PLOVER: Fast and Scalable Virtual Machine Fault-tolerance on Multi-core

Paper #135

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 scalable VM fault-tolerance on multi-core 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 service performance.

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 only transferring divergent pages. An extensive evaluation on PLOVER, the first VSMR system, shows that PLOVER was 1.5X∼3.3X faster than two popular primary-backup systems. Meanwhile, PLOVER consumed 2.4X∼7.3X less network bandwidth than both of them. PLOVER source code and evaluation results are released on `github.com/sosp17-p135`.

1 Introduction

The cloud computing paradigm enables a pervasive deployment of online services in virtualized infrastructures (e.g., Xen [21]). Meanwhile, virtual machines (VM) are 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 [14] rather than occasional. A fast and scalable VM fault-tolerance approach on multi-core is highly desirable for online services.

A dominant VM fault-tolerance approach is primary-backup (e.g., REMUS [32]). This approach propagates the updated states of a primary VM to a backup VM with physical time epochs. Within each epoch, it lets the primary runs a server program to process client requests, tracks updated VM states (e.g., dirty memory pages), and buffers network outputs. As the boundary of an epoch, a `syncvm` operation is periodically invoked to transfer dirty pages from primary to backup. Finally, the primary releases outputs. By doing so, this approach automatically ensures external consistency [32]: primary and backup have the same states and a primary failure will not be observed by clients.

Unfortunately, despite much effort [10, 32, 40, 63, 79], achieving fast and scalable VM fault-tolerance on multi-core remains an open problem [4, 33, 40, 79] . A main reason is that primary-backup often has to transfer an excessive amount of dirty pages, which greatly defers the release of outputs and degrades program performance.

For instance, if a program updates 20K dirty pages each of 4KB within a 100ms epoch, transferring these pages consumes a huge network bandwidth of 6.4Gbps. Both our evaluation (§6) and prior study [32, 37, 40, 60] show that many programs access even more dirty pages on over four CPU cores. vSphereFT-6.5 [13], a latest primary-backup product, permits up to four vCPUs per VM and only two of such VMs per physical host [4]. Therefore, to enable fault-tolerance, people often have to sacrifice multi-vCPU speedup and VM consolidation [29].

As a service includes multiple programs (e.g., a website deployed in one VM can include a `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 [30, 43, 51, 59] runs the same program on a few physical hosts (or replicas), and it uses a distributed consensus protocol (typically, PAXOS [57]) to enforce the same, totally ordered inputs across replicas. For efficiency, SMR typically uses Multi-PAXOS [56]: one replica works as the "leader", which orders and proposes incoming inputs; the other replicas work as the "follower", which simply agree on inputs. Recent SMR systems [30, 43, 51, 59]) incur low performance overhead with popular programs on 16 CPU cores.

However, to ensure external consistency, SMR requires extra mechanisms to resolve divergent execu-

tions (i.e., multithreading nondeterminism [60]). Prior SMR systems provide two types of mechanisms. First, EVE [51] requires program developers to manually annotate variables shared by threads, and it detects divergent variable states at runtime. Second, REX [43] and CRANE [30] enforce same order of inter-thread synchronization (e.g., locks) across replicas. If no data race occurs, determinism is ensured; otherwise developer diagnosis [52, 74] may be needed. Neither mechanism is fully automatic as developer intervention may be needed.

Our key observation is that by enforcing the same total order of inputs for the same VM, almost all updated memory pages across the VM replicas are the same and they do not need to be transferred. Intuitively, if a key-value server replicated across replicas processes the same order of requests, all its replicas should contain roughly the same data in memory. Empirically, we used a PAXOS implementation [30] to enforce same order of client requests for 8 popular server programs running on two VMs, and we found that 52.4% to 83.1% of dirty pages in the two VMs were the same.

This paper presents Virtualized SMR (VSMR), a new SMR approach that can achieve fast and scalable VM fault-tolerance on multi-core. VSMR enforces same input order for the same replicated VM. It then periodically invokes a `syncvm` operation to efficiently compute updated page hashes, to compare them across replicas, and to transfer only the divergent pages. By doing so, PLOVER automatically enforces same execution states across replicas and ensures external consistency. By greatly reducing the amount of dirty pages that need to be tranferred, VSMR makes VM fault-tolerance much faster and more scalable on multi-core.

We implemented PLOVER,[1] the first VSMR system in Linux. PLOVER integrates a fast, RDMA-powered PAXOS implementation from an SMR system CRANE [30]. PLOVER intercepts inbound network packets in the KVM QEMU hypervisor [78] 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. Nevertheless, PLOVER does not need to modify the underlying PAXOS protocol, so it is generic to work with other consensus protocols. To maintain same file system state across replicas, PLOVER leverages a well-engineered primary-backup system's approach [10] (see §5.3).

We evaluated PLOVER on 11 widely used programs, including 8 servers (e.g., `SSDB` [82] and `Tomcat` [2]) and 3 dynamic language interpreters (e.g., PhP). We put these programs into 7 practical services, including Django CMS [6], a content management system (CMS) that consists of `Nginx` [71], Python, and `MySQL` [17]. We compared PLOVER with two well-engineered primary-backup systems QEMU-MicroCheckpoint [10] (for short, MC) and COLO [33]. Evaluation shows that:

1. PLOVER throughput is 1.5X∼3.3X faster than MC and COLO except the `PgSql` [77] program. PLOVER incurs less than 25% performance overhead on 4 out of 7 programs, and it is scalable on 4∼16 vCPUs.

2. Meanwhile, PLOVER saves much network bandwidth. PLOVER consumes 2.4X∼7.3X less bandwidth than MC and COLO. PLOVER's bandwidth advantage is stable on diverse benchmark scales.

3. PLOVER is robust to various failure scenarios.

Our conceptual contribution is a new VSMR approach, which automatically achieves much faster and more scalable VM fault-tolerance. Our other contributions includes the PLOVER implementation and an evaluation on popular programs, including a first effort to show that SMR can practically support diverse online services. Moreover, by efficiently enforcing same VMs across hosts, PLOVER can be broadly applied to other research areas. For instance, page-level false-sharing [23, 61] is a notorious performance problem in multithreading execution and replay [37, 55]. PLOVER can be an effective template to alleviate this problem, because most false-shared memory pages across host 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) [1] 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 [1] and RoCE [8]) are commonplace within a datacenter due to their ultra low latency and decreasing costs. RMDA's ultra low latency not only comes from 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 [5] and Arrakis [75]).

