SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2018, Vol. 61 092110:1–092110:15
https://doi.org/10.1007/s11432-017-9292-9

Asymmetric virtual machine replication for low latency and high available service

Rong CHEN & Haibo CHEN*

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240*, China*

Received 16 July 2017/Revised 25 September 2017/Accepted 14 November 2017/Published online 20 June 2018

Abstract Providing fault tolerance support to client-to-server applications is critical in the data center and cloud computing environments. Virtualization provides a direct way of achieving high availability by encapsulating the protected applications into the virtual machine and by periodically checkpointing the entire virtual machine (VM) state to the backup replication. However, existing VM replication solutions suffer from either excessive checkpointing overhead and network latency or unnecessary CPU resources consumption in backup replication. In this study, we exploit the ingredients of output packets and consider that the replication system maintains external consistency if the pre-released packets originate the already synchronized states. Furthermore, we transform the active-active primary and slave VM combination into an active-semiactive one by shrinking the number of active virtual CPUs (vCPUs) in the slave VM. The former optimization mechanism improves the performance in read-mostly client-to-server networked applications, whereas the latter one relieves the problem of double scheduling in the slave host. Therefore, we proposed the COLO++ system which is built over COLO and is a non-stop service solution with coarse-grained lock-stepping VMs for client-to-server systems. The two plus signs represent two of the optimizations. Experimental results using COLO++ implemented on KVM and Linux depict that it achieves nearly native VM performance under read-mostly workloads, as well as lower scheduling overhead in backup replication.

Keywords virtualization, fault tolerance, VM replication, memory, CPU scheduling

Citation Chen R, Chen H B. Asymmetric virtual machine replication for low latency and high available service. Sci China Inf Sci, 2018, 61(9): 092110, https://doi.org/10.1007/s11432-017-9292-9

1 Introduction

High availability (HA) is a critical feature for modern data centers and cloud computing environments. Any downtime of an application service may result in property damage and loss of customer loyalty. Generally, high available services are implemented using high available clusters [1], and user requests will be directed to active hosts in such clusters. However, some HA solutions rely on highly specialized hardware or software design1) to operate failover process.

Replication is a standard approach to solve fault tolerance, since it provides redundancy to achieve high availability. Once a failure occurs in a replica, the services that function using it can be taken over by other replicas [2]. Virtualization [3, 4] provides a method to encapsulate application services. Generally speaking, a virtual machine (VM) can encapsulate any kind of applications and can function on any type of hardware. A VM replication-based HA system can be implemented once and can be applied to many architectures.

* Corresponding author (email: haibochen@sjtu.edu.cn)
1) S. Abood. Hp non stop server. 2002. http://www.hp.com.

A generalized solution for HA services is to replicate a VM to tolerate hardware fail-stop failure. Lock-step replication [5] synchronously propagates every single instruction from primary VM to slave VM to achieve consistent states. Additionally, it suffers from significant overhead due to non-deterministic memory accesses in symmetric multi-processing (SMP) VMs. Instead of running two hosts in synchronously lock-step mechanism, other VM based replication HA systems, such as Remus [6] and COLO [7], replicate the VM states in an asynchronous manner. They utilize a live migration technique [8] to checkpoint the VM states into a backup physical host, including CPU states, memory states and devices states. The objective is to hide the system states from the outside world before the checkpoint is committed.

Remus suffers from significant overhead due to high frequency checkpointing, but the active/passive model saves the hardware resources of slave host. COLO reduces the checkpoints to improve its performance by utilizing a packet similarity comparison. However, it triggers the slave VM to execute like a shadow of the primary VM, which consumes the CPU resources and leads to double scheduling problems. Furthermore, packets generated by slave VM need to rendezvous with the packets in primary VM to perform the comparison. This leads to overhead in packets that are transferring over the network, especially for read-mostly client-to-server networked applications.

In this study, we propose COLO++, which is an enhanced VM replication system built over COLO. As the name implies, two optimizations are applied in the proposed system: (1) Pre-release clean output packets to skip over the comparison and checkpointing. (2) Shrinking the number of vCPUs (virtual CPUs) to relieve the scheduler stress in the slave hypervisor.

This study makes the following contributions:

- We analyze two points in the existing VM replication system (Section 2) and provide a new design to address the issues of the existing system (Section 3).
- Set of techniques to efficiently implement the system on kernel-based virtual machine (KVM) (Section 4).
- Set of evaluations to confirm the effectiveness of COLO++ (Section 5).

The remainder of this study is organized as follows: Section 2 introduces the issues of existing VM replication approaches and the motivation for our study. Section 3 describes the overall design and architecture of COLO++. Section 4 presents the details of system implementation. We present the evaluation results of COLO++ in comparisons to the existing solutions in Section 5. Related work is discussed in Section 6. Finally, we conclude the paper in Section 7.

2 Background and motivation

Generally, the primary VM will periodically synchronize its states to the slave VM in the VM replication model. States are the data of users on the server and application runtime environment, which includes CPU states, memory states, and device states. The VM replication system can be treated as a black box, which appears to the user as a single standalone VM with hardware fail-stop fault tolerance support. In a system having weak consistency, user data is more likely to be lost when a failure occurs. Therefore, high consistency is more desirable in a VM replication system, to provide seamless crash recovery and data protection.

2.1 External consistency

A typical VM replication system achieves high availability by blocking the output network until checkpointing is done. There are many steps in the process of VM executing to obtain consistency among VM states: (1) The VM executes normally while waiting for the user requests. (2) The replication system buffers response packets, and triggers a checkpoint to synchronize the dirty states. (3) After checkpointing is performed, the output packets are released. If the VM replication system releases packets before triggering checkpoints, consistency semantics may be violated.

