HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay

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

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

Abstract

HyCoR is a fully-operational fault tolerance mechanism for multiprocessor workloads, based on container replication, using a hybrid of checkpointing and replay. *HyCoR* derives from two insights regarding replication mechanisms: 1) deterministic replay can overcome a key disadvantage of checkpointing alone – unacceptably long delays of outputs to clients, and 2) checkpointing can overcome a key disadvantage of active replication with deterministic replay alone – vulnerability to even rare replay failures due to an untracked nondeterministic events. With *HyCoR*, the primary sends periodic checkpoints to the backup and logs the outcomes of sources of nondeterminism. Outputs to clients are delayed only by the short time it takes to send the corresponding log to the backup. Upon primary failure, the backup replays only the short interval since the last checkpoint, thus minimizing the window of vulnerability. *HyCoR* includes a "best effort" mechanism that results in a high recovery rate even in the presence of data races, as long as their rate is low. The evaluation includes measurement of the recovery rate and recovery latency based on fault injection. On average, *HyCoR* delays responses to clients by less than 1ms and recovers in less than 1s. For a set of eight real-world benchmarks, if data races are eliminated, the performance overhead of *HyCoR* is under 59%.

1 Introduction

For many applications hosted in data centers, high reliability is a key requirement, demanding fault tolerance. The fault tolerance techniques deployed are preferably application-transparent in order to avoid imposing extra burden on application developers and facilitate use for legacy applications. 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 transfer of the primary replica state (checkpoint), at the end of every execution *epoch*, to an inactive backup, so that the backup can take over if the primary fails [30]; and (2) active replication, where the backup mirrors the execution on the primary so that it is ready to take over. The second approach is challenging for multiprocessor workloads, where there are many sources of nodeterminism. Hence, it is implemented in a leader-follower setup, where the outcomes of identified nondeterministic events, namely synchronization operations and certain system calls, on the primary are recorded and sent to the backup, allowing the backup to deterministically replay their outcomes [34, 42].

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

This paper presents a novel fault tolerance scheme, based on container replication, called *HyCoR* (Hybrid Container Replication). *HyCoR* overcomes the two disadvantages above using a unique combination of periodic checkpointing [30,54], externally-deterministic replay [22, 28, 40, 44, 50], user-level recording of nondeterministic events [40, 42], and failover of network connections [20, 23]. A critical feature of *HyCoR* is that the checkpointing epoch duration does not affect the response latency, enabling *HyCoR* to achieve sub-millisecond added delay (§6.2). This allows adjusting the epoch duration to trade off performance and resource overheads with recovery latency and vulnerability to untracked nondeterministic events. The latter is important since, especially legacy applications, may contain data races (§6.3). *HyCoR* is focused on dealing with data races that rarely manifest and are thus more likely to remain undetected. Since *HyCoR* only requires replay during recovery and for the short interval since the last checkpoint, it is inherently more reslilient to data races than schemes that rely on replay of the entire execution [34]. Furthermore, *HyCoR* includes a simple timing adjustment

mechanism that results in a high recovery rate even for applications that include data races, as long as their rate of unsynchronized writes is low.

Replication can be at the level of VMs [26, 30, 31, 46, 51], processes [34, 42], or containers [54]. We believe that containers are the best choice for mechanisms such as *HyCoR*. Applying *HyCoR*'s approach to VMs would be complicated since there would be a need to track and replay nondeterministic events in the kernel. On the other hand, with processes, it is difficult to avoid potential name conflicts upon failover. A simple example is that the process ID used on the primary may not be available on the backup. While such name conflicts can be solved, the existing container mechanism already solves them efficiently.

With *HyCoR*, execution on the primary is divided into epochs and the primary state is checkpointed to an inactive backup at the end of each epoch [30, 54]. Upon failure of the primary, the backup begins execution from the last primary checkpoint and then deterministically replays the primary's execution of its last partial epoch, up to the last external output. The backup then proceeds with live execution. To support the backup's deterministic replay, *HyCoR* ensures that, before an external output is released, the backup has the log of nondeterministic events on the primary since the last checkpoint. Thus, external outputs are delayed only by the time it takes to commit the relevant last portion of the log to the backup.

Combining checkpointing with externally-deterministic replay for replication is not new [27, 34, 37, 40, 42]. However, Respec [40] requires an average external output delay greater than half an epoch and is based on *active* replication. [27, 37] do not provide support for execution on multiprocessors. See §8 for additional discussion. Furthermore, these prior works do not provide an evaluation of recovery rates and are not designed or evaluated for containers.

We have implemented a prototype of *HyCoR* and evaluated its performance and reliability using eight benchmarks. We obtained the source code for NiLiCon [54], and it served as the basis for the implementation of checkpointing and restore. The rest of the implementation is new, involving instrumenting standard library calls at the user level, user-level agents, and small kernel modifications. With 1s epochs, *HyCoR*'s performance overhead was less than 59% for all eight benchmarks. With more conservative 100ms epochs, the overhead was less than 68% for seven of the benchmarks and 145% for the eighth. *HyCoR* is designed to recover from fail-stop faults. We used fault injection to evaluate *HyCoR*'s recovery mechanism. For all eight benchmarks, after data races identified by ThreadSanitizer [7] were resolved, *HyCoR*'s recovery rate was 100% for 100ms and 1s epochs. Three of the benchmarks originally included data races. For two of these, without any modifications, with 100ms epochs and *HyCoR*'s timing adjustments, the recovery rate was over 99.4%.

