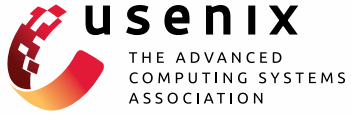

Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services

Jacob R. Lorch and Andrew Baumann, ***Microsoft Research;***
Lisa Glendenning, ***University of Washington;***
Dutch Meyer and Andrew Warfield, ***University of British Columbia***

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lorch

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services

Jacob R. Lorch, Andrew Baumann, Lisa Glendenning*, Dutch T. Meyer†, and Andrew Warfield†
*Microsoft Research, *University of Washington, †University of British Columbia*

Abstract

Many services need to survive machine failures, but designing and deploying fault-tolerant services can be difficult and error-prone. In this work, we present Tardigrade, a system that deploys an existing, unmodified binary as a fault-tolerant service. Tardigrade replicates the service on several machines so that it continues running even when some of them fail. Yet, it keeps the service states synchronized so clients see strongly consistent results. To achieve this efficiently, we use *lightweight virtual machine replication*. A lightweight virtual machine is a process sandboxed so that its external dependencies are completely encapsulated, enabling it to be migrated across machines. To let unmodified binaries run within such a sandbox, the sandbox also contains a library OS providing the expected API. We evaluate Tardigrade's performance and demonstrate its applicability to a variety of services, showing that it can convert these services into fault-tolerant ones transparently and efficiently.

1 Introduction

Tolerating machine failure is a key requirement of many services, but achieving this goal remains frustratingly complex. Many services that do not otherwise require a distributed system must, in some form, consistently replicate the critical aspects of their system state on different hosts. This requirement is particularly burdensome for simple applications that could, if not for the risk associated with a single point of failure, be deployed on a single machine running commodity software.

Tools exist to support developers writing fault-tolerant services, such as replicated state machine libraries [6, 22] and coordination services for consistent metadata storage [5, 18]. However, even when this is possible, supporting the semantics of these tools requires the efforts of expert systems designers, and puts significant demands on the service's design.

A promising alternative is asynchronous virtual machine replication (VMR), as used in the Remus system [11]. This approach transparently protects an arbitrary service by running it in a replicated VM. Externally observable consistency is achieved by buffering network output until a checkpoint of the system state that created the output has been replicated. Output buffering means that client-perceived latency will increase as the time to capture and disseminate a checkpoint increases, motivating techniques to reduce the size of these checkpoints.

To address this, we introduce the concept of asynchronous *lightweight* VM replication (LVMR), which uses lightweight virtual machines (LVMs) in place of VMs [3, 30]. A lightweight VM provides encapsulation with a smaller memory footprint because background operating system services are outside of the container. This substantially reduces the time to create and replicate checkpoints, leading to a reduction in both service latency and replication bandwidth.

An LVM has a higher-level interface between guest and host than a VM, so some techniques used in VMR do not directly translate. We implement LVMR as an extension that interposes on an existing, general binary interface between an LVM guest and host. This requires dealing with non-determinism in the interface, using existing calls to quiesce the system so a consistent snapshot can be captured, and checkpointing through the interface.

To demonstrate the practicality of our design, we implement it as a system we call Tardigrade. We show that, through reasonable optimizations like in-memory checkpointing, identification of hot pages, and delta encoding, the cost of checkpointing can be made low. We find that client-perceived latency impact for a simple application is $\sim$11 ms on average, with a 99.9th quantile latency under 20 ms. Furthermore, this latency does not skyrocket when external processes like OS updates run on the host.

Tardigrade uses primary-backup replication to survive as many faults as there are backups. These faults must be external to the service, e.g., power loss, disconnection, or system crash, since replication cannot mask faults that cause the primary to corrupt replicated state. Instead of relying on synchrony assumptions, we use a variant of Vertical Paxos [20] for automatic failure recovery. This permits the use of an unreliable failure detector to decide when to fail over the active replica, and allows replicas to communicate over a standard network.

A key design goal of Tardigrade is that it permits simpler design of fault-tolerant systems. We demonstrate this by encapsulating and evaluating three existing services that were developed without fault tolerance as a first concern.

In summary, the contributions of this paper are:

- We introduce the idea of asynchronous LVM replication (LVMR) and describe a complete design of a system supporting it.
- We illustrate the practicality of our design by implementing it in the Tardigrade system, applying optimizations to make it performant, and evaluating the resulting performance.
- We demonstrate how LVMR makes it easy to deploy fault-tolerant services. To show this, we replicate the FDS metadata service [27], ZooKeeper, and Apache without changing their binaries.

2 Background and Motivation

In this section, we provide background needed to understand the motivation and design of our system. First, §2.1 evaluates the cost of replicating OS background state in VMR. §2.2 describes the concept of a lightweight VM (LVM), which we use in Tardigrade. Finally, §2.3 overviews the specific LVM system used by Tardigrade, Bascule [3], which we chose because of its extensibility.

2.1 Overheads in Asynchronous Virtual Machine Replication

Remus [11] introduced asynchronous virtual machine replication (VMR). In VMR, the protected guest software is encapsulated in a VM, and snapshots of its state are frequently sent to a backup host across the network. On the backup, the VM image is resident in memory and begins execution immediately upon primary failure.

The amount of time the guest is suspended during the snapshot is minimized using *speculative execution* [26], in which the guest executes while the most recent snapshot of its state is asynchronously replicated. To prevent externally-observable inconsistencies, Remus buffers the output of speculative execution, i.e., network packets, and releases it only when the state that produced it is durably replicated. This mechanism bounds the minimum latency of the system observed by clients by the amount of time required to take and replicate a snapshot.

To understand these overheads, we test two Xen-based hosts connected by a 1 Gb/s network. We use RemusDB, the highest performing version of Remus available, to protect a Windows Server 2012 guest.

First, we measure the cost of the suspend/resume operation Remus uses in isolation—without replication or network buffering—on an idle guest. We find it is 10 ms regardless of checkpoint interval. This is due to the overhead of suspending an unmodified guest VM, which requires the guest and each virtualized device to synchronize its internal state, e.g., flushing processor caches to RAM via an ACPI interrupt. Note that Linux can be paravirtualized to perform this synchronization much more quickly, in <1 ms.

Next, we measure the effect of common OS background tasks on latency. Figure 1 evaluates ping response

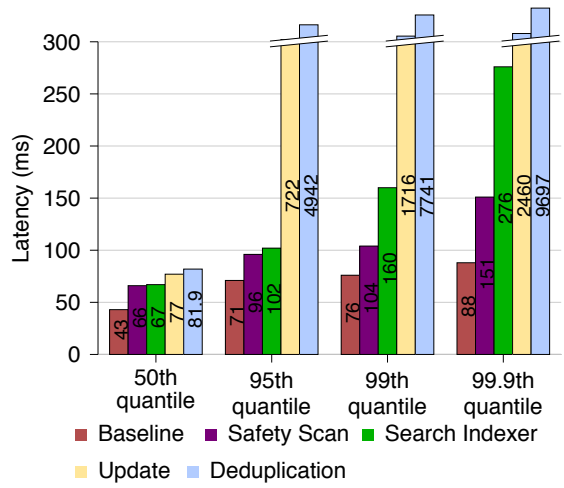

Figure 1: Effect of background tasks on ping latency of Windows Server 2012 under Remus protection

time for both an idle baseline and when a single background OS task is running. We set a 50-ms checkpoint interval for each case. This is a conservative choice. While a lower interval would provide improved latency, the benefits would be seen across all configurations, and the workload throughput would suffer because the experiment would spend more time in a suspended state.