Below is an example that violates the rules: A key/value store service is running on the VM replication system. In this scenario, the original value of “foo” equals to 10. The user requests to change this value

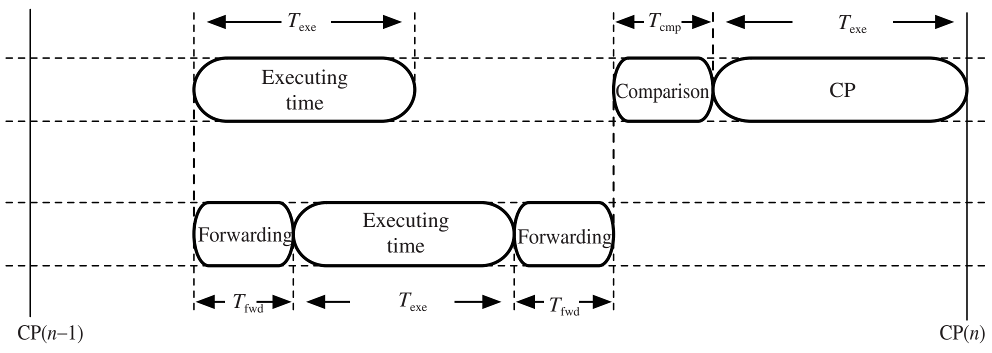

Figure 1 Roundtrip latency.

from 10 to 20 and obtains the correct response. Subsequently, a hardware fail-stop occurs in the primary VM, and the slave VM begins to control the entire system. Since the system does not synchronize the VM states before responding to the user, inconsistency occurs when the second user obtains a response from the system for the value of "foo". Additionally, this value will be 10 instead of 20 in this scenario, which is confusing to the user. This problem is known as the output commit problem [9].

2.2 Remus and COLO

Remus [6] is a novel VM level replication system that can be used to obtain HA using commodity hardware. The model of the Remus replication system is an active primary VM/passive slave VM. Thus, the primary VM executes actively, whereas slave VM is suspended until the primary VM crashes. After the failure of primary VM, the slave VM controls the execution. From the perspective of a client, the system provides services seamlessly.

The primary VM periodically performs checkpoints on the slave VMs with high frequency to achieve strong consistency. The checkpoint process is based on live migration techniques [8]. Remus will block every output network packet until the checkpoint is acknowledged by the slave VM to achieve consistency. However, this replication model introduces significant latency overhead, even though it provides high availability. Services running on such a system will suffer from response time delays due to the longer lasting processes. As previously mentioned, such a replication model will introduce significant overhead, which will introduce an important benefit i.e., Remus only executes the slave VM after failover. All the vCPUs are frozen in the slave VM. Therefore, the scheduler does not have the additional burden of forcing the system to run.

COLO [7] is a coarse-grained lock-stepping VM replication method whose implementation is based on Remus. Since the slave VM in Remus is not actively executing, it leads to passive checkpoints. COLO performs active checkpointing by actively executing both primary and slave VMs in parallel. The slave VM is equipped with identical hardware resources as the primary VM, which includes number of vCPUs and memory size to make both VMs run at a relative consistent pace.

As depicted in Figure 1, COLO handles the request of the client by the following steps.

(1) The primary VM receives the client request and forwards them to the slave VM.

(2) Both VMs execute the client request simultaneously.

(3) The primary VM compares response packets from both primary VM and slave VM. Subsequently, it releases one copy of the two packets if both packets are observed to be identical; otherwise, it triggers the Remus checkpoint.

Such a design minimizes the number of checkpoints. Therefore, the overall latency overhead is reduced. COLO obtains higher performance by offering tradeoffs in two respects: First, Remus possesses strong consistency by blocking the output network packets. However, this incurs a high response latency. COLO is tuned to obtain a higher performance. However, the consistency of COLO is compromised. Secondly, COLO runs primary and slave VMs concurrently, which leads to hardware resource overhead in the slave host.

Table 1 Performance gap between single VM and COLO system

	Single VM	COLO	Round-trip packet	Checkpointing
Latency (ms)	3746	6921	1080	1968

2.3 Issues and motivations

Packets synchronize latency. COLO needs to collect packets from the primary and slave VMs to perform comparison or checkpoints, which leads to a noticable slowdown in performance. We evaluate COLO and the single VM system, respectively, by running the memcached service to demonstrate the performance gap. Further, we use Yahoo! Cloud Serving Benchmark (YCSB) at the client side to issue 10000 requests and observe the overall runtime.

The result is illustrated in Table 1. We can observe that the memcached performance of the COLO system is approximately 2 times slower than that of the single VM. We investigated two procedural problems for handling client requests to understand the factors that slow down the overall performance of COLO.

The first factor is the roundtrip packet latency. The primary VM receives the request, executes the instructions, and generates the response packet. Subsequently, the primary VM waits for the response of the slave VM. Further, the primary VM compares the two packets. The slave VM receives the request packet from the primary VM and generates a response packet that will be forwarded to the primary VM. If the COLO comparing module in the primary VM wants to access the packets of both primary and slave VMs, it is required to wait for the completion of instructions that are executing a time plus one packet roundtrip time between the primary and slave hosts. The latency of the roundtrip packet between the two hosts is depicted in Table 1. When the client issues 10000 requests, each request incurs a round-trip packet latency, which produces significant overhead.

The second factor is the checkpointing latency. COLO reduces service response latency by running both the primary and slave VM in parallel. Both the VMs generate response packets that will be collected by the COLO comparing module in the primary VM. This module compares the output packets and makes the following decisions: (1) If the contents of the two packets are identical, it means that the states of both the VMs are consistent. Consequently, COLO can continue providing services. (2) If the content of the compared packets is divergent, COLO will terminate all services and initiate the checkpoint routine. Since the checkpoint routine synchronizes the replication states by terminating both VMs, it reduces the performance of the entire system considerably. The last cell in Table 1 provides the latency of the checkpoint routine. While running memcached services, we evaluated that the requests of 10000 clients will incur 23 checkpoints.

CPU resources consumption. Unlike Remus, the slave VM behaves exactly like the primary VM in the COLO VM replication system. The slave VM has an identical configuration as the primary VM, which includes the number of vCPUs and memory size. Additionally, all the vCPUs in the slave VM are in complete use as a normal VM.

The purpose of running the primary and slave VMs simultaneously is to compare the output packets and to reduce the service response latency. However, when running CPU-intensive applications in COLO, there are few network packets that are being sent and received, which means that the COLO system does not gain any benefit from this scenario. Furthermore, it consumes the scheduling resources of the vCPUs in the slave host, which doubles the scheduling problems.

In virtualized environments, the double scheduling problem is described to be a guest OS that is scheduling processes on vCPUs and a hypervisor that is scheduling vCPUs on physical CPUs. Due to the semantic gap between the schedulers in the guest OS and hypervisor, this introduces many problems such as lock holder preemption (LHP) [10].