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

Section 2 presents two key building blocks for *HyCoR*: NiLiCon [54] and deterministic replay [22, 28, 40, 44, 50]. An overview of *HyCoR* is presented in §3. *HyCoR*'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 *HyCoR* and of our prototype implementation are described in §7. §8 provides a brief overview of related work.

2 Background

HyCoR integrates container replication based on periodic checkpointing [30, 54], described in §2.1, and deterministic replay of multithreaded applications on multiprocessors, described in §2.2.

2.1 NiLiCon

Remus [30] introduced a practical application-transparent fault tolerance scheme based on VM replication using high-frequency checkpointing. NiLiCon [54] is an implementation of the Remus mechanism for containers. With NiLiCon, the active primary container and passive backup container are on different hosts. Execution on the primary is divided into epochs. At the end of each epoch, the primary is paused and an incremental checkpoint, containing all state changed since the last checkpoint, is sent to the backup. To track the memory state changes, NiLiCon sets all the memory pages to be read-only at the beginning of the epoch. Thus, the first write to a page in an epoch triggers a page fault exception, allowing NiLiCon to record the page number and restore its original access permissions.

As presented in §1, external outputs (server replies to clients) are initially buffered and are released only after the commitment on the backup host of the checkpoint of the epoch that generated the outputs. If the primary fails, the checkpoint on the backup host is used to restore the container. Since no external outputs are released prior to checkpoint commitment, consistency between the containers and external clients is guaranteed, even if the workload is non-deterministic.

NiLiCon's implementation is based on a tool called CRIU (Checkpoint/Restore in User Space) [5] with optimizations that 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 verion of the DRBD module [9].

To ensure consistency of the checkpointed state, CRIU utilizes the Linux kernel's *freezer* [4] feature to stop the container so that its state does not change during checkpointing. A virtual signal to all the threads in the container causes them to pause. A thread executing a system call immediately exits the system call and is paused before returning to user mode. Once all the threads are paused, checkpointing proceeds.

NiLiCon relies on CRIU to preserve established TCP connections across failover, using a special repair mode of the socket provided by the Linux kernel [19]. After setting a socket to repair mode, a user-level process can obtain or modify TCP state, such as sequence numbers and buffered packets.

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 [28]. 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 accesses to the same memory location by different processors. Without hardware support, the cost of providing deterministic replay of all such events is prohibitive. Hence, a common practical approach is to support deterministic replay only for programs that are data-race-free [43]. For such programs, as long as the results of synchronization operations are deterministically replayed, the ordering of shared memory accesses are preserved. This involves much lower overhead since the frequency of synchronization operations is much lower than normal memory accesses.

The recording of nondeterministic events can occur at different levels: hardware [36, 52], hypervisor [32, 33, 37], OS [35, 39], or library [43, 47]. Without dedicated hardware support, the recording must be done in software. It is advantageous to record the events at the user level, thus avoiding the overhead for entering the kernel or hypervisor [40].

The degree to which the replay must recover the details of the original execution depends on the use case [28]. To support seamless failover with replication, it is sufficient to provide *externally deterministic replay* [40]. 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.

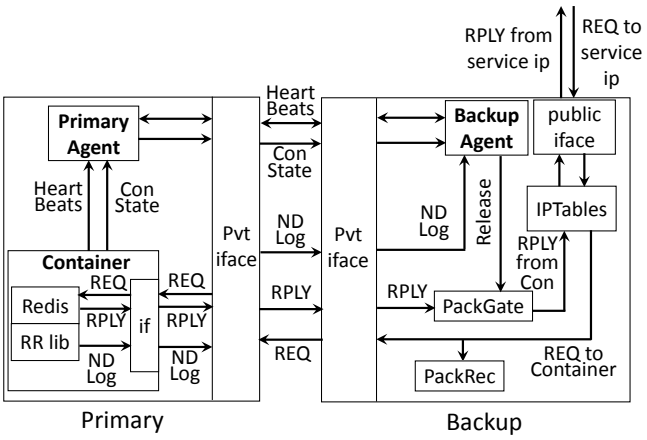

Figure 1: Architecture of *HyCoR*. (ND log: non-deterministic event log).

3 Overview of *HyCoR*

HyCoR provides fault tolerance by maintaining a primary-backup pair with an inactive backup that takes over when the primary fails. As discussed in §1, this is done using a hybrid of checkpointing and deterministic replay [27]. The basic checkpointing mechanism uses the implementation that we have obtained from the NiLiCon [54] authors.

Figure 1 shows the overall architecture of *HyCoR*. The primary records nondeterministic events: operations on locks and nondeterministic system calls. The record and replay are done at the user level, by instrumentation of glibc source code. When the primary executes, the instrumented code invokes functions in a dedicated RR (Record and Replay) library that create logs used for replay. There is a separate log for each lock. For each thread, there is a log of the nondeterministic system calls it invoked, with their arguments and return values. Details are presented in §4.1.

