COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service

YaoZu Dong[1,2], Wei Ye[1,2], YunHong Jiang[1], Ian Pratt [4], ShiQing Ma[1,2], Jian Li[3], HaiBing Guan[1*]

[1] Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China.
[2] Intel Asia-Pacific R&D Ltd., China
[3] School of Software, Shanghai Jiao Tong University, China.
[4] Bromium Inc., USA
eddie.dong@intel.com, li-jian@sjtu.edu.cn, hbguan@sjtu.edu.cn

Abstract

Virtual machine (VM) replication provides a software solution of for business continuity and disaster recovery through application-agnostic hardware fault tolerance by replicating the state of primary VM (PVM) to secondary VM (SVM) on a different physical node. Unfortunately, current VM replication approaches suffer from excessive overhead, which severely limit their applicability and suitability. In this paper, we leverage the practical effect of networked server-client system that PVM and SVM are considered as in the same state only if they can generate the same response from the clients' point of view, and this is exploited to optimize performance. To this end, we propose a generic and highly efficient non-stop service solution, named as "COLO" (COarse-grained LOck-stepping virtual machine) utilizing on-demand VM replication. COLO monitors the output responses of the PVM and SVM, and rules the SVM as a valid replica of the PVM according to the output similarity between PVM and SVM. If the responses do not match, the commit of network response is withheld until PVM's state has been synchronized to SVM. Hence, we ensure that the system is always capable of failover by SVM. Although non-determinism may mean a different internal state of SVM from that of the PVM, it is equally valid and remains consistent from external observations. Unlike earlier instruction level lock-stepping deterministic execution approaches, COLO can easily support Multi-Processors (MP) involving workloads with the satisfying performance. Results show that COLO significantly outperforms existing approaches, particularly on server-client workloads such as online databases and web server applications.

*Corresponding author.

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2523630

1 Introduction

Surviving hardware failure is critical to achieve non-stop service for networked client-server systems in data-center and cloud computing environments. Software-Based Replication (SBR) model provides an OS- and application-agnostic high availability as well as a high flexible redundancy solution in virtualized environment. SBR model enables non-stop service through virtual machine (VM) replication, which allows cloud service providers to deliver dependable infrastructure as a service. Generally speaking, SBR model replicates the primary virtual machine (PVM) in a specific frequency to a secondary VM (SVM), and use the SVM to take over the service once a fault state of the PVM is detected. It has shown significant advantages than the hardware-implemented fault tolerance solutions, such as HP Non-Stop server [4], which require redundant components and specific system design to maintain and switch from the faulty component to the backup [15][24][32]. Besides its expensive price and low industrial popularity, hardware-implemented fault tolerance solutions can only provide the physical server level replication rather than indicated VM replication, which reduce the flexibility, elasticity and scalability features in cloud computing and data-center environment.

Conventionally, one kind of SBR approach replicates VM state at instruction level using a technique known as lock-stepping [13] [18] [30] [31] [21], where PVM and SVM execute in parallel for deterministic instructions. In contrast, lock-stepping is applied for non-deterministic instructions, achieving the same state at the boundary of those instructions. The instruction level lockstepping method is only suitable for the single processor VM, and their performance will degrade dramatically to one-seventh for the guest with more than one processor due to the high non-deterministic execution results of each memory access instruction [19]. Thus, the solution of lock-stepping at the boundary of each instruction is inefficient.

Another kind of SBR approach replicates the VM state at the boundary of each epoch with periodic checkpoints, that is, the PVM state is replicated to the SVM periodically, such as Remus [17]. Remus buffers the output packets until a completion of successful checkpoint in order to enable the failover from the replica in case of the hardware failure on PVM hosting physical node. However, periodic checkpointing suffers from the extra network latency due to output packet buffering, and the heavy overhead due to the high checkpoint frequency.

Note that the perfect matching between PVM state and SVM state is an overly strong condition. On the other hand, the non-stop service can be achieved only if the SVM can successfully take over the service while respecting the application semantics at the time of hardware failure (no matter the machine state is identical or not). In this case, it is a key point to identify the boundary for the divergence between machine states in order to determine when the SVM becomes an invalid replica. From the client's point of view, the SVM is qualified as a valid replica of the PVM as long as the PVM and SVM can generate the identical responses. If so, the SVM can successfully take over when the PVM is in a hardware failure, so that it provides the unnoticeable high availability service (transparency), according to the service semantics.

Though execution of non-deterministic instructions may cause immediate differences in machine states, the PVM and SVM will likely generate identical outputs in a short interval. For instance, the TCP timestamp typically uses the system ticks, and its value is non-deterministic. However, the timestamp becomes different only after the accumulation of a sufficiently large clock drift between the PVM and SVM. Therefore, we use *output similarity* or *response similarity* [22] to analyse and quantify the divergence between PVM and SVM.

In this paper, we propose COLO, a non-stop service solution with coarse-grained lock-stepping VMs for client-server systems. The PVM and SVM execute in parallel, and the inbound packets from clients are delivered to both. COLO receives the outbound packets from both the PVM and SVM and compares them before allowing the output to be sent to clients.

The SVM is qualified as a valid replica of the PVM, as long as it generates identical responses to all client requests. Once the differences in the outputs are detected between the PVM and SVM, COLO withholds transmission of the outbound packets until it has successfully synchronized the PVM state to the SVM. COLO transfers the incremental VM states during checkpointing, for efficiency. As execution continues the PVM and SVM will again start to diverge due to non-determinism, and COLO will continue to compare the output until it determines resynchronization is required. Thus, COLO ensures the SVM is always a valid replica of the PVM from the point of view of all clients, and can take over if the PVM fails.

The contributions of this paper are as follows:

- Taking advantage of output similarity between the PVM and the SVM, we propose a novel solution for non-stop service for server-client systems, based on coarse-grained lock-stepping VMs.
- We discuss the design and implementation of coarse-grained lock-stepping on Xen and compare it with the periodic checkpointing solution, Remus.
- We explore how performance can be improved through modifications of TCP stack for improving determinism and hence output similarity, which can effectively reduce the frequency of VM replication.
- We conduct a comprehensive set of experiments to measure the performance of COLO for different usage scenarios.

The evaluation results show that COLO has potential to scale well with the increasing number of virtual CPUs. COLO can achieve the native comparable performance in SysBench testing [33] with CPU and memory workloads and achieve 80% of native performance running Kernel Build workload. In the same testing scenario, COLO outperforms Remus by 29%, 92% and 203% respectively in task completion time of Kernel Build. In FTP server [34] GET and PUT benchmarks, COLO achieves 97% and 50% of native performance, outperforming Remus by 64% and 148%, respectively. In Web Server tests with WebBench [34], COLO achieves up to native performance, outperforming Remus by 69% when running WebBench with up to 256 concurrent threads. In the pgbench PostgreSQL data base benchmark COLO achieves 82.4% of native performance on average and 85.5% of native peak performance, outperforming Remus by 46% and 34%, respectively.

Note that COLO is built on top of Xen [10] and its incremental VM check-pointing solution, Remus [17], but the solution itself is generic enough to be implemented in other hypervisors. In COLO system, the replicas are executed in the context of a PVM with one or more backup replicas dynamically configured in an n-modular redundant fashion. The necessary patches of COLO have been posted on the Xen mailing list, and the original idea was presented in Xen summit 2012 [3].

The rest of the paper is organized as follows: Section 2 gives a brief introduction of Xen and Remus passive-checkpointing approach. In section 3 and 4, we describe the coarse-grained lock-stepping VM (COLO) approach as well as the detailed implementation. Section 5 presents the evaluation results, and Section 6 shows related work. We conclude the paper and describe future work in Section 7.

2 Background: Xen and Remus

Xen implements a split device driver model, with a frontend (FE) driver running in the guest communicating with a backend (BE) driver running as a service in domain 0, as shown in Figure 1. In the split device driver model, the FE driver communicates with the BE driver through shared memory for bulk data transfers, and uses an event channel (indexed by a per-guest port), for notifications. Direct page table mode is used in the Xen paravirtualized (PV) guest, which cooperatively works with hypervisor to manage both the hardware and guest view of page tables [10], though this is not an architectural requirement of the COLO approach.