Here is a simple example, which illustrates the LHP problem, as depicted in Figure 2. vCPU-0 and vCPU-1 run on pCPU-0 and pCPU-1, respectively. Initially, vCPU-0 acquires the lock and is preempted by the hypervisor. vCPU-1 waits for vCPU-0 to release its lock, but it can only acquire the lock after vCPU-0 is rescheduled and exits the critical section. In this example, vCPU-1 is required to wait for an

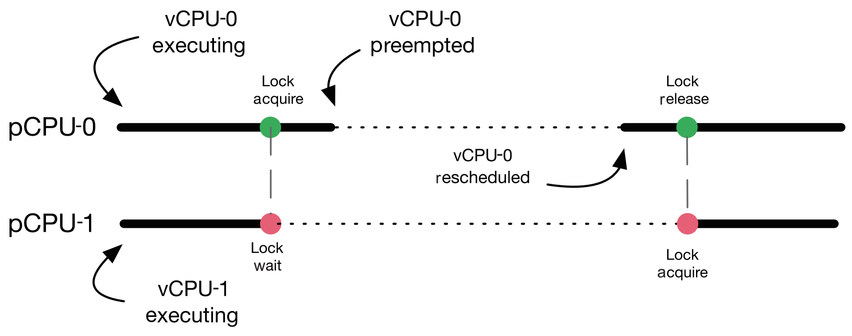

Figure 2 (Color online) Lock holder preemption problem.

additional rescheduling period to acquire the necessary lock.

In a VM replication system, we do not require both primary and slave VMs to be running in a CPU-intensive scenario, since the computing tasks can be performed using a single VM. When the computing tasks are completed or after a long period has elapsed, a checkpoint is triggered to save the states to slave replication. Since there will be very few network packets that are available during execution, checkpoints are not a concern before the packets are flushed. Therefore, making the slave active is not necessary in such a scenario.

Objective. In this study, we propose COLO++. We were motivated by the problems elaborated in the two scenarios using COLO. The two plus signs represent the two COLO optimizations, respectively. Using the first plus, we consider that each output packet consists of several pieces of memory contents. Before flushing the output packets, a comparison or checkpoint is required, which will produce significant overhead. We propose a method to decide whether to release an output packet, according to its constituents. If such an output packet originates from clean memory, this packet can be safely released without violating external consistency. Using the second plus, we consider that it is not necessary for the slave VM to occupy too many CPU scheduling resources. When running CPU-intensive applications, few output packets are generated, and shrinking the number of vCPUs could reduce the severity of the double scheduling problem. Furthermore, only a small part of vCPUs will handle the client requests when running a network intensive application. Therefore, reducing the number of active vCPUs will not affect the efficiency of providing services. In the next section, a more detailed description of COLO++ will be provided.

3 Design overview

In our design, COLO++ is a primary VM active/slave VM semi-active replication system. Due to the semantic gap between guest VM and hypervisor, it is hard to decide whether or not releasing a particular output packet without making sure the consistency of system is not violated. So in the traditional way, replication system does checkpoints or packets comparing before releasing them. As we known, there are always tradeoffs between generality and performance. In COLO++, it trades generality to get better performance. Applications could provide some hints to guest OS and hypervisor, to help make the decision about packets releasing. Besides, in order to ease the double scheduling problem in slave host, we use vCPU-freezing technique to make less vCPU active in slave VM.

In our design, COLO++ is a primary VM active/slave VM semiactive replication system. Due to the semantic gap between the guest VM and hypervisor, it is difficult to decide if it is required to release a particular output packet without ensuring that the consistency of the system is not violated. Therefore, the replication system performs checkpoints or packet comparison before releasing them using the traditional approach. As we know, tradeoffs always occur between generality and performance. In COLO++, generality is exchanged to achieve better performance. Applications could provide some hints to the guest OS and hypervisor to aid the decision about releasing packets. Moreover, we use a vCPU-freezing technique to make the number of active vCPUs to be less in the slave VM, which eases the double

Table 2 COLO++ interfaces

Interface	Function
CLPP_PROTECT	Notify guest OS that a specific memory region needs to be tracked
CLPP_UNPROTECT	Untrack a specific memory region
CLPP_ENTER	Indicate that COLO++ is now handling user requests
CLPP_EXIT	Exit requests handling routine

scheduling problem in the slave host.

3.1 Interfaces

The objective of identifying clean/dirty output packets is to establish a connection between packets and memory regions. This is a method to track all the memory regions that the application could touch. However, this method is inefficient and it will mostly obtain an incorrect result while identifying a clean or dirty packet. Since the virtual memory space of a process consists of the stack, heap, data section, and many such fields, different data types are stored to make the application perform correctly. The stack maintains the running environment and stores several temporal data such as local variables. The heap is the dynamic memory region, and memory could be allocated using malloc and new interfaces. For the data section, it stores nonzero global variables.

Every step of the process will update its memory more or less. For example, when calling a function, the return address and parameters of this function will be pushed into the stack and pollute the memory. Therefore, it will never be enough to track all the memory regions of the application to determine whether the output network packets are dirty.

We observe that only dirty packets are obtained using this method. However, when the application services handle requests from clients, most of the polluted memory regions have no use of the output packets. It is recommended to track the memory regions that really matter to identify clean/dirty packets accurately. COLO++ leverages several interfaces by allowing applications to provide hints to the guest OS and hypervisor to track the memory regions that are related to output packets.

Table 2 lists the interfaces that will be translated into syscalls under certain circumstances. The CLPP_PROTECT interface is used to provide hints to the guest OS and hypervisor that a specific memory region should be tracked. When the tacked memory region is polluted, the output packet generated from this memory region will be identified as a dirty packet. CLPP_UNPROTECT is an API to declare that this memory region does not require more tracking. This API is used in the scenario where a specific memory region has been reclaimed. Therefore, it is necessary to untrack this region. CLPP_ENTER is used for notifying the guest OS that the request handling process is initiating, while CLPP_EXIT performs the opposite function.