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

HyCoR does not guarantee recovery in the presence of data races. Specifically, unsynchronized accesses to shared memory during the epoch in which the primary fails may cause replay on the backup to fail to correctly reproduce the primary's execution, leading the backup to proactively terminate. However, *HyCoR* includes a simple "best-effort" mechanism that increases the probability of success in such circumstances for application with a low rate of unsynchronized accesses to shared memory (§4.6). With this mechanism, the order and timing of returns from nondeterministic system calls by *all* the threads is recorded during execution on the primary. During replay, the recorded order and relative timing are enforced.

If the primary fails, network connections must be main-

tained and migrated to the backup [20, 21, 23, 53]. Like CoRAL [20, 21], requests are routed through backup by advertising the service IP address in the backup. Unlike FT-TCP [23, 53] or CoRAL, replies are also routed through the backup, resulting in lower latency (§4.3).

As with most other state replication work [30, 46, 51, 54], *HyCoR* 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. Handling of primary failures have already been discussed. If the backup fails, the primary configures its network, advertises the service IP address, and communicates with the clients directly.

4 Implementation

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

The kernel modifications include: an ability to record and enforce the order of access to key data structures (§4.1); support for a few variables shared between the kernel and RR library, used to coordinate checkpointing with record and replay (§4.2); and a new queueing discipline kernel module used to pause and release network traffic (§4.3).

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. §4.6 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, *HyCoR* 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 is recorded to handle the trylock variants as well as errors. During replay, in most cases synchronization operations must actually be performed in order to properly enforce the correct semantics. However, if the recorded return value indicates that the thread failed to acquire the lock, the thread directly returns instead of trying to acquire the lock. For each lock, the total ordering of returns from each operation is enforced. This is not really necessary for reader-writer locks and trylock operations. However, it simplifies the implementation and there are minimal negative consequence since these events are relatively rare.

For each thread, there is a log of invoked system calls. The log entry includes the parameters and return values. During replay, the recorded parameters are used to detect divergence (replay failure). For some functions, such as gettimeofday(), replay does not involve the execution of the function and the recorded return values are returned. However, as discussed in §4.4, functions, such as open(), that involve the manipulation of kernel state, are actually executed during replay.

A key challenge is the replay of system calls that are causally dependent. These functions interact with the kernel and *HyCoR* does not replay synchronization within the kernel [39]. Thus, for example, if two threads invoke open() at approximately the same time, without user-level synchronization, the access within the kernel to the file descriptor table may occur in a different order during record and replay. As a result, during replay, each thread would not obtain the same file descriptor as it did during the original execution.

To meet the above challenge, *HyCoR* uses a modified version of the Rendezvous mechanism in Scribe [39]. Specifically, the kernel is modified to maintain an access sequence number for each shared kernel resource, such as the file descriptor table. Each thread registers the address of a per-thread variable with the kernel. When the thread executes a system call accessing a shared resource, the kernel increments the sequence number and copies its value to the registered address. At the user level, this sequence number is attached to the corresponding system call log entry. During replay, the before and after hooks enforces the recorded execution order.

4.2 Integrating Checkpointing with Record/Replay

Two aspects of *HyCoR* complicate the integration of checkpointing with record/replay: I) the RR library and the data structures it maintains are in the user level and are thus part of the state that is checkpointed and restored with the rest of the container state; and II) checkpointing is triggered by a timer in the agent, external to the container [54], and is thus not synchronized with the recording of nondeterministic events on the primary.

Based on the implementation described so far, the above complications can lead to the failure of *HyCoR* in two key scenarios: (1) a checkpoint may be triggered while the RR

library is executing code that must not be executed in the replay mode, such as sending the nondeterministic event log to the backup; (2) a checkpoint may be triggered while a thread's execution falls between the beginning of a before hook and the end of an after hook, potentially resulting in a state from which replay cannot properly proceed;

To handle Scenario (1), *HyCoR* prevents the checkpoint from occurring while any application thread is executing RR library code. Each thread registers with the kernel the address of a per-thread *in_rr* variable. In user mode, the RR library sets/clears the *in_rr* when it respectively enters/leaves the hook function. An addition to the kernel code that handles the *freezer* virtual signal (§2.1) prevents the thread from being paused if the thread's *in_rr* flag is set. However, the virtual signal remains pending. To prevent checkpointing from being unnecessarily delayed, after checkpointing is requested by the agent, threads are paused immediately before entering or after returning from RR library code. A *checkpointing* flag, shared between the agent that controls checkpointing and the RR library code, is used by the agent to indicate that checkpointing is requested, causing the RR library code to invoke a *do nothing* system call, thus allowing the virtual signal to pause the thread.

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

Replay is performed in the before hook (§4.1). During replay, if the after hook finds that *in_hook* is set, that indicates that checkpointing occurred between the before and after hooks. Thus, if *current_phase* is *replay* and *in_hook* is set, the after hook passes control back to the before hook. This allows the system call to be correctly replayed. For system calls that are actually executed during replay (§4.1), there is a need to determine whether the system call was actually invoked during the record phase. If it was, the system call must not be invoked again during replay. This required determination is accomplished based on the *syscall_skipped* flag.

