Enhancing TCP Throughput of Highly Available Virtual Machines via Speculative Communication

Balazs Gerofi
The University of Tokyo
bgerofi@il.is.s.u-tokyo.ac.jp

Yutaka Ishikawa
The University of Tokyo
ishikawa@is.s.u-tokyo.ac.jp

Abstract

Checkpoint-recovery based virtual machine (VM) replication is an attractive technique for accommodating VM installations with high-availability. It provides seamless failover for the entire software stack executed in the VM regardless the application or the underlying operating system (OS), it runs on commodity hardware, and it is inherently capable of dealing with shared memory non-determinism of symmetric multiprocessing (SMP) configurations. There have been several studies aiming at alleviating the overhead of replication, however, due to consistency requirements, network performance of the basic replication mechanism remains extremely poor.

In this paper we revisit the replication protocol and extend it with *speculative communication*. Speculative communication silently acknowledges TCP packets of the VM, enabling the guest's TCP stack to progress with transmission without exposing the messages to the clients before the corresponding execution state is checkpointed to the backup host. Furthermore, we propose *replication aware congestion control*, an extension to the guest's TCP stack that aggressively fills up the VMM's replication buffer so that speculative packets can be backed up and released earlier to the clients. We observe up to an order of magnitude improvement in bulk data transfer with speculative communication, and close to native VM network performance when replication awareness is enabled in the guest OS. We provide results of micro-, as well as application-level benchmarks.

Categories and Subject Descriptors C.4 [*Performance Of Systems*]: Fault-tolerance; D.4.5 [*Operating Systems*]: Reliability—Fault-tolerance, Checkpoint/restart

General Terms Design, Reliability, Performance

Keywords Virtualization; Hypervisor; Checkpoint; Recovery; Fault Tolerance

1. Introduction

With the recent increase in cloud computing's prevalence, the number of online services deployed over virtualized infrastructures, i.e., over Virtual Machines (VM), has experienced a tremendous growth. At the same time, the latest hardware trend of growing component number in current computing systems (e.g., datacenters, high-end computer clusters, etc.) renders hardware failures common place rather than exceptional [32]. Hence, designing software with fault-resilience in mind has become a major concern.

Replication at the Virtual Machine Monitor (VMM) layer is an attractive technique to ensure fault tolerance in virtualized environments, primarily, because it provides seamless failover for the entire software stack executed inside the virtual machine, regardless the application or the underlying operating system (OS).

There are currently two main approaches to primary-backup based replication of virtual machines. Log-replay records all input and non-deterministic events of the primary machine so that it can replay them deterministically on the backup node in case the primary machine fails [4], [25]. Although this solution provides high efficacy to uni-processor virtual machines, its adaption to virtual SMP environments, i.e., virtual machines with multiple virtual CPUs, is cumbersome, because it requires determining and reproducing the exact order in which CPU cores access the shared memory [24]. While virtual SMP environments are gaining increasing importance [20], it has been shown that log-replay imposes super-linear performance degradation with the increasing number of virtual CPUs on various workloads [11].

The other approach, checkpoint-recovery based replication, captures the entire execution state of the running VM at relatively high frequency in order to propagate changes to the backup machine almost instantly [6, 9, 12, 19, 35]. This solution, essentially, keeps the backup machine nearly up-to-date with the latest execution state of the primary machine so that the backup can take over the execution in case the primary fails [6]. One of the main strengths of checkpoint-recovery based replication is its inherent ability to tackle with multi-core configurations [12].

However, any fault tolerant system needs to ensure that the state from where an output message is sent will be recovered despite any future failure, which is commonly referred to as the *output commit* problem [27]. As a consequence of such requirement, output of the running VM needs to be held back, i.e., disk I/O and network traffic have to be buffered and can be released only after the backup machine acknowledged the corresponding update [6, 9, 19].

Several recent studies have explored the domain how to accelerate the failure free period of checkpoint-recovery based replication, focusing mainly on decreasing its computational overhead [12, 19, 35]. Communication performance, due to the consistency requirements mentioned above, remains notoriously poor [6]. Workloads, which require high-bandwidth TCP data transfer are particularly hurt, because TCP transmission is unable to progress as a result of the constant delay in acknowledgments from its peers.

In this paper we focus on the TCP communication performance of *replicated virtual machines*, i.e., virtual machines that are being replicated via checkpoint-recovery based replication. We make the following contributions:

VEE'12, March 3–4, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-1175-5/12/03...$10.00

- We revisit the basic replication protocol and extend it with *speculative communication*. Speculative communication fabricates and delivers speculative acknowledgments to the VM in response to its TCP packets so that the guest's TCP stack can progress with TCP transmission. In order to ensure consistency, speculative messages are not exposed to the client machines until the VM's corresponding execution state is checkpointed and the update is acknowledged by the backup host.
- We propose *replication aware congestion control*, an extension to the guest OS' TCP stack that aggressively fills up the replication buffer of the VMM so that speculative packets can be sent to the backup host and released earlier to the clients.
- Finally, a rigorous evaluation of the introduced mechanisms is given using two different interconnects, Gigabit Ethernet and Infiniband [1] (a high-performance interconnect that is gaining popularity also in data-centers) as the network for replication.

Experimental results show up to an order of magnitude improvement in bulk data transfer with speculative communication, and close to native network performance when replication awareness is enabled in the guest OS and replication is performed over Infiniband.

We begin with detailing the basic VM replication protocol in Section 2. Section 3 describes the design of our proposed mechanisms and Section 4 provides insight to the details of implementation. Experimental evaluation is given in Section 5. Section 6 surveys related work, Section 7 provides further discussion along with future plans, and finally, Section 8 concludes the paper.

2. Background and Motivation

This section provides an overview of the basic virtual machine replication protocol focusing in particular on communication aspects. It also motivates the need for further investigation how to improve communication performance.

2.1 Basic Replication Protocol

As mentioned above, checkpoint-recovery based replication of virtual machines is delivered by capturing snapshots of the running VM at relatively high frequency so that changes can be reflected to the backup machine almost instantly.

Between checkpoints the VM executes in log-dirty mode, i.e., write accessed pages are recorded so that when the checkpoint is taken only pages that were modified in the most recent execution phase need to be transferred, along with execution context of the virtual devices and vCPUs. One phase of dirty logging and transferring the corresponding changes is often called a *replication epoch* [6, 19, 35].

VM downtime during replication epochs, i.e., the period while the VM is stopped, can be eliminated almost entirely. Instead of leaving the VM paused and transferring the update synchronously, data can be buffered first and transferred in an asynchronous fashion (which we call the *basic asynchronous replication protocol* in the rest of this paper). This way the VM is stopped only for the buffering period, and execution of the next epoch can overlap the actual data transfer [6]. Copy-on-write (COW) can be used to omit VM downtime even while buffering the update [12].

