PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance

Cheng Wang[*], *Xusheng Chen*[*], *Weiwei Jia, Boxuan Li,*
Haoran Qiu, Shixiong Zhao, and Heming Cui
The University of Hong Kong

Abstract

Cloud computing enables a vast deployment of online services in virtualized infrastructures, making it crucial to provide fast fault-tolerance for virtual machines (VM). Unfortunately, despite much effort, achieving fast and multi-core scalable VM fault-tolerance is still an open problem. A main reason is that the dominant primary-backup approach (e.g., REMUS) transfers an excessive amount of memory pages, all of them, updated by a service replicated on the primary VM and the backup VM. This approach makes the two VMs identical but greatly degrades the performance of services.

State machine replication (SMR) enforces the same total order of inputs for a service replicated across physical hosts. This makes *most* updated memory pages across hosts the same and they do not need to be transferred. We present Virtualized SMR (VSMR), a new approach to tackle this open problem. VSMR enforces the same order of inputs for a VM replicated across hosts. It uses commodity hardware to efficiently compute updated page hashes and to compare them across replicas. Therefore, VSMR can efficiently enforce identical VMs by transferring only divergent pages. An extensive evaluation on PLOVER, the first VSMR system, shows that PLOVER's throughput on multi-core is 2.2X to 3.8X higher than three popular primary-backup systems. Meanwhile, PLOVER consumed 9.2X less network bandwidth than both of them. PLOVER's source code and raw results are released on `github.com/hku-systems/plover`.

1 Introduction

The cloud computing paradigm enables a pervasive deployment of online services in virtualized infrastructures (e.g., Xen [26]). Meanwhile, a virtual machine (VM) is incorporating more and more virtual CPUs (vCPU) on multi-core hardware because online services process many requests concurrently. This rapid growth of cloud computing components implies that hardware failures become commonplace [18] rather than occasional. A fast and multi-core scalable VM fault-tolerance approach is highly desirable for online services.

Primary-backup (e.g., REMUS [36]), a dominant VM fault-tolerance approach, works in a physical time slot manner. In each slot, it runs a service in the primary VM to process client requests, tracks updated VM states (e.g., dirty memory pages), and buffers network outputs. When a slot ends, a `syncvm` operation is invoked to transfer dirty pages from the primary to backup. Once the transfer succeeds, network outputs are sent to clients. By doing so, primary-backup ensures *external consistency* [36]: primary and backup have the same states and a primary failure will not be observed by clients.

Unfortunately, despite much effort [13, 36, 42, 64, 82], achieving fast and multi-core scalable VM fault-tolerance remains an open problem [5, 38, 42, 82] . A main reason is that the primary-backup approach often has to transfer an excessive amount of dirty memory pages, which greatly degrades the performance of a service and occupies prohibitive network bandwidth.

For instance, if a program updates 20K dirty memory pages within a 100ms slot, transferring these pages consumes a huge network bandwidth of 6.4 Gbps. Both our evaluation (§6) and prior study [36, 40, 42, 60] show that many programs access even more dirty pages on over four CPU cores. vSphereFT-6.5 [17], a latest primary-backup product, permits up to four vCPUs per VM and only two of such VMs per physical host [5]. Therefore, to enable fault-tolerance, people often sacrifice multi-vCPU speedup and VM consolidation [34].

As a service includes multiple programs (e.g., a website deployed in one VM can include an `Nginx` web server, a Python interpreter, and `MySQL`), and a program scales better on more CPU cores and accesses more memory, this problem becomes even more challenging.

Another approach, state machine replication (SMR), appears a promising solution for this open problem. SMR [68] models a program as a deterministic state machine and replicates it on different physical hosts (or replicas). It uses a distributed consensus protocol (typically, PAXOS [56]) to enforce the same total order of program inputs across replicas, making them perform the same sequence of state transitions. SMR systems [35, 45, 50] often incur low performance overhead with popular programs on 16 CPU cores.

However, to ensure external consistency, SMR requires extra mechanisms to resolve divergent executions (i.e., multithreading nondeterminism [60]) across replicas. Extisting SMR systems provide two major mechanisms. First, EVE [50] requires program developers to manually annotate variables shared by threads, and it detects divergent variable states at runtime. Second,

[*]The first two authors contributed equally to this work.

REX [45] and CRANE [35] enforce the same order of inter-thread synchronization (e.g., locks) across replicas. If no data race occurs, determinism is ensured; otherwise, developers' diagnosis [51, 75] may be needed. Therefore, neither of the two mechanisms is fully automatic.

Our key observation is that by enforcing the same total order of inputs for a VM replicated across hosts, almost all updated memory pages across the hosts are the same and they do not need to be transferred. Intuitively, if a VM containing a key-value service is replicated across hosts and it receives the same order of requests, all these hosts should contain roughly the same data in memory. Empirically, we enforced the same total order of client requests for 8 diverse services running on two VMs, and 72% to 97% of the services' dirty pages were the same after processing these requests.

This paper presents Virtualized SMR (VSMR), a new SMR approach that can achieve fast, multi-core scalable VM fault-tolerance. VSMR enforces same total order of network inputs for a VM replicated across hosts. It then periodically invokes a `syncvm` operation to efficiently compute updated page hashes, to compare them across the replicas, and to transfer only the divergent pages.

In a conceptual level, VSMR replicates an entire guest VM as a state machine and achieves the strengths of both SMR and primary-backup. By transferring only those divergent pages, VSMR automatically and efficiently ensures external consistency. Leveraging the powerful fault-tolerance of PAXOS, VSMR tackles a notorious "split-brain problem" (§2.2) in primary-backup systems.

We implemented PLOVER,[1] the first VSMR system in Linux. PLOVER uses APUS [92], a fast, RDMA-powered PAXOS implementation. PLOVER intercepts inbound network packets in the KVM QEMU hypervisor [80] and replicates them to other VM hypervisors using PAXOS. PLOVER's `syncvm` operation (§4) is built on top of PAXOS for robustness, and it uses RDMA to efficiently compare page hashes across replicas. PLOVER does not modify the underlying PAXOS protocol, so it is generic to work with other fast consensus protocols [58, 78].

We evaluated PLOVER on 12 widely used programs, including 8 servers (e.g., `SSDB` [85] and `Tomcat` [3]) and 4 dynamic language interpreters (e.g., PHP). We group these programs into 8 practical services, including DjCMS [7], a content management system (CMS) that consists of `Nginx` [73], Python, and `MySQL` [22]. We compared PLOVER with three well-engineered primary-backup systems QEMU-MicroCheckpoint [13] (for short, MC) , COLO [38], and STR [63]. Evaluation shows that:

1. On average, PLOVER's throughput is 2.2X higher than MC, STR, and COLO on 4-vCPU VMs, 3.8X higher on 16-vCPU VMs. Compared to unreplicated executions, PLOVER's overhead on response time is modest. PLOVER has reasonable CPU usage.

2. PLOVER consumes 9.2X less network bandwidth than both MC, STR, and COLO on average. It enables consolidating multiple fault-tolerant VMs on one host.

3. PLOVER is robust to various failures.

Our major contribution is VSMR, a new SMR approach, which automatically achieves much faster and more scalable VM fault-tolerance. Our other contributions include the PLOVER implementation and an extensive evaluation on diverse, sophisticated online services. Moreover, by efficiently enforcing the same VM across hosts, PLOVER can be broadly applied to other research areas. For instance, page-level false-sharing [28, 61] is a notorious performance problem in multithreading replay [40, 54, 67]. PLOVER can be an effective template to alleviate this problem, because most false-shared pages across the record and replay hosts should have the same contents and they do not need to be transferred.

The remaining of the paper is organized as follows. §2 introduces the background of RDMA, VM, and PAXOS. §3 gives an overview on PLOVER's architecture and its advantages over the primary-backup approach. §4 presents PLOVER's runtime system. §5 describes implementation details, §6 presents evaluation results, §7 introduces related work, and §8 concludes.