The four background tasks we evaluate are common and important services for Windows file servers:

1. *Safety Scanner* protects the OS against malware.
2. *Search Indexer* manages an index of file contents and properties to optimize lookups.
3. *Windows Update* fetches and applies critical patches to the OS.
4. *Single Instance Storage* deduplication improves storage efficiency.

The most costly background activity we measure is deduplication, which shows maximum incremental checkpoint sizes of 691 MB after compression. This causes delays of more than seven seconds to commit states, during which all communication is buffered. This is the primary contributor to the high ping times observed in all tests. Even Safety Scanner, which dirties memory at the most modest rate, still produces a more than 50% increase in median latency.

These results support the intuition that checkpointing the state of non-critical OS background services in a VM has a significant cost in both replication bandwidth and service latency.

2.2 Lightweight VMs

A traditional virtual machine provides the abstraction of a dedicated machine complete with kernel mode, multiple address spaces, and virtual hardware devices. It is therefore able to run traditional OSes, perhaps lightly modified for paravirtualization, and can host multiple applications. In contrast, a lightweight VM (LVM) is con-

structed from a single isolated user-mode address space, referred to as a *picoprocess* [13]. An LVM typically runs only a single application along with a library OS (LibOS), which provides the application with the APIs on which it depends. Past work on Drawbridge [30] refactored an existing monolithic OS, Windows, to create a self-contained LibOS running in a picoprocess yet still supporting rich desktop applications.

Despite being able to run unmodified applications, an LVM typically has substantially lower overhead than full VMs. This is because it elides most OS components not needed to implement application-facing interfaces, such as the file system, device drivers, and service processes. For example, the Drawbridge authors reported a disk footprint of 64MB and working set of 16MB for their Windows 7 LibOS [30]. Compared to typical VM footprints measured in gigabytes, this small scale makes lightweight VMs attractive for efficient replication.

2.3 Bascule

Bascule [3] is an architecture for LibOS extensions based on Drawbridge. It defines a narrow binary interface, the *Bascule ABI*, consisting of 40 downcalls and 3 upcalls implementing primitive OS abstractions: virtual memory allocation and protection, exception handling, threads and synchronization mechanisms, and finally an I/O stream abstraction used for files and network sockets. All interaction between a LibOS and the host must traverse the ABI. Bascule extensions, such as our checkpointer (§3.3), are loaded in-process with the LibOS and application, and interpose on the ABI. Since the Bascule ABI is designed to be independent of host OS and guest LibOS, and to enable arbitrary nesting of implementations, extensions support a variety of platforms, and may be composed at runtime.

3 Design

This section presents the design of Tardigrade. We start with an architectural overview, then discuss each of the pieces of that architecture in turn. Our design assumes a fail-stop model: server machines will fail only by stopping, not by acting arbitrarily.

3.1 Overview

Figure 2 illustrates the architecture of the Tardigrade system. On each machine, we run an *instance* that will at various times act as a primary service replica, a backup service replica, or a spare. The *orchestrator* coordinates these instances to ensure they act in concert as a consistent, fault-tolerant service, using a variant of Vertical Paxos [20]. The orchestrator uses an unreliable failure detector to get hints about which instances have failed.

Each instance makes use of two subcomponents to do its job: a Bascule component consisting of a host and guest, and a *network filter*. The Bascule component runs the service in a lightweight VM. The network filter only releases guest output when the checkpoint of a state following its generation has been durably replicated.

The Bascule guest contains, like typical Bascule guests, an unmodified application running atop a library OS mimicking the OS the application expects. We leave these components unchanged, and add a *checkpointer* below the library OS that lets the Bascule guest checkpoint its state or restore its state from a checkpoint.

3.2 Orchestration

The orchestrator manages the instances using the Vertical Paxos protocol [20] and is divided into two components: the unreliable failure detector and the view manager. Note that our terminology differs slightly from that of Vertical Paxos: we call the master an *orchestrator* and call ballots *views*.

A *checkpoint* is a snapshot of the LVM's state. A *full* checkpoint is a self-contained checkpoint, while an *incremental* checkpoint reflects only changes that have been made to the LVM during an inter-checkpoint interval, also known as an *epoch*. An incremental checkpoint thus describes how to go from a *pre-state* to a *post-state*.

A *view* is an assignment of roles to instances; instances can take on three roles. When *primary* it runs the service and responds to client requests. When *backup* it records checkpoints of the primary's state so that it can become primary if needed. When *spare* it simply waits until it is needed as a primary or backup.

The primary of the first view starts a fresh LVM and disseminates a full checkpoint, then transitions to periodically taking incremental checkpoints. Checkpointing involves the following steps: *quiescing* the guest, capturing the checkpoint, resuming guest execution, and sending the checkpoint to backups.

A received checkpoint is *applicable* at a backup if the backup can restore it. A full checkpoint is always applicable, and an incremental checkpoint is applicable if the backup can recreate the incremental checkpoint's pre-state. For instance, if a backup has a full checkpoint and the three incremental ones following it, then all four of these are applicable.

Once a checkpoint is applicable at all backups, it is *stable*. That is, as long as one of the backups or the primary remains alive, the system can proceed.

When the primary learns that a checkpoint is stable, it *decides* the checkpoint. That is, it considers the checkpoint to describe the next official state in the sequence of service states. Thus the service's logical lifetime is divided into epochs punctuated by decided checkpoints; we call an epoch decided when its ending checkpoint is decided. Once an epoch has been decided, it is safe to send any network packets the primary generated during that epoch, because it will never be necessary to roll back to an earlier state.

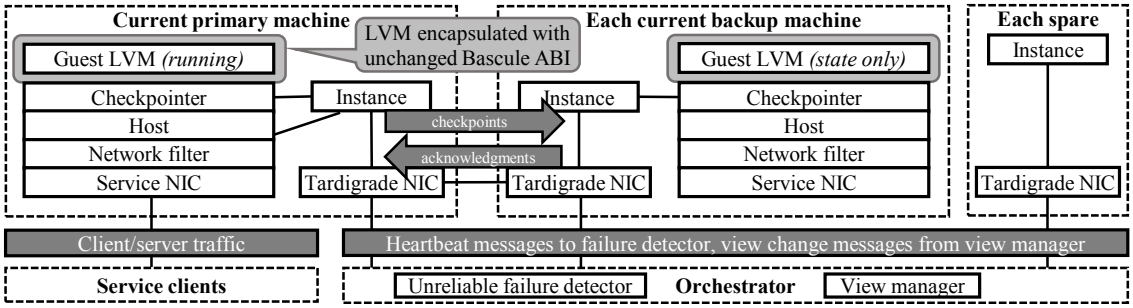

Figure 2: Overview of Tardigrade architecture for replicating lightweight virtual machines (LVMs)

When a backup learns that a checkpoint is decided, it can *apply* the checkpoint to its state. For a full checkpoint, the backup starts a new LVM and initializes its state to match. For an incremental checkpoint, the backup performs the operations necessary to transform the current pre-state into the post-state. We implement a queue of applicable checkpoints at a backup that is processed asynchronously with sending acknowledgements to the primary; the queue size tuning knob balances latency of checkpoint acknowledgements in the steady state with recovery time in the relatively rare failover event.