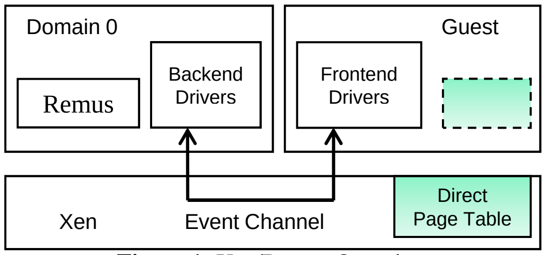

Figure 1: Xen/Remus Overview

Remus implements an active PVM/passive SVM model, that is, the PVM executes actively, while the SVM is suspended until a PVM failure passes control to the SVM. Remus periodically takes checkpoints of the VM state, including CPU, memory, and device states, replicating them from the PVM in the primary node to the SVM in the secondary node. Remus transfers periodic checkpoints based on Xen live migration techniques [16].

Remus uses heartbeat monitoring, which is a software component running on both the primary and secondary nodes, to periodically check platform availability. When the primary node suffers a hardware fail-stop failure, the heartbeat stops responding, the secondary node will trigger a failover as soon as it determines the absence of heartbeat.

Remus only executes the SVM after failover, when a hardware fail-stop failure is detected in the primary node. Because the SVM in Remus is not actively executing during each epoch, we refer it as a passive checkpointing, and say COLO performs active checkpointing because it actively executes the SVM after each checkpoint. This make COLO as a revolution architecture design for software based replication, which will be described in the next section.

3 COLO Design and Principles

In this section, we introduce the design principle of COLO, a VM-based coarse-grained lockstepping solution, implementing based on Remus and Xen PV guests. We explain how to use the output similarity to determine the checkpointing time for the networked server-client system and discuss the efficiency in replication overhead reduction. Then, we highlight the implementation challenges and solutions how to extend and optimize the Xen passive checkpointing solution for active-checkpoints by improving the output similarity between PVM and SVM.

3.1 Output Similarity Model

As introduced before, COLO initiate a server-client system in the PVM and SVM at exactly the same state and then stimulate them with the same incoming events. Then, the identical results should be produced for a specific interval, which depends on the deterministic execution performance of PVM and SVM. We now introduce how to detect the output divergence between PVM and SVM with the response model.

Response Model: A client-server system can be considered as a request/response system, where a client sends its request to the server and the server responds to the client. The request and response packets (denoted as r and R, respectively) form a stream of packets, as shown in equation (1) & (2). r_i and R_i denote the i_{th} request and response packet, respectively.

$$r = \{r_0, r_1, r_2, ..., r_n, ...\} \tag{1}$$

$$R = \{R_0, R_1, R_2, ..., R_n, ...\} \tag{2}$$

The response packet for the current request can typically be determined by the request stream consisting of prior requests. That is, the n_{th} response packet is a function of the request stream. In previous literature [14][20], server hot swap solutions were actually built based on this assumption for certain applications and usage models (for example, a uniprocessor system running Apache server).

Output Similarity: In many cases, the response to the current request is determined by both the prior request stream and the execution of non-deterministic instructions (such as I/O, interrupts, and timestamp counter accesses). Consequently, the response packet R_n can be considered as a function of both the prior request stream and the execution results of non-deterministic instructions, as shown in equation (3) (where the execution result of non-deterministic instructions is denoted as a variant U). COLO duplicates the request stream to both PVM and SVM, however, the variant U differs by nature between the primary and secondary servers. R_k^p and R_k^s denote the k_{th} response packet from the PVM server and SVM server, respectively. More importantly, the result of memory accesses in a multiprocessor system is typically non-deterministic, which means that non-deterministic instructions are pervasive in modern server systems.

$$R_n = g_n(r_0, r_1, r_2, ..., r_n, U) \tag{3}$$

On the other hand, from the clients' points of view, every response stream delivered by equation (3) (no matter what U is), is a valid response according to the server application semantics or the service semantics.

Output Similarity Bound for Coarse-Grained Lock-Stepping: Even though the execution of non-deterministic instructions may cause immediate difference in VM machine states, the primary and backup servers may still generate identical outputs in a short term, i.e. the PVM and SVM have output (response) similarity, and we measure the output similarity as the number of identical output packets that PVM and SVM generate.

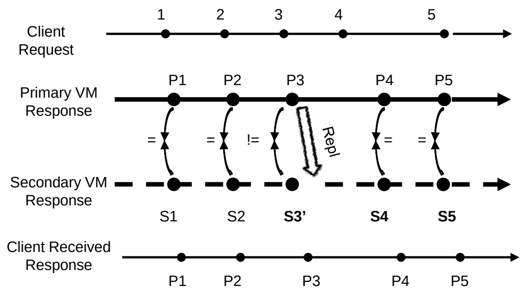

Figure 2: VM Based Coarse-grained Lockstepping

COLO implements an efficient and generic virtual server replication solution by taking advantage of output similarity. COLO runs both primary and secondary VMs in parallel, as long as both VMs generate identical responses to client requests. If the output diverges due to the accumulated results of non-deterministic instructions, as shown in Figure 2, then the SVM is no longer a valid replica of the PVM. At that point, COLO will initiate a coarse-grained lock-step operation: It replicates the PVM state to the SVM.

Failover: If the PVM fails, the replica (SVM) can provide failover and send its response stream to the client. From the client's point of view, the response stream consists of packets from PVM p (1_{st} to k_{th} packets of R^p) and the packets from SVM s (starting from the $(k+1)_{th}$ packet of R^s), as shown in Equation (4), where a failover is implemented by switching output packets from the PVM to the SVM at the $(k+1)_{th}$ packet.

$$C = \{R_1^p, ..., R_k^p, R_{k+1}^s, ...\} \tag{4}$$

3.2 COLO Overall Architecture

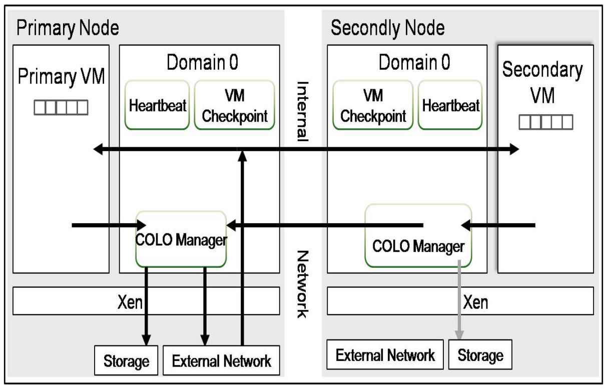

Figure 3: COLO Overall Architecture

The architecture of COLO is shown in Figure 3. It consists of a pair of networked physical nodes: the primary node running the PVM, and the secondary node running the SVM to maintain a valid replica of the PVM. PVM and SVM execute in parallel and generate output of response packets for client requests according to the application semantics. The incoming packets from the client or external network are received by the primary node, and then forwarded to the secondary node, so that both the PVM and the SVM are stimulated with the same requests (in terms of packet data and sequence).

The COLO manager are planned in the hypervisors both in the primary node and secondary node. The COLO manager on the secondary node consumes the output packets from the SVM, and forwards them to the COLO manager in the primary node. The COLO manager in the primary node receives the output of PVM and that of SVM, then it checks if SVM is a valid replica of the PVM according to the output similarity model, as shown in Figure 3.

If the SVM is a valid replica as of packet k, the primary node can then immediately release packet k, as shown in Figure 3 as well. If the response packets are divergent (the SVM is no longer a valid replica), the COLO manager forces a new checkpoint to forward the PVM state (including the not-yet-released output packets) to the SVM to regenerate an exact VM replica, and resumes execution.

3.3 Challenges of Output Similarity

Note that COLO can relax the checkpointing frequency based on output similarity model. Since the inbound request packets are duplicated to both PVM and SVM, the stimulation of both are exactly the same. Therefore, the divergence between PVM and SVM may be incurred only from the VM execution and indetermination of output connection level network protocol stack.