The advantage of RDMA latency is especially significant when transferring small message sizes. Benchmarks [3, 7] show that, with the same network interface

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

card (NIC), transferring messages with less than 2KB on RDMA is about 10X∼30X faster than on TCP. If message sizes become larger (e.g., over 8KB), RDMA latency is merely about 30% faster than TCP because network bandwidth becomes a bottleneck for both. This suggests that RDMA is attractive for invoking consensus on inputs and sending hashes of memory pages, and it is less beneficial on 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 [21, 53, 87] are widely used in clouds and datacenters due to their low performance overhead [40], platform independence, performance isolation [45], etc. For instance, the Linux KVM [53] uses hardware virtualization extensions to achieve nearly same performance as bare metal machines [40]. PLOVER uses KVM for three main reasons. First, KVM incurs little performance overhead compared to bare-metal. Second, QEMU hypervisor works in userspace is suitable for RDMA-based PAXOS to intercept inputs (RDMA currently only supports userspace memory). Third, the QEMU virtual threads that act as vCPUs are spawned from the QEMU main process, which enables PLOVER to monitor programs non-intrusively [84] without modifying guest OS.

Moreover, VM platform independence enables consolidation [29]: people can migrate many VMs [27, 70] 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 [10, 13, 32, 33, 62, 63, 79] 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 epoch to the backup before releasing outputs, the so called "output commit problem" [83]. Therefore, the major performance bottleneck of this approach is the time taken to transfer dirty pages, because local memory access speed can be 10X∼100X faster [10] 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 [40] shows that this transfer time can surpass an epoch time, greatly degrading performance.

vSphereFT used to take a record-replay approach [24, 80] for uni-vCPU, but it switches to the REMUS approach since vSphereFT 6.0 (verified in §6). If fault-tolerance is enabled, vSphereFT permits at most four vCPUs per VM and only two of such VMs per host [4]. This limitation hurts multi-vCPU speedup and VM consolidation.

2.3 PAXOS and SMR Systems

PAXOS [56, 57, 66] is a major protocol to enforce the same, totally ordered inputs across replicas. For efficiency, typical PAXOS implementations [66, 72] take the Multi-Paxos approach [56]: it elects a dedicated leader to invoke consensus on new inputs, and other replicas work as followers to agree on inputs. In PAXOS, the agreed value in each input is flexible, and PLOVER takes advantage of this flexibility. PAXOS can be used to maintain different roles consistently for different replicas [59, 69], and replicas with different roles can interpret the same agreed input value differently according to the (consistent) roles. E.g., NOPaxos's leader executes inputs; its followers agree on inputs and interpret inputs as no-operation (NOP).

To maintain roles for replicas consistently, PAXOS replicas send periodical heartbeats [66, 72] to other replicas and track the number of heartbeat failures with a threshold. If the threshold is reached on a local replica, 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, each node takes a configurable, randomized timeout [72, 76] and proposes itself as the leader. PLOVER (§3) leverages role consistency and configurable timeouts in PAXOS.

Three recent SMR protocols, NOPaxos [59], CRANE RDMA [19], and DARE [76] incur a low consensus latency of tens of μs. CRANE RDMA is an RDMA-powered implementation of the TCP-based PAXOS protocol in CRANE [30]. PLOVER chose CRANE-RDMA for three main reasons: (1) it provides a flexible `paxos(void *op)` API to propose a consensus request with `op` as the proposed value; (2) its consensus protocol includes a durable storage (DARE works purely in memory); and (3) it is open source (NOPaxos is not yet).

3 Overview

3.1 Deployment Suggestion

PLOVER deployment follows a typical SMR system: 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; having more GPU cores is even better. By running three replicas, PLOVER can tolerate hardware failures or network partitions on one replica. This fault-tolerance 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 [13, 32].

We suggest more CPU or GPU cores because PLOVER uses spare cores to compute dirty page hashes. Our evaluation used 24-core hosts and PLOVER performance was already reasonable. RDMA also becomes preva-

lent [67, 76]. 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 [59]) and using other OS bypass techniques (e.g., Arrakis [75]) to send page hashes.

3.2 PLOVER Architecture

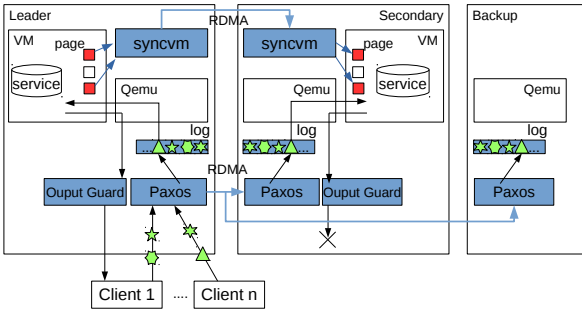

Figure 1: PLOVER *Architecture.* Key components are in blue, inputs are in green, divergent dirty pages are in red.

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 follower), PLOVER invokes PAXOS to consistently maintain three replica roles: leader, secondary, and follower. In PLOVER's underlying PAXOS level, both PLOVER's secondary and follower are "PAXOS followers" which simply agree on consensus requests. The only difference is on interpreting `syncvm` in the upper PLOVER system level: the PLOVER leader and secondary invole the `syncvm`, and the PLOVER follower interprets `syncvm` as NOP. We made this design decision because transferring divergent pages to only the secondary is efficient. Prior systems (e.g., NOPaxos [59]) also show that it is safe for replicas to use consistent roles to interpret the same agreed operation differently.

Second, to minimize service downtime during the leader's failures, PLOVER shortens its secondary's local PAXOS election timeout. This gives the secondary a much higher chance to become a new leader. The reason is that the secondary's states are more up-to-date than the follower. Shortening the secondary's election timeout is just one hint for PLOVER efficiency, not crucial for correctness. Even if the follower is elected as the new leader, PLOVER invokes a VM migration for it to get up-to-date states (see §5.4).

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 coordinator resides in all three replicas to agree on the same order of input requests with `syncvm` operations for all replicas. On PLOVER starts, PAXOS elects one replica as the leader, which is dedicated to receive and order client requests. The leader also periodically invokes a consensus on a `syncvm` if it detects that all programs running in local VM finish processing requests and become idle, effectively reducing page divergence (§4.3).

