Resilire: Achieving High Availability at the Virtual Machine Level

Peng Lu

Preliminary Examination Proposal Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Robert P. Broadwater
Paul E. Plassmann
C. Jules White
Danfeng Yao

April 20, 2012
Blacksburg, Virginia

Keywords: Virtual Machine, High Availability, Live Migration, Checkpointing, Resumption

Resilire: Achieving High Availability at the Virtual Machine Level

Peng Lu

(ABSTRACT)

High availability is a critical feature of data centers, cloud, and cluster computing environments. Replication is a classical approach to increase service availability by providing redundancy. However, traditional replication methods are increasingly unattractive for deployment due to several limitations such as application-level non-transparency, non-isolation of applications (causing security vulnerabilities), complex system management, and high cost. Virtualization overcomes these limitations through another layer of abstraction, and provides high availability through live virtual machine (VM) migration: a guest VM image running on a primary host is transparently checkpointed and migrated, usually at a high frequency, to a backup host, without pausing the VM; the VM is resumed from the latest checkpoint on the backup when a failure occurs. A virtual cluster (VC) generalizes the VM concept for distributed applications and systems: a VC is a set of multiple VMs deployed on different physical machines connected by a virtual network.

In this dissertation, we first reduce live VM migration downtimes through a memory synchronization technique, called FGBI. FGBI reduces the dirty memory updates which must be migrated during each migration epoch by tracking memory at block granularity. Additionally, it determines memory blocks with identical content and shares them to reduce the increased memory overheads due to block-level tracking granularity, and uses a hybrid compression mechanism among dirty blocks to reduce the migration traffic. We implement FGBI in the Xen VM and compare it with two state-of-the-art VM migration solutions including LLM and Remus, on benchmarks including the Apache webserver and the SPEC benchmark suite. Our experimental results reveal that FGBI reduces the downtime by as much as $\approx$77% and $\approx$45% over LLM and Remus respectively, with a performance overhead of $\approx$13%.

We then present a lightweight, globally consistent checkpointing mechanism for VC, which checkpoints the VC for immediate restoration after (one or more) VM failures. VPC predicts the checkpoint-caused page faults during each checkpointing interval, in order to implement a lightweight checkpointing approach for the entire VC. Additionally, it uses a globally consistent checkpointing algorithm, which preserves the global consistency of the VMs' execution and communication states, and only saves the updated memory pages during each checkpointing interval. Our experimental results reveal that VPC reduces the solo VM downtime by as much as $\approx$45% and reduces the entire VC downtime by as much as $\approx$50% over competitors including VNsnap, with a memory overhead of $\approx$9% and performance overhead of $\approx$16%.

The dissertation's third contribution is a VM resumption mechanism, called VMresume, which restores a VM from a (potentially large) checkpoint on slow-access storage in a fast and efficient way. VMresume predicts and preloads the memory pages that are most likely to be accessed after the VM's resumption, minimizing otherwise potential performance degradation due to cascading page faults that may occur on VM resumption. Our Xen-based implementation and experimentation reveals that VM resumption time is reduced by an average of $\approx$ 57% and VM's unusable time is reduced by as much as $\approx$ 68% over native Xen's resumption mechanism.

Based on these results, we propose two major post-preliminary research directions: 1) live VM migration in virtualized systems without a hypervisor and 2) live VM migration in wide area networks (WANs). To eliminates security vulnerabilities for the entire system, we propose to remove

the hypervisor from the virtualized system, and provide a new VM migration mechanism in such non-hypervisor environment. Besides, to implement a efficient migration mechanism while reducing the downtime across the WAN, we propose to develop a novel record/replay mechanism, which reduces the network traffic due to transferring the dirty disk data from the source to the target machine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High availability is increasingly important in today's data centers, cloud, and cluster computing environments. A highly available service is one that is continuously operational for a long period of time [23]. Any downtime experienced by clients of the service may result in significant revenue loss and customer loyalty.

Replication is a classical approach for achieving high availability through redundancy: once a failure occurs to a service (e.g., failure of hosting node), a replica of the service takes over. Whole-system replication is one of the most traditional instantiations: once the primary machine fails, the running applications are taken over by a backup machine. However, whole-system replication is usually unattractive for deployment due to several limitations. For example, maintaining consistent states of the replicas may require application-level modifications (e.g., to orchestrate state updates), often using third-party software and sometimes specialized hardware, increasing costs. Further, since the applications directly run on the OS and therefore are not isolated from each other, it may cause security vulnerabilities, especially in cloud environments, where applications are almost always second/third-party and therefore untrusted. Additionally, such systems often require complex customized configurations, which increase maintenance costs.

Virtualization overcomes these limitations by introducing a layer of abstraction above the OS: the virtual machine (VM). Since applications now run on the guest VM, whole-system replication can be implemented easily and efficiently by simply saving a copy of the whole VM running on the system, which avoids any application modifications. Also, as a guest VM is totally hardware-independent, no additional hardware expenses are incurred. Due to the VM's ability to encapsulate the state of the running system, different types of OSes and multiple applications hosted on each of those OS can run concurrently on the same machine, which enables consolidation, reducing costs. Moreover, virtual machine monitors (VMMs) or hypervisors increase security: a VMM isolates concurrently running OSes and applications from each other. Therefor, malicious applications cannot impact other applications running on another OS, although they are all running on the same machine.