2 Background

2.1 RDMA

RDMA (Remote Direct Memory Access) [2] can directly write from the userspace memory of a host to the userspace memory of a remote host, bypassing the OS and CPU on both hosts. RDMA architectures (e.g., Infiniband [2] and RoCE [11]) are commonplace within a datacenter due to their ultra low latency and decreasing costs. RDMA's ultra low latency comes from not only its OS bypassing feature, but also its dedicated network stack implemented in hardware. RDMA latency is several times smaller than software-only OS bypassing techniques (e.g., DPDK [6] and Arrakis [77]).

The advantage of RDMA latency is especially significant when transferring messages of small sizes. Benchmarks [4, 10] show that, with the same network interface card (NIC), transferring messages of less than 2KB on RDMA is about 10X$\sim$30X faster than on TCP. If the message size becomes larger (e.g., over 8KB), RDMA latency is merely about 30% faster than TCP because network bandwidth becomes a bottleneck for both. This

[1] The Pacific golden plover is well known for her strong tolerance to the extreme weather in Alaska.

suggests that RDMA is attractive for invoking consensus on inputs and sending hashes of memory pages, and it is less beneficial for transferring pages. PLOVER uses RDMA for invoking PAXOS consensus, exchanging page hashes across replicas, and transferring divergent pages.

2.2 Virtual Machine and Its Fault-tolerance

VMs [26, 52, 91] are widely used in clouds and datacenters due to their low performance overhead [42], platform independence, performance isolation [47], etc. For instance, KVM [52] is an accelerator that uses the hardware virtualization features of various CPUs, while QEMU [80] emulates the hardware for VMs. PLOVER uses KVM-QEMU for three main reasons. First, KVM-QEMU incurs little performance overhead compared to bare-metal. Second, QEMU works in userspace and is suitable for RDMA-based PAXOS to intercept inputs (RDMA currently only supports userspace memory). PLOVER uses QEMU's `tap_send()` API to intercept network inputs. Third, the QEMU virtual threads that act as vCPUs are spawned from the QEMU main process, which enables PLOVER to monitor programs running in a guest VM non-intrusively [87] without modifying the guest.

Moreover, VM platform independence enables consolidation [34]: people can migrate many VMs [32, 72] to a small number of physical hosts to save energy and ease management. However, consolidation also implies that many VMs are prone to hardware failures. Therefore, a fast, scalable, and network bandwidth friendly VM fault-tolerance approach is highly desirable.

Existing VM fault-tolerance systems [13, 17, 36, 38, 63, 64, 82] are mainly based on the primary-backup approach. To maintain external consistency, the primary must transfer the dirty memory modified by a program within one time slot to the backup before releasing outputs, the so called "output commit problem" [86]. Therefore, the major performance bottleneck of this approach is the time taken to transfer dirty pages, because local memory access speed can be 10X$\sim$100X faster [14] than network speed. As real-world programs become increasingly scalable on multi-core and access more memory per second, this bottleneck becomes even more significant. An evaluation [42] shows that this transfer time can be much bigger than a `syncvm` time slot, greatly degrading the performance of services.

Synchronization Traffic Reduction (STR) [63] is a heuristic for reducing the number of transferred pages. It runs both primary and secondary VMs in parallel to process the same network inputs in the same order. STR uses a 25ms `syncvm` interval and only transfers divergent pages in each `syncvm` operation. However, both our evaluation and STR's show that this heuristic is ineffective because of the static `syncvm` interval (§6.2).

COLO is a primary-backup system deployed in Huawei [20]. It runs the same service on both primary and backup, compares per-connection network outputs, and does a `syncvm` if there is any network output divergence. COLO can safely skip the `syncvm` operation if network outputs remain identical. Nevertheless, both our evaluation and COLO's show that its performance severely degrades when the number of client connections is large.

vSphereFT used to take a record-replay approach [29, 83] for uni-vCPU, but it switches to the REMUS approach since vSphereFT 6.0 [9, 17]. If fault-tolerance is enabled, vSphereFT permits at most four vCPUs per VM and only two of such VMs per host [5]. This affects multi-vCPU speedup and VM consolidation.

Since primary-backup has only two replicas, when network partition occurs, neither the primary nor backup can determine whether the other one fails forever or is temporarily partitioned. Therefore, they both may serve client requests, breaking external consistency. This is the notorious "split-brain problem" [23, 24, 83].

2.3 PAXOS and SMR Systems

PAXOS [55, 56, 68] is a major protocol to enforce the same, totally ordered inputs across replicas. For efficiency, typical PAXOS implementations [68, 74] take the Multi-Paxos approach [55]: it elects a dedicated leader in each view to invoke consensus on new inputs, and other replicas work as witnesses to agree on inputs. In PAXOS, the value of each agreed input is flexible, and PLOVER takes advantage of this flexibility. PAXOS can be used to maintain different roles consistently for different replicas [58, 71], and replicas with different roles can interpret the same agreed input value differently according to the (consistent) roles. E.g., the leader of NOPaxos [58] executes inputs; its witnesses agree on inputs and interpret inputs as no-operation (NOP).

To maintain roles for replicas consistently, PAXOS replicas send periodical heartbeats [68, 74] to other replicas and track the number of heartbeat failures with a threshold. If a replica finds that its threshold is reached, it suspects the replica on the other end failed and it invokes a new consensus (e.g., leader election); otherwise, a replica can safely intercept inputs or logical operations on its own safely. During leader election, the node with the most up-to-date state wins [74, 78].

Three recent SMR protocols, NOPaxos [58], APUS [92], and DARE [78] incur a low consensus latency of tens of μs. PLOVER uses APUS for three main reasons: (1) it provides a flexible `paxos_op(void *val)` API to propose a consensus request with `val` as the proposed value; (2) its consensus protocol includes a durable storage (DARE works purely in memory); and (3) it is open source.

3 Overview

3.1 Deployment Suggestion

PLOVER's deployment follows typical SMR systems: three replicas are connected with RDMA networks, and each replica runs a PLOVER VM instance containing a set of programs. We suggest each replica have 16+ CPU cores. By running three replicas, PLOVER can tolerate hardware failures or network partitions of one replica. This fault-tolerance guarantee is sufficient because: (1) a VM can already tolerate various failures in guest OS, and (2) tolerating one failure is a common guarantee in VM fault-tolerance systems [17, 36].

We suggest more CPU cores because PLOVER uses spare cores to compute dirty page hashes. Our evaluation used 24-core hosts and PLOVER performance was already reasonable. In addition, RDMA becomes prevalent [69, 78]. RDMA is just a requirement for current PLOVER implementation, not a requirement for VSMR. One can implement VSMR using other fast PAXOS protocols (e.g., NOPaxos [58]) and using other OS bypass techniques (e.g., Arrakis [77]) to send page hashes.

3.2 PLOVER Architecture

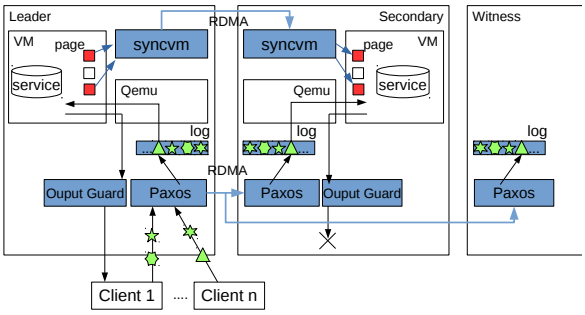

Figure 1: PLOVER *Architecture.* Key components are in blue, inputs are in green, divergent dirty pages are in red.

We designed PLOVER to be simple and generic for various PAXOS implementations. To this end, PLOVER has two unique features compared to regular SMR systems.

First, unlike regular SMR systems which maintain two replica roles (leader and witness), PLOVER invokes PAXOS to consistently maintain three replica roles: leader, secondary, and witness. In PLOVER's underlying PAXOS level, both PLOVER's secondary and witness are "PAXOS witnesses" which simply agree on consensus requests. The only difference is about interpreting `syncvm` in the upper PLOVER system level: the PLOVER leader and secondary involve the `syncvm`, and the PLOVER witness interprets `syncvm` as NOP. We made this design choice because transferring divergent pages to only the secondary is efficient.