When a failure occurs, the backup machine rolls back to the latest consistent state it received from the primary host and restarts the most recent replication epoch's execution phase. Since the execution path on the backup may be different (particularly, when multiple vCPUs access shared memory) than that executed on the primary machine, the primary machine's output, which may depend on such non-deterministic steps, have to be buffered and held back until the corresponding checkpoint is acknowledged by the backup.

This property of the replication protocol is consequence of the so-called *output commit* problem [27], which postulates that any fault tolerant system needs to ensure that the state from where a message is sent will be recovered despite any future failure.

2.2 TCP Communication

Transmission Control Protocol (TCP) is the most widely used network protocol for reliable data transfer over the Internet. TCP relies on acknowledgments to ensure data have been transferred entirely, and at the same time acknowledgments play a crucial role in adapting the transmission bandwidth to the properties of the underlying physical network.

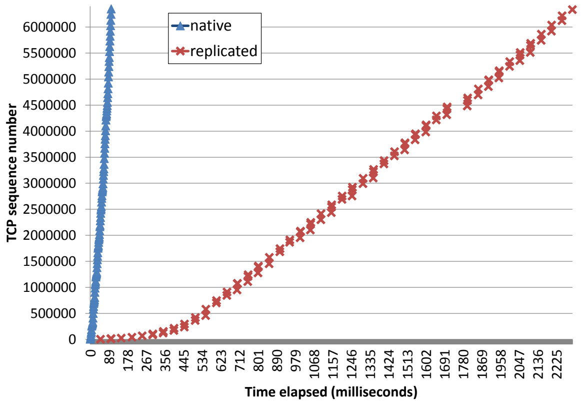

Figure 1. TCP bulk data transfer of the native and the replicated virtual machines. *Replication epoch length is set to 50 milliseconds in this experiment and Gigabit Ethernet is used as the underlying network between the VM and the client machine.*

Taking a closer look at how *output commit* affects the TCP performance reveals that the attainable network bandwidth is substantially lower compared to the native execution. Figure 1 demonstrates the difference between the native and the replicated virtual machine's TCP throughput, as recorded on the client side. The figure shows how TCP progresses with sequence numbers with respect to the time elapsed. One of the key observations is the grouped fashion of the transfer when replication is enabled. As seen, groups of packets appear approximately in every 50 milliseconds on the client machine, which is the length of the replication epoch in this experiment.

In each replication epoch the VM's TCP stack attempts to transmit a number of packets to the client, which are buffered and held back, i.e., delayed by the VMM (in order to guarantee output commit) until the end of the epoch. Consequently, acknowledgments of these packets are also delayed which prevents the VM's TCP stack to increase the congestion window at the same pace how the native execution would do. As a result of this quenched transmission, the VM's TCP stack believes the underlying network provides significantly lower bandwidth than it does in reality. In this experiment we obtain less than 6 MB/s TCP throughput from the replicated virtual machine, as opposed to the 110 MB/s of the native execution. In both scenarios, the physical link between the VM and the client machine was Gigabit Ethernet. Clearly, such degraded network performance is unacceptable for online services where clients expect reasonable bandwidth and short response times from the server machines.

3. Design

Motivated by the scenario presented above, this section discusses the idea of *speculative communication* and the role of *shadow sock-*

ets. Moreover, *replication aware congestion control*, a modification to the VM's TCP stack, which further improves attainable network bandwidth is also presented.

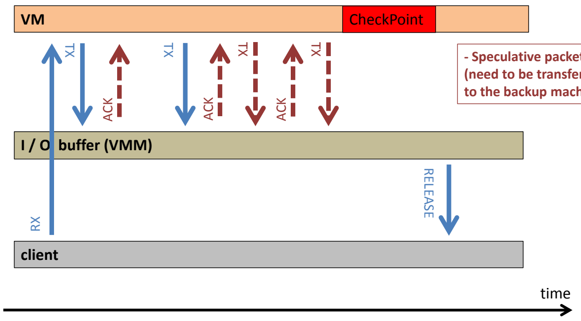

Figure 2. Speculative communication of replicated virtual machines. *Dashed arrows represent speculative acknowledgments of the VMM and speculative packets of the VM. Speculative communication allows the VM to progress with its TCP streams further than it can under the basic replication protocol. Speculative packets, however, need to be transferred to the backup host.*

3.1 Speculative Communication

The mechanism of *speculative communication* is shown in Figure 2. The basic idea is that instead of simply buffering network packets the VMM tracks established TCP connections and fabricates *speculative acknowledgments* that appear exactly the same as if the client sent them. The acknowledgments are submitted to the VM, which enables the VM's TCP stack to progress with data transfer and transmit more packets during the same period of time. In order to ensure consistency, the packets from the VM are not transferred immediately to the client machines, but are only buffered first, hence the name *speculative communication*. Once the corresponding checkpoint is acknowledged by the backup, the packets can be released. This mechanism allows the replicated VM to attain higher network throughput and better response times, however, it also opens up various questions and issues.

3.1.1 Determinism and Fault-Tolerance

One might be wondering whether or not allowing the VM to progress with its TCP streams breaks the conditions of fault tolerant execution. Since speculative packets are not exposed to the clients until the corresponding execution state's checkpoint is acknowledged by the backup machine, such packets do not depend on any non-deterministic steps of the VM that cannot be recovered in the future if a failure occurs. However, when a checkpoint is taken, the VM's TCP stack (encompassed by the execution context of the virtual machine) does represent a state where retransmission of these packets is already impossible, due to the fact that they have been acknowledged and the TCP stack has likely removed them from its retransmission queue. Consequently, the packets which are acknowledged by the VMM also need to be transferred to the backup machine so that they can be transmitted over the network again in case the primary machine fails. The main goal of speculative communication is to overlap the actual transfer on the network with the backup data transfer of the next epoch's speculative packets. (Note, that the replication takes place over a separate NIC.)

3.1.2 Correctness

Another issue is how buffered packets are released, i.e., actually transferred to the clients. TCP ensures that every single byte of a stream is received correctly (assuming no permanent error occurs on the network), and packets are retransmitted if they get lost or corrupted during the transfer. When the VMM acknowledges a TCP packet to the VM, the VM's TCP stack cannot be counted on for possible retransmissions any more, because it believes the packet reached its destination. Once data are buffered by the VMM, it becomes the VMM's responsibility to transfer that data properly and conforming to the TCP semantics. For this reason, the VMM maintains *shadow sockets* of each established TCP stream of the replicated VM. Shadow sockets ensure both necessary packet retransmissions and correct TCP congestion control of the speculative stream. The mechanism of shadow sockets, along with the necessary alternations to network packets of both incoming and outgoing traffic will be explained below.