Each instance periodically sends a heartbeat to the orchestrator. After detecting a primary or backup failure, the orchestrator proposes a new view, which includes the identity of the new primary and backups. The new primary is, if available, the primary from the previous view; otherwise, it is a backup from the previous view.

If the new primary was previously a backup, it stops accepting checkpoints from the previous view and finishes applying checkpoints it has acknowledged. The new primary then takes a full checkpoint, disseminates it to the new backups, then asks the orchestrator to *activate* the view. The orchestrator activates the view by deciding that it follows the previous active view; then, the new primary considers the initial checkpoint stable and proceeds with further checkpoints.

When a backup is elevated to primary, the initial state the backup uses is guaranteed to match a state of the previous view's primary at or beyond the last state the primary knew to be stable. However, in the case it is beyond the last state the primary knew to be stable, the primary will not have released the network traffic generated between its last stable checkpoint and the backup's initial state. Note that this same technique was used in Remus [11]; the insight is that losing the network output of state that was successfully replicated mimics the case of the actual network dropping packets, and services are typically written to be robust to unreliable networks.

When a spare gets a message with the initial full checkpoint of a new view, it saves and acknowledges the checkpoint. When the primary later tells the spare that the view is activated, the spare becomes a backup.

Note that an orchestrator may propose a new view but never activate it. If machines fail during the view change, the orchestrator may propose a different new view to succeed the current view. It will eventually activate only one, and the initial checkpoints of the aborted views can be discarded.

3.3 Checkpointer

Our checkpointer is designed as a Bascule extension and is responsible for both capturing and applying checkpoints. In Bascule, a guest LibOS uses a PAL to translate the guest's ABI calls into underlying system calls. The checkpointer extends the system by interposing between the two. In other words, it provides the Bascule ABI to the guest, and satisfies the requests the guest makes by passing them on to the PAL. From this position, it can track all system objects (e.g., files, threads, synchronizers) that the guest uses, and it can virtualize system objects so that they are portable across machines.

A naive checkpoint would be a list of all ABI calls made by the guest and a snapshot of its CPU state and memory contents. However, this is impractical for two reasons. First, it would produce extremely large checkpoints. Second, ABI calls are not deterministic, so just replaying them will not necessarily bring about the same state on the new machine. Instead, the checkpointer inspects the current state and produces a list of actions that can reproduce that state.

3.3.1 Memory tracking

The checkpointer tracks the following information for each memory region allocated by the LVM: location, protection, and which pages may have been modified during the current epoch. This metadata is stored in an AVL tree where each node represents a memory region.

Identifying a subset of memory that has not been modified in the preceding epoch is essential for generating efficient incremental checkpoints. Ideally the checkpointer would use hardware such as page-table dirty bits for this, but it is not accessible through the Bascule ABI. Instead, the checkpointer uses a standard technique for tracking memory modifications. First, before each epoch the checkpointer write-protects all writable pages using the `VirtualMemoryProtect` ABI call. During the

epoch, when the guest writes a protected page, the checkpointer intercepts the triggered access violation exception. The exception handler restores the original page protection, sets the corresponding dirty bit in the metadata tree, and suppresses the exception from the guest.

Many optimizations of this general design are possible; §4.1 discusses the optimizations we implemented.

One complication is that although we can suppress access violation exceptions incurred by the guest, we cannot suppress exceptions incurred by the host. This can happen when the guest passes the host a pointer to memory the checkpointer has write-protected. So, before passing any guest pointers to the host, the checkpointer ensures that they are not write-protected. For pointers to small objects, like an integer, the checkpointer simply substitutes pointers to its own stack, and then copies the values into the guest-supplied pointers when the call returns. For pointers to large objects, like a buffer, the checkpointer touches all the pages in advance so that any exceptions are incurred by it rather than the host.

3.3.2 File-change tracking

For each mutable private file system (FS) of the LVM, the checkpointer tracks which parts have potentially changed during the last epoch. For each file, the checkpointer tracks possible changes to its existence, its metadata, and its blocks. Note that it does not track the actual contents of those changes, as they can be read directly from the FS during checkpointing.

Operations that can potentially change a file include open, delete, rename, map, and write. However, all write ABI calls are asynchronous, so the checkpointer tracks changes due to a write only when the call has completed. Before a write completes, a checkpoint will capture the ongoing write to replay on restore, so there is no need to capture the actual change to the file. For similar reasons, the checkpointer does not track changes due to mapped files until the region is unmapped.

3.3.3 Quiescence

To take a consistent snapshot of an LVM's state, each of its threads must first *quiesce*, i.e., pause so that it stops mutating state. Additionally, because the Bascule ABI does not provide a way for one thread to capture another's state, each quiescing thread must also capture its own state with standard x86 instructions and store it in a location accessible to the checkpointer.

We use different methods to quiesce a thread, depending on which of three states it is in: the middle of a blocking ABI call to the host, in the middle of a non-blocking but nevertheless uninterruptible operation, or neither. A thread in the latter state is quiesced by raising an interrupt in the thread; the checkpointer's interrupt handler will initiate quiescence. Handling the first two states is more involved.

Before a thread enters a non-blocking but uninterruptible state, such as a non-blocking ABI call or a code section that mutates checkpointer-tracked state, the thread acquires a *checkpoint guard*. This is essentially a per-thread lock that is re-entrant since a thread holding a guard may take an exception that itself requires a guard. We implement the checkpoint guard as a simple atomic counter. If a quiescence interrupt occurs while a thread holds the guard, then the interrupt is ignored and the thread sets a flag to quiesce when the guard is released. This will happen shortly, since by assumption the thread is in the middle of only non-blocking operations.

The final case to consider is when the thread has entered a blocking ABI call. Fortunately, there are only two indefinitely blocking ABI calls: `ObjectsWaitAny`, which waits for one of an array of handles to be signaled, and `StreamOpen`, which can block when asked to open an outgoing TCP connection.

When the guest calls `ObjectsWaitAny`, the checkpointer adds an additional handle to the list of handles to be waited on; this extra handle is to the *quiescence-requested* event. When the checkpointer initiates quiescence, it sets this event, thereby waking any such blocked threads. When a thread returns from `ObjectsWaitAny`, it quiesces if the quiescence-requested event is set. Since the wait call was prematurely terminated, the thread repeats the call upon resuming; if the wait had a relative timeout, then the thread reduces it by the amount of time already spent waiting and/or checkpointing.

When the guest calls `StreamOpen` with parameters for opening an outgoing TCP connection, the thread first captures its state and marks itself as quiesced. The thread then proceeds with the blocking call since it will not hold up any checkpoints that occur during the call. Upon return, the thread waits until any concurrent checkpointing completes, then rescinds its claim to be quiesced and proceeds with execution. Note that this approach would work for arbitrary calls, not just `StreamOpen`, that do not mutate guest state. However, it requires an expensive thread capture on each call, so we do not use it for `ObjectsWaitAny` where a more lightweight solution exists.

3.3.4 Dealing with non-determinism of Bascule ABI

The Bascule ABI has several sources of non-determinism. The checkpointer must hide them so that restoring a checkpoint results in a replica of the checkpointed state.