On a successful consensus for an input or `syncvm` operation, PAXOS appends the operation to a consensus log. All external inputs are forwarded to the VM's inbound NIC in the same total order across replicas. Enforcing same order of inputs are effective on enforcing same memory states across for the replicated VMs. §6.4 shows that, by only enforcing same order of realistic workload inputs for different VM replicas for 11 programs, 52.4% ∼83.1% of the programs' memory are already the same and do not need to be transferred.

PLOVER's output guard works similar to the one in primary-backup. The guard in all three replicas buffers all program outputs since the last `syncvm` operation. On a `syncvm` successes, the leader's guard release outputs to clients, and the secondary and follower discard outputs.

The `syncvm` component is invoked when a `syncvm` operation is appended to the consensus log. The leader interprets a `syncvm` with three steps: it (1) exchanges dirty page bitmaps with secondary and computes hashes of local dirty physical pages, (2) receives hashes from the secondary and compare hashes, and (3) transfers only the divergent pages. The secondary interprets a `syncvm` operation with the first two steps as the leader, it then receives divergent pages from the primary. §4.4 describes our `syncvm` protocol in detail.

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

3.3 Comparing PLOVER and primary-backup

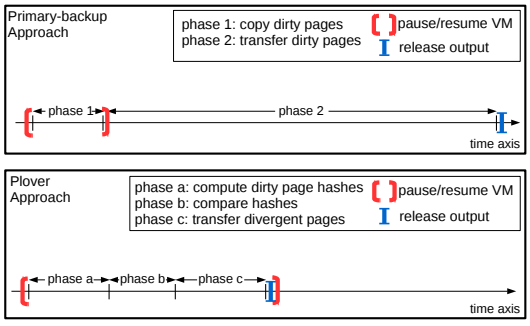

Figure 2: *Comparing `syncvm` operation in VSMR and MC.*

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 resolve network partition out-dated leadership. In contrast, primary-backup is known not able to handle network partition or out-dated leadership, the so called "split-brain problem" [80]. Because hardware failures may cause transient network partitions (e.g., NIC or network switch errors), PAXOS's strong fault-tolerance is increasingly useful.

PLOVER is fast when the leader VM and the secondary VM process requests at roughly the same speed, then they both can start `syncvm` almost at the same time. This equal-speed requirement is just for improving performance (i.e., if the secondary finishes processing requests later than the leader, the leader will wait at `syncvm` for some cycles), not for correctness. Primary-backup has the same requirement because its backup frequently applies received dirty pages to guest OS. §6.6 discusses this requirement in detail.

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 [10], a recent REMUS-based implementation developed in QEMU [78]. Our evaluation (§6.1) verified that vSphereFT-6.5 [13] uses a similar approach as MC. 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 on copying dirty pages to another memory region; and (2) $t_{transfer}$, time taken on transferring dirty pages. The primary typically 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 (confirmed in our evaluation).

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 OS and releases outputs after transferring the divergent pages. PLOVER resumes the guest after the transfer because it saves the page copy time by using RDMA to directly write divergent pages to secondary.

Compared to primary-backup, phase (a) in PLOVER can be fast by leveraging parallel CPU/GPU, and phase (b) can be fast by leveraging RDMA. Phase (c) can also be fast if most dirty pages between the PLOVER leader and secondary are the same. Our evaluation shows that PLOVER's $t_{divergent}$ is 13.1X faster than MC's $t_{transfer}$.

4 The PLOVER Runtime System

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

API	Argument	API Semantic
`paxos_second`	secondary ID	Propose the secondary
`paxos_input`	input packets	Propose client requests
`paxos_sync`	`syncvm`	Propose a `syncvm` operation
`paxos_nop`	NIL (empty)	Propose a NOP operation

Table 1: *Consensus operations in* PLOVER.

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 follower) that has been agreed by PAXOS, `vid` is the current PAXOS view ID [66], `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 four SMR operations in total: `paxos_second`, `paxos_input`, `paxos_syncvm`, and `paxos_nop`.

The `paxos_second` API is invoked every time after a new PLOVER leader is elected or after the secondary is suspected to fail. This API is invoked by the PLOVER leader to ensure that a new secondary is consistent agreed among PLOVER replicas. The PLOVER leader invokes this API by checking the host ID of the latest replied consensus message in the underlying PAXOS and proposes this ID as the new secondary. This operation complies with PAXOS safety guarantee even if the proposed host fails immediately after the consensus is reached, because the leader's `syncvm` operations can detect the new secondary's fail by incrementing n_{err} (§4.4) and starting a new election.

The `paxos_input` operations are invoked by the leader when an inbound network packet arrives at local hypervisor. Both PLOVER secondary and follower act as "PAXOS followers" to agree on the proposed packet, a standard PAXOS consensus process for new inputs.

The `paxos_syncvm` and `paxos_nop` operations appear in pairs for the `syncvm` operation (§4.4) between PLOVER leader and secondary. When the primary determines the programs running in local VM have finished processing inputs and become idle (§4.3), it invokes a consensus on `syncvm` by invoking a `paxos_syncvm` operation followed by a `paxos_nop`. This has two benefits. First, invoking a `syncvm` with consensus can divide the sequence of client requests into exactly the same for different replicas, greatly reducing memory divergence (confirmed in §6.4).

Second, adding a `paxos_nop` consensus operation

makes the PLOVER runtime system simple and generic to various PAXOS implementations. In practical PAXOS practice [66], the leader carries the acknowledgement of one input consensus in the consensus request of the next input in order to reduce PAXOS messages. Therefore, PLOVER invokes a `paxos_nop` next to a `paxos_syncvm` for its secondary to know this acknowledgement and to execute the `syncvm` immediately.

4.3 Efficiently Determining Epoch Boundary

Similar to primary-backup on ensuring external consistency [83], PLOVER leader must buffer all clients packets before a `syncvm` succeeds, including response contents and TCP ACKs. Client programs will stop sending new packets when their TCP congestion windows are met, even server programs have finished processing current requests and become idle. Existing primary-backup systems use a constant physical time epoch (e.g., 25ms in REMUS and 100ms in MC), and we found unnecessary program idle slots when running them.