Second, to minimize service downtime during the leader's failures, rather than letting the remaining nodes compete to be the new leader, PLOVER elevates its secondary to be the leader because the secondary's state is more up-to-date than the witness's. PLOVER has the same safety guarantee as PAXOS by ensuring there is one unique leader in each view and all the replicas are consistent with their roles (§4.5). To preserve the fault tolerance guarantee, the new leader will do a VM migration to the witness, elevate the witness to be the secondary, and then begin to serve client requests.

Figure 1 shows PLOVER's architecture with four key components: the PAXOS input coordinator (PAXOS), the consensus log (*log*), the output buffering guard (*guard*), and the `syncvm` component. The PAXOS coordinators reside in all three replicas to maintain a consensus log with the same order of SMR operations, including input requests, `syncvm`, and role changes (§4.2).

When PLOVER starts, PAXOS elects one replica as the leader, which is dedicated to receive and make consensus on client requests. When the leader receives a new network input, it invokes PAXOS to replicate this input on PLOVER's replicas. §6.3 shows that, by enforcing the same total order of realistic workload inputs for different VM replicas for 8 services, 72% ∼97% of the programs' memory are already the same and do not need to be transferred.

The leader periodically invokes consensuses on `syncvm` operations to synchronize the VM states of the VMs. PLOVER uses an adaptive algorithm to determine the intervals between two `syncvm` operations based on current workloads, which effectively reduces transferred states and improves performance (§4.3). On successful consensus on a `syncvm` operation, the `syncvm` components of the leader and secondary interpret it with three steps: (1) they exchange dirty page bitmaps and compute hashes of each dirty physical pages concurrently; (2) the leader receives hashes from the secondary and compare hashes; (3) the leader transfers only the divergent pages. §4.4 describes our `syncvm` protocol in detail.

The guards on both leader and secondary buffer network outputs since the last `syncvm` operation. When a new `syncvm` succeeds, the leader's guard releases outputs to clients, while the secondary discards outputs.

PLOVER ensures external consistency. Suppose the leader fails in the n_{th} slot (i.e., PLOVER has finished $n-1$ `syncvm` operations), the secondary becomes the new leader, and the old leader and the new leader have the same states in the last $n-1$ slots. Since the old leader's output in the n_{th} slot has not been released by PLOVER, clients will not observe any inconsistency even if the new leader's state in the n_{th} slot differs from the old leader's. Thus, the new leader can take over without perturbing clients.

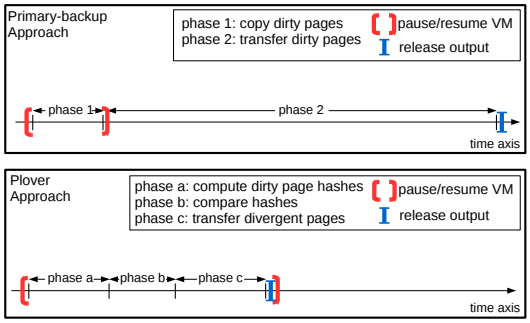

Figure 2: *Comparing `syncvm` operation in VSMR and MC.*

3.3 Comparing PLOVER and primary-backup

We design PLOVER to gain the same fault-tolerance strength as regular SMR. By using PAXOS to maintain the roles of replicas, PLOVER can consistently maintain a leader. By running three PAXOS replicas, PLOVER is also able to consistently detect an outdated leader caused by transient network partitions. In contrast, primary-backup is known unable to handle network partition, the so called "split-brain problem" (§2.2). Because hardware failures may cause transient network partitions (e.g., NIC or network switch errors), PAXOS's strong fault-tolerance is increasingly useful.

To illustrate why PLOVER can be faster than a typical primary-backup approach, Figure 2 shows the leader or primary's workflow in PLOVER and in MC [13], a recent REMUS-based implementation developed in QEMU [80]. Within a primary-backup `syncvm` operation, the time taken in MC's primary can be divided into two major phases: (1) t_{copy}, time taken for copying dirty pages to another memory region; and (2) $t_{transfer}$, time taken for transferring dirty pages. The primary resumes its guest VM after phase (1), but it must release outputs after phase (2) for external consistency. $t_{transfer}$ is often much longer than t_{copy} and becomes the bottleneck (§6.2).

The time taken in a PLOVER leader's `syncvm` operation can be divided into three major phases: (a) $t_{compute}$, time taken to compute hashes for local dirty pages; (b) $t_{compare}$, time taken to compare hashes between leader and secondary; and (c) $t_{divergent}$, time taken to transfer only divergent pages. PLOVER resumes its guest VM and releases outputs after transferring the divergent pages. PLOVER resumes the guest VM after the transfer because it saves the page copy time by using RDMA to directly write divergent pages to the secondary.

Compared to primary-backup, PLOVER's phase (a) can be fast by leveraging CPU cores, and phase (b) can be fast by leveraging RDMA. Phase (c) can be fast if most dirty pages between the PLOVER leader and secondary are the same. Our evaluation shows that PLOVER's $t_{divergent}$ is up to 12.8X faster than MC's $t_{transfer}$.

API	Argument	API Semantic
`paxos_second`	secondary ID	Propose the secondary
`paxos_input`	input packets	Propose client requests
`paxos_syncvm`	`syncvm`	Propose a `syncvm` operation

Table 1: PLOVER's *consensus operations, all built on top of* APUS's *`paxos_op(void *val)` API (§2.3).*

4 The PLOVER Runtime System

This section introduces PLOVER's runtime System. Table 1 shows all the three types of consensus operation APIs that PLOVER's leader invokes.

4.1 Terminology Setup

A PLOVER replica maintains a <`role`,`vid`,`log`, n_{err} > tuple on its local QEMU hypervisor. `role` is a replica's role (leader, secondary, or witness) that has been agreed by PAXOS; `vid` is the current PAXOS view ID [68]; `log` is the current PAXOS consensus log (§3.2), and n_{err} is the current number of communication failures recorded in PAXOS (e.g., a PAXOS heartbeat failure will increment n_{err} by 1). `vid`, `log`, and n_{err} are all exposed from the underlying PAXOS implementation, and PLOVER only updates n_{err} if a `syncvm` has an error. In short, PLOVER runs on top of PAXOS without modifying its implementation.

4.2 SMR Operation Types

As an SMR system, all PLOVER operations run on top of the underlying PAXOS protocol. PLOVER has three SMR operations in total: `paxos_second`, `paxos_input`, and `paxos_syncvm` (Table 1).

The `paxos_second` API is invoked by the PLOVER leader to assign one replica as the secondary, ensuring that the new secondary is consistently agreed among PLOVER replicas. This API is invoked when a new PLOVER leader is elected or the secondary is suspected to fail. PLOVER's leader randomly proposes a replica in its current PAXOS group as the secondary. This operation complies with PAXOS safety guarantee even if the proposed secondary fails, because the leader's `syncvm` operations can detect the new secondary's failure by incrementing n_{err} (§4.4).

The `paxos_input` operations are invoked by the leader when inbound network packets arrive at local hypervisor. Both PLOVER secondary and witness act as "PAXOS witnesses" to agree on the proposed packet, achieving a standard PAXOS consensus.

The `paxos_syncvm` is used to invoke a `syncvm` operation in PLOVER. When the primary finds that the service running in local VM has finished processing inputs and become idle (§4.3), it invokes a consensus on `syncvm` by invoking a `paxos_syncvm` operation. Invoking a `syncvm` with consensus is beneficial: it makes the leader and secondary receive exactly the same sequence of client requests between every two consecutive `syncvm`

operations, greatly reducing memory divergence (§6.3).

4.3 Efficiently Determining Slot Boundary

Similar to primary-backup for ensuring external consistency [86], PLOVER leader must buffer all outbound packets before a `syncvm` succeeds, including client responses and TCP ACKs. Client programs will stop sending new packets when their TCP congestion windows are met, even server programs have finished processing requests and become idle. This leads to unnecessary time slots. In practical workloads with concurrent connections, arrival times of requests are often unpredictable, thus a static `syncvm` time slot configuration (e.g., 25ms in REMUS and 100ms in MC) can often cause an idle service and unnecessary time slots.