The simplest and most widespread source of non-determinism is handle identifiers. Because the host can assign arbitrary identifiers to handles, there is no guarantee it will assign the same ones during restoration of a checkpoint as were used at the time of checkpoint. So, the checkpointer virtualizes handles by maintaining a

mapping between guest virtual handles and host handles and by translating handles in ABI calls.

A subtle source of non-determinism is address space layout randomization (ASLR) [29, 35]. Host ASLR can rearrange the contents of a read-only binary file, so even if a primary and backup have duplicate file contents they may diverge. To handle this, the checkpointer interposes on the guest ABI call that maps binary files. Before returning to the guest, the checkpointer performs all necessary relocation to reflect the address where the file actually got mapped. In other words, the checkpointer ensures the guest's binary-mapping ABI is deterministic even though the host-provided ABI is not.

One source of non-determinism requires a small change to the ABI. In the original Bascule ABI, when the guest creates an HTTP request queue, the host assigns it a non-deterministic ID that is then used by the guest in subsequent calls to open HTTP requests. We address this by changing Bascule's ABI so that the guest assigns the ID instead. This is the only case where we modify the ABI. Modifying existing host and LibOS implementations to support Bascule's new ABI should be relatively straightforward: our modifications to the Windows host and LibOS constitute only ∼250 lines.

3.4 Networking

The IP address clients use to connect to the replicated service is the *service address*. Each instance devotes a NIC to the service that is separate from the NIC it uses to communicate with the orchestrator and other Tardigrade instances. We call this the *service NIC*. Only the primary sends traffic on the service NIC, including ARP packets, so the network and clients see only one machine using the shared address at a time.

To interpose on the service NIC, we implement a network filter that suppresses non-primary output and buffers primary output during epochs. The buffered output of the primary is released only when the following checkpoint is stable.

Buffering interacts with the current ABI in surprising ways, as discussed in §4.3. One consequence is that, until the ABI changes from a socket-based to a packet-based interface, TCP connections are broken on a failover event.

4 Implementation

Our implementation includes the following components, with line counts measured by SLOCCount [34].

- Bascule checkpointer extension: 17,056 lines of C, plus 1,226 lines of Python to produce automatically-generated hook functions (not separately counted).
- Network filter, implemented as a Windows kernel-mode driver: 1,329 lines of C.
- Orchestrator and instance: 683 and 2,718 lines of C#, respectively, plus 1,346 lines of common code used by both for inter-communication.
- Plugin to let instance and checkpointer extension communicate, using Bascule host's support for extending the stream namespace: 2,773 lines of C++.

A limitation of our current implementation is that the orchestrator runs on a single machine, so it is a single point of failure for the system. To improve fault-tolerance, our plan is to divide the orchestrator into two components: the unreliable failure detector and the view manager. The failure detector does not require consistent state, so it can be made fault-tolerant using simple stateless mechanisms; however, the view manager will be redesigned as a state machine and run with a replicated state machine library [6, 22].

The remainder of this section overviews some lessons learned during the implementation of Tardigrade.

4.1 Memory checkpointing optimizations

This subsection describes the optimizations we use to improve the performance of memory checkpointing.

The first optimization reduces checkpoint size by calculating updates to memory at a finer granularity than a page using a twin-diff-delta technique [2]. In this technique, the exception handler that executes when a write-protected page is first written in an epoch stores a copy of the pre-write contents of the page. Then, the checkpoint at the end of the epoch uses delta encoding [17] to capture the difference in the page content more precisely.

The next optimization selectively disables write-protection for hot pages. Our heuristic for deciding that a page is hot is exceeding a threshold for the number of consecutive epochs that the page has been written to, defaulting to three. When write-protection is disabled for a hot page, the checkpointer simply assumes that it is always dirty. However, a side-effect of this mechanism is that the checkpointer cannot detect when the hot page is no longer being written by the guest, so every epoch we flip each hot page to cold with a fixed probability. We found performance to be fairly insensitive to this value of in a broad range; we default to the value $1/16$ which lies within that range. An alternative would be to use twin-diff-delta to detect when a hot page has not been written in a given epoch; we plan to investigate this in future work.

Another important optimization uses parallelism to reduce the time to snapshot memory changes. Our checkpointer maintains several threads, roughly one per core, and disseminates independent memory-snapshotting tasks to them via a shared task queue. We could have parallelized other checkpointing operations besides memory snapshotting, but found it generally not to pay off: other operations are so quick or rare that queuing and scheduling time overwhelms the benefits of parallelization. So,

the only other snapshotting operation we parallelize is capturing thread states.

Normally, an epoch ends when (1) the previous epoch's checkpoint has been disseminated to the backups and acknowledged, and (2) the current epoch's duration is greater than some minimum, typically set to zero. Our final optimization, *checkpoint capping*, can end an epoch earlier based on the rate of memory dirtying in the guest. Checkpoint capping mitigates the effect of rapid memory dirtying on increased time to take a checkpoint, which is especially problematic for guests running in platforms with garbage collection. The goal of checkpoint capping is to automatically end the epoch before the resulting checkpoint can get too large. However, during the epoch it is infeasible to efficiently and precisely predict the potential checkpoint size while accounting for additional optimizations like delta encoding. So, the heuristic we use is to prematurely end the epoch once the number of dirtied pages reaches a configurable threshold.

4.2 In-memory checkpoints

We found that writing checkpoints to files on a Windows host dominated the cost of checkpoint capture and dissemination, even when the files are not stored on disk. So, instead of using files, each instance shares a memory region with the checkpointer extension. The checkpointer captures checkpoints directly to this shared memory, and the instance copies checkpoints received over the network directly into it as well. The section defaults to 1 GB; if this is too small for a particular checkpoint it uses a file instead.

4.3 Breaking connections

The Bascule ABI supports networking through a socket interface. To open a TCP or UDP socket, the guest opens a specially-named stream. To send or receive on that socket, the guest writes or reads the stream handle.

This socket-based interface presents a challenge: since TCP session state is in the host rather than in the guest, it cannot be seen or modified by the checkpointer. Therefore, when restoring an LVM on a new machine, the TCP state will be different and the guest will not be able to communicate over existing connections. To address this problem, the checkpointer breaks all TCP connections before restoring the guest. This is implemented by restoring TCP and HTTP streams as a handle to a special event that is always signalled. When the restored guest starts running and calls an operation on such a handle, the operation will return immediately with `STATUS_CONNECTION_RESET`.

Our hypothesis is that services are written to recover from such transient disconnections, and we find that this hypothesis holds for the services evaluated in §5. A cleaner solution would be to modify the ABI to support checkpointing guest networking state so that connections can be transparently migrated across hosts.

This has taught us a lesson about the design of Bascule. The socket ABI was chosen for expedience, since it obviated the need to build a network stack in the LibOS. The ABI designers were aware that this choice was problematic for compatibility and portability; our work on Tardigrade demonstrates that it also interferes with migration. We believe that the path forward for the Bascule networking interface is to use packets rather than sockets as the interface between guest and host, and we are working with the Bascule development team to realize this goal. An additional benefit of a packet-based interface is to enable packet buffering within the checkpointer extension itself instead of requiring an external network filter.