3.2 Shadow Sockets

Shadow sockets play a central role in performing speculative communication. The VMM maintains a hash table of shadow sockets based on IP addresses and TCP port numbers of the given TCP flow. A simplified view of the shadow socket infrastructure is shown in Figure 3.

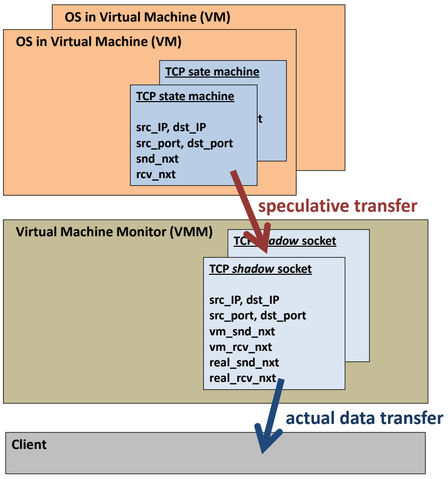

Figure 3. High-level overview of the shadow socket infrastructure. *Shadow sockets track TCP flows and perform speculative communication for the guest's TCP connections. They also are responsible for the actual data transfer to the client.*

Most importantly, a shadow socket stores information about the sequence numbers of a TCP stream, both from the point of view of the VM and the real network (i.e., the client). Essentially, speculative communication offloads the actual TCP transmission from the VM's TCP stack to the shadow socket of the VMM. Although data are transferred (during speculative communication) via TCP from the VM to the VMM, it is merely a memory copy, without any network transmission involved. The VMM stores the payload of the TCP packets from the VM and during packet release phase it transfers the data through the corresponding shadow socket. Since we utilize the Linux kernel's TCP stack to perform this task, as it will be detailed in Section 4, the actual TCP transmission over the network behaves entirely conforming to standard TCP semantics. In fact, it does so as if it was one of the host's non-virtualized TCP connections. It is also worth mentioning, that incoming traffic doesn't go through the shadow socket, but the payload is passed to the VM directly. However, special care has to be taken how incoming, outgoing and speculative packets are handled, which we describe in the next sections.

3.2.1 Connection Tracking

Our current implementation assumes that any TCP activity of the guest is initiated after the replication has been enabled. This makes TCP connection tracking slightly less complicated, because an established flow can be identified by the acknowledgment of the SYN-ACK packet. The VMM creates shadow sockets for any TCP flow once a SYN packet is identified (regardless if it originates from the VM or from the clients). When the connection enters the established state, the VMM starts performing speculative communication for the given flow.

Shadow sockets are destroyed in three possible scenarios. The most common ways are if both parties of the connection sent their respective FIN packets or if there is no activity on the flow for more than a user specified time interval. Besides these, RST flags can also indicate a connection break-down.

3.2.2 Traffic Handling

As shown in Figure 3, we use the same set of notations for TCP sequence numbers with the TCP protocol specification [31]. vm_snd_nxt denotes the next sequence number the guest OS sends, while vm_rcv_nxt is the next sequence number the VM expects on an incoming packet. Notice, that vm_snd_nxt, at the same time, is the highest valid ACK number the VM accepts. These values have to be updated according to the traffic to and from the VM. For example, when a TCP packet is sent by the VM, the corresponding shadow socket's vm_snd_nxt field has to be updated so that it stores the sequence number that equals to $seq + tcp\_length$ of the given packet. Similarly, vm_rcv_nxt will store the acknowledgment field of the outgoing packet, since by definition, that is the next sequence number the VM expects to receive. Looking at the sequence numbers of the actual TCP flow on the network, $real\_snd\_nxt$ represents the sequence number the shadow socket sends next, and $real\_rcv\_nxt$ is the next sequence number the VMM expects from the client.

The main source of complications with respect to incoming, outgoing and speculative packets is the fact that the VM's TCP stack can be ahead of the actual data transmission, due to speculative communication. Using the notations above, this means that effectively vm_snd_nxt is ahead of $real\_snd\_nxt$ when speculative communication is in action. This has several implications on how packets can be passed to and from the VM.

Speculative Acknowledgments. Speculative acknowledgments always contain vm_rcv_nxt as their sequence nr. and vm_snd_nxt as the acknowledgment. Essentially, they acknowledge the entire *in-flight* sequence interval of the VM, i.e., the sequence interval the VM believes currently resides on the network. There are two more important issues worth mentioning with regards to speculative acknowledgments.

First, it is infeasible to acknowledge every single outgoing packet of the VM one-by-one, because of the high frequency of packets during a high-bandwidth data transfer. For this reason we employ a similar technique that is widely used in TCP implementations, i.e., acknowledging smaller than maximum segment size (MSS) packets directly, but scheduling ACKs based on timers if packets with MSS size are frequent.

Second, while speculative packets are continuously transferred in the background to the backup machine (because they are part of the given epoch update), the VMM keeps track of the overall buffer size used and indicates to the VM once the buffer limit has been reached. The limit of the buffer is determined by the replication frequency and the available bandwidth between the primary and the backup machines. The VM is notified to stop transferring data by means of announcing zero window in the speculative acknowledgment. In order to resume data transfer, each shadow socket for which zero window has been announced is marked and an acknowledgment with non-zero window is sent in the beginning of the next replication epoch.

Incoming Traffic. When speculative communication is active, incoming packets cannot be passed directly to the VM's network stack, because the VM's TCP state machine may expect higher acknowledgments than that in the actual traffic on the network. This is due to the speculative acknowledgments the VMM submitted. Therefore, the ACK field in incoming packet's TCP header needs to be updated according to the shadow socket's vm_snd_nxt field. Otherwise, passing the packet would cause the VM to possibly discard it with the reason of invalid acknowledgment. Modifying the TCP header's ACK field also requires updating the TCP and IP checksums, which is performed by the VMM before passing the packet to the VM.

Outgoing Traffic. Special attention needs to be payed for outgoing packets from the VMM to the clients as well. Imagine a scenario where the client sends a packet to the VM which has non-zero payload. This will cause the VM to send an acknowledgment to the client, which is in turn buffered by the VMM. Instead of releasing this packet, the VMM needs to update the $real\_rcv\_nxt$ field of the shadow socket according to the ACK field of the packet's TCP header and send an acknowledgment with the correct sequence number, i.e., $real\_snd\_nxt$. Otherwise, the client TCP stack may receive packets that, on one hand, have sequence number ahead of the expected one (because $real\_snd\_nxt$ may be behind vm_snd_nxt), on the other hand, the client may believe that the original packet (with the payload) was lost on the network and retransmit packets unnecessarily.