3.3.1 VM State Divergence

The divergences of VM states between PVM and SVM in COLO are workload dependent. In other words, different workload with CPU, I/O, disk, memory access requirements will result in different VM divergence state and output similarity, and consequently result in a different checkpointing frequency. 1) Every replication of PVM to SVM makes them in the same state. Then, COLO can deal with deterministic CPU intensive computing applications perfectly since the same input will produce the same results. In this case, COLO can achieve almost the same performance as native system without duplication. 2) If COLO hosts the I/O intensive workload such as key-value database server or other device I/O operations, the COLO deals with device state lockstepping with difference methods, which will be introduced in COLO implementation in Section 4.4. Briefly, take the example of key-value database server, PVM and SVM normally produce the same response to the ‘GET’ requests. However, the ‘PUT” request may introduce different data values in PVM and SVM, and COLO considers the state of storage device as an internal state of the guest to replicate. Therefore, COLO snapshots the local storage state as part of the VM checkpoint, which will degrade the I/O performance in comparison to the native performance. However, our experiments in Section 5 illustrate that COLO still outperforms the original checkpointing scheme (Remus) significantly in PUT operation.

3.3.2 Connection Level Output Similarity

A server of a networked system may have multiple TCP connections and may respond to multiple client requests concurrently, which imposes additional challenges for maximizing output similarity. First, even if the response packets from each TCP connection are identical, the sequence of response packets across different TCP connections may be different between PVM and SVM, resulting from the non deterministic instruction execution. However, each TCP connection runs independently, and can recover from failover for each the TCP protocol, according to the model described in subsection 3.1. Furthermore, the TCP/IP stack is designed to be able to deal with the reordering of the packets within and across TCP connections. For example, a network router may take different routes for different packets per internal policy in a conventional network system, and therefore it is possible that the network packets from different TCP connections may arrive at the destination out of order.

COLO implements the per TCP connection response packet comparison, and considers the SVM as a valid replica, if the response packets of each TCP connection from the PVM and SVM are identical, regardless of the packets ordering across TCP connections. This can be considered in terms of packets on different connections overtaking the next expected packet on a particular connection. A timeout mechanism is implemented to force a VM checkpoint if a TCP connection observes packet(s) from one VM are not matched by corresponding packet(s) from the other VM within a certain time period, e.g., 200ms, in order to guarantee the correctness and forward progress.

Therefore, TCP/IP protocol should be modified to enhance the output similarity, which can make PVM and SVM produce the identical outputs when treating with the same incoming events with the same state. The implementation of COLO should also provide the active checkpointing method, failover mode in case of the detection of PVM failure state, as well as the device lockstepping. The detailed implementation will be introduced individually in the next section.

4 COLO Implementation

4.1 TCP Modification for Response Similarity

Although most machine state differences between the PVM and SVM will not immediately generate response differences, some may lead to a high frequency of active checkpointings which can severely impact COLO performance. Improving output similarity, or maximizing the duration of SVM as a valid replica of the PVM, is critical to COLO performance. The longer output similarity is preserved, the less often we have to execute checkpointing.

The TCP connection is the dominant paradigm of reliable communication in modern client-server systems. Minimizing the divergence of TCP response packets from machine state differences due to non-deterministic instruction execution is critical to the output similarity of the SVM. In practice, the TCP stack employs non-deterministic instructions in the generation of response packet headers, which may result in different packets even the packaging response data are identical. Examples of such packet-level differences are: timestamps, TCP window size changes, the timing of ACK transmissions, and the coalescing of small messages into a single packet.

COLO modifies the guest OS’s TCP/IP stack in order to make the behavior more deterministic. A series of techniques are employed to improve response similarity, including coarse grained timestamps, a more deterministic ACK mechanism, a more deterministic small message delivery mechanism, and quantized window size notifications. The changes made to the guest TCP/IP stack are minimal, just tens of lines modified (57 insertions and 18 deletions). Our work is based on Linux TCP protocol stack but it can easily be applied to other OSes. It may be possible to achieve even better performance without changes to the guest TCP/IP stack by having a similarity comparison function that operates transparently over re-assembled TCP streams, but this is the subject of future work and not discussed further here.

4.1.1 TCP Timestamp Coherence:

TCP connections support an optional timestamp, which can easily lead to differing output packet headers between the PVM and SVM. Applications can create a TCP connection with or without timestamps, depending on the usage model. When using timestamps, packets coming from the different VMs may attach different timestamps even though packet data is the same. This is because the timestamps are derived from the time stamp counter or other timer sources, which may be different in the two VMs as a result of non-deterministic instructions.

COLO makes use of coarse-grained timestamps to reduce the output packet divergence introduced by the timestamp. The typical timestamp used in TCP headers comes from system ticks in Linux, which is based on an OS tick in units of 1ms, 4ms, or 10ms, depending on the tick frequency the guest OS uses. The TCP stack may observe an accumulated tick difference after a certain number of ticks (tens, hundreds, or more) in the guest OS, depending on the time virtualization policy and synchronization between the hosts. On the other hand, the TCP stack does not require high timestamp accuracy. Rather, it is mostly used to identify a timeout, or log the events with timestamps with accuracy requirement of only hundreds of milliseconds. In COLO, we modify the guest OS TCP software stack to use coarse-grained timestamps with a granularity of *128ms* to increase the output similarity.

4.1.2 TCP Acknowledgement Coherence:

Generation of TCP acknowledge (ACK) packets may cause divergence in response packets. TCP connections use ACK packets to acknowledge receipt of received data packets, but for efficiency, the packet may be deferred for some time in case the ACK packet may piggyback onto an output data packet. However, the policy is based on fine-grained internal state, such as the fine-grained size of the internal reassembly buffer, which may lead to a high possibility of output divergence.

COLO optimizes the TCP protocol to use a highly deterministic ACK mechanism with coarse-grained internal state to improve output similarity through controlling the release of ACK packets.

4.1.3 Packet Size Coherence with Nagle Algorithm:

Delivery of small messages in TCP stack may pose additional challenges to the output similarity. Although server applications will typically generate identical response streams to the identical client requests, the timing with which the application writes chunks of additional data to the socket may be different, and the way in which the data may be split into packets may be different. For example, the timing of packets received from the client that adjusts the advertised client window may result in the stream being segmented in different ways. Further, timers involved in the implementation of Nagle's algorithm may affect how small messages are grouped.

COLO attempts to make this segmentation of the stream into packets more deterministic by adjusting Nagle parameters to encourage more aggressive coalescing of short messages into full packets.

4.1.4 TCP Notification Window Size Coherence:

The notification window size is used by the receiver to tell the sender the amount of data it is able to accept for efficient data transmission, as a form of flow control. If the client sends data too quickly, or the server application consumes data too slowly, the server will notify the client with a smaller window size, throttling the client. On the other hand, if the client sends data too slowly or the server application consumes data too quickly, the server will notify the client with a bigger window size, suggesting a higher sending speed for the client.

Quantization of notification window size is proposed in COLO to reduce packet divergence. If the notification window size is large enough (larger than 255), COLO masks the 8 least significant bits, otherwise it rounds down to the nearest power of 2. Therefore, the notification window size in the packet is more likely to be the same in the PVM and SVM, then the output similarity is improved.

4.2 Active-Checkpointing

COLO requires both the PVM and SVM to be executed in parallel at runtime, which has additional challenges under Xen. First, both the PVM and SVM in COLO may generate dirty memory pages. Log-dirty mode [16] is used to track the dirty pages of the PVM (D^p), and the dirty pages of the SVM (D^s), for efficient tracking of dirtied pages. A VM checkpoint requires the SVM to update the delta memory set, ($D^p \bigcup D^s$), but transmitting the whole union delta set is suboptimal. Second, passive-checkpointing resumes the device from the predefined initial state, when a failover occurs, but active-checkpointing may generate dirty states on the fly in the SVM.

COLO solves the memory checkpointing issue by keeping a local copy of the previous checkpoint's memory contents, and reverting locally modified memory pages to the previous checkpoint before applying the delta memory pages from the PVM. Therefore, only D^p is transmitted, saving CPU and network resources. For device state, COLO uses the device suspend/resume process that was introduced by live migration [16] to gracefully bring both the PVM and SVM to the initial state, and rebuilds the machine state using active-checkpointing.