4.4 Network buffering

Network buffering effectively increases the round-trip time of connections to the server, increasing the bandwidth-delay product of each connection. For TCP connections, this necessitates both a large window size and a large buffer for sent but unacknowledged packets. A Windows host detects this high delay and adjusts the TCP window size in response, but it does not automatically increase the send buffer size.

To fix this, we update the `DefaultSendWindow` registry setting so the send buffer size exceeds the expected bandwidth-delay product. Note that this setting should also be managed on any client that sends significant traffic, because network buffering on the server delays acknowledgment of client packets, causing the client to buffer sent packets for up to two epochs.

5 Evaluation

5.1 Methodology

The machines we use in our experiments are Dell PowerEdge R710 rack servers. Each is configured with two quad-core 2.26 GHz Xeon E5520s, 24 GB RAM, two Broadcom BCM5709C NetXtreme II Gigabit Ethernet NICs, and a Seagate Constellation ST9500530NS 500 GB SATA disk. All the NICs are connected to a single 48-port switch.

Except when otherwise specified, we use four machines: the primary and orchestrator. the backup, the spare, and the client. On each machine, Tardigrade uses one NIC and the replicated service uses the other. The client machine runs Windows Server 2008 R2 Enterprise, and the other machines run Windows Server 2012 R2 Datacenter. To minimize latency, we configure the system to checkpoint as frequently as possible, i.e., to initiate a checkpoint as soon as the previous one is stable.

Our evaluation covers a range of microbenchmarks and real-world services. The microbenchmarking experiments use a simple ping server that listens on a UDP

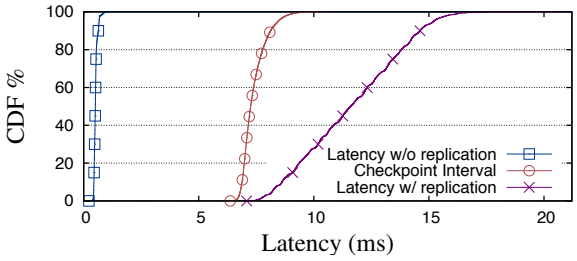

Figure 3: CDF of latency seen by ping client, alongside CDF of checkpoint interval

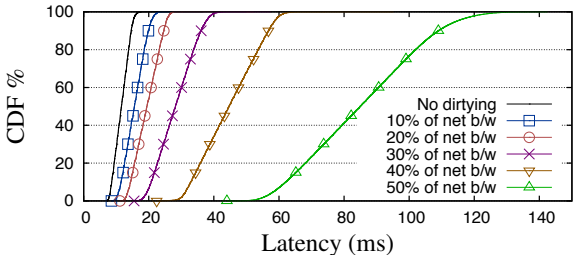

Figure 4: Effect of memory dirtying rate on CDF of latency seen by ping client

port and responds with pongs. This ping server can be configured to dirty memory at a given rate by looping through a 100 MiB region one byte at a time, incrementing each byte modulo 256. To ensure the ping service achieves this dirtying rate, the microbenchmarks disable the checkpoint-capping optimization described in §4.1. Also, the memory-dirtying algorithm accounts for real time: e.g., when it is unscheduled, it makes up for the lost time by dirtying memory until caught up. The client of the ping service sends 100,000 requests, one every 2 ms.

5.2 Latency impact

Our first experiment evaluates the base latency overhead of Tardigrade by running the ping server without memory dirtying and measuring the ping response times seen by the client. Figure 3 shows the CDF of this latency.

Running the service in Tardigrade increases the average latency from 0.5 ms to 11.6 ms, but the 99.9% quantile latency is not substantially higher, only 17.5 ms. The proximate cause of this latency is the interval between consecutive checkpoints, whose CDF is also shown in Figure 3. As expected, the average service latency is the baseline service latency plus 1.5 times the average checkpoint interval. After all, if the server sends a packet at time t, Tardigrade will release that packet when the incremental checkpoint covering t is stable, i.e., at the end of the subsequent interval.

We also measured CPU utilization during this experiment to evaluate CPU overhead. We found that the baseline utilization of the unprotected service was 7.3%, that running the service in Bascule slightly increases utilization to 7.9%, and that running it in Tardigrade modestly increases utilization to 13.5%. Most of the observed utilization increase in Tardigrade is from the orchestrator and primary instance processes.

5.3 Effect of dirtying rate on latency

Next, we evaluate the effect of memory-dirtying rate on latency; we expect that higher memory-dirtying rates will increase checkpoint size, thereby increasing the time required to replicate each checkpoint over the network. Figure 4 shows the latency observed by the client as the

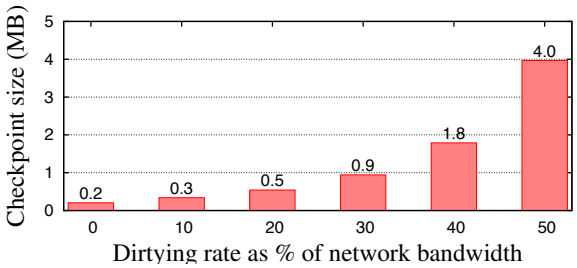

Figure 5: Effect of memory dirtying rate on average checkpoint size of ping server.

memory-dirtying rate increases from 0% to 50% of the network speed, i.e., from 0 to 512 Mb/s.

When the dirtying rate is 10% of the network bandwidth, i.e., 100 Mb/s, the client latency is reasonable, even at the 99.9th quantile. However, as the rate of memory dirtying rises, the latency seen by clients rises nonlinearly. Indeed, Figure 4 only goes to 50% because a dirtying rate of 60% gives average latency over half a second.

This latency is not due to CPU time. The primary machine's CPU utilization generally decreased as the dirtying rate increased, presumably since more time was spent waiting for the network and the backup.

Figure 5 shows the cause: as expected, average checkpoint size increases as dirtying rate increases. We see that the non-linearity of the increase in client latency tracks the non-linearity of the increase in average checkpoint size. This non-linear effect occurs because larger checkpoints take longer to disseminate, leading to to longer periods the service running asynchronously with checkpoint dissemination, which leads to even larger checkpoints. Note that this feedback loop stabilizes to an equilibrium; we do not see it increase with time and cause the distribution to diverge. We expect equilibrium as long as the memory-dirtying rate does not exceed the rate of checkpoint capture and dissemination.

These results suggest that asynchronous replication is a poor fit for workloads with sustained memory-dirtying rates that are a significant fraction of the network bandwidth. Fortunately, as shown later in this section, there are useful services with tractable memory-dirtying rates.

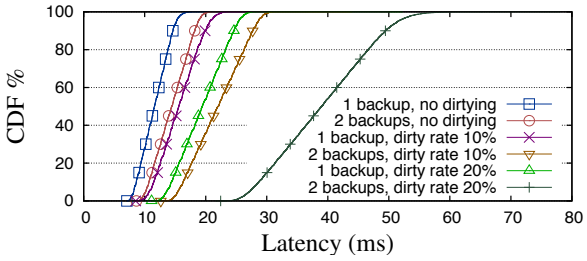

Figure 6: Effect on client latency of adding a backup. Memory-dirtying rate is represented as a percentage of network bandwidth.