3.3 Replication Aware Congestion Control

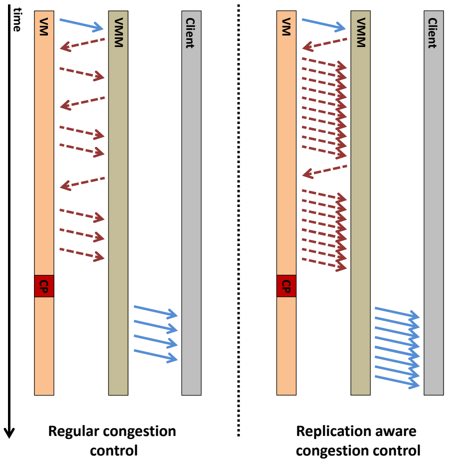
time
VM
VMM
Client
CP
Regular congestion control
VM
VMM
Client
CP
Replication aware congestion control

Figure 4. Replication aware TCP congestion control of the guest OS. *Replication aware congestion control aggressively transmits a large number of packets in order to fill up the replication buffer as soon as possible.*

As mentioned above, speculative communication essentially offloads the actual TCP transmission to the VMM's shadow sockets. However, since speculative packets have to be transferred to the backup machine first, the VMM should buffer speculative packets as fast as possible. There are two main factors that determine how fast the VM transfers packets to the VMM's buffer, the receive window size announced in the speculative acknowledgments and the congestion window size of the guest's TCP stack. Our first approach towards gathering as many packets as possible from the VM was to announce large receive window size (2MB in our current implementation) in the speculative acknowledgments.

Unfortunately, standard congestion window control algorithms (such as Vegas [3] or CUBIC [13]), which are widely used in operating systems' network stacks, are not designed for the conditions speculative communication promotes. On the other hand, congestion control policy in TCP stack implementations is often easy to alternate. Linux, for instance, provides a standard kernel interface to grow or shrink congestion window size in response to acknowledgments and different policies for congestion control can be easily implemented as loadable kernel modules.

Several prior studies have explored the idea to make the guest operating system aware of virtualization. In this paper we follow the same path, but we take the approach to the next level, where the guest OS is not only aware of virtualization, but also of high availability (i.e., it knows it runs in a replicated VM). In order to better cooperate with speculative communication we developed a very simple congestion control algorithm with a low slow start threshold followed immediately by a relatively large congestion window. Figure 4 indicates the difference between regular and replication aware congestion controls. Our congestion control algorithm aggressively transmits groups of packets together, filling up the speculative communication buffer much faster than operating under standard congestion window algorithms. In Section 5 we provide experimental results demonstrating the impact of this approach.

4. Implementation

We chose the Linux Kernel Virtual Machine (KVM) [17] as the platform of this study. KVM takes advantage of the hardware virtualization extensions so that it achieves nearly the same performance with the underlying physical machine.

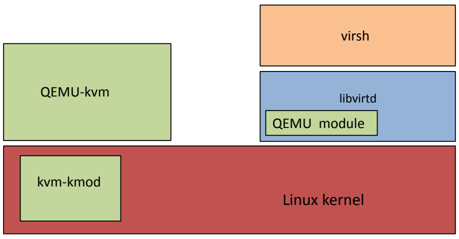

Figure 5. The Linux kernel virtual machine architecture.

Figure 5 depicts the high-level KVM architecture. The most important components are the *kvm* kernel module and *qemu-kvm*, a KVM tailored version of QEMU. On top of these, *libvirtd* is an often used facility for managing virtual machines, for which *virsh* provides a command line interface. A major advantage of the KVM architecture is the full availability of user-space tools in the QEMU process, such as threading or libraries. Similarly, the *kvm* kernel module can directly make use of any functionality provided by the Linux kernel, such as drivers or the TCP stack.

4.1 Replication Logic and Disk I/O Buffering

The replication logic is entirely implemented in *qemu-kvm*, leveraging a great amount of the live migration code.

For disk I/O buffering we modified the virtio driver of *qemu-kvm*. The disk I/O buffer behaves also as a hash table that operates on sector granularity so that read requests referring to sectors which are already buffered can be accessed consistently.

4.2 Speculative Communication and Shadow Sockets

Most of the speculative communication functionality is implemented in user-space, i.e., in the *qemu-kvm* process. Both incoming and outgoing packets are examined in the networking code of QEMU. Speculative packets are put on a *backup-queue* first, which is continuously processed by a background thread that transfers the packets to the backup machine. Packets that have been acknowledged are then put on the *transfer-queue*. However, the transfer-queue is only processed once the corresponding epoch's update is also acknowledged by the backup machine.

Shadow sockets, on the other hand, have components both in a user-space and in the Linux kernel. The user-space component is mainly responsible for tracking TCP flows from the VM's viewpoint (i.e., it takes care of vm_snd_nxt and vm_rcv_nxt) and handles speculative acknowledgments. Shadow sockets in *qemu-kvm* are created immediately once a SYN packet is detected and are maintained on a hash table so that a particular flow can be quickly identified. Kernel components are, however, only allocated when the connection enters the established state.

For the kernel-space component of shadow sockets we modified the *kvm* kernel module and extended it with special purpose *ioctl()* calls in order to create and close sockets, and also to perform write on them. Socket creation raises certain complications, because it requires imitating the steps of a connection establishment, without actually putting any packets onto the network. Nevertheless, for most of the functionalities of shadow sockets we directly leverage the underlying TCP implementation of the Linux kernel. This makes several things significantly easier. Because shadow sockets are identical to the non-virtualized TCP connections of the host, the Linux kernel completely exempts us from maintaining and identifying connections (such as hashing and looking up sockets for packets). It also handles the actual TCP transmission, and thus, we don't need to worry about packet retransmissions, congestion control, fairness among flows, and other aspects of the TCP protocol itself. We did need, however, an additional feature so that the rcv_nxt field of a TCP socket can be advanced from user-space in case it is dictated by an acknowledgment of the VM.

4.3 Replication Aware Congestion Control

Replication aware congestion control is a kernel module of the guest OS. The Linux kernel provides a standard interface for developing alternative congestion control algorithms in the form of implementing a *tcp_congestion_ops* structure and registering it via the *tcp_register_congestion_control()* function. The linux kernel adjusts the congestion window either according to the *slow start* algorithm or according to *congestion avoidance*. The slow start threshold (*ssthresh*) is used to determine whether to use slow start or congestion avoidance algorithms. A congestion control algorithm is allowed to specify *ssthresh* and once the threshold is reached, it can adjust the actual congestion window (*snd_cwnd*) in face of an ACK. We use a very low slow start threshold (*ssthresh* is set to 2), followed by a constant large congestion window (*snd_cwnd* is set to 1024).

4.4 Transactional Updates

Another particular issue worth mentioning is the transactional nature of updating the backup machine. When replication data are sent to the backup host, *qemu-kvm* cannot just read and apply the changes directly, because a failure during the update would leave the backup machine in an inconsistent state.