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

4.3 Handling Network Traffic

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

Requirements (1) and (2) have been handled in connection with other mechanisms, such as [20, 21, 23, 53]. With *HyCoR*, this is done by mapping the advertised service IP address to the backup. Incoming packets are routed through the backup, where they are recorded by the PackRec thread in the agent, using the *libcap* library. Outgoing packets are also routed through the backup. To meet requirement (2), copies of the outgoing traffic are sent to the backup as part of the nondeterministic event log.

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

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

4.4 Transition to Live Execution

As with [34, 39] and unlike the deterministic replay tools for debugging [40, 48–50], *HyCoR* needs to transition from the replay mode to the live mode. The switch occurs when the backup replica finishes replaying the nondeterministic event log, specifically, when the last system call that generated an external output during the original execution is replayed. To identify this last call, after the checkpoint is restored, the RR library scans the nondeterministic event log and counts the number of system calls that generated an external output. Once replay starts, this count is decremented and the transition to live execution is triggered when the count reaches 0.

To support live execution, after replay, the kernel state must be consistent with the state of the container and with the state of the external world. For most kernel state, this is achieved by actually executing during replay system calls that change kernel state. For example, this is done for system calls that change the file descriptor table, such as open(), or change the memory allocation, such as mmap(). However, this approach does not work for system calls that interact with the external world. Specifically, in the context of *HyCoR*, these are reads and writes on sockets associated with a connection to an external client. As discussed in §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, *HyCoR* records the state changes on the sockets based on the nondeterministic event logs. When the replay phase completes, *HyCoR* updates all the sockets based on the recorded state. Specifically, the relevant components of socket state are: the last sent sequence number, the last acknowledged (by the client) sequence number, the last received (from the client) sequence number, the receive queue, and the write queue. The initial socket state is obtained from the checkpoint. The updates to the sent sequence number and the write queue contents are determined based on writes and sends in the nondeterministic event log. For the rest of the socket state, *HyCoR* cannot rely on the event log since some packets received and acknowledged by the kernel may not have been read by the application. Instead, *HyCoR* uses information obtained from PackRec (§4.3).

With respect to incoming packets, once the container transitions to live execution, *HyCoR* must provide to the container all the packets that were acknowledged by the primary but were not read by applications. During normal operation, on the backup host, PackRec keeps copies of incoming packets while PackGate extracts the acknowledgment numbers on each outgoing stream. If the primary fails, PackGate stops releasing outgoing packets and it thus has the last acknowledged sequence number of each incoming stream. Before the container is restored on the backup, PackRec copies the recorded incoming packets to a log. PackRec uses the information collected by PackGate to determine when it has all the required (acknowledged) incoming packets. Using the information from the nondeterministic event log and PackRec, before the transition to live execution, the packet repair mode (§2.1) is used to restore the socket state so that it is consist 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. To optimize performance, *HyCoR* includes performance optimizations, such as a specialized heap allocator for the logs and maintaining a list of logs that have been modified since the last time log entries were collected. However, such optimizations proved to be insufficient. Specifically, with one of our benchmarks, *Memcached*, under saturation, the performance overhead was approximately 300%.

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

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

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

When an application thread logs a system call that outputs

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

4.6 Mitigating the Impact of Data Races

Fundamentally, *HyCoR* is based on being able to identify all sources of non-determinism that are potentially externally visible, record their outcomes, and replay them when needed. This implies that applications are expected to be free of data races. However, since *HyCoR* only requires replay of short intervals (up to one epoch), it is inherently more tolerant to rarely manifested data races than schemes that rely on accurate replay of the entire execution [34]. As an addition to this inherent advantage of *HyCoR*, this section describes an optional mechanism in *HyCoR* that significantly increases the probability of correct recovery despite data races, as long as the manifestation rate is low.

HyCoR mitigates the impact of data races by adjusting the relative timing of the application threads during replay to approximately match the timing during the original execution. As a first step, in the record phase, the RR library records the order and the TSC (time stamp counter) value when a thread leaves the after hook of a system call. In the replay phase, the RR library enforces the recorded order on threads before they leave the after hook. As a second step, during replay, the RR library maintains the TSC value corresponding to the time when the after hook of the last-executed system call was exited. When a thread is about to leave a system call after hook, the RR library delays the thread until the difference between the current TSC and the TSC of the last system call is larger than the corresponding difference in the original execution. System calls are used as the basis for the timing adjustments since they are replayed (not executed) and are thus likely to cause the timing difference. This mechanism is evaluated in §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 [13] (version 1.0.1), a popular container runtime used in Docker.

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

Mechanisms like *HyCoR* are most useful for server applications. The mechanism is stressed by applications that manage significant state, execute frequent system calls and synchronization operations, and interact with clients at a high rate through many TCP connections. Hence, five of the benchmarks used were in-memory databases handling short requests: *Redis* [14], *Memcached* [11], *SSDB* [16], *Tarantool* [17] and *Aerospike* [2]. These benchmarks were evaluated with 50% read and 50% write requests to 100,000 100B records, driven by *YCSB* [29] clients. The number of user client threads ranged from 60 to 480.