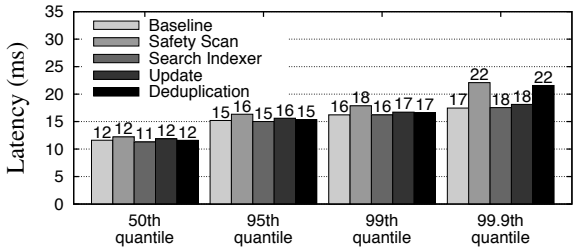

Figure 7: Effect of external processes on latency seen by ping client. Format mirrors that of Figure 1.

5.4 Effect of additional backup

Tardigrade can use multiple backups to tolerate overlapping failures of more than one machine; however, replicating to multiple backups increases the time needed for a checkpoint to be stable. To evaluate this effect, we measure the effect on latency of running the ping service with two backups instead of one. Note that our current implementation does not use IP multicast; if it did, this effect would be significantly reduced.

Figure 6 shows ping latency as a function of both dirtying rate and number of backups. Without dirtying, adding a backup increases latency by only a few milliseconds, which is still quite manageable. With a dirtying rate equivalent to 10% of the network bandwidth, the latency increase is higher than that incurred when adding the same amount of memory dirtying using a single backup. This non-linearity occurs for the same reason as in §5.3; as in those experiments, this acceleration reaches a stable equilibrium: we do not see the checkpoint interval increase with time.

5.5 Impact of external processes

Next, we evaluate the latency impact from resource-intensive processes running on the host, external to the LVM. This experiment uses the ping server without memory dirtying. Recall that we evaluated Remus using this experimental setup in §2, with results shown in Figure 1.

As expected, Figure 7 shows that the impact of external processes on the latency of a service running in Tardigrade is dramatically reduced compared with running in Remus. Indeed, the impact is nearly undetectable for the 99th quantile and below, and hardly noticeable at the 99.9th quantile. We find that external processes cause occasional higher checkpoint periods, likely due to scheduling contention. However, the checkpoint sizes remain unaffected, resulting in 99.9th-quantile latencies in Tardigrade under 25 ms despite external processes that caused 99.9th-quantile latencies of multiple seconds in Remus.

5.6 Failover time

To evaluate the time to recover the service when an instance fails, we use a variant of the client that measures failover times as a long period with no response, i.e., a period with zero service bandwidth. We run the experiment 100 times in each of two scenarios: primary failure and backup failure.

The median recovery time is 500 ms in the backup-failure scenario and 700 ms in the primary-failure scenario. The difference between these two scenarios reflects the time for the new primary to start running; because we keep each backup's in-memory state and objects up to date, this startup cost is only 200 ms. The remaining time is largely due to the 100 ms failure-detection timeout and the time to take a full checkpoint and disseminate it to the new backup. The median size of this full-checkpoint transfer is 26.9 MB, which the new backup takes 225 ms to download.

Note that many services will have larger memory footprints and thus commensurately longer failover times. For instance, the remainder of this section evaluates three real services in Tardigrade. Interruption with a failure at a random point induced full checkpoints of 36 MB for the metadata service, 170 MB for the coordination service, and 636 MB for the web service. Sending the latter over a 1 Gb/s link would take at least 5 s.

An operation that is not on the critical path for recovery is the new backup initializing its LVM. This is because we let the backup acknowledge checkpoints, including full ones, after queueing them for later application. When we prevent this by substantially decreasing the queue size, median response time for the backup-failure scenario becomes 8.8 s. This high figure demonstrates that checkpoint queueing and keeping the backup up-to-date significantly reduce failover delay.

5.7 FDS metadata service

In this and the following two subsections, we evaluate Tardigrade's performance on real services. First, we evaluate a custom configuration metadata service written by colleagues for the FDS research project [27]. In normal operation, this service experiences low traffic because it simply sends and receives periodic heartbeats and informs clients and disk servers when failures occur.

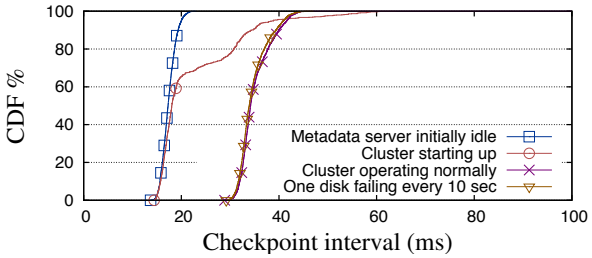

Figure 8: Distribution of checkpoint periods when running the FDS metadata service, during various phases

This experiment uses the FDS cluster's 10 Gb/s network so that the FDS cluster can run as normal but use our fault-tolerant metadata service. However, Tardigrade still uses a 1 Gb/s port for checkpoint dissemination so that the results are comparable to other results presented in this paper.

This experiment proceeds in several phases. First, the metadata service starts up and then idles for one minute. Then, the FDS cluster with 70 disk servers starts up, each of which must communicate with the metadata service to receive a role assignment. Then, ten clients begin executing FDS's stock write-intensive load-testing tool, and the cluster runs normally for two minutes. Finally, we kill one disk server process every ten seconds, provoking the metadata service to react to the departures.

Figure 8 shows the resulting distribution of checkpoint interval. Initially, the checkpoint interval averages 17.4 ms, reflecting an average checkpoint size of 0.9 MB. As the cluster comes online and requests assignments, the checkpoint interval increases but never goes above 64 ms. When the cluster is up and handling requests from disk servers and clients, checkpoint interval maintains a modest average of 35.2 ms, reflecting checkpoint sizes averaging 1.8 MB. This low activity is not surprising, since FDS was designed to reduce load on its metadata service by caching the metadata at participating parties. It is thus an ideal candidate for Tardigrade.

5.8 ZooKeeper coordination service

Our next real service is ZKLite, a custom in-memory implementation of the ZooKeeper server API, written in Java by one of the co-authors for a separate research project. We initialize the server state for the benchmark by creating a balanced binary znode tree of depth 10. The benchmark then executes 100,000 operations, where each operation either reads or writes the data of a random znode. Writes are done with probability $1/3$, and write a uniformly random amount of data between 0 and 10 KB. Operations are launched in parallel, with at most 100 outstanding at once. We report results for the last 90% of operations to reflect steady-state performance.

Since this service is written in a garbage-collected language, it experiences occasional periods of fast memory

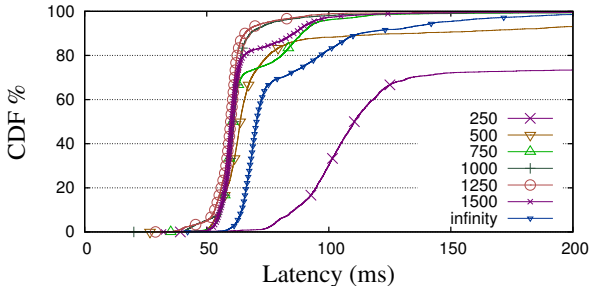

Figure 9: Effect on client latency distribution for ZKLite of number of pages dirtied per epoch before quiescence

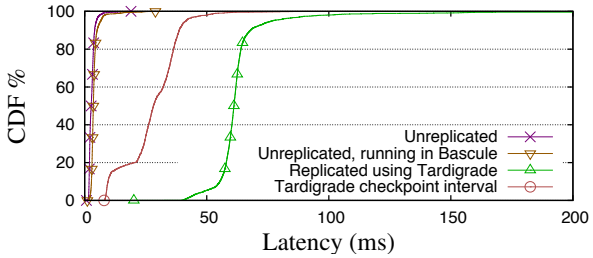

Figure 10: Client latency distribution for ZKLite

dirtying. So, we enable checkpoint capping as discussed in §4.1. A key parameter here is the number of pages dirtied in an epoch before triggering quiescence. If this parameter is low, the service spends a lot of time quiesced instead of processing client requests. On the other hand, a high parameter value results in large checkpoints and thus increased buffering time for outbound packets. Both manifest as high client latency. Figure 9 shows the effect on latency of various parameter settings; infinity means that checkpoint capping is turned off. We see that performance is improved substantially by the use of checkpoint capping, is best when quiescence occurs after about 1,000 pages dirtied per epoch, and is fairly insensitive to this parameter value over the range 750–1,500.