This implies that at the end of each replication epoch the backup machine needs to collect the updates first and then apply all changes together in a transactional fashion, only if all data were received successfully. Unfortunately, the network protocol of *qemu-kvm*'s live migration code doesn't support this by default.

For this reason we extended the *QEMUFile* object with a buffer and a flag that indicates that the file is in accumulating mode. The primary machine toggles this flag on the file corresponding to the backup connection and all subsequent writes are first buffered. We record the number of bytes to be transmitted and inform the backup machine in advance regarding the length of the update. It can then read the whole stream, store it in a buffer and toggle the backup file's flag to indicate that subsequent read operations issued by *qemu-kvm* should access the buffer instead of receiving data from the network.

5. Evaluation

In this section we present the evaluation of speculative communication and replication aware congestion control. We discuss bulk TCP throughput first, which is then followed by demonstration how application level performance is affected. We consider applications that are good candidates for deployment over virtualized infrastructures (i.e., in cloud computing environments) and for which high-availability is naturally expected.

5.1 Experimental Setup

Throughout our experiments the host machine of the replicated VM was equipped with a 4 cores Intel Xeon 2.2GHz CPU, with 2 hyperthreads per core (i.e., 8 hardware threads altogether), 6GBs of RAM and a 250GB SATA harddrive. The machine had two Intel 82546GB Gigabit Ethernet network interfaces. One of the physical network cards were bridged to the virtual machine and used for application traffic and the other was dedicated to the replication protocol for the experiments, when replication took place over Gigabit Ethernet. Moreover, a Mellanox MT26428 Infiniband QDR HCA was also present in both the primary and the backup hosts for the experiments utilizing Infiniband.

The host machines run Ubuntu server 9.10 on Linux kernel 2.6.37 and we used qemu-kvm 0.14.50 with kvm-kmod 2.6.37 as the basis of our implementation. For the virtual machines in each experiment we used the KVM virtio disk and network drivers. We do not present performance results on the native host machine, because in virtualized environments direct access to the underlying machines is normally not available. However, we had Intel's hardware MMU virtualization support, i.e. Extended Page Tables (EPT) enabled in all experiments. Unless stated otherwise, the VM had 1 GB of RAM allocated with memory ballooning support disabled.

5.2 TCP Throughput

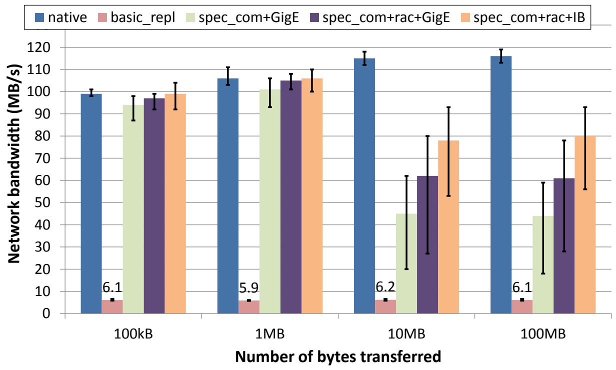

Figure 6. Bulk data transfer performance measured on the client machine. *The replication period is set to 50 milliseconds in this experiment.*

In our first experiment we measured the TCP network throughput from the virtual machine to a separate host. The VM and the client were connected through Gigabit Ethernet (i.e., theoretically the maximum throughput is around 120MB/s).

We ran the test under five different VM configurations, native virtual machine, VM replicated with the basic asynchronous replication protocol (see Section 2.1) over GigE, speculative communication enabled over GigE, speculative communication enabled over GigE with replication awareness in the guest kernel, and finally, speculative communication with replication aware congestion control in the guest, replicated over Infiniband. Replication period was set to 50 milliseconds, where replication was enabled. We used the *ttcp* utility [29] to perform these experiments. *ttcp* allows to define the number of bytes transferred over the link and we used four different scenarios, 100kB, 1MB, 10MB, and 100MB. Each experiment was executed multiple times and we report the average, the highest, and the lowest values (in form of error bars) we obtained. Figure 6 illustrates the results.

As seen, the native execution achieves up to 117MB/s, which is close to the maximum possible throughput of Gigabit Ethernet. Contrary, the throughput attained by the basic replication protocol is extremely low, approximately 6MB/s. Looking at the numbers for speculative communication, the figure reveals that speculative communication achieves near native network performance for short streams (100kB and 1MB), which is followed by a sudden drop in the performance settling around 40-50MB/s for longer transfers. Notice however, that speculative communication achieves up to 65MB/s throughput, which is an order of magnitude better performance compared to the basic replication protocol. Furthermore, replication awareness in the guest kernel further contributes up to 35% performance improvement compared to the only speculative communication case.

The sudden performance drop with longer streams is related to the replication period of the experiment. Because both 100kB and 1MB can be buffered under one replication epoch, the maximum achieved throughput is very close to the native execution, essentially, the transfer is merely double buffered. However, when the transfer spans multiple replication epochs (such as in case of the 10MB and 100MB transfers), speculative buffering is interrupted in every 50 milliseconds. Whereas replication aware congestion control in conjunction with Infiniband yields up to 95MB/s throughput even for longer streams, a similar tendency can be observed over Infiniband as well.

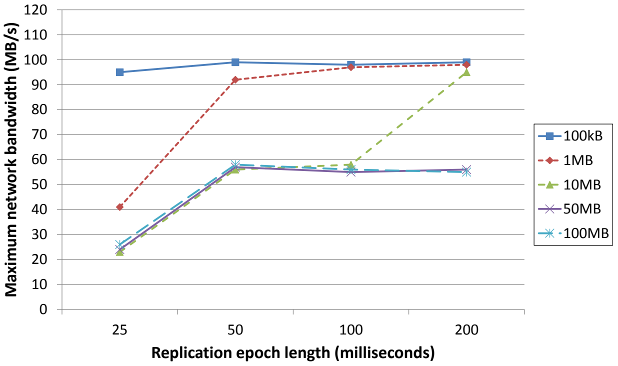

Figure 7. Bulk data transfer performance according to different stream sizes and replication epoch lengths. *Legend represents the number of bytes transferred. The experiments were executed over GigE, without replication aware congestion control in the guest OS.*

Another interesting observation is the relatively high deviation of the attainable bandwidth, especially for longer streams. We believe this is related to scheduling issues, similarly how related work describes it in case of TCP reception performance of virtualized environments [16].