4.3 Failover Mode

The heartbeat modules in both PVM and SVM are used to detect the failure of the physical nodes, as shown in Figure 3. With the advance of modern hardware technologies such as reliability, availability and serviceability (RAS) features [5], many hardware failures are self-corrected. Most hardware unrecoverable failures (such as power, memory, cache and PCIe traffic failure) will typically be fail-stop failures as opposed to undetected corruptions, i.e. Byzantine failures. And therefore, COLO is applicable to survive from most hardware failures.

COLO can tolerate hardware fail-stop failure. The SVM can successfully take control if the PVM fails after releasing the $(k-1)_{th}$ packet, but before the k_{th} packet, and the COLO manager of the secondary node does not consume the k_{th} packet yet, shown as area "A" in Figure 4, by releasing packets from the SVM at the k_{th} packet without noticeable difference from the client's perspective, even if the internal state is different.

If the fail-stop failure happens after the COLO manager of the secondary node consumes the k_{th} packet, but before the PVM starts to release the k_{th} packet (shown as area "B" in Figure 4), or if the fail-stop failure happens when the PVM is releasing the k_{th} packet (shown as area "C" in Figure 4), the SVM takes control and returns responses starting from the $(k+1)_{th}$ packet. The client may completely lose packet k. We rely on the network stack and application to recover from this type of error, which may happen in real network systems, as well. Consequently, COLO achieves non-stop service.

In case the hardware failure happens during a checkpoint (shown as area "D" in Figure 4), the SVM may resume from its local VM snapshot if the checkpoint is not completed yet, which takes control and returns responses starting from the $(k+1)_{th}$ packet as it does to area "B" as mentioned above, or from the newly successful VM snapshot, which would include packet k and releases packet k, as it does to area "A" as mentioned above, respectively.

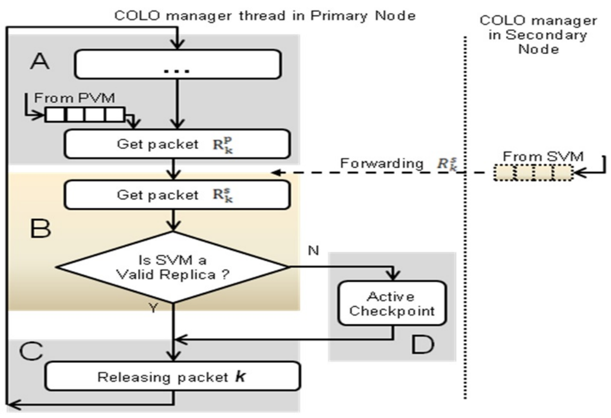

Figure 4: Execution and Checkpoint Flow in COLO

4.4 Device State Lock-Stepping

A VM may use a local storage or a remote storage device, and the PVM and SVM may share the remote storage with non-stop storagee service.

Local Storage: The local storage device can be viewed as an external interaction, like a network client, or an internal state of the guest. In the former, COLO has to forward write operations to local storage from the SVM to the PVM for comparison, treating the SVM as a valid replica of PVM, if and only if both the network packets and the write operations of local storage on the SVM are identical to those on the PVM. In the latter, COLO considers the state of local storage as part of the VM state, and snapshots the local disk state when taking a checkpoint. Therefore, COLO does not need to forward the write operations from SVM local storage to the PVM at runtime. Instead, it treats divergence of local storage devices like that of the CPU and memory, where differences may not immediately generate a difference in the response, allowing the SVM to still be considered a valid replica. However, it requires that the VM checkpointing process transmit the storage device state as well, and be able to roll back to a previous checkpoint, or update to a new checkpoint.

Forwarding and comparing storage write operations between the PVM and SVM saves the effort of supporting VM storage checkpointing. However it may increase the frequency of VM checkpointing if the write operations from the PVM and SVM are different. One may improve the storage packet similarity by implementing a higher level storage virtualization solution. For example a file system level PV solution is likely more deterministic than a block level PV solution. Snapshotting storage device state may add additional latency to the checkpoints. However, it may be more efficient if the guest storage write operations are less deterministic.

COLO considers the state of local storage device as an internal state of the guest, and snapshots the local storage state as part of the VM checkpoint, and plans to explore the other solution in future. For efficiency, COLO transmits just the deltas between VM checkpoints. To do this, COLO logs the guest's disk write operations, and transmits them from PVM to SVM when taking a checkpoint. Once the new checkpoint is received, both the PVM and SVM commit the logged write operations to maintain identical storage state between the PVM and SVM. To bound the required log size (and therefore the duration of log transmission time), COLO forces a VM checkpoint once the pre-allocated log buffer is filled.

Remote Storage: The PVM and SVM may use remote storage devices for exclusive and/or shared access, depending on the configuration. In the dedicated access use case, the SVM and PVM have their own storage partition, and can access its own remote partition independently. COLO views the dedicated accessed remote storage to be same as the local storage, and therefore applies same policy as that of local storage, mentioned in above subsection, for non-stop service.

In shared remote storage access case, COLO relies on the remote storage system to provide the non-stop storage service such as the Google file system [20], and views the interaction with remote storage same as that of client. COLO views the outbound disk I/O access requests and inbound disk response data, same as that of the outbound and inbound network packets, as part of the request and response stream shown in equations (1)-(4) in section 2, and applies the same policy with that of network packets for non-stop service. COLO forwards the write operations of remote storage from SVM to PVM, and compare them in addition to the network response packets to determine if a divergence happens. The SVM is viewed as a valid replica if and only if both network response packets and storage access operations are identical as of packets k_{th}. COLO enforces a VM checkpoint if a divergence is identified, or releases the access operation to the remote storage (and drop the write operation from SVM) in addition to the network packets, if the SVM remains to be a valid replica. In the meantime, COLO forwards the inbound remote storage data packets to both the PVM and SVM, so that both the PVM and SVM observe the same (both in contents and sequence) storage data.

Other Devices The other virtual devices, such as the display and serial device, are also considered guest internal state, and COLO treats them similarly with that of local storage for device state lock-stepping. These devices do not typically have large amounts of internal state (unlike storage devices), nor do they impact the client view of server responses (unlike network devices). Therefore, COLO can rely on the VM checkpointing process to synchronize the device state.

4.5 Discussion

COLO enables a new solution for application-agnostic, non-stop service, surviving from hardware fail-stop failures. The above mentioned efforts for enhancing the output similarity can effectively reduce the checkpointing frequency, and the evaluations prove this in the next section. Different applications may have different characteristics of output similarity, COLO shows great potential to achive a highly efficient and scalable non-stop service, at least in the workloads we tested, and we believe COLO can be equally efficiently applied to other workloads as well with reasonable optimization.

Note that additional performance optimization techniques may be applied to further improve the response similarity while reducing the cost of active-checkpoints. First, one may use hardware assisted paging technologies, such as extended page table (EPT), to avoid the expensive process of re-constructing the direct page tables of PV guests every VM checkpoint. Second, page compression and speculative asynchronous transmission of dirty pages may be used to reduce the cost of active-checkpoints. Third, one may explore support of unmodified guest OSes, such as Windows, with hypervisor support to repackage guest packets for best-effort deterministic TCP. COLO platform provides an efficient architecture to employ these potential technical candidates for output similarity enhancement in order to treat with various and complex applications in cloud computing and data-center.

Re-examining TCP/IP stacks to achieve deterministic behavior without effecting throughput or fairness is another ongoing research direction. Although the optimizations to improve the output similarity may be subjected to the feature of the TCP/IP stack the applications use, and may require additional tuning effort, COLO presents an efficient application-agnostic solution to non-stop service, to many of the applications if not most. High level device virtualization solutions with improved determinacy, such as the storage device mentioned in 4.3, may be used to simplify the solution of device state checkpointing, and therefore improve performance.