3.2 Architecture of COLO++

Figure 3 depicts an overview of COLO++ architecture. Currently, COLO++ is a two-VM replication system built over COLO. The VM that contacts the outside world is called the primary VM. Additionally, the primary VM periodically synchronizes its states to the slave VM to maintain external consistency. The entire system can be treated as a black box having only one entrance and exit point. When the primary VM crashes, the slave VM will take control of the system. COLO++ aims at two weak spots of COLO. The first one is the overhead produced while maintaining consistency, which leads to time-consuming packets performing roundtrip travel or even checkpointing. The second weak spot is the overhead of vCPU scheduling in the slave host, which doubles the scheduling problems. Motivated to solve these weak spots, COLO++ has two main functioning components, i.e., a packet analyzer and vCPU freezer. The application uses a special library called call wrapper, which is used to provide hints to the guest OS. The packet analyzer analyzes packets generated by the application and tags these packets as either clean or dirty. A clean packet is released by the hypervisor immediately, whereas a dirty packet should be subjected to packet comparison or checkpointing before being released. The vCPU freezer shrinks

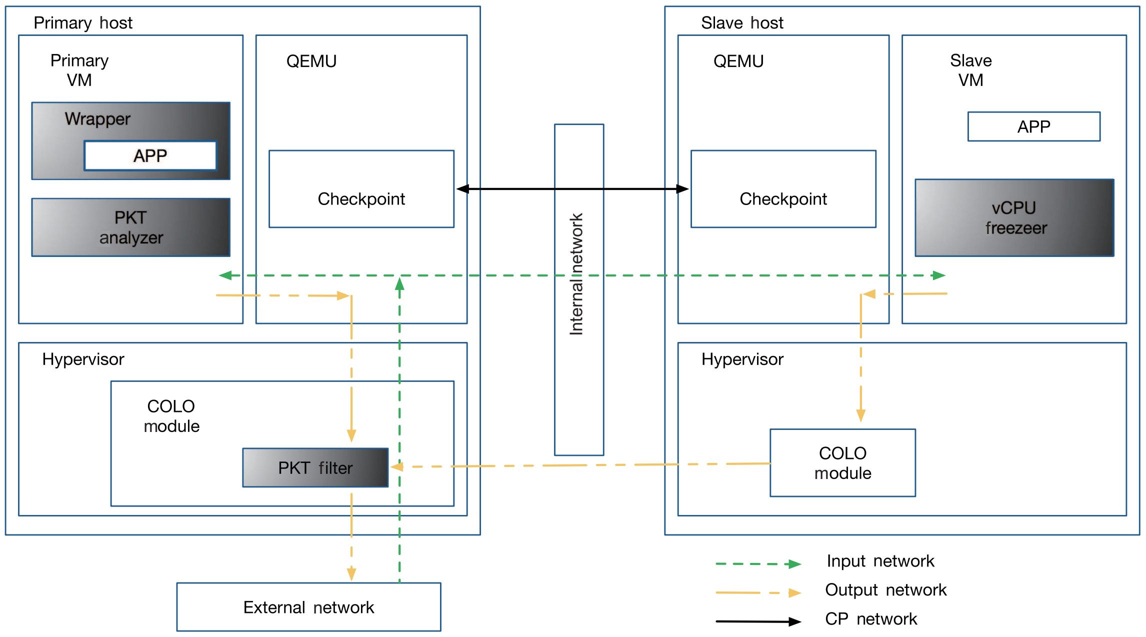

Figure 3 (Color online) Overall architecture of COLO++.

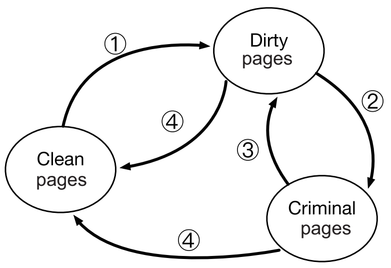

Figure 4 Dirty page states transition.

the number of active vCPUs in the slave VM, so that the hypervisor in the slave host could have fewer entities to schedule. Thus, the double scheduling problem is eliminated.

Actually, COLO++ improves on COLO and also absorbs the benefits of Remus. COLO compares the output packets, whereas COLO++ eliminates this step when the packet originates from a clean memory page. Remus waits for checkpointing but consumes zero CPU scheduling resources in the secondary VM.

3.3 Packets early releasing

The packet analyzer is the key component in identifying clean/dirty packets. Actually, it would be the best solution if we could figure out the exact mapping between the contents of output packets and memory regions. However, the construction of such mapping leads to significant overhead since it needs to track every memory operation and record the data flow of each memory piece, which will finally be generated in the output packets. Once mapping is constructed, it can accurately decide the destiny of output packets. Taint analysis [11] is a technique for tracking and recording data flow information. It requires to modify the binary file of an execution application. However, even if this can be performed, the application should still provide some hints about the destination of packet generation.

Instead of constructing the mapping, COLO++ uses a very coarse-grained method to make a decision about whether a packet is clean or dirty. The packet analyzer maintains a list of all consecutive memory regions that are hinted by the application, and these regions are managed using the granularity of a 4 KB sized page. Each of these pages can be classified into three groups as depicted in Figure 4. At the beginning of each epoch between adjacent checkpoints, all dirty memory is synchronized using a slave

VM. Therefore, all tracked pages are marked as clean pages. After executing instructions from clients, the updated pages will be treated as dirty pages (①). If any dirty page is accessed subsequently, it will be moved to the set of criminal pages (②). When any output packet is generated, COLO++ will check the criminal set. If the criminal set is not empty, this packet will be a dirty packet; otherwise it will be a clean one. In the next round of checkpoint, all the dirty pages and criminal pages will be moved to clean pages set (④).

This method will give a false positive, which will treat the clean packet as a dirty one. However, it will not violate the rule of external consistency because if a packet is marked as dirty, it will be subjected to packet comparison or checkpointing. Actually, COLO is a system that treats all the output packets as dirty ones.

A problem exists due to which for any dirty page going into the criminal set, all subsequent output packets will be treated as dirty packets in this epoch, even though there are only a few irrelevant pages in the criminal set. For read-mostly applications, only a few write operations will generate plenty of dirty packets. To eliminate the problem, COLO++ will clear the criminal set at the right time. Typically, the client issues a request, and the network service will create a new task to handle this request. The right time to clear all the criminal pages is when there are no task handling requests (③). Therefore, the previous request which introduces dirty pages into the criminal set will not affect the subsequent request that only affects the clean pages. The final two interfaces in Table 2 are used to detect whether it is the right time to clear the criminal set. CLPP_ENTER increases the reference counter by one, whereas CLPP_EXIT does the opposite. When the reference counter becomes zero, this indicates that the right rime is approaching.

3.4 Semiactive slave to save CPU

COLO++ shrinks the number of vCPUs in the slave VM to eliminate the double scheduling problem. There is a straightforward method to reduce the number of vCPUs in the slave VM by booting less vCPUs. However this method has several limitations. First, it is not functional when synchronizing CPU states between asymmetric VMs. Each CPU has its per cpu states like registers, variables, and run queue. In a VM replication system, the vCPU in the primary VM should contain the ghost vCPU in the slave VM. Otherwise, when doing checkpoints, it will confuse the system if it is observed that the vCPUs between the primary and slave VMs do not contain one-to-one mapping. Secondly, the slave VM will take control of the system when the primary VM crashes. When booting less vCPUs in the slave VM, it will confuse the clients with regard to the number of vCPUs after failover.