The evaluation also included a web server, *Lighttpd* [8], and two batch PARSEC [25] benchmarks: *swaptions* and *streamcluster*. *Lighttpd* was evaluated using 20-40 clients retrieving a 1KB static page. For *Lighttpd*, benchmarking tools SIEGE [15], ab [1] and wget [6] were used to evaluate, respectively, the performance overhead, response latency, and recovery rate. *Swaptions* and *streamclusters* were evaluated using the native input test suites.

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

For *swaptions* and *streamcluster*, recovery was considered successful if the output was identical to the golden copy. For *Lighttpd*, we used multiple wget instances that concurrently fetched a static page. Recovery was considered successful if all the fetched pages were identical to the golden copy. For the in-memory database benchmarks, the *YCSB* clients could not be used since they do not verify the contents of the replies and thus could not truly validate correct operation. Instead, we developed customized clients, using existing client libraries [3, 10, 12, 18], that spawns multiple threads and let each thread work on separate set of database records. Each thread records the value it stores with each key, compares that value with the value returned by the corresponding get operation and flags an error if there is a mismatch. Recovery was considered successful if no errors were reported.

A possible concern with the customized client programs is that, due to threads working on separate sets of database records, lock contention is reduced and this could skew the results. We compared the recovery rate and recovery latency results of the customized clients with the *YCSB* clients. For the *YCSB* clients, recovery was considered successful if replay succeeded and the clients finished without reporting errors.

The results were similar: the recovery rate difference was less than 2% and the recovery latency difference was less than 5%. In §6.3, we report the more robust results obtained with the customized client programs.

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

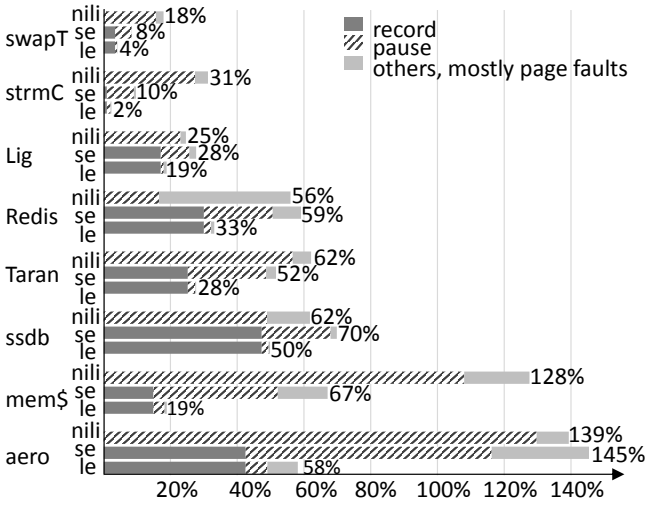

Figure 2: Performance overheads: NiLiCon, *HyCoR-SE*, *HyCoR-LE*.

6 Evaluation

This section presents *HyCoR*'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 *HyCoR* are evaluated: *HyCoR-SE* (short epoch) and *HyCoR-LE* (long epoch), with epoch durations of 100ms and 1s, respectively. Setting the epoch duration is a tradeoff between the lower overhead with long epochs and the lower susceptibility to data races and lower recovery time with short epochs. Hence, *HyCoR-LE* may be used if there is high confidence that the applications are free of data races. Thus, with the *HyCoR-SE* configuration, the data race mitigation mechanism described in §4.6 is turned on, while it is turned off for *HyCoR-LE*.

HyCoR 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 [54]. The short epochs of NiLiCon are required since, unlike *HyCoR*, the epoch duration with NiLiCon determines the added latency in replying to client requests (§2.1). In all cases, the "stock setup" is the application running in an unreplicated container.

6.1 Overheads: Performance, CPU Utilization

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

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

With four of the server benchmarks, the number of the worker threads cannot be configured (*Lighttpd*, *Redis*: 1, *Tarantool*: 2, *SSDB*: 12). *Mecached*, *Aerospike* were configured to run with four worker threads. For these applications, the number of the worker threads is set to four because, with our experimental setup, it was not possible to generate enough client traffic from YCSB to saturate more than four worker threads. For consistency, the two non-interactive benchmarks were also configured to run with four threads.

For each benchmark, the workload that saturates the cores in the stock setup was used for the stock, *HyCoR*, and NiLiCon setups. With NiLiCon, due to the long latencies it normally incurs for server responses (§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 2 shows the performance overheads of NiLiCon, *HyCoR-SE*, and *HyCoR-LE*, with the breakdown of the sources of over-

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

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

Table 1: CPU utilization overhead for *HyCoR-SE*. LogTH: logging thread. KerNet: kernel's handling of network packets. P: primary host and B: backup host.

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

Table 2: Response Latency in μs. S: Stock, H: *HyCoR-SE*, N: NiLiCon

head. Each benchmark was executed 50 times. The margin of error of the 95% confidence interval was less than 2%.