To avoid unnecessary time slots, PLOVER develops an adaptive-slot algorithm by inserting `syncvm` operations when its leader determines idle status of programs running in guest VM. PLOVER leverages QEMU's threading hierarchy to spawn an internal thread that checks the CPU usage of the guest VM to determine whether it is idle. §5.1 describes implementation details. This simple, non-intrusive algorithm helps PLOVER quickly proceed its slots, and our evaluation shows that this algorithm is effective in improving PLOVER's performance (§6.2).

4.4 Protocol for `syncvm`

PLOVER's `syncvm` contains three phases (§3.3). The first phase is `compute`. On executing the `paxos_syncvm` operation, the leader pauses its VM immediately, while the secondary does the pause when its programs become idle (§4.3). When both VMs are paused, leader and secondary exchange their dirty page bitmap and compute a union of the two bitmaps. Then, leader and secondary concurrently compute page hashes according to the union.

The second phase is `compare`. The secondary sends its hash list to the leader, and the leader does a comparison to identify all divergent pages.

The third phase is `transfer`. The leader uses RDMA to transfer all divergent pages to secondary and append a special EOF at the end. The secondary saves the pages in a static buffer, sends an ACK to the leader, and applies divergent pages to its guest VM in an atomic manner. This is crucial for PLOVER's correctness because if the secondary starts applying pages while receiving, and the leader fails in the middle, the secondary will end in a corrupted state. On receiving the ACK, the leader releases outputs since the last `syncvm` and resumes its guest VM.

All the three phases carry the sender's `vid` and the receiver checks `vid` as a standard PAXOS way [35, 68]. If any communication error happens during a `syncvm`, a local replica increments n_{err} by 1. If this replica is the leader, it re-invokes a `syncvm` consensus (§4.2). PAXOS will be involved once n_{err} reaches its re-election threshold. Although updating n_{err} in both PLOVER and in the underlying PAXOS may have data races, n_{err} is just a statistic variable and there is no a correctness issue.

4.5 Handling Replica Failures

PLOVER automatically tolerates one replica failure. If the secondary fails, the leader will invoke a standard VM migration to bring the witness's states up-to-date and elevate the witness to be the secondary. If the witness fails, no PLOVER actions are needed because the leader can continue to serve client requests and ensure fault tolerance.

If the leader is suspected to fail, a new leader will be elected. Because the secondary's state is more up-to-date, PLOVER ensures if a secondary is working normally, it will always be the new leader. To do so, PLOVER doesn't let the witness become the leader in the leader election. After the secondary becomes the leader, it will do a VM migration from itself to the witness, elevate the witness to be the secondary, and start to serve client requests.

4.6 Correctness

PLOVER is designed to handle the same failure model as regular SMR, where network messages may be lost but will not be corrupted, network may be partitioned, and hosts may fail. As an SMR system with three replicas, PLOVER can tolerate the failure of one replica.

PLOVER guarantees external consistency: if a client receives a reply for its requests, the execution states generating this reply will not be lost. Prior work [36, 38] shows that this guarantee is sufficient for VM fault-tolerance in a client-server model.

We give a proof sketch of PLOVER's external consistency guarantee in three steps. First, all replies are sent from PLOVER's leader. PLOVER's underlying PAXOS protocol ensures one strongly consistent leader among the replicas. Moreover, only the leader invokes `syncvm` operations and network outputs will not be released until a `syncvm` finishes.

Second, PLOVER does not affect the correctness of its underlying PAXOS. We made only two modifications to the underlying PAXOS protocol: always elevating the secondary to be the new leader and increasing n_{err} on a `syncvm` error (§4.1). These two modifications do not hurt PAXOS's correctness because it guarantees there is one unique leader in each view, and n_{err} is just a counter of observed communication errors on a local host.

Third, before sending out a reply, the leader has finished a `syncvm` and successfully replicated the states that generate this reply to the secondary. No matter which replica fails, the states will not be lost. Therefore, PLOVER ensures external consistency.

We also carefully designed PLOVER for reasonable

liveness. If the leader is alive, its `syncvm` operation has a timeout-and-retry mechanism and its program-idle determination (§4.3) has a bounded waiting time.

5 Implementation Details

Much of PLOVER implementation code was inherited from well-engineered VM systems [13, 38, 52], including replicating file system [38]. Our implementation found and fixed two new bugs that crashed MC [13]: one bug was an integer overflow on the number of dirty pages, the other was an inconsistent states between the PCI device and bus on restarting replicas. QEMU developers confirmed both our bug reports.

5.1 Determining Server Program Idle Status

When clients connect with services running in a VM fault-tolerance system (e.g., REMUS and PLOVER) using TCP, the system buffers network outputs and causes the clients' TCP windows to become full and to stop sending requests. This will result in an idle service and a wasted time slot. Therefore, a mechanism is needed to determine when the service is idle, so that a `syncvm` is invoked.

To efficiently find the idle status of a service, PLOVER creates a simple, non-intrusive algorithm without modifying guest OS. This algorithm uses the threading hierarchy of QEMU: all QEMU virtual threads (threads that emulate vCPUs) are spawned from the QEMU hypervisor process (§2.2). PLOVER creates an internal thread in the process to call `clock()`, which gets the total CPU clock of a process and its children. If PLOVER finds that the increment rate of this clock is as small as an vacant VM for a threshold (100μs), it finds the service idle.

This eliminates wasted time slots in PLOVER and lets services run almost in full speed. Moreover, because both the PLOVER leader and secondary finish processing current requests, their memory should be mostly the same. This simple algorithm is already effective for reducing page divergence (§6.3) and achieved reasonable performance overhead (§6.2) in our evaluation, and it can be further extended to handle straggler requests.

5.2 Computing Dirty Page Hashes Concurrently

We leveraged multi-core hardware and implemented a multi-threaded dirty page hash computing mechanism. The mechanism detects the number of CPU cores on local host creates same number of threads to compute hashes of dirty physical pages since the last PLOVER `syncvm` operation. We used Google's CityHash [43], because it is fast. Our evaluation shows that computing hashes has reasonable CPU footprint (§6.4) because it takes only about 6.3μs for each page.

5.3 Fast Consensus in Hypervisor

PLOVER uses APUS [92] to achieve consensus on network inputs among replicas. A naive approach for implementing this is to let APUS intercept network packets and synchronously achieve consensus in QEMU's inbound network device (e.g., TAP device). However, this approach causes severe performance degradation. QEMU's network is implemented in an event driven model. On receiving a network packet, the event handler needs to acquire a global lock and feed the packet into the VM. The whole process takes less than 1μs. On the other hand, APUS takes over 10μs to reach consensus. As a result, this naive approach would hold the global lock for a long period and block the handling of other events, causing great performance degradation to the VM.

To address this problem, we implemented a non-blocking consensus mechanism in QEMU. On receiving a network packet, rather than directly feed it into the guest VM, the event handler only appends the packet to a buffer. PLOVER asynchronously reads packets from the buffer, invokes APUS to achieve consensus, and leverages QEMU's event driven loop to feed the packet into the VM.

6 Evaluation

Our evaluation hosts were nine Dell R430 servers with Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. All NICs are Mellanox ConnectX-3 Pro 40Gbps connected with Infiniband [2]. To mitigate LAN/WAN network variance, all client benchmarks and VMs were run in these hosts. Running clients in WAN will further mask PLOVER overhead compared to unreplicated executions.

We evaluated PLOVER on 12 widely used programs, including 8 server programs (`Redis` [81], `SSDB` [85], `MediaTomb` [21], `Nginx` [73], `MySQL` [22], `Tomcat` [3], `PgSql` [79], and `lighttpd` [59]) and 4 dynamic language interpreters (`Node.js`, PHP, Python, and JSP). To be close to real-world deployments, we group these programs into 8 practical services, including DjCMS [7], a large, sophisticated content management system (CMS) consisting of `Nginx`, Python, and `MySQL`.