In this study, we use the vCPU freeze technique to dynamically reduce and resume the number of vCPUs in the slave VM. This technique is proposed in vCPU-bal [12] and vScale [13]. We use it in the VM replication scenario to eliminate the double scheduling problem at the slave host. Unlike vCPU-bal and vScale, the vCPU freezer does not affect the performance of the overall system in COLO++. In vCPU-bal and vScale, an agreement is reached for all VMs in one host by which the total number of vCPUs in this host will not exceed the number of physical CPUs. Therefore, the double scheduling problem could be eliminated. However in COLO++, we use this technique only in the slave VM. From the perspective of the client, the performance of the primary VM will not be compromised.

The objective is to make the guest OS schedule its scheduling entities to less CPUs, thereby excluding the hypervisor from vCPU scheduling. Typically, the OS scheduler will try to balance all tasks and distribute them evenly across all CPUs. When the vCPU is fully loaded, the scheduler in the hypervisor will treat this vCPU as a runnable task. If the vCPU has nothing to perform, it will enter an idle state and never consume CPU cycles in the hypervisor. Therefore, COLO++ moves the spread tasks for all vCPUs to a limited subset, whereas some of the vCPUs function normally and the rest enter an idle state.

There are two issues with regard to shrinking the number of vCPUs in the slave VM. The first issue is that the primary and slave VMs are nearly identical, including the CPU executing environments and instructions. This means that the scheduler in the slave VM acts identical to the one in the primary VM.

We need a special data structure, called the freezed_cpu_mask, that will not be synchronized between two VMs to move tasks into the limited subset of all vCPUs in the slave VM while leaving the primary VM unchanged. The schedulers in each VM will refer to their own freezed_cpu_mask and act accordingly. In the primary VM, the freezed_cpu_mask is set to empty. Therefore, the vCPU will not be frozen. However, in the slave VM, the administrator could configure the freezed_cpu_mask using the shared memory between the guest OS and hypervisor. The second issue is that even though we have an exclusive data structure that will not be synchronized after checkpointing, the run queue states of each vCPU still need to be synchronized, which leads to the busy scheduler serving no purpose. To address this issue, COLO++ extends the time limit of periodic checkpoints. This is because the checkpoint time length will not affect the performance of the CPU-intensive application scenario.

For pure CPU-intensive workloads, we can simply make the slave VM passive and perform checkpoints to synchronize the states, which falls back to the design of Remus. However, our semiactive design offers the system administrator a method to manage the CPU scheduling resource in the slave VM. Therefore, the number of active vCPUs will not be “ALL” or “NULL”, when the VM runs a mixed workload of both CPU and network intensive applications. In a network intensive application, the first optimization can be used to obtain better performance. So we cannot “shut down” all the vCPUs because we require one/some of them to do the packet generation work.

4 Implementation

We developed a prototype of COLO++ from the COLO system based on KVM/QEMU in Linux.

Dirty page tracking. To identify clean/dirty output packets, COLO++ classifies tracked pages into clean, dirty, and criminal sets. At the beginning of each epoch, COLO++ will write-protect all pages and mark them as clean pages. Any subsequent page fault transforms the page from a clean set to a dirty set. In a virtualized environment, hardware vendors introduce a new type of page table produced by Intel, named Extended Page Table (EPT). There will be an option to implement dirty page tracking inside the hypervisor through EPT. However, a walk through between two page tables is required to establish a connection between the guest virtual address (GVA) and the host physical address (HPA). These tables are the guest PT and EPT. In EPT, the consecutive memory range in the GVA space may be divided into several pieces. Moreover, it will be much more complicated to identify which process in the guest VM violates the EPT write-protect restriction, since the information inside the hypervisor only mentions which vCPU triggers the EPT page fault. COLO++ implements dirty page tracking inside the guest kernel, and it is much more convenient to maintain every tracked page and source process of the page faults. If the application process accesses the dirty pages, it will try to move the touched pages from the dirty set to the criminal set. COLO++ leverages the access bit on the page table entries to track the accesses of every tracked page. There are two methods to assign memory to VMs: static memory allocation and dynamic memory allocation. Dynamic memory management, such as memory ballooning and memory hotplug, will affect the size of unused memory. However, the access bit on the page table entries will not be polluted by such management techniques. Dirty page tracking inside the guest kernel could be safely applied to COLO++.

The normal procedure for handling a page write protection fault is: (1) The specific instruction tries to write to a write-protected page and triggers a page fault. (2) The kernel handles this fault and returns to the user space. (3) Re-executing the instruction triggers a page fault. There is one issue after step (3), where the access bit in the page table entry is already set. This means that every page already comes from a clean set to a dirty set and is moved into the criminal set immediately. To solve this problem, we used the debug register[2)] to set a breakpoint in the next instruction. Therefore, the guest OS could vary the access bit on the page table entry. As the number of dirty pages in every epoch increases, the system performance drops dramatically due to the process of handling page faults and debug breakpoints. To eliminate the overhead, we set a threshold for the dirty output packet rate (50%

2) x86 debug register. https://en.wikipedia.org/wiki/X86_debug_register.

by default and configurable). COLO++ functionality will be disabled when the dirty output packet rate reaches the threshold and will automatically fall back to the COLO system.

Packets filter. After successfully identifying clean/dirty packets, the guest OS needs to pass the information in order to tell the hypervisor whether to release packets in advance. Virtio [14] was chosen as the main platform for I/O virtualization in KVM. We extended the descriptor table of the send queue in the virtio-net module by one bitto represent clean or dirty packets, respectively. The information could be transferred from the guest front-end to the QEMU backend. However, QMUE is not the component who decides the destiny of the packets, since this is decided by the kernel module in the hypervisor call, where the COLO module is located. Fortunately, QEMU uses one I/O thread mode to handle the I/O request. To continue transferring this information, we developed a channel between QEMU and the COLO module. This is a shared buffer which indicates the mapping of clean/dirty information and the corresponding packet sending QEMU to the COLO module. Therefore, all components could be connected together to serve the purpose of COLO++.