The record overhead is caused by the RR library recording non-deterministic events. The pause overhead is due to the time the container is paused for checkpointing. The page fault overhead is caused by the page fault exceptions that track the memory state changes of each epoch (§2.1).

As shown by a comparison of the results for *HyCoR-SE* and *HyCoR-LE*, due to locality in accesses to pages, the pause and page fault overheads decrease as the epoch duration is increased. This comparison also shows that the fact that the data race mitigation mechanism is on with *HyCoR-SE* and of with *HyCoR-LE*, has no significant impact on the record overhead. With *HyCoR-SE*, the average incremental checkpoint size per epoch was 0.2MB for *Swaptions*, 15.6MB for *Redis*, and 41.2MB for *Aerospike*, partially explaining the differences in pause overhead, which is also affected by the time to obtain required kernel state [54]. With *HyCoR-SE*, the average number of logged lock operations plus system calls per epoch was 9 with *streamcluster*, 907 with *Tarantool*, and 2137 with *Aerospike*, partially explaining the differences in record overhead. However, the overhead of logging system calls is much higher than for lock operations. *Memcached* is comparable to *Aerospike* in terms of the rate of logged system calls plus lock operations, but has 341 compared to 881 logged system calls per epoch and thus lower record overhead.

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

Table 1 shows a breakdown of CPU utilization overhead with *HyCoR-SE*. The "others" row for the primary is the overhead for recording the nondeterministic events, handling the page fault exceptions for tracking memory changes, and collecting and sending the incremental checkpoints. The "others" row for the backup is the overhead for receiving and storing the nondeterministic event logs and the checkpoints. The KerNet row for the backup is the overhead for packet handling in the kernel, that includes the routing of requests and responses to/from the primary and the PackGate module.

In terms of CPU utilization overhead, the worst case is with *Redis*. A significant factor is the overhead for packet handling in the backup kernel (KerNet). We have found that this overhead is mostly due to routing, not PackGate. *Redis* involves only one worker thread and it receives and sends a large number of small packets, leading to this overhead. Techniques for optimizing software routing [38] can be used to reduce this overhead.

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

6.2 Response Latency

A key advantage of *HyCoR* compared to schemes based on checkpointing alone, such as Remus [30] and NiLiCon [54] is significantly lower response latency. Table 2 shows the response latencies with the stock setup, *HyCoR-SE* and NiLiCon. The number of client threads for stock and *HyCoR-SE* is separately adjusted so that the CPU load on the cores running application worker threads is approximately 50%. For NiLiCon, due to its long response latencies, it is not possible to reach 50% CPU usage. Instead, NiLiCon is evaluated with the same number of client threads as *HyCoR-SE*, resulting in CPU utilization of less than 5%, thus favoring NiLiCon. To evaluate the impact of response size, *Lighttpd* is evaluated serving both 1KB as well as 100KB files. Each benchmark is

executed 50 times. We report the average of the mean and the 99th percentile latencies of the different runs. For the average response latencies, the 95% confidence interval has a margin of error of less than 5%. For the 99th percentile latencies, it is less than 15%.

With *HyCoR*, there are three potential sources for the increase in response latency: forwarding packets through the backup, the need to delay packet release until the corresponding event log is received by the backup, and increased request processing time on the primary. With *HyCoR-SE*, the increase in average latency is only 156μs to 581μs. The worst case is with *Aerospike*, which has the highest processing overhead (Fig. 2) and a high rate of nondeterministic events and thus long logs that have to be transferred to the backup. The increase in 99th percentile latency is 371μs to 6ms. The worst case is with *Lighttpd* serving a 100KB file. This is because the request service time is much longer than with the other benchmarks and thus a checkpoint is more likely to happen in the middle of this time. The pause time for checkpoint of this benchmark is approximately 6ms. It should be noted that, in terms of increase in response latency, NiLiCon is not competitive, as also indicated by the results in [54].

With *HyCoR-LE*, the increase in the average response latency is from 40μs to only 343μs, due to the the lower processing overhead. The increase in the 99th percentile latency is under 534μs since checkpoint are much less frequent and thus less likely to interrupt the processing of a request.

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 *HyCoR*, 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 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. The 95% confidence interval margin of error is less than 0.1%.

Primary Failure Recovery Rate. Three of our benchmarks contain data races that may cause recovery failure: *Memcached*, *Aerospike* and *Tarantool*. Running *Tarantool* with *HyCoR-SE*, through 50 runs of fault injection in the primary, we find that, due to data races, in all cases replay fails and thus recovery fails. Due to the high rate of data race manifestation, this is the case even with the mechanism described in §4.6. Thus, we use a modified version of *Tarantool* in which the data races are eliminated by manually adding locks.

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

Table 3: Recovery rate and replay time (in ms). *HyCoR* with different levels of mitigation of data race impact.