We used popular workloads that make these services run at their peak throughputs and then collected results. Prior study [76, 88] shows that hardware errors occur more frequently when services have higher load, thus the fault-tolerance of PLOVER is more crucial. Table 2 shows our workloads. For each workload, we spawned different number of clients to saturate the services and collected the curve of throughputs for unreplicated executions.

For `Redis` and `SSDB`, each request contains a batch of 1K operations of 50% SET and 50% GET; for the other six services, each request contains one operation. We found sending operations in batches for `Redis` and `SSDB` made them reach peak throughput. For instance, when each request for `Redis` contains only one SET or GET operation, `Redis`'s throughput is only 43K oper-

ation/s for 64 connections; when each request is a 1K-operation batch, its throughput reaches a peak value of 481K operation/s for 64 connections.

Service	Workload
Redis	50% SET, 50% GET requests arriving in batches.
SSDB	50% SET, 50% GET requests arriving in batches.
MediaT	Concurrent requests on transcoding a 50MB video.
DjCMS	Concurrent requests on a dashboard page [8].
PgSql	PgBench [79] with TPC-B benchmark.
Tomcat	Concurrent requests on a shopping store page [15].
lighttpd	Concurrent requests using PHP to watermark images [1].
Node.js	Concurrent requests on a messenger bot [12].

Table 2: ***Eight services and workloads used in experiments.***

We compared PLOVER with four fault-tolerance systems: CRANE [35], an open-source SMR system among recent ones [35, 45, 50]; QEMU-MicroCheckpoint [13] (for short, MC), a REMUS-based primary-backup system carried in QEMU [80]; Synchronization Traffic Reduction (STR) [63], a primary-backup system designed to reduce the number of transfered pages; and COLO [38], a primary-backup system deployed in Huawei [20]. MC has an RDMA implementation [14], but it is being actively developed and not runnable on our hosts. We did not use REMUS because it was built before 2008 and did not run on our hosts. This section focuses on six questions:

§6.1: Can PLOVER correctly enforce deterministic executions by transferring only divergent pages?

§6.2: How fast is PLOVER compared to MC, STR, and COLO? How does it scale to multi-core?

§6.3: How effective is each PLOVER technique on reducing divergence of dirty pages?

§6.4: What is PLOVER's CPU footprint and how well does it support VM consolidation?

§6.5: Can PLOVER efficiently handle replica failures?

§6.6: What did we learn from VSMR and its implementation PLOVER? What are PLOVER's limitations?

6.1 Verifying Correctness

To check whether PLOVER can capture all divergent memory pages, we took Racey [48], a nondeterminism stress testing benchmark. Racey generates many data race accesses by using multiple threads to access an in-memory array concurrently without acquiring any locks, and it computes an output based on the array content.

We wrote a shell script to repetitively launch the Racey program in PLOVER leader VM for 3K times and appended its output to a file in local VM. We compared the files between PLOVER's leader and secondary and found the files had the same content. Thus, PLOVER indeed captured and transfered all divergent pages.

VMware's documentation [9, 17] states that vSphereFT-6.5 works similar to MC. Since vSphereFT is not open source and has restrictions on publishing evaluation results [16], we compared PLOVER with MC instead.

6.2 Performance and Scalability on Multi-core

Figure 3 shows PLOVER, MC, STR, and COLO's throughput on 8 services with different number of clients. MC used 100ms-slot (MC's default) and STR used 25ms-slot (STR's own default). All experiments ran on 4-vCPU per VM (unless specified) because COLO [38] and REMUS [36] evaluated up to 4 vCPUs per VM. On average, PLOVER's throughput is 2.2X higher than MC, STR, and COLO.

As the number of clients increases, PLOVER's throughput overhead becomes less obvious. The overhead mainly comes from the `syncvm` operations, which is determined by the `syncvm` frequency and the time spent on each `syncvm`. When the load on the service increases, the VM takes more time to be idle, so the `syncvm` frequency becomes smaller. On the other hand, the time spent on each `syncvm` remains almost the same because PLOVER only transfers divergent pages. Therefore the `syncvm` overhead becomes smaller when the number of client increases.

`PgSql` is the only service for which PLOVER is slower than COLO. COLO compares per-connection outputs between its primary and backup and skips `syncvm` if outputs did not diverge. `PgSql` ran SQL transaction workloads and its outputs were mostly the same. Except for `PgSql`, PLOVER was several times faster than COLO.

To analyze COLO, we also looked into `SSDB`, which had concurrent SET/GET requests. We found that COLO's output divergence was frequent when data dependencies exist among connections (i.e., GET requests frequently got different responses when SET and GET requests on the same key arrived at `SSDB` concurrently). When any output in any connection had an output divergence, COLO did a `syncvm`. COLO evaluation shows that it greatly slowed down when the number of client connections was large. PLOVER is not sensitive to outputs.

Intuitively, STR should perform better than MC because it only transfers divergent dirty pages in `syncvm`. However, our evaluation found that sometimes STR's throughput is lower than MC (e.g., 64 clients in `Redis`). This comes from two aspects. First, STR uses a static `syncvm` interval and this causes many divergent dirty pages to be transferred. Second, compared to MC, STR requires extra time to compute and compare dirty page hashes.

All eight services' unreplicated executions reach their peak throughput on 64 clients except for `PgSql`; `PgSql` reaches its peak throughput on 32 clients. For the remaining of the paper, we use the peak throughput points of unreplicated executions of each service as our sample points.

Figure 4 shows the response time of the four systems

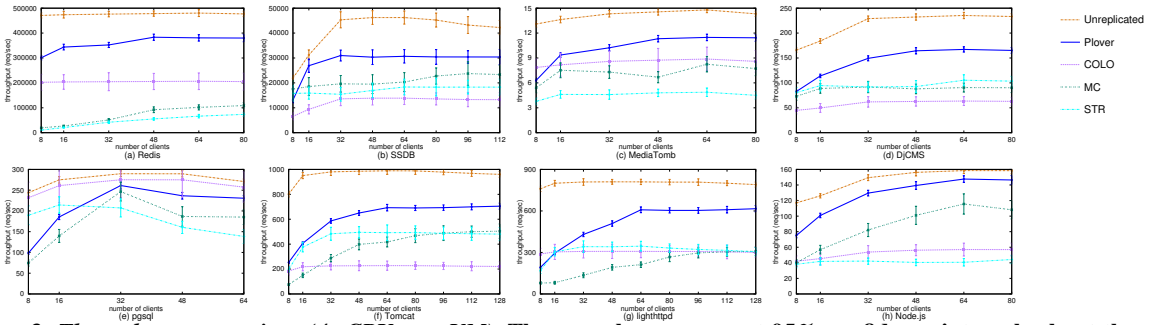

Figure 3: *Throughput comparison (4 vCPUs per VM).* The error bars represent 95% confidence intervals about the mean. For the remaining figures and tables, we use the peak throughput points of unreplicated executions as our sample points.

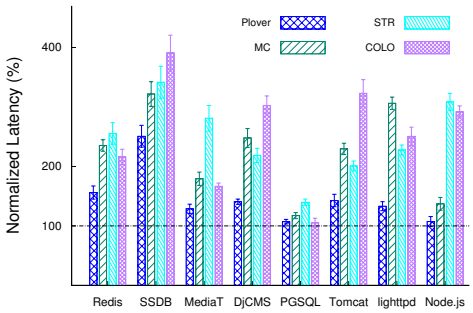

Figure 4: *Response times normalized to unreplicated executions (4 vCPUs per VM).* 100% means no overhead.

normalized to unreplicated executions. For six services (excluding `Redis` and `SSDB`), PLOVER's overhead of response time follows the same trend as the overhead of throughput because each client connection in these six services sends requests one by one. For `Redis` and `SSDB`, because the requests arrive in batches in order to saturate the two services, all four systems incur high overhead on response time. Specifically, PLOVER incurred the highest latency overhead for `SSDB`, because its same dirty page rate was only 77% (Table 3).

Figure 5 explains why PLOVER's performance was higher. PLOVER consumes 9.2X less bandwidth than MC, STR, and COLO on average. This reduction makes PLOVER the first VM fault-tolerance system that supports consolidating multiple VMs on a host (§6.4).