vCPU freezer. COLO++ leverages the guest OS scheduler to reduce the number of active vCPUs in the slave VM. There are several methods to achieve this. Initially, the administrator needs to control the minimal number of vCPUs. For the current implementation, the guest OS boots up using a predefined reserved physical consecutive memory region of 4 KB as the shared memory between guest and hypervisor. This memory region is mapped through EPT and the administrator maintains the freezed_cpu_mask in it. The guest OS scheduler refers to the freezed_cpu_mask as an instruction to scheduled tasks. Since this reserved memory region is a part of the guest physical memory space, it will be synchronized between the primary and slave VMs. Further, the second step is to exclude the shared page from synchronization. In KVM, there are two ways to collect dirty pages during VM migration, namely, page write protection and Page Modification Logging (PML) [15]. In our implementation, we used the former one. The KVM module maintains a bit mask to record all the dirty pages during this period and pass them to QEMU. QEMU sets up a checkpoint by synchronizing dirty pages indicated by this bit mask. To exclude the share page, we excluded the bit in bit mask during dirty page collection, so that the shared page will be not synchronized from there on. The final step is to actually schedule the tasks. In Linux, there are two main types of schedulable entities that can consume CPU cycles: threads and interrupt routines. vSacle [13] provides an extensive discussion on how to interfere with normal scheduling in every push or pull event when performing load balancing. One situation is that, it would confuse the clients if we moved the specific threads away, when the clients set affinities to the threads. The strategy of COLO++ is to keep the affinity in the primary VM, and reserve the affinity in the slave VM. Normally, setting affinities aims for better performance. In COLO++, the performance is mostly determined by the primary VM. The overall performance of the slave VM is weakened after reducing the number of vCPUs. Therefore, CPU affinity is less important in the slave VM. When the fail-stop happens to the primary VM and the slave VM takes control of the system, the slave VM will resume the thread affinities that were formerly disabled.

5 Evaluation

In this section, we describe the evaluation of COLO++ in two sections. In the first section, we evaluated the performance of COLO++ in the read-mostly network-intensive scenario to demonstrate the efficiency of prereleasing the clean output network packets. Secondly, we evaluated the benefit of shrinking the number of vCPUs in the slave VM. Note that the native single VM system without replication and the original COLO system are the two baselines. Additionally, we illustrated that COLO++ exhibits significant performance improvement than COLO in the read-mostly network intensive scenario, and it can also reduce the stress in the slave VM CPU resources without any performance drop, which can even improve performance.

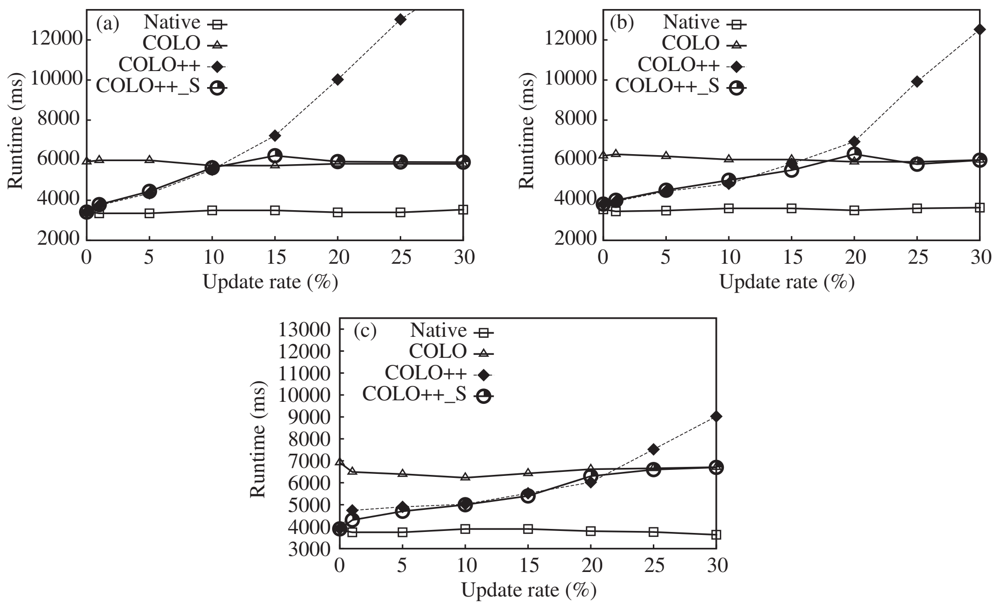

Figure 5 Relationship between update rate and performance (Runtime) in different packet sizes. (a) 1 KB; (b) 2 KB; (c) 4 KB.

5.1 Experimental setup

We evaluate COLO++ using two host machines having a 40-core Intel Haswell processor, 128 GB DDR3 memory. Two Intel 82599 10 Gigabit NIC [16] were used for the external network and internal checkpoint network. Moreover, we boot the guest VM with 16 vCPUs and 2 GB of memory and used virtio-net as the network device. The guest kernel version was Linux 3.14.35. The experiment run KVM as a hypervisor. The host kernel version was 3.18.10 and the QEMU version was 2.3.92. The client system had the same hardware and kernel configuration as the primary and slave hosts.

5.2 Network performance

This subsection will illustrate the experimental results of COLO++ in comparison with that of COLO in a network intensive application. The aim is to show that COLO++ outperformed COLO in the read-mostly benchmark.

Here we chose memcached [17] as the evaluated benchmark. We modified the code of memcached slightly to fit the COLO++ system. To speed up the allocation of objects, memcached reserves a large memory range in order to insert everything. The memcached object allocation procedure was managed by itself to reduce the number of library function calls like malloc. To adopt memcached in COLO++, it was initially required to use the COLO++ interface CLPP_PROTECT to notify that the reserved memory pool was used for serving the output network packets. Additionally, we used CLPP_UNPROTECT to declare when the memory region was reclaimed. Therefore, any read and write operation happening in this space could be monitored. Subsequently, we wrap the user request handling routines by CLPP_ENTER and CLPP_EXIT to provide hints that would indicate the timing of cleaning the criminal set. The effort for the adaptation is approximately 20 lines of code.

We used YCSB as the client to issue 10000 read/write requests to the replicated memcached system. Additionally, we set the number of record fields to 10, and the size of each field (in bytes) to 100, 200, and 400. Therefore, the size of objects was approximately 1 KB, 2 KB, and 4 KB, respectively. The results are depicted in Figure 5(a)–(c).