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

The second set of benchmarks consists of *Memcached* and *Aerospike*, used to evaluate the the data race mitigation mechanisms (§4.6). For these, to ensure statistically significant results, 1000 fault injection runs are performed with each benchmark with each setup. The results are presented in Table 3. For both the recovery rate and replay time, the 95% confidence interval is less than 1%. Without the §4.6 mechanism, the recovery rate for *HyCoR-LE* is much lower than with *HyCoR-SE*, demonstrating the benefit of short epochs and thus shorter replay times. Enforcing a total order of the recorded system calls in the after hook is not effective for *Memcached* but increases the recovery rate of *Aerospike* for both *HyCoR* setups. However, with the timing adjustments, both benchmarks achieve high recovery rates, even with *HyCoR-LE*. The total order of the system calls is the main factor that increase the replay time. Thus, there is no reason to not also enable the timing adjustments.

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

As an additional validation of *HyCoR*, the three bench-

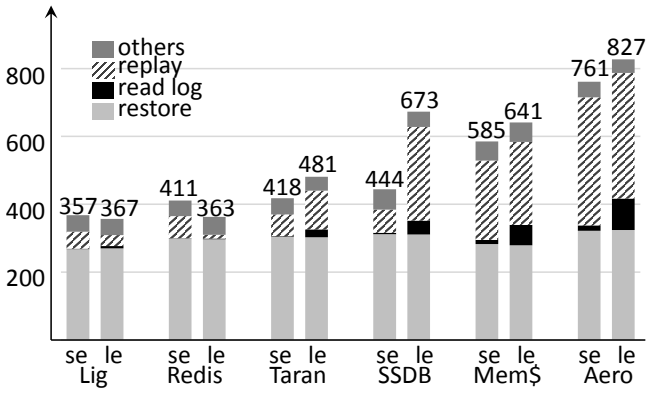

Figure 3: Recovery Latency (ms) breakdown with *HyCoR-SE* and *HyCoR-LE*.

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

Primary Failure Recovery Latency. Figure 3 shows a breakdown of the factors that make up the recovery latency with *HyCoR-SE* and *HyCoR-LE*. The batch benchmarks, *swaptions* and *streamcluster*, are not included since their execution times are above 60s, so their recovery latency is insignificant. With *HyCoR-SE*, the data race mitigation scheme is enabled, while with *HyCoR-LE* it is disabled. The 95% confidence interval margin of error is less than 5%. *Restore* is the time to restore the checkpoint, mostly for restoring the in-kernel states of the container (e.g., mount points and namespaces). *Read log* is the time to process the stored logs in preparation for replay. *Others* include the time to send ARP requests and connect the backup container network interface to the bridge.

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

7 Limitations

We have identified one inherent limitation of *HyCoR* and four limitations of the current research prototype implementation. An inherent limitation is that the mechanism used for mitigating the impact of data races (§4.6) is incapable of handling a high rate of racy accesses (§6.3). However, as discussed in §6.3, such data races are easily detectable and are thus easy to eliminate, even in legacy applications.

HyCoR does not currently support multiple processes. To overcome this limitation, the RR library would need significant enhancements, such as support for inter-process communications via shared memory. Techniques presented in [24] may be applicable. *HyCoR* also does not handle asynchronous signals. This may be resolved by techniques used in [39], that delay signal delivery until a system call or certain page faults.

HyCoR does not handle C atomic types, functions, intrinsics and inline assembly code that performs atomic operations transparently. In this work, such cases were handled by protecting such operations with locks. Specifically, this was done for *Aerospike* and *glibc*. Compiler support [42] is needed to overcome this limitation.

Recovery currently fails if a socket is created via accept() or connect() during replay. Resolving this limitation would require recording and restoring during replay various socket state components, such as the timestamp and window scale. This can be done by enhancing the RR library with code used in NiLiCon [54] to checkpoint and restore socket state.

8 Related Work

HyCoR builds on prior works on fault tolerance using replication, replication based on high-frequency checkpointing, replication based on deterministic replay, and network connection failover. In the works cited, a few are validated using fault injection [21, 30, 31, 41, 54].

As with *HyCoR*, there are other replay systems that support replay from a checkpointed state [39, 40, 48, 49] and the transition from replay to live execution [34, 39]. There are various works aimed at replay in the presence of data races while only recording system calls and synchronization operations [22, 40, 44, 50]. However, these works either require delaying outputs until the end of a long (tens of ms) epoch [40, 50], or requires lengthy offline processing to enumerate the order of memory accesses involved in the data races [22, 44], making them unsuitable for fault tolerance.

Early work on VM replication for fault tolerance is based on leader-follower active replication using deterministic replay [26]. This is combined with periodic checkpointing in [27], based on earlier work on using these technique for debugging [37]. Both of these works are focused on uniprocessor systems. Extending them to multiprocessors is impractical, due to the overhead of recording shared memory access order in the hypervisor [33,45]. Remus [30] focuses on multiprocessors and implements VM replication for fault tolerance using high-frequency checkpointing alone (§2.1). Tardigrade [41] applies Remus's algorithm to a lightweight VMs based on a library OS. NiLiCon [54] applies Remus's algorithm to containers. Phantasy [46] uses hardware features, PML and RDMA, to optimize Remus. Plover [51] 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 [31] uses active VM replication, comparing outputs before release to the client. On a mismatch, the state of one VM is updated with the other's. There is no mechanism to ensure that the backup's execution matches the primary's, resulting in high performance overhead and long response latencies for applications with significant nondeterminism.