Although COLO intends to provide application-agnostic solution to non-stop service, one may be able to combine COLO with minor application level modification to greatly improve performance with enhanced response determinacy. For example, the concurrency control schemes to improve the serial ordering of transactions in databases [34], can be reused in COLO to achieve a better tradeoff for an efficient non-stop service solution.

5 Evaluation

In this section, we examine the performance efficiency of COLO achieved by replication disturbance reduction with output similarity constraint using various benchmarks. We first verify the fairness of the modified TCP protocol to prove that COLO does not impact on the TCP connection attributes and features among multiple connections. Second, we compare the performance enhancement of COLO in comparison with Remus with multiple micro-benchmarks as well as illustrating their output similarity. Finally, we evaluate the efficiency of COLO for multi-processor hosted VM and recovery performance from the failover when executing web services and database. Note the native networked system without replication is used as the native baseline, and we illustrate that COLO has significant performance improvement than Remus.

Note that we did not execute the comparison between COLO and other commercial non-stop service solutions because we focus on the software based replication solutions. Moreover, commercial non-stop service solutions prohibit free publication of benchmark results in their license agreements.

5.1 Testbed Configuration

Each of the primary and backup 'server' systems is an Intel® Xeon^{TM} platform, equipped with an 8-core processor, running at 2.7GHz, with 128 GB memory. An Intel® 82576 Gigabit NIC and an Intel®82599 10 Gigabit NIC are used [5], in which the former one for the external network and the later one for the internal network. Except where stated, the guest runs PV Xen Linux with dedicated cores and 2 GB memory equipped with 1 VCPU, and uses a PV disk and a PV NIC driver. The experiment runs Xen 4.1 as the hypervisor and uses a 10 MB memory buffer to log the disk write operations. Domain 0 uses RHEL6U1 (kernel updated to 3.2) to take the advantage of the latest hardware, while the guest uses RHEL5U5 which support Xen 4.1 formally. The client system has the same hardware configuration as the server, but runs RHEL6U1 natively.

5.2 Impact of TCP/IP Modification

As aforementioned, TCP/IP protocol is modified to enhance the output similarity. We now verify that the modification does not impact the network connection fairness in comparison with the unmodified native TCP/IP in a WAN connection environments.

We use iperf to generate multiple concurrent TCP connections to evaluate the impact of our modifications through a company network in between Beijing and Shanghai. Seen from Figure 5 and Figure 6, the standard deviation among the concurrent TCP connections in COLO is close to that of native, which proves that the COLO modification on the TCP/IP protocol stack does not bring obvious fairness issues.

We have also executed the stress testing to compare

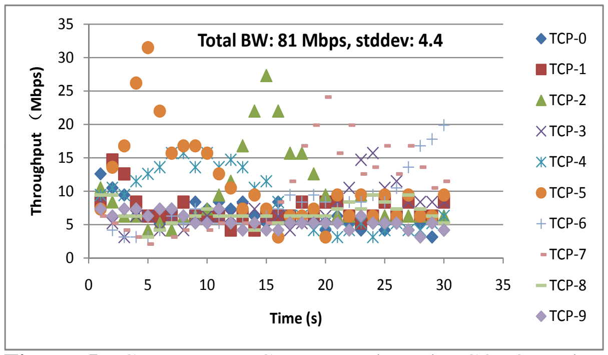

Figure 5: Concurrent TCP connections in COLO native TCP/IP stack in WAN

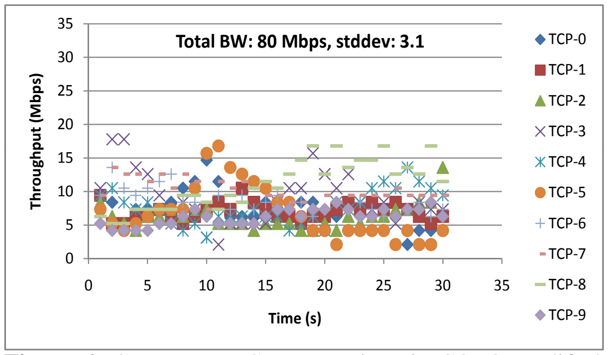

Figure 6: Concurrent TCP connections in COLO modified TCP/IP stack in WAN

the performance of modified TCP stack with native TCP protocol, and the results show that our modification does not introduce significant impact on connection performance. We do not illustrate here since the experimental results are similar,.

5.3 Performance and Output Similarity

This section will illustrate the experimental results of COLO in comparison with Remus evaluated by numerous benchmarks, including Sysbench, Kernel Build, Web Server, FTP Server, as well as Pgbench. These micro-benchmarks are representative for applications in cloud computing and data-center. The experimental results are organized according to the employed benchmark types, and every benchmark will be introduced individually together with the testing scenario.

5.3.1 Sysbench and Kernel Build

This subsection evaluate the CPU and Memory performance of COLO in comparison with Remus. The Sysbench and Kernel Build are used to evaluate the system performances, where Sysbench can produce CPU or Memory-intensive workload while Kernel Build can emulate a CPU and Memory mixed workload. The experimental data is normalized to that of native system to show the differences and performance efficiency.

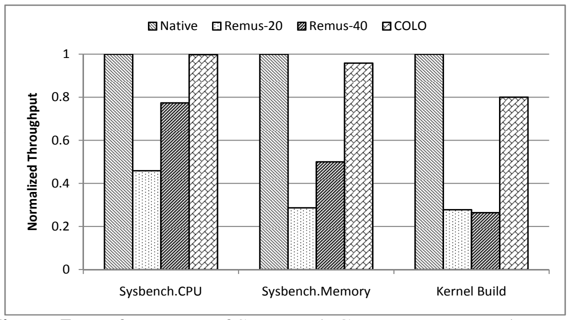

Figure 7: Performance of SysBench CPU, Memory and Kernel Build

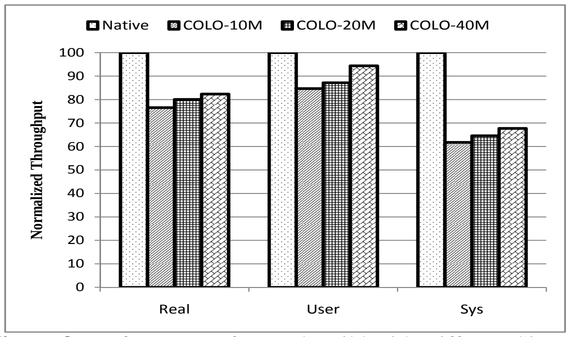

Figure 8: Performance of Kernel Build with Different Size of Log Buffer

As shown in Figure 7, COLO performance is similar to the native tested by SysBench [6] with CPU and memory-intensive workloads; and it achieve 80% of native performance when running kernel build. We configure the Remus checkpoint time interval by 20ms and 40ms, which are denoted by Remus-20 and Remus-40, respectively. These replication period configurations are recommended for application-transparent fault tolerance [12] with effective replication. It is shown in Figure 7 that COLO outperforms Remus-40 by performance improvement of 29%, 92% and 203%, respectively. This is because COLO reduces the replication overhead and the saved resources are used to process CPU and memory-intensive workloads. While, Remus-20 has the lowest performance due to the high replication overhead. The additional overhead for COLO in Kernel Build comes from the on-demand VM checkpoint when there are too many pending storage write operations.

Figure 8 illustrates the performance impact from the different buffer size when executing Kernel Build in COLO platform. Known that the larger the memory buffer, the better the performance of the kernel build due to the reduced VM checkpoint frequency. We set the buffer size with 10, 20 and 40MB, and the performance of COLO is denoted by COLO-10M, COLO-20M and COLO-40M, respectively, in Figure 8. Note that a larger buffer size can improve the performance, but having a larger memory buffer may lead to long transmission latency in VM checkpoints, which is unsuitable for some cases.