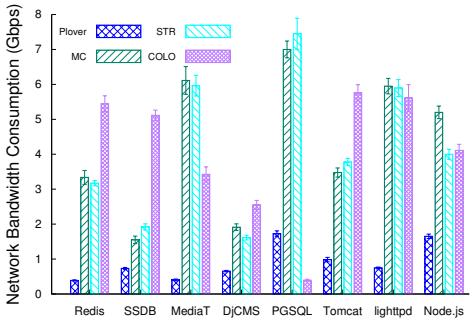

Figure 5: PLOVER *network bandwidth consumption compared with STR, MC and* COLO *(four vCPUs per VM).*

To understand PLOVER's performance, Table 3 shows its micro events. For all evaluated services, we observed that 72%∼97% pages between leader and secondary were the same. This greatly reduced page transferring time, a major performance bottleneck in primary-backup systems such as MC. The time between every two `syncvm` operations largely varied, which reflects that PLOVER can automatically detect service idle time (§5.1) for diverse workloads.

Table 4 shows that MC-25ms and MC-100ms have similar performance: a larger `syncvm` time slot accumulates more dirty pages, and thus much longer copy time and transfer time (§3.3). Combining Table 3 and Table 4 explains why PLOVER was much faster than MC: PLOVER only needs to transfer 3% ∼ 28% of dirty pages.

The results of STR-25ms and STR-100ms are similar. In our experiments, the throughput difference between these two systems was 18% on average. This is because STR uses a static `syncvm` interval, in which primary and secondary process different number of client requests. As a result, both STR-25ms and STR-100ms have low same dirty page rate and need to transfer most of the dirty pages. Therefore, we only focus on evaluating MC-100ms and STR-25ms (their own default settings) in the following sections.

To evaluate the effectiveness of RDMA in PLOVER's implementation, we changed PLOVER's dirty page bitmap and divergent page transfer mechanisms from RDMA to TCP (PLOVER-TCP). We ran the 8 services with PLOVER-TCP and found that, compared to PLOVER, PLOVER-TCP's overall throughput dropped by 2.1% ∼ 9.8%. We found that PLOVER-TCP increased the time spent in the two transfer mechanisms by 35.1% ∼ 74.2%. Because neither of the two mechanisms is PLOVER's performance bottleneck, PLOVER's high performance mainly stems from greatly reducing the pages that need to be transferred rather than RDMA.

We also evaluated PLOVER scalability on up to 16 vCPUs per VM. Figure 6 shows the scalability results on four services, normalized to PLOVER throughput on four vCPUs. The throughputs of the other four services were not scalable to multi-core (e.g., `PgSql` is I/O bound and its throughput increased by only 14.7% when we changed the number of vCPUs per VM from 4 to 16),

Service	Compute	Compare	Trans	Interval	Page	Same
Redis	3.5ms	1.9ms	2.8ms	153ms	13.5k	93%
SSDB	2.3ms	1.4ms	6.2ms	180ms	9.1k	77%
MediaT	7.9ms	4.0ms	17.1ms	914ms	29.2k	86%
DjCMS	0.9ms	1.3ms	3.3ms	90ms	3.6k	74%
PgSql	2.8ms	1.5ms	8.3ms	93ms	11.1k	76%
Tomcat	1.1ms	0.6ms	3.6ms	78ms	4.3k	72%
lighttpd	9.4ms	5.0ms	2.8ms	86ms	33.9k	97%
Node.js	9.6ms	5.5ms	28.8ms	375ms	37.8k	74%

Table 3: PLOVER *performance analysis for each* `syncvm` *operation (on average)*. "Compute" means the time of computing hashes for dirty pages; "Compare" means the time of comparing hashes between leader and secondary; "Trans" means the time of transferring divergent pages; "Interval" means the time between two `syncvm` detected by PLOVER (§4.3); "Page" means the number of dirty pages in each `syncvm`; "Same" means the same rate of dirty pages.

	MC-25ms			MC-100ms		
Program	Page	Copy	Transfer	Page	Copy	Transfer
Redis	6.1k	6.6ms	30.2ms	11.0k	11.9ms	35.1ms
SSDB	2.7k	2.9ms	7.8ms	4.8k	5.2ms	20.0ms
mediaT	4.6k	5.1ms	20.5ms	3.8k	4.2ms	16.5ms
DjCMS	2.8k	3.1ms	9.0ms	3.8k	4.1ms	13.2ms
PgSql	7.9k	8.5ms	39.0ms	8.2k	8.9ms	40.9ms
Tomcat	6.5k	6.5ms	15.6ms	12.2k	13.2ms	39.8ms
lighttpd	33.3k	23.9ms	53.5ms	33.9k	11.6ms	55.7ms
Node.js	11.3k	11.6ms	36.7ms	21.3k	14.9ms	42.5ms

Table 4: *MC performance analysis for each* `syncvm` *operation (on average) with 25ms and 100ms time slot*. "Page" means the number of dirty pages in each `syncvm`; "Copy" means the time for copying dirty pages (§3.3); "Transfer" means the time for transferring ditry pages.

so the four services do not need the 16-vCPU speedup. Overall, PLOVER scaled well for all four services, and its throughput was 3.8X higher than MC, STR, and COLO on 16-vCPU VMs. When the number of virtual CPUs increased from 1 to 16, the throughput for COLO, STR, and MC reached a bottleneck at 4 cores and even dropped for `SSDB` and `MediaTomb`. Prior study [36, 40, 42] points out a main reason of this huge drop: the number of dirty pages a primary-backup approach has to transfer will increase greatly when more vCPUs are added into one VM.

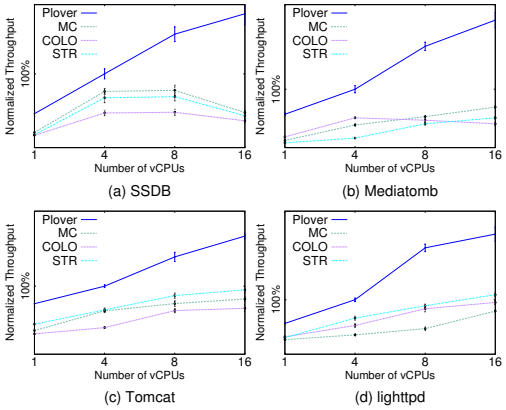

Figure 6: *Throughput scalability on the number of vCPUs per VM, normalized to* PLOVER*'s 4-vCPU throughput.* The more vCPUs per VM, the faster PLOVER than STR, MC and COLO.

6.3 Effectiveness of PLOVER Reduction Techniques

PLOVER's high performance is mainly brought by two techniques: same total order of inputs and efficiently determining service idle time (§4.3). To assess their effectiveness, we used three plans.

First (Plan1), we implemented a per-TCP-connection input forwarding mechanism between leader and secondary to order each TCP connection separately and used a 25ms `syncvm` time slot. Second (Plan2), we enforced a total order of inputs for all connections between leader and secondary, and used a 25ms `syncvm` time slot. Third (Plan3), we ran the full PLOVER.

For all three plans, we measured Same Dirty Page Rate (SDPR): the percentage of same dirty physical pages between two replicas. The difference between Plan1 and Plan2 shows the effectiveness of total ordering of network inputs between leader and secondary. The difference between Plan2 and Plan3 (PLOVER) shows the effectiveness of determining service idle time. When PLOVER is configured with a static `syncvm` interval (25ms), it has an average of 5.1% higher SDPR than Plan2. This shows that using PAXOS instead of STR to order network inputs incurs only a small cost.

Figure 7 shows that, the SDPR for 8 services differed by 25.5% on average between Plan1 and Plan2, and the difference between Plan2 and Plan3 was 29.2% on average. Both PLOVER's two techniques were quite effective on improving SDPR and the performance of services.