To avoid unnecessary idle slots, PLOVER develops an adaptive-epoch algorithm by inserting `syncvm` operations when its leader determines idle status of programs running in guest OS. PLOVER leverages QEMU's threading hierarchy to spawn an internal thread that periodically calls the Libc `clock()` for every 1ms, and it determines the programs as idle if the growth of `clock()` is lower than a threshold for 3ms. §5.1 describes implementation details. This simple, non-intrusive algorithm helps PLOVER quickly proceed its epochs with high 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 pause its virtual machine immediately, while the secondary does the pause when its programs become idle (§4.3). When both virtual machines are paused, the leader and secondary exchange their dirty page bitmap and computes a union of the two bitmaps. Then, the leader and secondary concurrently compute hashes of pages 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 save the pages in a static buffer, sends an ACK to the leader, and applies divergent pages to its guest OS in a transactional manner (§5.3). On receiving the ACK, the leader release the output since last `syncvm` and resumes its guest OS.

All the three phases carry the sender's `vid` and the receiver checks `vid` as a standard PAXOS way [30, 66]. If any communication errors happen 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 be racy, this variable is just a statistic and thus correctness does not matter.

4.5 Fault-tolerance Correctness

PLOVER is designed to provide the same fault-tolerance strength as regular SMR. Under an SMR failure model, requests may be lost but will not be corrupted, network may be partitioned, and hosts may fail.

PLOVER ensures that all three VM replicas receive the same total order of client requests by using PAXOS to intercept inbound packets in replicas' QEMU hypervisors (§4.2). By doing so, the programs being replicated across VM replicas will see the same order of inputs. Moreover, PLOVER's program outputs are made consistent to the external world by buffering network and disk outputs within PLOVER's epochs, each of which takes a `syncvm` as boundary (§4.4).

One major correctness concern may come from PLOVER's secondary, including its role and its actions, and we provide a correctness proof sketch. Because the secondary is used to do `syncvm` with the leader, correctness is ensured if both communication sides of a `syncvm` operation always know who is the up-to-date secondary. First, we prove the leader side. Because leader's role is guaranteed to be consistent by PAXOS, and only the leader proposes who is the secondary, so the leader always knows the up-to-date secondary. Moreover, leader uses n_{err} to detect a suspected failure of current secondary and it can propose a new one (to ease discussion, we name this new proposal a "secondary-change" operation in this proof).

Second, we prove the secondary side. Note that every replica must execute all operations agreed by PAXOS in order, and any "secondary-change" is a PAXOS consensus operation. Therefore, an up-to-date secondary must execute this "secondary-change", know its current role, and involve `syncvm` operations accordingly. An outdated secondary is harmless: it either executes this "secondary-change" and downgrades itself to a follower, or it cannot involve any `syncvm` operation because the leader knows and only works with the up-to-date secondary. In summary, the role and the actions of PLOVER's secondary are correct.

We also carefully designed PLOVER for reasonable liveness. If the leader fails, a new leader will be elected by PAXOS. The `syncvm` operation has a timeout-and-retry mechanism. If any communication failure happens during synchronization, the leaders abort the ongoing `syncvm`, increment its local n_{err}, and re-invokes another

syncvm consensus. Leader must not release output until a syncvm succeeds. Secondary also has the same timeout but it does not retry. It just executes next consensus operation. The program-idle determination implementation (§4.3) also has a bounded waiting time to tolerate replica speed variance (a break on this bounded wait time only increases page divergent rate).

5 Implementation Details

Much of PLOVER implementation code was inherited from well-engineered VM systems [10, 33, 53], including replicating file system [33]. Our implementation found and fixed two new bugs that crashed MC [10]: 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

Because most client-server connections are TCP, and all output buffer systems [13, 32, 33] will have server idle status when server programs finish processing current requests and clients' TCP windows are full.

To efficiently determining these status and inserting PLOVERsyncvm operations to minimize inter-replica page divergent rate, PLOVER develops a simple, non-intrusive algorithm without modifying guest OS. This algorithm leverages the threading hierarchy of QEMU: all QEMU virtual threads (threads that emulate vCPUs) are spawned from the QEMU main process (§2.2). PLOVER runs an internal thread in QEMU to call `clock()` every 1ms and compare the increment of processor time with a threshold (default is 1,200, ticked by the guest OS's idle process). If the increment is smaller than the threshold for three consecutive checking, PLOVER regards the guest VM as idle. Then, PLOVER inserts an syncvm, transfers only divergent pages, and releases outputs. This makes server programs run in almost full speed (§6.2). Moreover, because both the PLOVER leader and secondary's server programs finish processing current requests, their memory should be mostly the same. This algorithm is effective on reducing page divergence (§6.4).

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 PLOVERsyncvm operation. Via leveraging our 24-core machines, hash computation time does not appear a performance bottleneck (§6.2). One can leverage the emerging multi-GPU hardware to further speedup our mechanism, and we leave it for future work. We used Google's CityHash [41], because it is fast.

5.3 Transactional Memory and Disk Update

A naive way to transfer divergent pages is to let the leader directly use RDMA to directly write divergent physical pages to the secondary corresponding page address. However, this can cause an inconsistent memory state on the secondary if the leader crashes during sending pages. PLOVER secondary must apply the divergent pages transferred from leader in a transactional manner. Therefore, PLOVER uses RDMA to send divergent pages from the leader to a static, registered RDMA region on the secondary. The secondary firsts copies its local divergent pages to a local region and then applies the leader's transferred pages. This transactional memory update makes PLOVER secondary's memory consistent.

To maintain the same file system state across replicas, PLOVER takes the implementation [9] from QEMU MicroCheckpoint [10] (for short, MC), a primary-backup system. PLOVER maintains same file system state with two rules. First, only the PLOVER leader writes to all replicas' file systems; this has been implemented in MC by copying and forwarding leader's disk write operations to non-leaders, and non-leaders withold these operations until a PLOVERsyncvm. PLOVER non-leaders's VMs do not write directly to local disk but use a memory buffer to hold the modified disk sectors, which is also implemented in MC. Second, on a successful syncvm, PLOVER's non-leaders release the witheld disk operations, then PLOVER replicas have same file system state.