In order to further investigate the bandwidth drop effect we ran additional tests with different replication epoch lengths for different stream sizes. These experiments were all executed over Gigabit Ethernet, without replication aware congestion control. Figure 7 shows the maximum bandwidth we could obtain for each combination. One of the key observations is the fact that streams with longer sizes perform worse, because they span through several replication epochs. As seen, in case of a 1MB stream the smallest replication buffer it fits was the one corresponding to 50 milliseconds epoch length, while the 10MB could only fit the 200 milliseconds one. It is also shown that enlarging the replication epoch length generally increases the attainable bandwidth, because longer epoch size means less frequent checkpoints, which in turn lowers the overhead of the replication itself.

In terms of network performance the basic replication protocol would favor shorter replication epochs, so that buffered packets can be released earlier, however, short replication epochs introduce higher computational overhead. The main strength of speculative communication is its ability to allow long replication epochs and reasonable network performance at the same time.

5.3 SPECweb 2005

The first application we investigated was SPECweb's Banking workload, which emulates an Internet personal banking web-site, where clients are accessing their accounts, making transactions, etc [14]. Requests are transmitted over SSL throughout the whole benchmark. The SPECweb configuration requires at least three machines for running the experiments. One of the server hosts is the actual SPECweb application server, which is accompanied by a backend machine. These were deployed in two VMs residing on two separate physical machines. On the application server we used Apache 2.2.11 with the default configuration the Ubuntu server distribution provides. Besides these, another machine was utilized for running the SPECweb client side scripts.

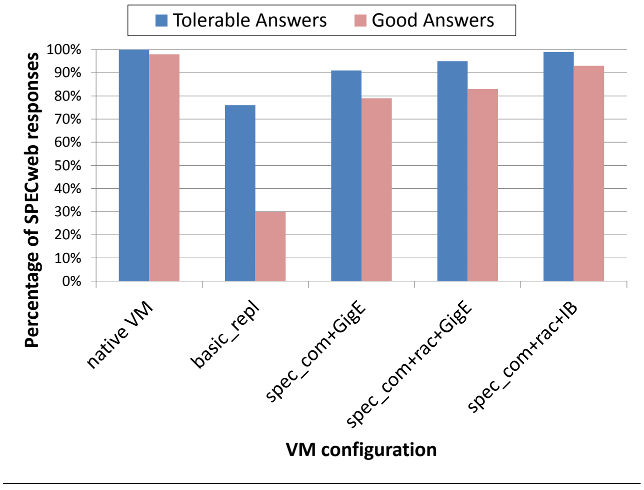

Figure 8. SPECweb2005 Banking results. *The replication period is set to 50 milliseconds and the number of simultaneous connections is 100.*

We replicated only the main SPECweb application server, for which another physical machine was utilized to serve as backup host. SPECweb reports two separate values for each experiment, the ratio of *good* and *tolerable* answers. First we tuned the SPECweb configuration so that 98% of the responses are categorized as *good* when executed on the native VM. In order to assess scalability aspects of the proposed improvements we set the number of simultaneous client connections to 100 in this setup. The same configuration was then used to measure the performance in case of the basic asynchronous replication over Gigabit Ethernet, speculative communication over GigE, speculative communication and replication aware congestion control over GigE, and finally, speculative communication with replication aware congestion control in the guest and replicated over Infiniband.

Figure 8 illustrates the ratio of answers we obtained from these experiments. As seen, the basic replication mechanism imposes substantial performance degradation, achieving only 30% of the *good* answers of the native execution. Webservers are good candidates for speculative communication, because a typical web interaction consists of a small request message which is then followed by a longer, bulk data transfer from the server process. Speculative communication allows the replicated server to possibly transmit the entire response into the VMMs buffer in one go, which can be transferred over the network during the next replication epoch, instead of spanning several epoch as it happens using the basic replication. We expected to see significant improvements due to this mechanism in case of SPECweb.

As the figure shows, speculative communication even over GigE yields 2.6X better performance in terms of *good* answers compared to the basic mechanism, achieving 80% of the native VM's performance. However, we do not observe significantly better performance when replication aware congestion control is enabled in the guest kernel. Further examination on SPECweb's attributes revealed that the average response size is approximately 50kB in this test, with the maximum size transfers not exceeding 150kB. Up to 150kB can be easily buffered even without replication aware congestion control, which explains why the two methods perform very similar in this benchmark.

Table 1. SPECweb Banking average response times (seconds)

Request Type	native VM	basic asynchr.	spec_com+ GigE	spec_com+ rac+IB
login	0.418	3.183	2.022	1.313
account_sum	0.105	2.356	1.389	0.720
check_detail	0.122	2.506	1.610	0.793
bill_pay	0.140	2.315	1.332	0.727
add_payee	0.129	1.820	0.960	0.473
payee_info	0.164	2.611	1.414	0.664
quick_pay	0.132	4.799	2.413	1.214
billpay_status	0.166	2.388	1.308	0.745
chg_profile	0.187	2.498	1.309	0.780
post_profile	0.107	2.412	1.328	0.888
req_checks	0.214	2.921	1.649	0.780
post_chk_order	0.157	2.392	1.156	0.607
req_xfer_form	0.179	2.089	1.238	0.687
post_fund_xfer	0.103	2.309	1.243	0.586
logout	0.139	1.845	0.822	0.522
check_image	0.016	0.827	0.382	0.179

Utilization of Infiniband, nevertheless, contributes another 13% increase in the attainable ratio of *good* answers, getting as close as 93% of the native execution. Infiniband provides significantly higher network bandwidth (up to 6X in our setup), which enables the backup machine to acknowledge a replication epoch's update in much shorter time. Consequently, the buffered reply can be also transferred with much smaller delay.

Table 1 provides further details on the average response time of each individual request types the benchmark consists of. As seen, the basic asynchronous replication renders the response times substantially longer than the native execution. Speculative communication alleviates the overhead with an average ratio of 57% shorter delay compared to the basic replication. We do not provide separate results for the replication aware congestion control case over GigE, because it's very close to the speculative communication only case. Infiniband, on the other hand, yields another 56% shorter delay in responses compared to the GigE case.

5.4 Hadoop

Hadoop [30] is an open-source implementation of the MapReduce (MR) [7] distributed data analysis tool, a framework often used in cloud computing environments. A Hadoop MR cluster consists of several worker nodes (called *TaskTrackers*) that are coordinated by a central entity, the *JobTracker*. While *TaskTrackers* may fail any time without bringing the entire job down, the failure of the *JobTracker* is fatal. Therefore, it is a good candidate for fault tolerant execution.

Our Hadoop cluster is relatively small scale, it consists of 16 worker nodes (i.e., TaskTrackers), each of them is a single vCPU VM with 1.2GB of RAM, residing on four different physical hosts equipped with Intel Xeon 2.2GHz Quad-core CPUs and 6GB of RAM. We used approximately 2GB of raw data on each node the as input. We evaluated the price of protecting the Hadoop master node via VM level replication through the MR/DB benchmark set [26]. MR/DB operates on random generated HTML data and it includes various common tasks that can be expressed either as SQL queries or as MapReduce computations, such as grep, select, aggregate and join.