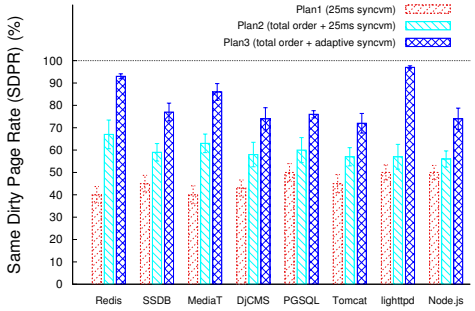

Figure 7: *Effectiveness of* PLOVER *techniques on reducing divergent pages.*

6.4 CPU Footprint and Consolidation

Figure 8 shows PLOVER's CPU footprint on the 8 services compared with unreplicated execution in KVM, MC and COLO. STR's CPU footprint is not included in the figure because it is similar to PLOVER's. Both PLOVER and COLO let their leaders and secondaries execute clients' requests concurrently. MC's secondary does not execute clients' requests but is busy applying updated states. Different from COLO and MC, PLOVER has a witness which consumed 7% $\sim$ 15% CPU to agree on network inputs without executing them.

Except for `Redis`, PLOVER's leader and secondary incurred 2.7% $\sim$ 9.2% and 5.3% $\sim$ 18.3% more CPU than

unreplicated executions, including computing hashes, comparing hashes and transferring divergent pages. PLOVER's CPU footprint was not significant for two reasons. First, computing hash for each page only took 6.3μs. Second, by transferring only divergent pages, PLOVER saved much CPU on transferring pages. For `Redis`, all three systems incurred obvious CPU footprint because `Redis` is single threaded, so its unreplicated execution only used 1 out of the 4 vCPUs.

We also evaluated PLOVER's performance on VM consolidation. We deployed one to five PLOVER leader VMs (each with 4 vCPUs) on a 24-core host, ran `PgSql` in each VM, and spawned the same number of clients for each VM. We found the total throughputs of all VMs in the host increased from 230 (one VM) to 1089 requests/s (five VMs) and the network bandwidth consumption increased from 1.8 Gbps to 10.1 Gbps. These results suggest that PLOVER is friendly to consolidating multiple fault-tolerant VMs on the same host due to its greatly reduced network bandwidth consumption compared to MC and COLO. Neither MC [13] nor COLO [38] evaluated consolidation. vSphereFT-6.5 [5] currently supports up to two 4-vCPU VMs on each physical host.

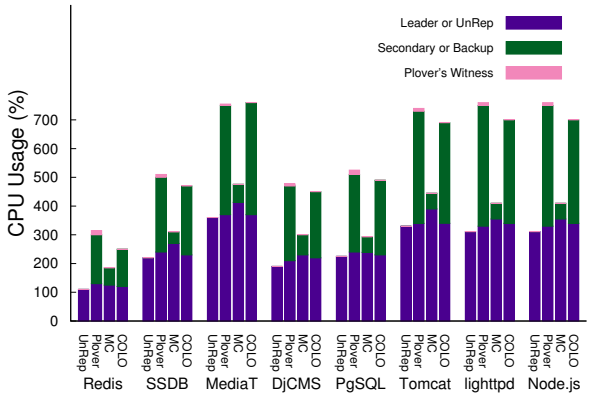

Figure 8: *CPU footprint on 4-vCPU VMs*. "UnRep" means unreplicated execution. PLOVER has 3 replicas; MC and COLO have 2. PLOVER and COLO have similar footprint.

6.5 Handling Hardware Failures

We measured the performance of PLOVER when various failures happened. We killed the leader, secondary, and witness in each experiment and monitored the real-time throughput of `Redis`. When the witness was killed, we did not observe performance impacts for `Redis`.

Figure 9 shows `Redis`'s throughput fluctuation when we killed the leader at the 3rd second and then added a new replica after a few seconds. The APUS leader election protocol [92] employed by PLOVER took a 100ms timeout to detect the leader's failure and 16.3μs to elect the secondary as the new leader. Then the new leader did a full VM migration to make the witness's guest VM up-to-date, which took about 2.8s. We also partitioned the leader out and then added it back after a second, and we found the new leader was elected almost as quickly as the leader's failure case without having a split-brain issue. Unlike existing SMR systems [35, 74] which need complex mechanisms to find the new leader's IP address, clients were not perturbed during a PLOVER leader election because VSMR replicates an entire guest VM (including its IP address) as a state machine.

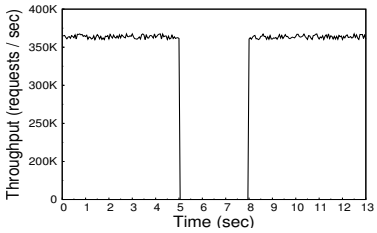

Figure 9: *`Redis`'s performance on handling* PLOVER*'s leader failure and adding a new replica*.

6.6 Lessons Learned

We found VM a promising abstraction to enforce same executions among SMR replicas. This owes to three main reasons. First, the VM abstraction can efficiently and systematically capture state changes in the guest OS, including both userspace and kernel memory. We also found VM useful on synchronizing systems nondeterminism (e.g., enforcing same physical times and ASLR layouts) across our replicas. Second, a VM carries a rich set of management primitives (e.g., migration), which makes SMR recovery easy to implement.

Third, a VM itself has several transparency features that SMR needs. For instance, the same VM replicated on different physical hosts can maintain the same IP and MAC addresses, making client connections transparently switch to the new PLOVER leader if current leader fails. In contrast, traditional SMR implementations (e.g., Raft [74]) require complex mechanisms to find the new leader. In this regard, VSMR makes SMR simpler.

PLOVER has two limitations. First, it requires the leader VM and the secondary VM to occupy the same amount of computation resources, so that they can finish processing current requests roughly at the same time and do `syncvm` efficiently. We deemed this requirement reasonable due to three reasons: (1) it is much easier to achieve in VM deployments than on bare-metal, because VMs have performance isolation and they will not overuse resources; (2) PLOVER has greatly reduced network bandwidth consumption, a major resource that may cause performance contention among VMs on the same host; and (3) requiring primary and backup to run on roughly the same amount of computation resources is a common requirement in primary-backup systems [17].

Second, as an SMR approach, VSMR requires three replicas and thus it consumes more CPU than primary-backup systems. Our evaluation shows that PLOVER's CPU consumption is compatible to COLO, about twice as

big as unreplicated executions. Nevertheless, PLOVER's robust fault-tolerance, modest performance overhead, and low network bandwidth consumption could make its extra CPU usage worthwhile.

7 Related Work

VM-based Fault-tolerance. Existing VM-based fault-tolerance systems [13, 36, 42, 62–64, 82] typically take the primary-backup approach: they propagate incremental updates from the primary VM to the backup VM. Primary-backup is more scalable than the log-replay [83] approach on multi-core because the latter needs to record exact interleavings of shared memory access. However, all typical VM fault-tolerance systems (REMUS, COLO, MC and vSphereFT) evaluated up to 4 vCPUs per VM. As most programs scale to more and more cores and access increasingly larger memory working set, transferring these dirty pages becomes a notorious problem [36, 38, 42] for the primary-backup approach, greatly degrading program performance and hijacking excessive network bandwidth.

Four recent papers aim to alleviate the open problem. First, COLO [38] lets primary and backup compare per-TCP-connection outputs and avoid dirty page propagation if no outputs diverge. COLO has effectively scaled the Remus-based approach to up to four vCPUs per VM. As shown in both COLO's and our evaluation, when the number of client connections is large or when data dependency among connections exists, COLO does many more `syncvm` operations than REMUS. PLOVER is not sensitive to output divergence.

Second, Gerofi et al. [42] shows that using copy-on-write during the dirty memory copying (t_{copy} in §3.3), primary-backup can resume VMs faster than REMUS; this work also shows that using a 10Gbps RDMA NIC can transfer dirty page faster than using a 1Gbps Ethernet NIC. Another latest work [82] also shows that RDMA can mitigate t_{copy}. These two works [42, 82] are complementary to PLOVER because PLOVER focuses on greatly reducing the amount of transferred dirty pages.

Third, Adaptive-Remus [37] shows that REMUS can monitor its output buffer and do a `syncvm` once noticing outputs. This work improves REMUS's performance by 29% when the number of client connections was small. However, with many connections, it will invoke a `syncvm` for almost every network output and incur prohibitive performance overhead.