5.4 Handling Replica Failures

PLOVER automatically tolerates one replica failure, and it leverages the VM migration feature [27, 70] to handle replica failures. If the follower's hardware fails, no PLOVER actions are needed because one non-leader failure does not affect PAXOS performance. If the secondary's hardware fails, the leader invokes a consensus on a new secondary (§4.2. Once consensus is reached, the leader does a VM migration provided by KVM for its own guest VM state to the new secondary.

A failure of current PLOVER leader triggers a standard PAXOS leader election. Because the secondary timeout has already been shortened by PLOVER (§4.2), most of the time the secondary will propose itself as the new leader earlier than the follower. Once it is elected, it proposes the other replica as a new secondary (§4.2) and invokes a VM migration to transfer all guest OS states to the new secondary. Even if the follower gets elected as a new leader (election timeout has randomness), it also proposes the other replica as a new secondary and asks for a VM migration from the other one. If one replica fails forever, PLOVER allows developers to add a new host as a follower to form a three-replica group.

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 [1]. The `ping` latency between every two replicas is 84 μs (the TCP/IP over RDMA round-trip latency). To mitigate LAN/WAN network variance, all client benchmarks and VMs were ran in these hosts. A larger network latency will further mask PLOVER overhead (e.g., input consensus and `syncvm` operations) compared to unreplicated executions.

We evaluated PLOVER on 11 widely used programs, including 8 programs (`Redis`, `SSDB`, `MediaTomb`, `Nginx`, `MySQL`, `Tomcat`, `PgSql`, and `Mongoose`) and 3 dynamic language interpreters (PhP, Python, and JSP). To be close to real-world deployments, we group these programs into 7 services, including Django CMS [6], a large, sophisticated content management system (CMS) that consists of `Nginx`, Python, and `MySQL`. For all 7 services, we ran them with real-world workloads or popular performance benchmarks collected from Internet. All workloads used 16 to 100 concurrent connections in order to saturate the server programs regardless the number of vCPUs. For instance, for `Redis` and `SSDB`, we spawned Redis-bench to run 100 concurrent connections. For Django CMS, `Tomcat`, and `Mongoose`, we used them to run three dynamic script languages with webpages generated by themselves [6] or collected from Internet, and we use `ApacheBench` [16] to run concurrent requests. Table 2 shows all services with workloads. These services all update or store important data or files, thus PLOVER's strong fault-tolerance is attractive to them.

Service	Programs	Workload
`Redis`	Self	50% SET, 50% GET
`SSDB`	Self	50% SET, 50% GET
`MediaTomb`	Self	Transcoding videos
Django CMS	`Nginx`, Python, `MySQL`	Concurrent HTTP
`Tomcat`	`Tomcat`, JSP	Concurrent HTTP
`PgSql`	Self	PGBench
`Mongoose`	`Mongoose`, PhP	Concurrent HTTP

Table 2: ***7 Services and workloads.***

We compared PLOVER with three recent fault-tolerance systems: CRANE [30], an open-source SMR system among recent SMR systems [30, 43, 51]; QEMU-MicroCheckpoint [10] (for short, MC), a recent Remus-based primary-backup system developed by the QEMU team; and COLO [33], a primary-backup system developed by Intel and Huawei. MC is developing an RDMA version to speedup page transfer, but this version was not runnable on our hosts yet. Using RDMA to transfer many dirty pages would not get much speedup over TCP (up to 30%, see §2.1), and the network bandwidth consumption remains huge. We did not use REMUS [32] because it was developed in 2008 and did not run on our hosts.

We also collected evaluation results for vSphereFT-6.5 [13], the latest VMWare product based on primary-backup. Because we are waiting for VMWare's approval [12] on publishing results, our evaluation only verified (§6.1) whether vSphereFT's approach is equivalent to MC's. We provided exact benchmark command line in §6.3 to evaluate an `SSDB` key-value server, and people can assess PLOVER's performance advantage over vSphereFT by running vSphereFT on their own.

The remaining of this section focuses on six questions:

§6.1: Can PLOVER correctly enforce deterministic executions by only transferring divergent pages? Is vSphereFT's approach equivalent to MC's?

§6.2: How does PLOVER performance compare to REMUS and COLO? How does it scale to multi-core?

§6.3: How does PLOVER, REMUS, COLO performance scale with the amount of dirty pages by varying workload scales?

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

§6.5: Can PLOVER efficiently handle replica failures?

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

6.1 Verifying Correctness

To check whether PLOVER can capture all divergent memory pages, we took Racey [46], 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 same content. Thus, we were confident that PLOVER captured and transfered all divergent pages. We then ran Racey in CRANE. As CRANE uses Parrot [31] to enforce deterministic synchronization only, the files on two CRANE replicas had different contents.

We also verified whether vSphereFT-6.5 [13] works similar to MC. vSphereFT's documentation is a bit confusing because its published paper [80] in 2010 used a log-replay approach for uni-vCPU. We wrote a toy TCP server program that takes a request and spawns a number `nt` of threads (one to four), each of which modifies one byte in 10K distinct pages. The number of dirty pages in this program is linear to the number of threads spawned. We ran this program PLOVER, MC, and vSphereFT (trial version). We found that the number of pages that PLOVER transferred was a constant value of about 220 (including kernel network buffer) per 100ms regardless of thread count. For an idle Ubuntu kernel,

the number of dirty pages per 100ms was about 82∼110. This reflected that PLOVER did not transfer dirty physical pages if they didn't diverge across replicas.

For MC, its network bandwidth consumption was about 3.1Gbps when `nt=1` and 10.9Gbps when `nt=4` on our 40Gbps NIC, which roughly matched the number of dirty pages our toy program generated. The bandwidth consumption trend of vSphereFT was similar to that in MC, although we found that vSphereFT used several clever optimizations (e.g., if a memory page was dirty across different vSphereFT epochs but its content did not change, vSphereFT did not transfer the page). Then, we believed that vSphereFT uses a similar approach as MC.

6.2 Performance and Scalability on Multi-core

Figure 3 shows PLOVER, MC, and COLO throughput on 7 services. All results are normalized to unreplicated executions ran in KVM. Since workloads we ran all spawned concurrent requests with tight loops, the higher the throughput, the smaller the response time. In this figure, all three systems had four vCPUs per VM because COLO and REMUS evaluated up to four vCPUs [33]. Overall, PLOVER incurred less than 25% performance overhead for four out of 7 services, and it was faster than MC and COLO for 1.5X∼3.3X except for `PgSql` (COLO was 29.1% faster than PLOVER). we found PLOVER faster than MC on all evaluated programs.