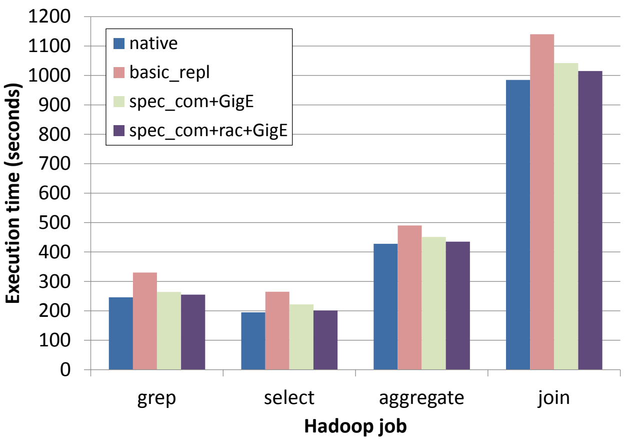

Figure 9. Hadoop MR/DB benchmark runtimes. *Experiments were run on a Hadoop cluster with 16 worker nodes and a separate master node protected via VM level replication.*

We ran four configurations for each benchmark, measuring execution running on native VM, basic asynchronous replication over GigE, speculative communication over GigE, and finally, speculative communication over GigE with replication aware congestion control enabled in the guest's kernel. We executed all tests five times and report the average values.

Figure 9 illustrates the results. As seen, the basic replication mechanism induces up to 34% overhead compared to the native execution, which is significant considering the scale of the system. Since the Hadoop JobTracker merely orchestrates the entire job, the protected VM's dominant action is communication rather than computation. Just as the case with webservers, the JobTracker is also a good candidate for speculative communication, because speculative communication prevents messages to span multiple replication epochs, which would otherwise slow down the entire communication. As the figure shows, speculative communication, even over Gigabit Ethernet, renders the overhead of replication between approximately 5% and 13%. With replication aware congestion control enabled in the guest's TCP stack, the overhead is further alleviated down to 3%, which is almost the same with the native execution. No experiments were run over Infiniband for these benchmarks.

6. Related Work

6.1 Virtual Machine Migration

Checkpoint-recovery based fault tolerance captures snapshots of the running VM at high frequency, often leveraging the live migration support of the underlying Virtual Machine Monitor (VMM). Thus, VM live migration is closely related to checkpoint-recovery based replication. Solutions, such as *Xen* [5], *KVM* [17], and *VMware's VMotion* [23] all provide the capability of live migrating VM instances. Pre-copy is the dominant approach to live VM migration [5, 23]. It initially transfers all memory pages then tracks and transfers dirty pages in subsequent iterations. When the amount of data transferred becomes small or the maximum number of iteration reached, the VM is suspended and finally, the remaining dirty pages and the VCPU context is moved to the destination machine. VM replication, on the other hand, leaves the VM running in precopy mode at all times so that dirty pages are logged and the entire execution state can be reflected to the backup node at the end of each replication epoch [6, 9].

Performance improvement to VM migration has been the focus of several prior studies. Xian et al. showed how data deduplication can be exploited to accelerate live migration [34], while *Microwiper* [10] proposed ordered propagation of dirty pages to transfer them according to their rewriting rates, reducing service downtime during the migration. High performance interconnects have also been used in the context of virtual machine migration, Huang et al. presented RDMA based migration over Infiniband [15],

6.2 Virtual Machine Replication

Bressoud and Schneider [4] introduced first the idea of hypervisor-based fault tolerance by executing the primary and the backup VMs in lockstep mode, i.e., logging all input and non-deterministic events of the primary machine and having them deterministically replayed on the backup node in case of failure. Whereas Bressoud and Schneider demonstrated this technique only for the HP PA-RISC processors VMware's recent work implements the same approach for x86 architecture [25]. These works, however, can handle only uni-processor environments. Deterministic-replay imposes strict restrictions on the underlying architecture and its adaption to multi-core CPU environment is cumbersome, because it requires determining and reproducing the exact order in which CPU cores access the shared memory.

In the context of deterministic (i.e. replayable) SMP execution, solutions on different abstraction levels have been proposed. *Flight Data Recorder* [33] is a hardware extension that enables deterministic replay for SMP environments, but it is unclear what degree of concurrency they can handle without significant performance degradation. Runtime system level solutions, such as *Respec* [18] and *CoreDet* [2] ensure deterministic execution of multi-threaded applications, but their main weakness compared to VM level solutions is the inability to provide fault tolerance for an entire software stack (including the operating system), which is encompassed by a virtual machine. *SMP-ReVirt* [11] exploits hardware page protection to detect and accurately replay sharing between virtual CPUs of a multi-core virtual machine, however, their experiments report superlinear slowdown with the increasing number of virtual CPUs.

Checkpoint-recovery based solutions such as *Remus* [6] and *Paratus* [9] can overcome the problem of multi-core execution by capturing the entire executions state of the VM and transferring it to the backup machine. Although most of the data transfer can be overlapped with speculative execution, transferring updates to the backup machine at very high frequency still comes with great performance overhead. *Kemari* [28] follows a similar approach to *Remus*, but instead of buffering output during speculative execution, it updates the backup machine each time before the VM omits an outside visible event.

Improving the performance of checkpoint-recovery based VM replication has become an active research area recently. Lu et al. [19] proposed fine-grained dirty region identification to reduce the amount of data transferred during each replication epoch, while Zhu et al. [35] improved the performance of log-dirty execution mode by reducing read- and predicting write-page faults. Gerofi et al. [12] utilized Infiniband RDMA for replicating multiprocessor virtual machines. All the above mentioned studies in the domain of checkpoint-recovery based VM replication, however, deal mostly with computational overhead. To the contrary, we focus on the communication aspect of highly available virtual machines.

6.3 Virtualized Network I/O

Alleviating the overhead of network device virtualization has been also an active research domain in recent years. Menon et al. proposed packet aggregation which reduces per-packet handling overhead between the guest and the VMM [21]. TwinDriver suggested tighter cooperation between the guest and the VMM by moving some of the functionalities of the network driver into the host [22]. Dong et al. proposed efficient interrupt coalescing for network I/O virtualization and virtual receive side scaling to effectively leverage multi-core processors [8]. The closest work resembling ours is vSnoop [16]. vSnoop offloads TCP acknowledgment handling to the VMM so that TCP reception performance doesn't suffer due to scheduling delays of virtual machines. The main difference between vSnoop and our work, is that vSnoop deals with TCP reception as opposed to TCP transmission, which eliminates the need of shadow sockets. Moreover, while our study also exploits the idea of offloading functionalities from the guest VM to the virtual machine monitor, we are focusing on improving network performance of virtualized environments with the additional feature of high availability.