Another set of works use deterministic replay, discussed in §2.2, for active replication of multiprocessor workloads [34, 40, 42]. As with *HyCoR*, the primary records the outcomes of nondeterministic events and logs them to the backup. Rex [34] requires the application to be data race free and requires manual modifications of the application source code to use a specified API. As with *HyCoR*, output to the external world can be released only after the backup receives the non-deterministic log. Execution divergence, due to a data race or some other unlogged nondeterministic event, can cause failure. Castor [42] handles data races by buffering the output to the external world until the backup finishes replaying the associated log. If divergence, due to a data race, prevents the backup from continuing replay, the backup's state is synchronized with the primary's.

Comparing *HyCoR* with Rex and Castor, the key difference is the use of checkpointing versus an active replica. The disadvantages of *HyCoR* are periodic pauses for checkpointing during normal operation and higher recovery latency. To explain the advantages of *HyCoR*, two cases are considered: (1) the applications are assumed to be free of data races, and (2) there may be some data races. For the former case, it should be noted that, for applications with a large number of synchronization operations, replay may be slower than the original execution. Thus, under heavy load, the active replica is a performance bottleneck [34]. For example, both Rex and *HyCoR* are evaluated with *Memcached*, and the performance overheads are 40% and 19%, respectively.

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, with Castor as with *HyCoR*, a data race can cause recovery to fail. Specifically, with Castor, if the primary fails while transferring its state to the backup, 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 backup state and thus have low recovery rate (like *HyCoR*) and also high performance overhead. For applications with a lower rate of racy memory accesses, such as *Memcached* and *Aerospike*, execution divergence is less likely. Whether *HyCoR* or Castor have higher recovery rate depends on the rate of execution divergence and the cost of synchronizing the state. For example, based on the recovery rate for *Memcached* shown in Table 3 for the "stock" *HyCoR-SE*, the probability of execution divergence in 50ms (half an epoch) 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 99.5% with *HyCoR* (Table 3). 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 *HyCoR*. For programs with higher memory working set, the state transfer time would be larger and thus the recovery rate advantage of *HyCoR-SE* would also be larger.

9 Conclusion

HyCoR is a unique point in the design space of application-transparent fault tolerance schemes for multiprocessor workloads. By combining checkpointing with externally deterministic replay, it facilitates trading off performance and resource overheads with vulnerability to data races and recovery latency. Critically, the response latency is not determined by the frequency of checkpointing, and sub-millisecond added delay is achieved with all our server applications. As we have found (§6.3), legacy applications may still have data races. *HyCoR* targets data races that are most likely to remain undetected and uncorrected, namely, rarely-manifested data races. Unlike mechanism based strictly on active replication and deterministic replay [34], *HyCoR* is not affected by data races that manifest during normal operation, long before failure. For handling data races that manifest right before failure, *HyCoR* introduces a simple best effort mechanism that significantly reduces of the probability of the data races causing recovery failure. *HyCoR* is a full fault tolerance mechanism. It can recover from primary or backup host failure and includes transparent failover of TCP connections.

This paper describes key implementation challenges encountered in the development of *HyCoR* and outlines their resolution. The extensive evaluation of *HyCoR*, based on eight benchmarks, included performance and resource overheads, impact on response latency, as well as recovery rate and latency. The recovery rate evaluation, based on fault injection, subjected *HyCoR* to particularly harsh conditions by intentionally perturbing the scheduling on the primary, thus challenging the deterministic replay mechanism (§5). With high checkpointing frequency (*HyCoR-SE*), *HyCoR*'s throughput overhead is less than 68% for seven of our benchmarks and 145% for the eighth. If the applications are known to be data race free, with a lower checkpointing frequency (*HyCoR-LE*), the overhead is less than 59% for all benchmarks, significantly outperforming NiLiCon [54]. With data race free applications, *HyCoR* recovered from all fail-stop failures. With two applications with infrequently-manifested data races, the recovery rate was over 99.4% with *HyCoR-SE*.

References

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

[2] Aerospike. https://https://www.aerospike.com/. Accessed: 2020-08-07.

[3] Aerospike C Client. https://www.aerospike.com/apidocs/c/. Accessed: 2020-08-07.

[4] cgroup freezer. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt. Accessed: 2020-08-07.

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

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

[7] Google threadsanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual. Accessed: 2020-08-07.

[8] Home - Lighttpd. https://www.lighttpd.net/. Accessed: 2020-08-07.

[9] 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.

[10] libMemcached. https://libmemcached.org/libMemcached.html. Accessed: 2020-08-07.

[11] memcached. https://memcached.org. Accessed: 2020-08-07.

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

[13] opencontainers/runc. https://github.com/opencontainers/runc. Accessed: 2020-08-07.

[14] Redis. https://redis.io. Accessed: 2020-08-07.

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

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

[17] Tarantool - In-memory DataBase. https://tarantool.io. Accessed: 2020-08-07.

[18] Tarantool C client libraries. https://github.com/tarantool/tarantool-c. Accessed: 2020-08-07.

[19] TCP connection repair. https://lwn.net/Articles/495304/. Accessed: 2020-08-07.

[20] 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.

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

[22] 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.

[23] 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.

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.