COLO compares per-connection outputs between its primary and backup, and it skips transferring memory if outputs did not diverge. `PgSql` ran SQL transaction workloads and its outputs were mostly the same in our evaluation. Except for `PgSql`, PLOVER was several times faster than COLO. We looked into `SSDB`, which had concurrent SET/GET requests, and we found that COLO's output divergence was frequent when there were data depenency between 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 slowed down when the number of client connections was large. PLOVER is not sensitive to outputs.

In short, We found COLO faster than PLOVER when the number of client connections was small and connections had little dependency with each other; we found PLOVER faster than COLO when the number of connections was large or connections had data dependency.

Figure 4 explains why PLOVER's performance was higher. PLOVER reduced the network bandwidth consumption 2.4X∼7.3X compared to MC and COLO. In short, PLOVER was up to 3.3X faster than both MC and COLO; meanwhile, it saved up to 7.3X network bandwidth than both of them.

To understand PLOVER's performance, we analyzed

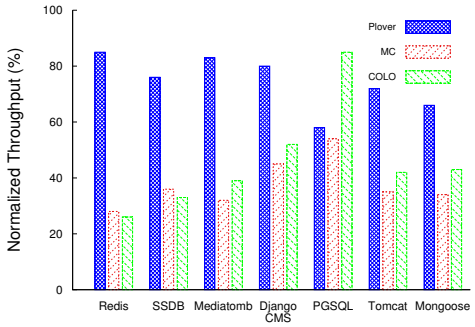

Figure 3: PLOVER, *MC, and* COLO *throughput normalized to unreplicated executions (all used four vCPUs per VM).*

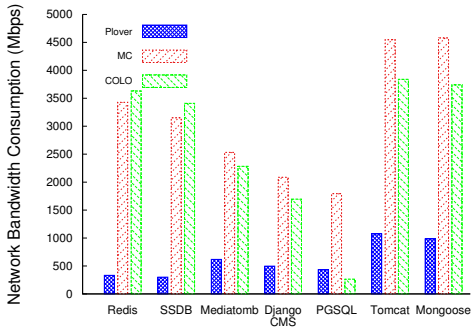

Figure 4: PLOVER *network bandwidth consumption compared with MC and* COLO *(all used four vCPUs per VM).*

its micro events in Table 3 and Table 4. Table 3 shows the mean divergent pages per PLOVER epoch between leader and secondary when workloads ran at peak performance. For all evaluated services, we observed that 73.2%∼91.1% 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 and COLO.

Table 4 shows the breakdown of PLOVER's micro performance events. The time for computing hash values for dirty pages takes around 1/3 of the total time. However, this can be alleviated when more cores are added to a machine or a better hash function is leveraged. The divergent page transfer time is much smaller because the dirty pages needed to be transferred has been greatly reduced (Table 3). Table 5 explains the performance gap between PLOVER and MC: MC's main performance bottleneck was transferring dirty pages. PLOVER's page hash computation time was smaller than MC's memory copying time because it used multithreading to compute hashes (§5.2). Prior work [79] also shows that copying dirty memory to another region on local host could be costly.

Service	Leader Dirty	Secondary Dirty	Divergent
`Redis`	21180	22334	2047
`SSDB`	18467	19645	1751
`MediaTomb`	32358	33995	7873
Django CMS	15502	15882	3747
`Tomcat`	41711	41987	9844
`Mongoose`	40367	39288	8669
`PgSql`	10043	10032	2610

Table 3: *Divergent rate of dirty pages in* PLOVER.

One caveat is that unlike MC which has a passive backup, each PLOVERsyncvm (and also COLO's) pays device resetting time (mainly for NIC and PCI) on the secondary because its secondary also actively runs. The time cost was constant and it was 6.2ms ∼9.8ms in evaluation. §6.6 discusses how to reduce this time cost.

We also evaluated PLOVER scalability on up to 16 vCPUs. Prior work [19, 31, 40, 90] showed that most of these programs reached peak throughput between 8∼16 cores. Figure 5 shows the scalability results on four programs, normalized to PLOVER throughput on four vCPUs. The throughput of the other three services were not scalable to multi-core (e.g., PgSql is IO bound and its throughput increased by only 14.7% when we changed the number of vCPUs per VM from 4 to 16), so Figure 5 did not include them. When the number of virtual CPUs increased from 1 to 16, the throughput for both COLO and MC reach a bottleneck at 4 cores and may even drop as virtual CPU. One main reason was that primary-backup's dirty page transfer time increased as more vCPUs were added (e.g., Tomcat's page transfer time surpassed MC's 100ms epoch time in Table 5).

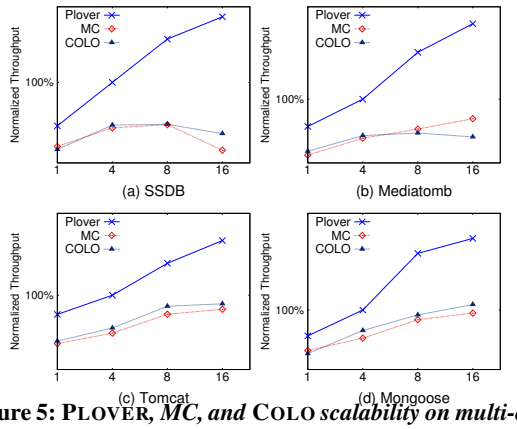

Figure 5: PLOVER, *MC, and* COLO *scalability on multi-core.*

6.3 Scalability to Dirty Pages

To test PLOVER's scalability to the number of dirty pages of a service, we run Redis within PLOVER and MC, as shown in Figure 6 with SSDB and MediaTomb. Other services' results were similar. For SSDB, our Redis-bench workload command line is "-n 1000000 -c 8 -P 500 -t TYPE -r RANGE". We spawned two terminals, each of which replaced "TYPE" to SET and GET. We then varied the key range "RANGE" from 100 to 100M. A larger key range means key-value requests contain more randomly generate keys, causing the size of SSDB's hash table as well as dirty memory pages to increase accordingly. In real-world, keys of key-value stores are large because they represent user or product IDs (e.g., in Amazon). PLOVER performance was not sensitive to SSDB's key range. MC's throughput dropped dramatically as the key range increased because it needed to transfer more dirty pages; PLOVER's performance was much more stable, and we think that is because SSDB across replicas had roughly same data in-memory.