Figure 5 illustrates the performance (Runtime) of COLO++ when running memcached having different memcached object sizes. The native line in each figure denotes the workload performance when running on an unmodified VM/hypervisor and nonreplicated VM system. For the unmodified VM/hypervisor and

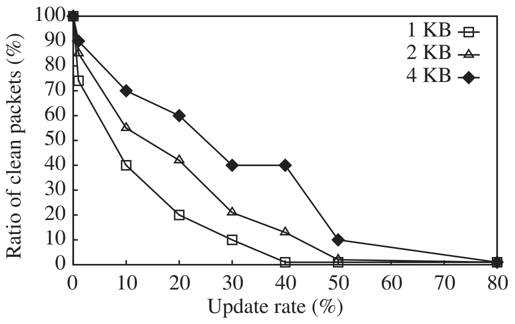

Figure 6 Clean packets ratio.

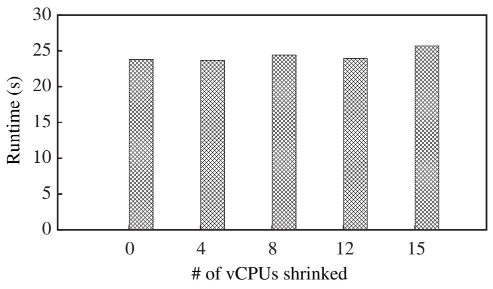

Figure 7 Performance of Sysbench CPU.

nonreplicated VM system, none of the above results will perform better than this benchmark. Therefore, a lower bound is formed.

Figure 5(a) illustrates the performance of memcached with 1 KB sized objects. Memcached is a key/value store service. Typically, user requests are put/update and get operations. When the update rate is zero, it means that all requests simply read the contents from the key/value store. Therefore, none of the operations will pollute the tracked memory in memcached. If all the tracked pages are in a clean set, this means they are already synchronized to the slave VM. It is safe to release the packets generated from the tracked memory, in advance. The zeroed update rate contributes to the prerelease of all response packets without comparison or checkpoints. Therefore, the performance of the zeroed update rate is close to native performance. As the update rates increase, the performance drops due to the overhead introduced from tracking dirty pages, i.e., page write protection faults and debug trap exceptions. COLO++_S exhibits the result of switching between the COLO++ and COLO system when the dirty output packets reach the threshold (50% by default). We can observe that the worst case performance in the COLO++ system will not exceed the bound of the COLO system.

As we can observe from Figures 5(b) and (c), the performance of COLO++ varies with different object sizes. When the object size is aligned to the page size (4 KB), there will be much less influence on the neighbors if a write operation is applied to an object. These three figures depict that COLO++ could still outperform COLO if the update rates reach 10%, 15%, and 21%, in the scenarios of 1 KB, 2 KB, and 4 KB sized objects, respectively.

To further understand the reason of COLO++ outperforming COLO, we evaluated the relationship between clean packet and data update rates. The results depict that the lower the update rate is, the cleaner the generated packets will be. As illustrated in Figure 6, the x-axis indicates the update rate and y-axis indicates the clean packet ratio. Additionally, it is demonstrated again that the object size will affect response performance.

5.3 Benefit of vCPU shrinking

In this subsection, we exhibit the benefit of vCPU shrinking that was used in the COLO++ system. Firstly, there are 2 parts of evaluations, and the results depict that the CPU performance of the COLO++ system will not be compromised while vCPU shrinking is enabled. Secondly, we exhibit that the vCPU shrinking mechanism improves the CPU performance of the colocated VMs in the same host.

We used sysbench [18] as the testing benchmark. The result is depicted in Figure 7, having a different number of vCPUs being shrunk, and the performance of the sysbench CPU is rarely changed.

In Subsection 5.2, several extra VMs were booted up to keep the overall number of vCPUs in the system at twice the number of physical CPUs to better demonstrate the benefit gained by the colocated VMs. Therefore, the average length of each CPU run queue will be 2, and the hypervisor will keep scheduling vCPUs to trigger the double scheduling problem.

We run the PARSEC benchmark suit [19] in a colocated VM equipped with 16 vCPUs to observe whether the vCPU shrinking mechanism of COLO++ will improve the performance of this VM. The

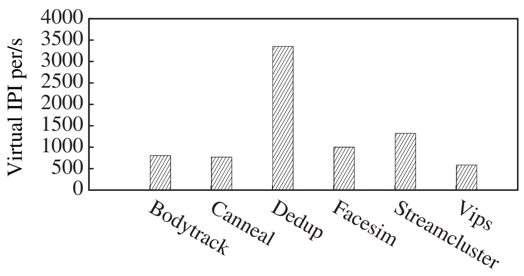

Figure 8 Average IPI per second happens in VM when running PARSEC benchmark suit.

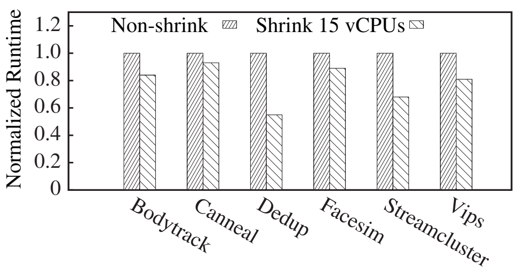

Figure 9 Performance of PARSEC benchmark suit.

PARSEC benchmark suit has many applications in different areas including financial analysis, computer vision, enterprise storage, and many other fields. Most of them are integrated with pthread to provide concurrency. Therefore mutual exclusion primitives, such as mutex and conditional variable, are largely used in order to guarantee parallelism safety. In Linux, most of these primitives will be translated into inter-processor interrupts (IPIs). In a virtualized environment, the virtual IPI could only be injected when the specific vCPU is scheduled. Therefore, the path of IPI sending and receiving mechanisms rely heavily on the effectiveness of the hypervisor scheduler. The larger is the number of vCPUs in the run queue, the longer the virtual IPI will have to wait. We selected several applications from the PARSEC benchmark suit, and it could be observed from Figure 8 that these applications have a significant amount of virtual IPIs sending and receiving operations that are performed during runtime. Since most of them perform a lot of memory operations during execution, it means that a large number of mutual exclusive primitives will be used. Therefore, it is natural to see why such many IPIs burst.

Figure 9 depicts the performance of a 16-vCPU colocated VM when running PARSEC applications. As we can observe, these applications benefit from the vCPU shrinking mechanism. Bodytrack, canneal, facesim, and vips gain approximately 10% to 20% in performance improvement. Additionally, the execution time of dedup is reduced by more than 45% and over 30% for streamcluster. If we observe the corresponding Figure 8, we could observe that vCPU shrinking eases the double scheduling problem of the hypervisor, while virtual IPI delays are mostly avoided.