Note that other services and workloads may have different ideal values for this parameter. For instance, if a service has low load, it can accommodate being frequently unscheduled, and thus may perform better with a low cap. If a service has high baseline latency, then the effect of network buffering will be relatively inconsequential, and a higher cap may be best.

Having established what parameter to use for checkpoint capping, we compare performance under Tardigrade to baseline performance. Figure 10 shows the results of benchmarks run under three setups: unreplicated; unreplicated, but running in Bascule; and in Tardigrade. The Bascule-only line shows that the overhead of running in Bascule contributes little to the higher latency seen, so as expected it is asynchronous replication that contributes most to latency.

As discussed in §5.2, outbound buffering causes de-

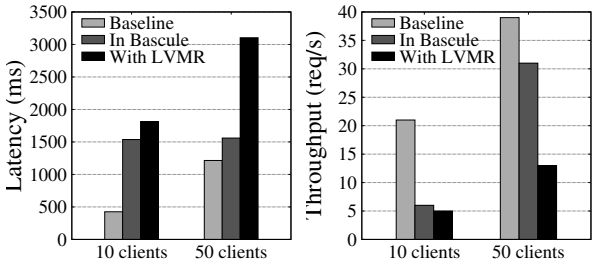

Figure 11: Performance of MediaWiki in Apache with different numbers of client threads

lay of 1.5 times checkpoint interval duration. That duration, also shown in Figure 10, accounts for most of the increased latency; the rest is due to overhead of replication such as handling access-violation exceptions due to memory tracking. The effect of checkpoint capping manifests at the high end of the latency distribution; during periods of high memory dirtying, the service is frequently quiesced, delaying client requests.

The lessons we draw are as follows. Tardigrade can replicate ZKLite under modest load, but at a noticeable latency cost, on the order of 60 ms. The service's use of garbage collection leads to periods of high memory dirtying, which temporarily cause even higher latency. Checkpoint capping can mitigate this problem but not eliminate it, by reducing the time spent buffering packets at the cost of delaying execution of the service.

5.9 Web service

Our web service is the popular MediaWiki running on Apache. Its use of dynamic PHP-generated pages instead of static content is typical for a modern website, and stresses Tardigrade by causing mutation of the service's in-memory state. We use Apache version 2.4.7, PHP v5.5.11, and MediaWiki v1.22.5 backed by a SQLite database. We enable the Alternative PHP Cache for intermediate code and MediaWiki page data.

We operate the server in three modes: normal, within Bascule, and within Tardigrade. We benchmark the service using multiple worker threads on the client, each of which repeatedly fetches the 14-KiB main page over a persistent HTTP connection, waiting for the completion of each fetch before initiating the next. We measure the system once it has reached steady state.

Figure 11 shows results under two load conditions: either 10 or 50 client threads. We see that some of the overhead of Tardigrade comes from running in an LVM and some is due to replication. In particular, the Bascule LVM adds significant latency to this workload because each request issues many small file I/Os, primarily `stat` calls, each of which requires an RPC to a separate security monitor process. This overhead is effectively amortized through batching and pipelining at 50 client threads, giving a significant increase in throughput with little added latency. In contrast, the latency overhead due to replication does increase with load: as load increases, so does the memory-dirtying rate and thus the checkpoint size and interval. With 10 client threads, the checkpoint interval average is 54.4 ms, but with 50 client threads it balloons to 475 ms. We conclude that web services may need modest load to be amenable to LVMR.

5.10 Complexity of services

The real services we use require no modifications to run under Tardigrade, supporting our hypothesis that Tardigrade can make unmodified binaries into fault-tolerant services. In the cases of the FDS metadata service and ZKLite, this also supports our hypothesis that Tardigrade can reduce code complexity and developer effort. According to the CodePro plugin for Eclipse, ZKLite is 24,082 lines of code, less than the 30,889 lines for the Apache ZooKeeper server. The smaller count reflects the fact that ZKLite does not have any code for dealing with failures. Further, FDS's metadata service was written two years ago on the assumption that someday it could be rearchitected for fault tolerance, but in that time no one at Microsoft has found the time to do so. Running it within Tardigrade makes the service fault-tolerant with no developer effort.

6 Discussion and Future Work

§6.1 distills the results of our evaluation into a categorization of services that may be good candidates for LVMR. Then, §6.2 discusses directions for future work.

6.1 Candidate service characteristics

Providing fault tolerance at the virtualization layer saves development effort at the expense of runtime performance. Based on our evaluation, we offer some guidance on the classes of applications for which the overhead of Tardigrade may or may not be reasonable.

An important characteristic to consider is the rate and magnitude of memory dirtying. If the memory-dirtying rate is significant relative to the network bandwidth, then LVMR will spend too much time taking checkpoints and transmitting them to backups. Our evaluation suggests that memory-dirtying rates above $\sim$40% of network bandwidth will cause significant delays. Also, as shown in §5.8, occasional bursts of memory dirtying, e.g., due to garbage collection, manifest as occasional periods of high latency even with checkpoint capping.

There are other reasons a service may not be a good candidate for LVMR. If a service must remain available despite software bugs within the service itself, then LVMR is not applicable. Also, if the service can tolerate very high latencies, then LVMR's main benefit relative to VMR is moot.

One class of promising candidate services for LVMR is metadata and coordination services. These critical services are usually required to be both highly available and

strongly consistent because entire distributed systems depend on them. Also, because these services tend to be centralized and therefore a potential bottleneck, system designers often use techniques such as client caching and coarse-grained synchronization to minimize service load. Another favorable characteristic is that the rate of state mutation in such services tends to be low; e.g., the ratio of read to write operations in a typical ZooKeeper workload varies between 10:1 and 100:1 [18].

Another class of good candidate services is niche web applications with a small number of users, e.g., web sites internal to an organization such as requisitioning systems and charity event managers.

An example of a service that is not a good candidate for LVMR is a DBMS. We ran SQL Server inside Tardigrade but found its performance to be poor due to large checkpoints. For such workloads, customization may be necessary, as done by RemusDB [24].

6.2 Future work

Our experience building Tardigrade has highlighted areas for improvement in the underlying LVM technology, Bascule. For instance, we discussed the difficulties caused by having a non-deterministic ABI in §3.3.4 and how Bascule would be improved by offering a packet-based network interface in §4.3. We hope that these lessons can inform the design of lightweight process container technologies to support migration.