Fourth, Tardigrade [62] uses lightweight VM (LVMs) to decrease the memory footprint on the primary to reduce checkpoint costs. On the other hand, PLOVER focuses on transferring only the divergent pages between primary and secondary to alleviate the checkpoint overhead. Besides, Tardigrade typically runs a single-process application, while PLOVER runs multiple processes (programs) in a guest VM.

State Machine Replication (SMR). Fault-tolerance is an essential technique in distributed systems [27, 29, 68]. SMR [68] is a powerful fault-tolerance technique: it typically uses PAXOS [55, 56, 68, 71, 89]) to enforce a total order of inputs for the replicated service, tolerating various failures. Many PAXOS implementation protocols [30, 31, 35, 68] exist. Consensus is widely used in datacenters [19, 49, 94] and worldwide Internet [33, 65]. Much work is done to improve specific aspects, including commutativity [66, 71], understandability [56, 74], and verification [44, 93].

To make SMR work with modern parallel programs, extra mechanisms are needed to ensure same program executions across replicas. Existing SMR systems propose a few fast mechanisms, including annotating global variables in program code [50] and enforcing same order of inter-thread synchronization [35, 45]. These mechanisms have shown reasonable performance on real-world programs, but they may require developer intervention (e.g., incorrect annotation or data races). Moreover, these mechanisms only enforce best-effort determinisms on userspace, not in kernel. PLOVER implements the new VSMR approach to realize an automatic, faster, and more scalable SMR system.

Multi-core Replay. Deterministic replay [25, 39–41, 46, 53, 54, 70, 75, 84, 90] aims to replay the exact recorded executions. Scribe tracks page ownership to enforce deterministic memory access [54]. Respec [57] uses online replay to keep multiple replicas of a multithreaded program in sync. In these record-replay systems, a false-sharing problem exists: recording becomes expensive even if multiple threads access different portions of same page. As most false-shared pages should have same contents, PLOVER may mitigate this problem.

8 Conclusion

We have presented VSMR, a novel SMR approach that makes VM fault-tolerance much faster and more scalable on multi-core. We have described PLOVER, the first VSMR system implementation and its evaluation on a wide range of real-world server programs and services. PLOVER runs several times faster than three popular primary-backup systems and it saves much bandwidth. PLOVER has the potential to greatly improve the reliability of real-world online services, and it can be applied to other research areas (e.g., multi-core replay).

Acknowledgments

We thank Jay Lorch (our shepherd) and anonymous reviewers for their many helpful comments. This paper is funded in part by a research grant from the Huawei Innovation Research Program (HIRP) 2017, HK RGC ECS (No. 27200916), HK RGC GRF (No. 17207117), and a Croucher innovation award.

References

[1] Adding watermarks to images using alpha channels. http://php.net/manual/en/image.examples-watermark.php.

[2] An Introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf.

[3] Apache tomcat. http://tomcat.apache.org/.

[4] Comparison of 40G RDMA and Traditional Ethernet Technologies. https://www.nas.nasa.gov/assets/pdf/papers/40_Gig_Whitepaper_11-2013.pdf.

[5] Configuration Maximums (vSphere 6.5). https://www.vmware.com/pdf/vsphere6/r65/vsphere-65-configuration-maximums.pdf.

[6] Data Plane Development Kit (DPDK). http://dpdk.org/.

[7] django cms - enterprise content management with django. https://www.django-cms.org/en/.

[8] Django fluent dashboard. https://github.com/django-fluent/django-fluent-dashboard.

[9] Fault Tolerance Performance in vSphere 6. https://blogs.vmware.com/performance/2016/01/vsphere6-fault-tolerance-perf.html.

[10] Implementing TCP Sockets over RDMA. https://www.openfabrics.org/images/eventpresos/workshops2014/IBUG/presos/Thursday/PDF/09_Sockets-over-rdma.pdf.

[11] Mellanox Products: RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.

[12] Pokdex messenger bot for pokmon go. https://github.com/zwacky/pokedex-go.

[13] QEMU MicroCheckpoint. https://wiki.qemu.org/Features/MicroCheckpointing.

[14] RDMA migration and rdma fault tolerance for QEMU. http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf.

[15] Simple shopping store. https://github.com/SaiUpadhyayula/SimpleShoppingStore.

[16] VMware End User License Agreements. http://www.vmware.com/download/eula.html.

[17] VMware vSphere 6 Fault Tolerance: Architecture and Performance. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere6-FT-arch-perf.pdf.

[18] Which Hardware Fails the Most and Why. http://www.storagecraft.com/blog/hardware-failure/.

[19] Why the data center needs an operating system. https://cs.stanford.edu/~matei/papers/2011/hotcloud_datacenter_os.pdf.

[20] Huawei FusionSphere. https://www.youtube.com/watch?v=yvsVuLAOhCo, 2014.

[21] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[22] MySQL Database. http://www.mysql.com/, 2014.

[23] Intermediate Course In Operating System: High Availability. www.cs.cornell.edu/ken/book/New%20514%20slide%20set/12-HighAvailability.ppt, 2015.

[24] The OTHER way of recovering from VMware ESXi Split Brain. https://www.pei.com/2017/02/way-recovering-vmware-esxi-split-brain/, 2017.

[25] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[27] K. P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, 1985.

[28] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, 1993.

[29] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[30] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[31] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[32] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, 2005.

[33] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Oct. 2012.

[34] A. Corradi, M. Fanelli, and L. Foschini. Vm consolidation: A real case based on openstack cloud. *Future Gener. Comput. Syst.*, Mar. 2014.

[35] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[36] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[37] M. P. da Silva, R. R. Obelheiro, and G. P. Koslovski. Adaptive remus: adaptive checkpointing for xen-based virtual machine replication. *International Journal of Parallel, Emergent and Distributed Systems*, 32(4):348–367, 2017.

[38] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[39] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, Dec. 2002.

[40] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, Mar. 2008.

[41] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[42] B. Gerofi and Y. Ishikawa. Rdma based replication of multiprocessor virtual machines over high-performance interconnects. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, 2011.

[43] `https://github.com/google/cityhash`.

[44] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, Oct. 2011.

[45] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[46] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[47] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, 2006.

[48] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`, 2009.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[50] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[51] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[52] `http://www.linux-kvm.org/`.

[53] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[54] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[55] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[56] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[57] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[58] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Fast replication with nopaxos: Replacing consensus with network ordering. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2016.

[59] `https://www.lighttpd.net/`.

[60] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[61] T. Liu, C. Tian, Z. Hu, and E. D. Berger. Predator: Predictive false sharing detection. *SIGPLAN Not.*, 49(8), Feb. 2014.

[62] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[63] M. Lu and T.-c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 534–543. IEEE, 2009.

[64] M. Lu and T.-c. Chiueh. Speculative memory state transfer for active-active fault tolerance. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

[65] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[66] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, 2014.

[67] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[68] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[69] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2013.

[70] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, Mar. 2009.

[71] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[72] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[73] Nginx web server. `https://nginx.org/`, 2012.

[74] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[75] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[76] S. Pertet and P. Narasimhan. Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, page 48, 2005.

[77] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014.

[78] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[79] Postgresql. `https://www.postgresql.org`, 2012.

[80] `http://www.qemu.org`.

[81] `http://redis.io/`.

[82] V. A. Sartakov and R. Kapitza. Multi-site synchronous vm replication for persistent systems with asymmetric read/write latencies.

[83] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, Dec. 2010.

[84] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[85] `ssdb.io/`.

[86] R. E. Strom, D. F. Bacon, and S. Yemini. *Volatile logging in n-fault-tolerant distributed systems*. IBM Thomas J. Watson Research Division, 1987.

[87] S. Suneja, C. Isci, V. Bala, E. de Lara, and T. Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. *SIGMETRICS Perform. Eval. Rev.*, June 2014.

[88] S. Technologies. Transient error protection. 2005.

[89] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[90] `http://www.vmware.com/solutions/vla/`.

[91] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[92] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of the Eighth ACM Symposium on Cloud Computing (Santa Clara, CA, USA*, 2017.

[93] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, Apr. 2009.

[94] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.