6 Related work

The replication system can be implemented either through hardware-based replication or software-based replication. However, hardware-based replication has some disadvantages in comparison to the software-based method, since it is expensive and not general. HP company recommends the HP Non-Stop server, which requires additional external equipment and special OS design to operate the failover process [20–22]. Additionally, the instruction level lockstep replication at the hardware level forces primary and slave hosts to execute the exact same CPU instructions. Even though it obtains reasonably good performance [23,24], its industrial popularity is quite low due to its low level implementation. Moreover, hardware-based replication can only provide physical machine level fault tolerance instead of just one virtual machine running on it.

For software replication, virtual machine level replication is proposed. The fine-grained instruction lockstep VM replication is implemented using various software [5, 25–27] and suffers from significant overhead due to nondeterministic memory accesses in SMP virtual machines. Subsequently, Remus [6] and its variant [28] uses the VM migration technique [8] to perform periodical checkpoints and synchronize the VM states. Therefore, the overhead of the instruction lockstep replication could be avoided. Other optimizations are proposed to reduce the overhead of memory migrations over VMs [29, 30]. COLO [7] further optimizes Remus with regard to the overhead of frequent checkpointing through packet similarity comparison.

A multi-tier application could use multiple VMs to support the entire service. This typically consists of three tiers: the presentation layer (web tier), business logic layer (App tier) and data access layer (DB tier) [31]. When migrating a multitier application to other data centers, a problem of correlated VM migrations is caused [32]. However, we can easily add COLO++ fault tolerance support to such a VM cluster. As long as each VM follows the external consistency protocol, the overall consistency will not be compromised.

7 Conclusion

This study proposed an asymmetric VM replication solution for HA with an active primary VM and semiactive slave VM combination. COLO++ exploits the sources of network output packets and classifies them into clean and dirty categories. By considering the ingredients of packets, network roundtrip latency could be avoided, while maintaining external consistency. Moreover, COLO++ shrinks the number of vCPUs in the slave VM to relieve the stress of the hypervisor scheduler and ease the double scheduling problems.

References

1 Jiang B, Ravindran B, Kim C. Lightweight live migration for high availability cluster service. In: Proceedings of the 12th International Conference on Stabilization, Safety, and Security of Distributed Systems, New York, 2010. 420–434
2 Mullender S. Distributed systems. United States of America: ACM Press, 1993: 12
3 Kivity A, Kamay Y, Laor D, et al. Kvm: the Linux virtual machine monitor. In: Proceedings of the Linux Symposium, Ottawa, 2007. 1: 225–230
4 Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization. In: Proceedings of the ACM SIGOPS Operating Systems Review, New York, 2003. 164–177
5 Bressoud T C, Schneider F B. Hypervisor-based fault tolerance. ACM Trans Comput Syst, 1996, 14: 80–107
6 Cully B, Lefebvre G, Meyer D, et al. Remus: high availability via asynchronous virtual machine replication. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, San Francisco, 2008. 161–174
7 Dong Y Z, Ye W, Jiang Y H, et al. Colo: coarse-grained lock-stepping virtual machines for non-stop service. In: Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, 2013. 3
8 Clark C, Fraser K, Hand S, et al. Live migration of virtual machines. In: Proceedings of the 2nd Symposium on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2005. 2: 273–286
9 Elnozahy E N M, Alvisi L, Wang Y M, et al. A survey of rollback-recovery protocols in message-passing systems. ACM Comput Surv, 2002, 34: 375–408
10 Friebel T, Biemueller S. How to deal with lock holder preemption. In: Proceedings of Xen Summit North America, Boston, 2008. 164
11 Enck W, Gilbert P, Han S, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst, 2014, 32: 5
12 Song X, Shi J, Chen H, et al. Schedule processes, not VCPUs. In: Proceedings of the 4th Asia-Pacific Workshop on Systems, New York, 2013. 1
13 Cheng L, Rao J, Lau F. vScale: automatic and efficient processor scaling for SMP virtual machines. In: Proceedings of the 11th European Conference on Computer Systems, New York, 2016. 2
14 Russell R. Virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Oper Syst Rev, 2008, 42: 95–103
15 Intel®. Page modification logging for virtual machine monitor white paper. Intel Whitepaper, 2015. https://www.intel.com/content/www/us/en/processors/page-modification-logging-vmm-white-paper.html
16 Intel® 82576 and 82599 Gigabit Ethernet controller datashee. Intel Whitepaper, 2002. https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html
17 Fitzpatrick B. Distributed caching with memcached. Linux J, 2004, 2004: 5
18 Kopytov A. SysBench: a system performance benchmark. http://sysbench.sourceforge.net, 2004
19 Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, New York, 2008. 72–81
20 Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery. ACM Trans Comput Syst, 2002, 20: 398–461
21 Lamport L, Shostak R, Pease M. The Byzantine generals problem. ACM Trans Program Lang Syst, 1982, 4: 382–401
22 Schneider F B. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput Surv, 1990, 22: 299–319
23 Bernick D, Bruckert B, Vigna P D, et al. NonStop/spl reg/advanced architecture. In: Proceedings of the International Conference on Dependable Systems and Networks, Yokohama, 2005. 12–21

24 Webber S, Beirne J. The stratus architecture. In: Proceedings of the 21st International Symposium on Fault-Tolerant Computing, Montréal, 1991. 79–85

25 Jeffery C M, Figueiredo R J O. A flexible approach to improving system reliability with virtual lockstep. IEEE Trans Dependable Secure Comput, 2012, 9: 2–15

26 Scales D J, Nelson M, Venkitachalam G. The design of a practical system for fault-tolerant virtual machines. ACM SIGOPS Operat Syst Rev, 2010, 44: 30–39

27 Reiser H P, Kapitza R. Hypervisor-based efficient proactive recovery. In: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems. Washington: IEEE Computer Society, 2007. 83–92

28 Minhas U F, Rajagopalan S, Cully B, et al. RemusDB: transparent high availability for database systems. VLDB J, 2013, 22: 29–45

29 Lu M, Chiueh T. Fast memory state synchronization for virtualization-based fault tolerance. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Lisbon, 2009. 534–543

30 Zhu J, Dong W, Jiang Z F, et al. Improving the performance of hypervisor-based fault tolerance. In: Proceedings of the International Symposium on Parallel and Distributed Processing, Atlanta, 2010. 1–10

31 Huang D, He B, Miao C. A survey of resource management in multi-tier web applications. IEEE Commun Surv Tut, 2014, 16: 1574–1590

32 Liu H, He B. VMbuddies: coordinating live migration of multi-tier applications in cloud environments. IEEE Trans Parallel Distrib Syst, 2015, 26: 1192–1205