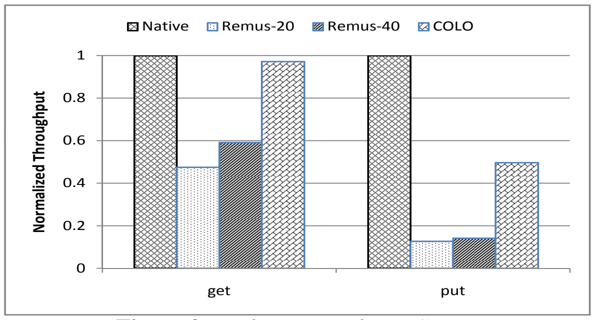

Figure 9: Performance of FTP Server

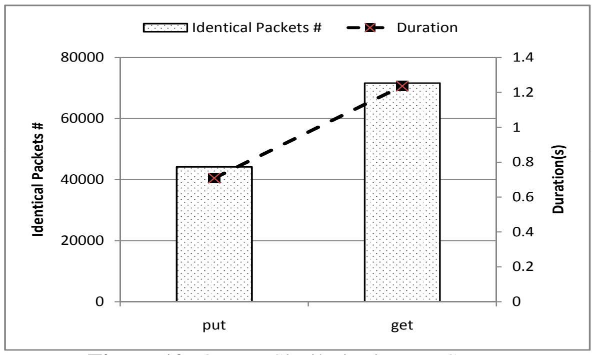

Figure 10: Output Similarity in FTP Server

5.3.2 FTP server

We evaluate the performance of COLO when executing FTP server [1] as the networked system workload on COLO architecture. Figure 9 illustrates the relative throughput performance when transmitting a 300MB file, where the native performance is normalized as 1. It is shown that COLO achieves 97% of native performance in FTP GET benchmarks, outperforming Remus-40 by 64%. That is because COLO can release the output packets much faster than Remus (without output buffering method as Remus) and spend less CPU cycles in the VM checkpoint. In FTP PUT benchmarks, COLO achieves 50% of native performance, outperforming Remus-40 by 148%. Remus-20 has also the lowest performance due to the same reason mentioned before.

Note the checkpoint frequency is significantly reduced by enhancing the output similarity from TCP/IP coherences efforts. We evaluate the average checkpointing time interval, and the results are shown in Figure 10. It is seen that COLO achieves an average output similarity time of 1236ms (during 71.6K packets transmission time interval) in FTP GET, and achieves 709ms (during 44.2K packets transmission time interval) in FTP PUT benchmarks, respectively. FTP PUT benchmarks incur more performance overhead due to the reduced output similarity and the additional VM checkpoints due to the accumulated disk write operations.

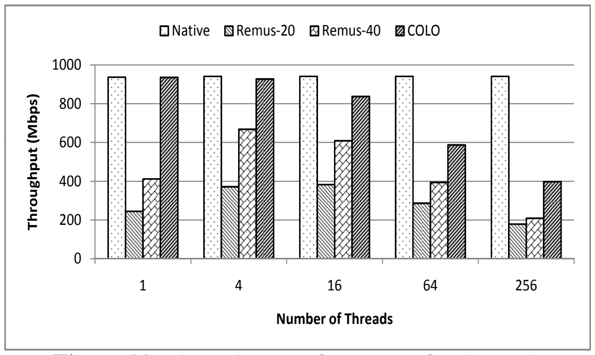

Figure 11: Throughput Performance of WebBench

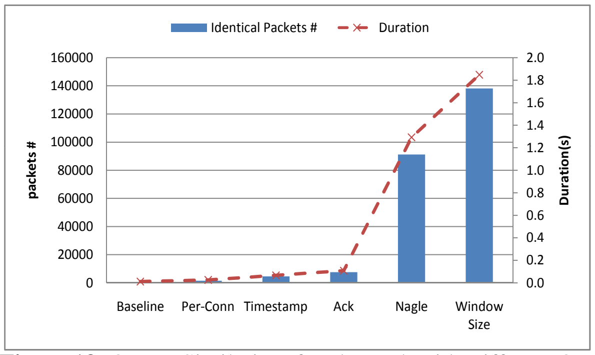

Figure 13: Output Similarity of WebBench with Different Optimizations

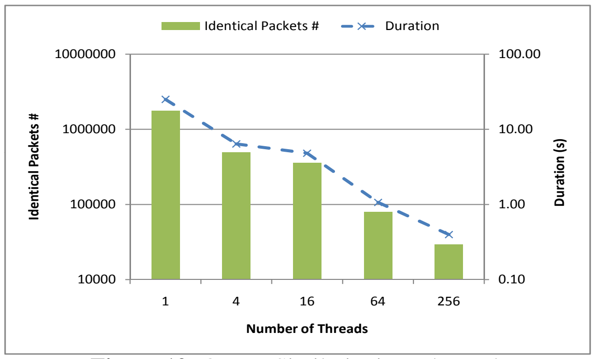

Figure 12: Output Similarity in WebBench

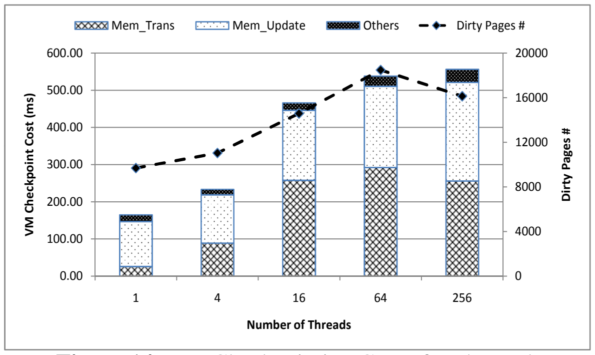

Figure 14: VM Checkpointing Cost of WebBench

5.3.3 Web Server

We evaluate the performance of COLO when hosting the web service workload tested by WebBench [2] micro-benchmark, and the results are shown in Figure 11. Apache Web server is installed in COLO, and client runs WebBench with a varying number of concurrent requesting threads (vary from 1 to 256 threads) on the client. Remus suffers from the excessive VM checkpoint cost, and the extra network latency due to the output packet buffering. While, COLO achieves much higher performance than Remus. For example, COLO achieves 100%, 99% and 89% of native performance, and it outperforms Remus-40 by 127%, 39%, and 38%, in cases involving one, four, and sixteen threads, respectively.

Figure 12 illustrates the output similarity duration and the transmitted similar packets number in logarithmic scale. In the case of 64 and 256 threads, COLO achieves 62% and 42% of native performance, outperforming Remus-40 by 49% and 91%, respectively. In average, COLO outperforms Remus by 69% due to the enhanced output similarity. In Figure 12, it also illustrates that the checkpointing frequency of COLO changes dynamically along with the different number of the concurrent request threads. This proves that COLO executes the checkpointing in on-demand manner according to the output similarity observation. This feature makes COLO a totally novel architecture in comparing with the traditional periodic checkpointing schemes, such as Remus.

As of the current implementation, the performance of COLO drops moderately when the thread number reaches 256 due to the reduced output similarity. However, it is not an architectural limitation, and we believe the performance can be further improved with additional engineering efforts.

Figure 13 shows performance improvement achieved by the TCP/IP modification for coherence enhancement methods introduced in Section 4.1. The performance improvements are illustrated individually by accumulated performance improvement in Figure 13, when running WebBench with sixteen concurrent requesting threads on the client. It can be seen that the packet size coherence enhancement with Nagle algorithm and TCP acknowledge window size coherence enhancement algorithm have the largest contribution in output similarity enhancement. In combing all these TCP/IP coherence enhancement algorithms, the modified TCP/IP stack significantly improves the output similarity by 1160x in packet number, or 753x in output similarity duration.