7. Discussion and Future Directions

7.1 Scheduling and Epoch Length

We have already pointed out in Section 5.2 that one of the main strengths of speculative communication is the fact that high network throughput can be attained even with longer replication epochs. Generally, the longer the replication epochs are the smaller the overhead of the replication itself is. Therefore, as a general policy, one could enforce as long replication epochs as possible in order to minimize the overhead. However, under the basic replication protocol network I/O benefits from shorter epochs so that pending packets can be released earlier. As speculative communication opens up the way to prolong replication epochs but progress with network transmission at the same time, determining the optimal epoch length could be based on other aspects of the workload, such as the amount of dirtied memory pages, number of written disk I/O sectors, etc. In the future we intend to explore the idea of how workload adaptive checkpoint scheduling and speculative communication could interplay in an efficient manner, finding the optimal epoch lengths of the replication dynamically based on the workload executed in the VM.

7.2 Extended Replication Awareness

We have shown that making the congestion control algorithm of the guest's TCP stack replication aware yields significant improvements. Whereas this particular modification focuses solely on network performance, the guest's kernel could be further extended so that other aspects of the replication were also addressed.

General replication awareness in the guest kernel would open up various directions for further optimization. For example, if the guest kernel was aware of the checkpoint frequency (i.e., the exact time when the next checkpoint will be taken), it could schedule disk operations so that they are scattered among multiple checkpoints in a balanced way or even delay them according to other factors of the give epoch's update, such as dirtied memory pages.

A more pervasive approach of replication awareness in the guest's TCP stack would allow to transfer (i.e., buffer in the VMM) exactly the number of packets the replication buffer can handle, without having to announce zero window to indicate when the buffer is full. Furthermore, if the guest's TCP stack knew when the next checkpoint is going to be taken, it could simply transmit a large number of packets without expecting an acknowledgment until the next epoch, which would exempt us from having to transfer those packets to the backup machine, because the TCP stack would be aware of the fact that they are still not acknowledged.

8. Conclusions

Checkpoint-recovery based virtual machine replication is attractive, because it provides high availability for the entire software stack executed in the VM. Due to the output commit problem, however, network performance of the basic replication method is extremely poor.

In this paper we have revisited the replication protocol and extended it with *speculative communication*. Speculative communication enables the VM to progress with TCP transmission without exposing the messages to the client machines before the corresponding execution state is acknowledged by the backup host. We have introduced the notion of *shadow sockets*, which track TCP flows, perform speculative communication for the guest and effectively offload TCP transmission into the virtual machine monitor. Moreover, we have proposed *replication aware congestion control*, an extension to the guest's TCP stack that cooperates better with speculative communication. While we have focused on improving TCP transmission performance of replicated virtual machines, speculative communication is also applicable for TCP reception.

We have shown that speculative communication yields up to an order of magnitude better performance of bulk TCP transmission, and close to native network performance when replication aware congestion control is enabled in the guest's TCP stack and the replication is performed over Infiniband. We also presented measurements regarding the impact of speculative communication on applications, which naturally fit virtualized environments, and at the same time, require high-availability. SPECweb attains close to 80% of the native performance with speculative communication, and over 90% when replicated over Infiniband. We observe close to native performance with speculative communication of the Hadoop master node even when replication takes places over Gigabit Ethernet.

Acknowledgments

This work has been supported by the Core Research for Evolutional Science and Technology (CREST) project of the Japan Science and Technology Agency (JST).

References

[1] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 53–64. ACM, 2010. ISBN 978-1-60558-839-1.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on selected Areas in communications*, 13:1465–1480, 1995.

[4] T. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[6] 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*, NSDI'08, pages 161–174, 2008. ISBN 111-999-5555-22-1.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[8] Y. Dong, Y. Zhang, and G. Liao. Optimizing Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive Side Scaling. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, 2011.

[9] Y. Du and H. Yu. Paratus: Instantaneous Failover via Virtual Machine Replication. In *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, GCC '09, pages 307–312. IEEE Computer Society, 2009.

[10] Y. Du, H. Yu, G. Shi, J. Chen, and W. Zheng. Microwiper: Efficient Memory Propagation in Live Migration of Virtual Machines. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 141–149, Washington, DC, USA, 2010. ISBN 978-0-7695-4156-3.

[11] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, 2008. ISBN 978-1-59593-796-4.

[12] 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, pages 35–44, 2011.

[13] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42:64–74, July 2008. ISSN 0163-5980.

[14] R. Hariharan and N. Sun. Workload Characterization of SPECweb2005. `http://www.spec.org/workshops/2006/papers/02_Workload_char_SPECweb2005_Final.pdf`, 2006.

[15] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 11–20, Washington, DC, USA, 2007. ISBN 978-1-4244-1387-4.

[16] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007. URL `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`.

[18] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. ASPLOS '10, pages 77–90. ACM, 2010. ISBN 978-1-60558-839-1.

[19] M. Lu and T. cker 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, 2009.

[20] R. McDougall and J. Anderson. Virtualization performance: perspectives and challenges ahead. *SIGOPS Oper. Syst. Rev.*, 44:40–56, December 2010. ISSN 0163-5980.

[21] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2006. USENIX Association.

[22] A. Menon, S. Schubert, and W. Zwaenepoel. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 301–312, New York, NY, USA, 2009. ACM.

[23] M. Nelson, B. H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 25, Berkeley, CA, USA, 2005. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1247360.1247385`.

[24] G. Pokam, C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas. Hardware and Software Approaches for Deterministic Multi-Processor Replay of Concurrent Programs. In *Intel Technology Journal*, volume 13, issue 4, pages 20–41, 2009.

[25] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44:30–39, December 2010.

[26] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53:64–71, January 2010.

[27] Strom, R.E. and Bacon, D.F. and Yemini, S.A. Volatile logging in n-fault-tolerant distributed systems. In *Fault-Tolerant Computing, Eighteenth International Symposium on*, pages 44 –49, Jun 1988.

[28] Y. Tamura. Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT. Technical report, NTT Cyber Space Labs, 2008.

[29] Test TCP (TTCP): Benchmarking Tool and Simple Network Traffic Generator. http://www.pcausa.com/Utilities/pcattcp.htm, 2010.

[30] A. TM. Hadoop. `http://hadoop.apache.org`.

[31] Transmission Control Protocol. Protocol Specification. `http://www.ietf.org/rfc/rfc793.txt`, 1981.

[32] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[33] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135. ACM, 2003. ISBN 0-7695-1945-8.

[34] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 88 –96, 2010.

[35] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li. Improving the Performance of Hypervisor-based Fault Tolerance. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –10, 2010.