There are a number of potential future directions for Tardigrade. First, we believe we can improve performance of guests by making slow operations appear to complete before they actually have, as done by Speculator [26]. Tardigrade already supports buffering output until a speculative operation has completed, and could be extended to support rollback in the case of operation failure. A second direction is exploring how to tune the library OS to improve Tardigrade performance: essentially, this would be to LVMs what paravirtualization [33] is to VMs. Third is grouping multiple LVMs into checkpoint domains to coordinate checkpointing among them, thereby essentially performing distributed snapshots [7] for LVMs. This may improve performance as we would not have to buffer network traffic between LVMs in the same checkpoint domain.

7 Related Work

This section first surveys related work that transparently provides fault tolerance using encapsulation, then briefly discusses alternate approaches to designing and implementing fault-tolerant services.

Encapsulation-based fault-tolerance

The key insights of Bressoud and Schneider [4] were that a VM is a well-defined state machine, and that implementing state machine replication [32] at the VM level is attractive in terms of engineering and time-to-market costs compared to implementations at the hardware, operating system, or application levels. Their system enforced deterministic execution of a primary and backup VM in lock-step through capture and replay of input events by the hypervisor.

VMware's server virtualization platform vSphere [31] provides high availability for a VM using primary-backup replication. Supporting multiprocessors can incur a high performance cost in a deterministic record-and-replay approach; instead, the most recent release of vSphere, 6.0, executes the same instruction sequence simultaneously in both VMs. This approach enables vSphere 6.0 to add support for up to 4 virtual CPUs in a protected VM.

Napper et al. [25] and Friedman and Kama [15] implemented fault-tolerant Java virtual machines using an approach similar to that of Bressoud and Schneider. This choice of hypervisor reduces overhead compared to virtual machine monitors that execute desktop operating systems, but also limits the class of applications.

Replication may be achieved by copying the state of a system instead of replaying input deterministically. State copying applies to multiprocessors and does not require control of non-determinism, but replicating state typically requires higher bandwidth than replicating inputs. In contrast to the replaying VM replication systems discussed above, Cully et al. [11] implemented state-copying primary-backup VM replication in Remus by building on live migration in the Xen virtual machine monitor [10]. Remus uses techniques such as pipelining execution with replication to address the high overheads of checkpointing VM state. Later work [24] established still further gains by compressing the replicated data, and other gains specific to paravirtualized systems. Tardigrade builds on the approach used in Remus and further reduces overhead.

Additional optimizations to VM replication have been proposed [23, 36, 37]. Although some optimizations are specific to a virtualization platform, others, such as speculative state transfer, may apply to Tardigrade and are topics for future work.

An LVM is similar in many regards to the containers actively being developed for Linux, including Docker [12] and its associated kernel support from LXC/LXD [21]. While there has been previous work on process-level migration in a research context [14, 28], the current popular interest in Linux containers makes us believe LVMR may be valuable outside the research community. Container interfaces appear to be closing the gap between the strong but efficient runtime isolation that is achieved by LVM and LXC/LXD and the desire to more easily deliver and manage the lifecycles of entire application stacks in production environments. There is already active open-source work on providing live container mi-

gration for Linux [8], and the work described in this paper is a natural next direction. Containers are actively used to manage large-scale distributed applications today, so integrating LVMR into those environments would ease the development complexity associated with critical, central components like those described in §6.1.

Designing services to be fault-tolerant

Virtualization-based fault tolerance is useful not only for protecting unmodified legacy applications, but also for reducing the cost and effort of developing new fault-tolerant services. In this section we overview some alternate, non-transparent approaches and their complexities.

One approach is to write the service as a serializable, deterministic state machine and rely on a library such as BFT [6] or SMART [22] for replication. However, there are several common errors the developer can make that will invisibly undermine the library's consistency guarantees, such as failing to serialize all relevant data, or writing non-deterministic code [1]. More recent work on Eve [19] has shown how to lift the requirement of determinism, but still requires annotation of which objects need serialization and, for performance, which operations are likely to commute.

Another approach is to micromanage the persistence of state to a reliable storage backend at the application level. One class of systems exemplifying this approach is transactional databases. Such customization is likely to yield good performance but requires careful engineering, including non-trivial checkpointing and recovery mechanisms [16]. A range of backend solutions are available, from locally-administered disk arrays [9] and distributed file systems to cloud-hosted services.

8 Conclusions

This paper describes asynchronous lightweight virtual machine replication, a technique for automatically converting an existing service into one that will tolerate machine failures. Using LVMs instead of VMs has the advantage that only changes to the application's state need to be replicated to backups before network output can be released. This leads to reasonable client-perceived latencies, even at the 99.9th quantile, and even when external processes share the host. To demonstrate the practicality of LVMR, we implemented it in the Tardigrade system. Tardigrade is not suitable for services that require extremely low latency or that modify memory at high rates. But, for many other services, the benefit of transparent fault tolerance will be a welcome aid to developers who lack the time, inclination, or expertise to correctly make their services consistent and fault tolerant.

9 Acknowledgments

The authors thank James Mickens and Jeremy Elson for helping us with FDS cluster configuration, Brian Zill and Jitu Padhye for answering networking questions, and Reuben Olinsky for help with Bascule. We are grateful to the anonymous reviewers for their helpful comments, and particularly to our shepherd, Jeff Chase, whose substantial feedback improved the paper.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, 2002.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. Lorch, B. Bond, R. Olinsky, and G. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proc. EuroSys '13*, pages 239–252, 2013.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, Feb. 1996.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI*, 2006.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, Feb. 1985.

[8] Checkpoint/Restore in Userspace. http://criu.org/.

[9] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, pages 273–286. USENIX, 2005.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. NSDI*, pages 161–174, 2008.

[12] Docker. https://www.docker.com/.

[13] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, pages 339–354, Dec. 2008.

[14] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Softwware Practice and Experience*, 21(8):757–785, July 1991.

[15] R. Friedman and A. Kama. Transparent fault-tolerant Java virtual machine. In *Proc. Reliable Distributed Systems*, pages 319–328. IEEE, 2003.

[16] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.

[17] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proc. OSDI*, 2008.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX ATC*, 2010.

[19] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proc. OSDI*, 2012.

[20] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. In *Proc. PODC*, 2009.

[21] Linux Containers. https://linuxcontainers.org/.

[22] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. EuroSys*, 2006.

[23] M. Lu and T. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Proc. Dependable Systems Networks*, pages 534–543. IEEE, 2009.

[24] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *VLDB*, 22(1):29–45, 2013.

[25] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant Java virtual machine. In *Proc. Dependable Systems and Networks (DSN)*, pages 425–425. IEEE Computer Society, 2003.

[26] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, 2005.

[27] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. OSDI*, 2012.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. OSDI*, 2002.

[29] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[30] D. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Proc. ASPLOS XVI*, pages 291–304, 2011.

[31] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[33] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. OSDI*, 2002.

[34] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[35] O. Whitehouse. An analysis of address space layout randomization on Windows Vista. Technical report, Symantec, 2007.

[36] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li. Improving the performance of hypervisor-based fault tolerance. In *Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[37] J. Zhu, Z. Jiang, Z. Xiao, and X. Li. Optimizing the performance of virtual machine synchronization for fault tolerance. *IEEE Transactions on Computers*, 60(12):1718–1729, 2011.