Figure 14 shows the overhead of VM checkpoints with different number of threads as well as the polluted dirty page number. As shown, one of the major overheads come from memory transmission, due to the large number of dirty memory pages as a result of improved output similarity. The average memory transmission speed is approximately 2Gbps only in the curren-

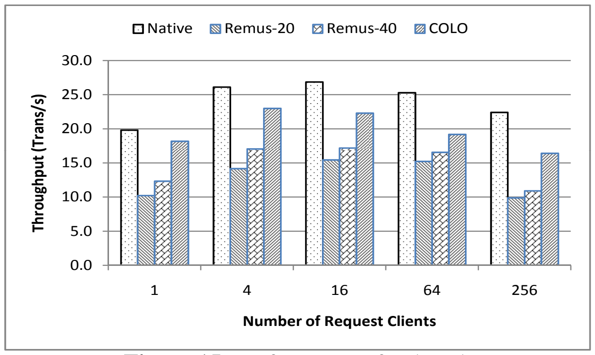

Figure 15: Performance of Pgbench

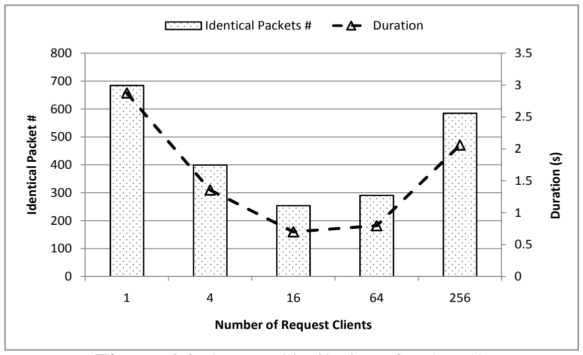

Figure 16: Output Similarity of Pgbench

t implementation, demonstrating the large potential for improvement by reducing the VM checkpoint cost.

Another important cost comes from the guest page table page updating, which involves slow memory page pin and unpin operations in Xen PV guests [10]. The cost can be completely eliminated in the hardware assisted virtual machine, where the two-dimensional page tables are used [11], as a direction for improvement.

5.3.4 PostgreSQL

This subsection evaluate COLO performance with PostgreSQL, which is one of most advanced open source Database, for emulating the applications in data-center access. Figure 15 shows the performance of the PostgreSQL when being exercised by pgbench (loosely based on TPC-B) running on the client. COLO achieves 82.4% of native performance in average and 85.5% of native peak performance, which outperform Remus by 46% and 34%, respectively.

The output similarity of the PostgreSQL database server is shown in Figure 16. In the case of one, four, and sixteen client requests, the output similarity drops, but it remains much greater than the duration of the VM checkpoint as shown in Figure 17, and therefore COLO is able to achieve very good performance. As the number of client requests increases, the identical packets generated per TCP connection keep dropping slowly. However, the overall output similarity re-bounces and the cost of the VM checkpoint jumps, largely due to the increased number of dirty memory pages. Dirty memory page transmission and guest page table updating dominate the cost of the VM checkpoint (80% in the case of 64 client requests and 84% in the case of 256 client requests), demonstrating the large potential for improvement opportunities, as well as further reducing the VM checkpoint cost and therefore improving performance.

Figure 18 shows the breakdown of output similarity improvement with different optimizations, when running pgbench with 64 concurrent client requests. As in Web Bench, modifying guest TCP/IP stack significantly improves the output similarity by 10.3x in packet number, or 10.4x in duration.

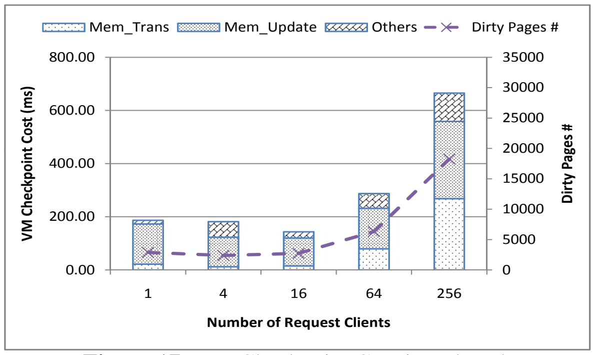

Figure 17: VM Checkpoint Cost in Pgbench

5.4 COLO Scalability for MP-Guest

This subsection will study the COLO performance when the number of VCPU owned by the guest and the number of task threads scale up. WebBench and Pgbench are used to test COLO performance in the cases of the web server and the database server workloads, respectively.

Figure 19 illustrates performance of COLO with WebBench for Multi-Processor guest with 2VCPU and 4VCPU, whose data are compared with 1CPU relatively. We vary the concurrent thread number from 1 to 256. It is shown that the performance of MP-guests does not degrade for the guest with multiple processors in comparison with the single processor guest. The COLO with MP can even perform better than the single processor guest in leveraging the MP resources. It proves that COLO in MP-guests does not show additional overhead and has the potential to scale well with more VCPUs.

Figure 20 shows the performance of the PostgreSQL database server when being exercised by Pgbench in MP-guests, with different number of client requests (from 1 to 256 requests). The performance of PostgreSQL in MP-guests is similar to that of guest with uniprocessor, demonstrating that COLO can scale well with more VCPUs as well.

5.5 COLO FailOver Time

This subsection will discuss the recovery mode agility of COLO, which means the time that SVM of CO-

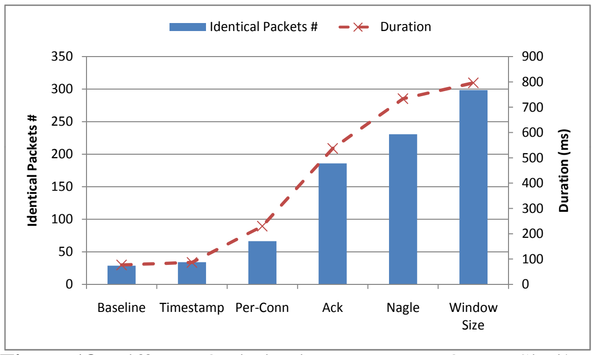

Figure 18: Different Optimizations Impact on Output Similarity with Pgench

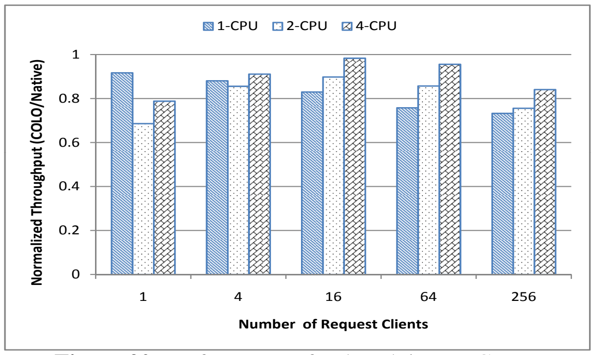

Figure 20: Performance of Pgbench in MP-Guests

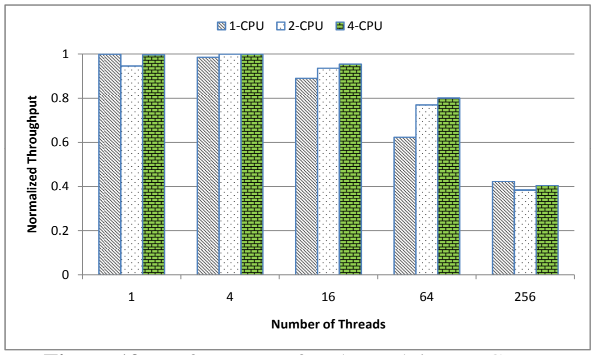

Figure 19: Performance of WebBench in MP-Guests

LO need to execute failover in case of the PVM failure. WebBench and Pgbench are used to evaluate the COLO failover time. WebBench in the client runs with 16 threads, and the PostgreSQL database server running pgbench in client with 16 concurrent requests. We have experimented with 100+ tests of failover and the SVM can successfully take over in all tests. Our experimental results show that the SVM in COLO can successfully take over and resume the service in 2.1s for the Web server and 1.5s for the PostgreSQL database server.

In summary, the evaluation results in this section illustrate that COLO performs significantly better than Remus, when executing the workloads fo Web server, FTP server as well as database server. The performance improvements are achieved by the checkpoint overhead mitigation from the output similarity, which is also illustrated in this section. The employed benchmarks can represent all major application scenarios in current cloud computing and data-center infrastructure, so that the efficiency of COLO can be verified.

5.6 COLO Performance Limitation

As aforementioned, COLO maintains the storage device as the internal state of the guest OS, and the snapshoting of storage state as part of the VM checkpointing will introduce the heavier overhead. This degrades the performance for I/O intensive workload, such as storage access especially for writting operations. This effect has been illustrated by the evaluations with benchmarks as Kernel Build (Figure 7-8), FTP server (Figure 9), and Pgbench (Figure. 15). More efficient storage and other device lock-stepping schemes should be explored in the future. However, this can be considered as a reasonable cost for achieving high availability, and COLO outperforms Remus significantly in these experiments resuls which verifies the efficiency of COLO.

