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[35] 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.

[36] 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*, pages 211–224, Boston, MA, USA, December 2003.

[37] 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*, page 121–130, Seattle, WA, USA, March 2008.

[38] 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*, pages 161–174, Amsterdam, The Netherlands, April 2014.

[39] 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*, page 193–208, San Diego, CA, USA, December 2008.

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

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

[42] 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.

[43] 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*, page 155–166, New York, New York, USA, June 2010.

[44] 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*, page 77–90, Pittsburgh, Pennsylvania, USA, March 2010.

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

[46] 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.

[47] 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*, page 693–708, Xi'an, China, April 2017.

[48] 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.

[49] 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*, page 97–108, Washington, DC, USA, March 2009.

[50] 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, page 177–192, Big Sky, Montana, USA, October 2009.

[51] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *USENIX 1995 Technical Conference*, pages 213–224, New Orleans, LA, January 1995.

[52] 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*, page 551–564, Denver, CO, USA, June 2016.

[53] 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.

[54] 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.

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

[56] 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*, pages 29–44, Boston, MA, USA, June 2004.

[57] 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*, page 15–26, Newport Beach, California, USA, March 2011.

[58] 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*, pages 483–499, Renton, WA, April 2018.

[59] 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*, page 122–135, San Diego, California, USA, May 2003.

[60] 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.

[61] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *16th European Conference on Computer Systems*, pages 540–555, Virtual, April 2021.

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

[63] Diyu Zhou and Yuval Tamir. HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay. *Computing Research Repository*, arXiv:2101.09584 [cs.DC], January 2021.