Service	Consensus	Compute	Compare	Transfer
Redis	10.2μs	5.0ms	1.8ms	2.5ms
SSDB	9.9μs	4.5ms	1.6ms	1.7ms
MediaTomb	9.6μs	8.6ms	3.3ms	9.5ms
Django CMS	9.8μs	4.0ms	1.1ms	4.6ms
Tomcat	9.4μs	9.3ms	4.2ms	12.4ms
Mongoose	9.1μs	8.8ms	3.7ms	11.7ms
PgSql	11.3μs	2.2ms	0.8ms	3.2ms

Table 4: PLOVER *micro performance event breakdown.*

Program	Copy	Transfer
Redis	11.3ms	50.6ms
SSDB	10.8ms	48.1ms
MediaTomb	17.2ms	72.4ms
Django CMS	7.8ms	38.2ms
Tomcat	24.3ms	105.7ms
Mongoose	23.2ms	92.1ms
PgSql	5.3ms	33.2ms

Table 5: *MC micro performance event breakdown.*

We also evaluated vSphereFT running with SSDB, and we are waiting for approval to release the results. People can run vSphereFT-6.5 on their own with the command line and compare PLOVER and vSphereFT performance. vSphereFT uses a similar approach to MC.

We also varied the number of clients in MediaTomb to change the number of dirty pages. Our workloads used MediaTomb to concurrently transcode a 30MB video randomly collected from Internet from avi to mpeg, so MediaTomb throughput was not high. The number of dirty pages in MediaTomb is proportional to the number of concurrent requests. We ran MediaTomb with PLOVER and MC, and spawned from 1 to 20 clients. Figure 6 shows that PLOVER is much more scalable than MC. MC slowed down as more concurrent requests were added due to its longer page transfer time (this also explains why MC did not scale well on multi-core in Figure 5).

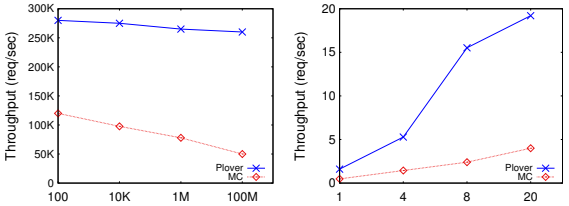

Figure 6: PLOVER *and MC scalability on increasing the amount of dirty pages.*

6.4 Effectiveness of PLOVER Reduction Techniques

The low divergent page rate in PLOVER is a result of two main techniques: (T1) enforcing same total order of inputs; and (T2) efficiently determining when server programs have finish processing requests and become idle. To assess the effectiveness of the two techniques, we chose three plans. First (Plan1), we turned off both T1 and T2 in PLOVER and used a constant epoch time of 100ms (same as MC's). Second (Plan2), we turned on T1 only in PLOVER, disabled T2, and used the same constant physical epoch time. Third (Plan3), we ran PLOVER

with both T1 and T2 enabled. Our metric was Same Dirty Page Rate (SDPR): the percentage of same dirty physical pages between two replicas among all dirty pages. Therefore, the difference between Plan1 and Plan2 shows the effectiveness of T1 in PLOVER, and the different between Plan2 and Plan3 shows the effectiveness of T2.

As shown in Figure 7, the SDPR for 7 services differs up to 19.2% between Plan1 and Plan2, and the difference between Plan2 and Plan3 was up to 32.1%. Note that the higher the SDPR rate, the reduction factor increases even much higher. For instance, if one technique improves SDPR from 80% to merely 90%, the reduction of network bandwidth consumption increases from 5X to 10X. We also tried enabling only T2 for both `Redis` and `SSDB`, and their SDPR rates was close to their Plan1 results. A possible reason is key-value stores are sensitive to inputs; a slight variance on input sequence affects SDPR rates much. Enforcing the same total order of inputs for these two programs improved their SDPR rates. Overall, by combining both T1 and T2, PLOVER greatly reduced page divergence rate than the constant physical epoch time approach (Plan1) used in typical primary-backup systems (e.g., MC).

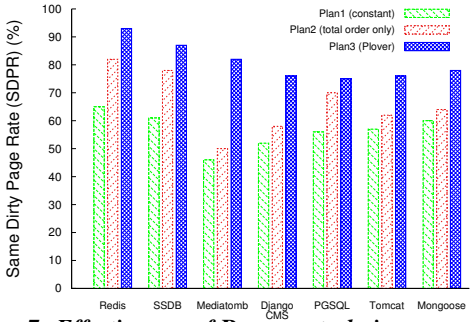

Figure 7: ***Effectiveness of* PLOVER *techniques on reducing divergent pages.***

6.5 Handling Hardware Failures

We measured the performance of PLOVER when various failure happens. We manually killed leader, secondary, and follower in each experiment with a `Redis` server running in guest VM and monitored the real time throughput from its clients. When the follower was killed, `Redis` did not show observable performance variance.

Figure 8 shows the fluctuation of the throughput when we killed only the leader. On each re-election successes, PLOVER does a VM migration to make the new leader or secondary's guest OS up-to-date (§5.4), so it took about 2.8s to make both leader and secondary up-to-date. A leader election of the underlying RDMA-based PAXOS implementation took slightly over one second to detect a host failure (heartbeat timeout was 1s) and electing a new leader took about 10.7μs [19]. When the new leader started to work, PLOVER's throughput increased slightly because consensus had only two replicas left. The fluctuation on killing the secondary was similar .

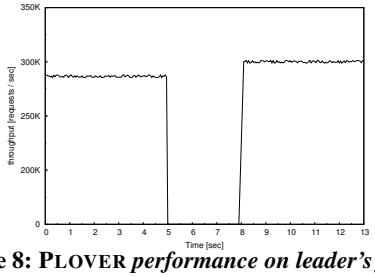

Figure 8: **PLOVER** ***performance on leader's failure.***

6.6 Lessons

We found VM a reasonable abstraction to enforce same executions for SMR replicas. This owes to three main reasons. First, the VM abstraction can efficiently and systematically capture state change 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 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 leader if current leader fails. In contrast, traditional SMR implementations [66] require complex protocols to discover the new leader. Third, a VM carries a rich set of management primitives (e.g., migration), which makes SMR recovery easy to implement (§5.4).