Moreover, COLO suffers from scalability problem along with the concurrent thread number increases as shown in Figure 11-12, where it under-performs native one. These are introduced by the non-deterministic multi-thread scheduling in the guest OS, where different thread execution sequence may commit the different result orders to network output protocol stack and generate output divergence between PVM ans SVM. This is the prevailing problem for all lock-stepping solutions that maintain the replications standby to take over the service immediately for failover. It can be considered as the withstood cost for active lock-stepping, and this is the reason why some current active lock-stepping solutions normally support single process VM [19]. In this paper, we have enabled COLO to support MP-guest efficiently, but there is still further improvement potentials for output similarity as our future work.

We have also illustrated how our proposed output similarity optimization methods can improve the performance one by one in Figure 13 and Figure 18. This verifies that our work concentrated on the crucial points and of course other sophisticated optimization methods can be applied to COLO as discussed in Section 4.5.

Although COLO achieves high availability effectively for the above cases, it may have the cases where the outbound response packets may diverge largely between the PVM and SVM, which therefore may make COLO not work well. Figure 21 shows an example that we upgrade the guest Linux from RHEL5U5 (kernel version 2.6.18) to SLES11 (kernel version 3.0.13), which uses sendfile with unblock writing scheme rather than our previous

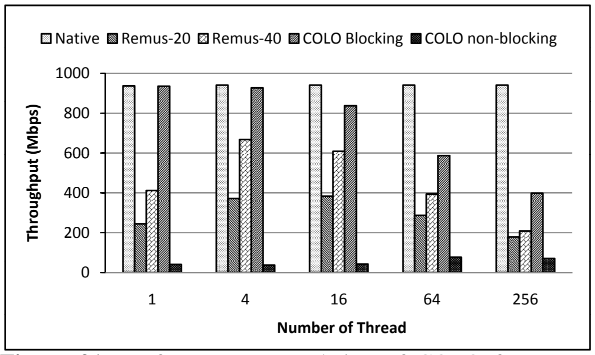

Figure 21: Performance Degradation of COLO from non-block Sending in WebBench

blocked writing experimental scenarios. We re-conduct the experiments in Figure 11, and compare the COLO performance with blocking and non-blocking sendfile schemes. It is observed that the checkpointings are incurred about per 100-200 packets even with COLO coherence efforts, which dramatically degrade the COLO performance (worse than Remus) as shown in Figure 21.

6 Related Work

OS and application-specific solutions are proposed for fault tolerance and non-stop service. TFT [12] interposed a supervisor agent between the application and the operating system to coordinate replicas at the system call interface for deterministic computing without considering multi-processor situations. PLR [33] used the additional runtime systems for process level redundancy in single-thread models. Determinator [9] modified the OS to support deterministic parallel computation, introducing a new parallel programming model. They all suffered from usage model limitations and/or OS and application modifications. xsyncfs [29] is an externally synchronous file system to provide high-performance synchronous I/O with software-only approaches.

Deterministic replay for multi-threaded programs is explored. Scribe [23] provided application record-replay with new operating system mechanisms, rendezvous and sync points to record non-deterministic interactions. ODR [8] was a replay system that reproduces bugs, relaxing its fidelity guarantees. SMP-Revirt [19] replayed execution of multiprocessor virtual machines for debugging. Dthread [25] enforced determinism in the face of data races and deadlocks by exploding multi-threaded applications into multiple processes, with private, copy-on-write mappings to shared memory. None of them provided an efficient solution for non-stop service.

TCP-specific solutions were proposed for non-stop service. ST-TCP [27] and CoRAL [7] modified the TCP protocol excessively to tolerate TCP server failures, and FT-TCP [35] modified select applications to be free from non-deterministic instructions, respectively. They require extensive software engineering efforts.

VM replication enables application-agnostic, non-stop service, surviving hardware fail-stop failures. A fine-grained instruction level VM lock-stepping [13][30][31][21] solution are proposed, however it suffers from excessive lock-stepping overhead due to non-deterministic memory accesses in MP-guests. Periodic checkpointing is then proposed to address the excessive overhead, by checkpointing VM state per epoch [17][28]. Additional optimizations are conducted to reduce the overhead of memory checkpoints [26][36], however they suffered from extra network latency due to output packet buffering and the overhead of frequent checkpointing. Neither of them provides highly efficient, non-stop service for different usage models.

7 Conclusion and Future Work

This paper proposed a revolutionary solution for high availability with active/active virtual machine with Coarse-grained lock-stepping method. COLO provides an efficient and generic application-agnostic solution to non-stop service for networked client-server systems, and provides efficient dependable infrastructure as a service for a variety of workloads in cloud computing environments. Using output similarity between the primary and the secondary VMs, it eliminated the excessive lock-stepping overhead in traditional instruction level lock-stepping solutions [18][31][21], which suffer from excessive instruction replay overhead for memory accesses in MP-guests. Compared to periodic checkpointing solutions, COLO eliminates the overhead due to frequent checkpointing and extra-long network latency, due to output packet buffering [11].

COLO can construct replicas for non-stop service, and the initial results show that it is a very attractive direction for continuing research and industry experiments. We plan to further improve COLO performance for production use and extend support to Windows guests. We intend to further study on the response similarity of different network workloads and additional optimizations to optimize similarity as discussed in Section 4.5. We expect to investigate the use of COLO in disaster recovery situations with long distance deployment, and with multiple secondary servers to survive from t fail-stop failures with $t+1$ replicas.

Acknowledgements

The other members who contributed to this study include Kun Tian, Jiangshan Lai and Congyang Wen. We would like to thank Richard Uhlig for his excellent insights and mentoring; and would like to thank Honesty Young for his great support to the project. This work was supported by Major Program of Chinese Ministry of Education (No. 313035) and Singapore NRF CREATE programme E2S2.

References

[1] Very secure ftp daemon (vsftpd). http://www.nlm.nih.gov/mesh/jablonski/syn drome_title.html.

[2] Web bench,. http://home.tiscali.cz/ cz210552/ webbench.html.

[3] Xen summit 2012. http://www-archive.xenproject.org/xensummit/xs12na_talks /agenda.html.

[4] S. Abood. Hp non stop server. http://www.hp.com, Jun 2002.

[5] S. Abood. Intel®82576 and 82599 gigabit ethernet controller datasheet,. http://www.intel.com, Jun 2002.

[6] S. Abood. Sysbench. http://sysbench.sourceforge.net/, Jun 2002.

[7] N. Aghdaie and Y. Tamir. Coral: A transparent fault-tolerant web service. *Journal of Systems and Software*, 82(1):131–143, 2009.

[8] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206. ACM, 2009.

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

[10] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[11] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGARCH Computer Architecture News*, 36(1):26–35, 2008.

[12] T. C. Bressoud. Tft: A software system for application-transparent fault tolerance. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 128–137. IEEE, 1998.

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

[14] N. Burton-Krahn. Hotswap-transparent server failover for linux. In *USENIX LISA'02: Sixteenth Systems Administration Conference*, pages 205–212, 2002.

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

[16] 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*, pages 273–286. USENIX Association, 2005.

[17] 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.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[19] 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*, pages 121–130. ACM, 2008.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[21] C. M. Jeffery and R. J. Figueiredo. A flexible approach to improving system reliability with virtual lockstep. *Dependable and Secure Computing, IEEE Transactions on*, 9(1):2–15, 2012.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 237–250. USENIX Association, 2012.

[23] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on

commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 155–166. ACM, 2010.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[25] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[26] 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.

[27] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 373–382. Citeseer, 2003.

[28] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: Transparent high availability for database systems. In *Proc. of VLDB*, 2011.

[29] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *In OSDI*, 2006.

[30] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 83–92. IEEE, 2007.

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

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

[33] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(2):135–148, 2009.

[34] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[35] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud. Engineering fault-tolerant tcp/ip servers using ft-tcp. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 393–402. Citeseer, 2003.

[36] 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. IEEE, 2010.