PLOVER's current implementation has a few performance limitations. First, compared to primary-backup systems [10, 13, 33], Each PLOVER`syncvm` need to apply leader's device states on the secondary and to restart the secondary's devices. COLO also needs this device restart for its backup. The restart took a constant time for all evaluated programs and it varied between 6.2ms to 9.8ms. Most of this time was spent on restarting the NIC and PCI devices. As future work, we plan to apply only divergent device states without restarting them, which may reduce this time down to sub milli-seconds.

As mentioned in §3.3, VSMR is fast when the leader VM and the secondary VM process requests at roughly the same speed, the they both can finish current requests in the same epoch and start `syncvm` almost at the same time. We deemed this requirement reasonable because: (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 copies CPU and device states to secondary, making them having same states, including cache status; (3) PLOVER has greatly reduced network bandwidth consumption, a major resource that may cause performance contention among VMs on the same host; and (4) requiring primary and backup to run at the same CPU hardware type and roughly the same speed is already a common requirement in primary-backup systems (e.g., vSphereFT [13]).

7 Related Work

VM-based Fault-tolerance. Existing VM-based fault-tolerance systems [10, 32, 40, 62, 63, 79] 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 approach because the latter needs to record exact interleavings of shared memory access. vSphereFT [80] initially used log-replay for unit-vCPU and it switches to primary-backup since vSphere-6.0 and supports up to four vCPUs per VM [4]. REMUS [32] shows that most of states are dirty memory pages updated by programs running in the primary within a physical time epoch (e.g., 25ms in Remus [32] and about 100ms in MC [10], a latest Remus-based implementation on KVM). As most programs scale to multi-core and access increasingly larger memory working set, transferring these dirty pages become a notorious problem [32, 33, 40] for the primary-backup approach, greatly degrading program performance and hijacking excessive network bandwidth.

Three recent papers aim to alleviate the open problem. First, COLO [33] lets primary and backup compare per-TCP-connection outputs and avoid dirty page propagation if outputs don't diverge. COLO has shown to effectively scale the Remus-based approach to up to four vCPUs. As shown in both COLO's and our evaluation, when the number of client connections is large or when data dependency among connections exist, COLO invokes even much more dirty page transfer than Remus (i.e., if any output packet diverges in any connection, COLO invokes a `syncvm`). PLOVER is not sensitive to output divergence.

Second, Lu el [62, 63] shows that the transferred pages between the primary and backup's memory within the same physical epoch can be reduced by 1.6X∼3.6X by simply comparing them via constant physical epochs, which matches the "Plan1" results in our evaluation (§6.4). Our evaluation shows that by enforcing total-ordered inputs for programs, PLOVER got a much higher reduction.

Third, Gerofi el [40] shows that using copy-on-write during the dirty memory copying (t_{copy} in §3.3), primary-backup can resume VM sooner 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 [79] also shows that RDMA can mitigate t_{copy}. These two work [40, 79] are complementary to PLOVER because PLOVER focuses on greatly reducing the amount of dirty pages that need to be transferred.

State Machine Replication. SMR takes a synchronous replication approach: it uses a distributed consensus protocol (typically, PAXOS [56, 57, 66, 69, 85]) to enforce a total order of inputs for the replicated application. Various PAXOS implementation protocols [25, 26, 30, 66] exist. Consensus is essential in datacenters [15, 47, 92] and worldwide Internet [28, 64]. Much work is done to improve specific aspects, including commutativity [65, 69], understandability [57, 72], and verification [42, 91].

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 varibles in program code [51] and enforcing same order of inter-thread synchronization [30, 43]. 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. Other approaches that enforce OS-level determinism either lack backward-compatibility [20] or incur prohibitive overhead [22]. PLOVER leverages the VM abstraction to build an automatic, faster, and more scalable SMR system.

Multi-core Replay. Deterministic replay [18, 36, 37, 39, 44, 54, 55, 68, 74, 81, 86] aims to replay the exact recorded executions. Scribe tracks page ownership to enforce deterministic memory access [55]. Respec [58], 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.

RDMA Techniques. RDMA architectures, including Infiniband [1], RoCE [8], and iWRAP [11], are commonplace in datacenters. RDMA are used to speed up high performance computing [38], key-value stores [34, 48, 49, 67], transactional systems [35, 50, 88], and file systems [89]. For instance, FaRM [34] leverages RDMA to build a fast DHT. FaRM works in a primary-backup manner [32, 73]. In general, PAXOS-based replication provides better reliability than primary-backup.

8 Conclusion

We have presented VSMR, a novel 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 two 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).

References

[1] An Introduction to the InfiniBand Architecture. `http://buyya.com/superstorage/`

chap42.pdf.

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

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

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

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

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

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

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

[9] QEMU file system block replication. https://github.com/qemu/qemu/blob/master/docs/block-replication.txt.

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

[11] RDMA iWARP. http://www.chelsio.com/nic/rdma-iwarp/.

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

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

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

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

[16] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html, 2014.

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

[18] 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.

[19] Fast and Scalable PAXOS on RDMA. HKU CS technical report No. 2017-03. http://www.cs.hku.hk/research/techreps/document/TR-2017-03.pdf, 2017.

[20] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[21] 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.

[22] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[32] 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.

[33] 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.

[34] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[35] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[36] 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.

[37] 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.

[38] M. P. I. Forum. Open mpi: Open source high performance computing, Sept. 2009.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

[49] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. Aug. 2014.

[50] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2016.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

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

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

[58] 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.

[59] 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.

[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] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

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

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

[73] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[74] 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.

[75] 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.

[76] 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.

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

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

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

[80] 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.

[81] 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.

[82] `ssdb.io/`.

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

[84] 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.

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

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

[87] 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.

[88] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, SOSP '15, Oct. 2015.

[89] G. G. Wittawat Tantisiriroj. Network file system (nfs) in high performance networks. Technical Report CMU-PDLSVD08-02, Carnegie Mellon University, Jan. 2008.

[90] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[91] 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.

[92] 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.