Besides these benefits, as a VM and its hosted applications are separated from the physical resources, another appealing feature of a VM is the flexibility to manage the workload in a dynamic way. If one physical machine is heavily loaded and a running VM suffers performance degradation due to resource competition, the VM can be easily migrated to another less loaded machine with available resources. During that migration, the applications on the source VM are still running, and if they are network applications, their clients therefore do not observe any disruption. This application- and client-transparent migration process is called "live migration" and is supported by most state-of-the-art/practice virtualization systems (e.g., Xen [9], VMware [50], KVM [4]). For small size systems, live migration can be done manually (e.g., by a system administrator). In a large scale system such as a cloud environment, it is done automatically [53].

1.1 VM Checkpointing

To provide benefits such as high availability and dynamic resource allocation, a useful feature of virtualization is the possibility of saving and restoring an entire VM through transparent checkpointing. Modern virtualization systems (e.g., Xen [9], VMware [50], KVM [4]) provide a basic checkpointing and resumption mechanism to save the running state of an active VM to a checkpoint file, and also, to resume a VM from the checkpoint file to the correct suspended state. Unlike application-level checkpointing [30, 38], VM-level checkpointing usually involves recording the virtual CPU's state, the current state of all emulated hardware devices, and the contents of the running VM's memory. VM-level checkpointing is typically a time consuming process due to potentially large VM memory sizes (sometimes, it is impractical as the memory size may be up to several gigabytes). Therefore, for solo VM checkpointing, often a lightweight methodology is adopted [20, 32, 40, 41, 57], which doesn't generate a large checkpoint file.

Downtime is the key factor for estimating the high availability of a system, since any long downtime experience for clients may result in loss of client loyalty and thus revenue loss. Figure 1.1 illustrates a basic fault-tolerant protocol design for VM checkpointing, showing the downtime problem under the classical primary-backup model. The VM is running on primary host and its memory state is check-pointed and transferred to the backup host. When the VM fails, the backup host will take over and roll-back each VM to its previous check-pointed state.

Under the Primary-Backup model, there are two types of downtime: I) the time from when the primary host crashes until the VM resumes from the last check-pointed state on the backup host and starts to handle client requests (shown as D1 in Figure 1.1); II) the time from when the VM pauses on the primary (to save for the checkpoint) until it resumes (shown as D2 in Figure 1.1).

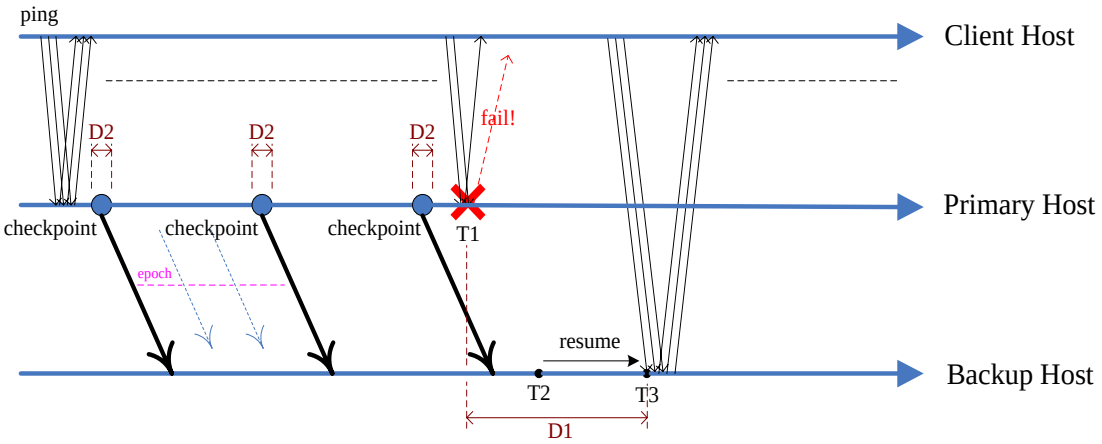


Figure 1.1: Primary-Backup model and the downtime problem (T_1: primary host crashes; T_2: client host observes the primary host crash; T_3: VM resumes on backup host; D_1 ($T_3 - T_1$): type I downtime; D_2: type II downtime).

1.2 Checkpointing Virtual Cluster

The checkpointing size also affects the scalability of providing high availability when checkpointing multiple VMs together. A virtual cluster (VC) generalizes the VM concept for distributed applications and systems. A VC is a set of multiple VMs deployed on different physical machines but managed as a single entity [55]. A VC can be created to provide computation, software, data access, or storage services to individual users, who do not require knowledge of the physical location or configuration of the system. End-users typically submit their applications, often distributed, to a VC, and the environment transparently hosts those applications on the underlying set of VMs. VCs are gaining increasing attraction in the "Platform as a Service" (PaaS; e.g., Google App Engine [3]) and "Infrastructure as a Service" (IaaS; e.g., Amazon's AWS/EC2 [1]) paradigms in the cloud computing domain.

In a VC, since multiple VMs are distributed as computing nodes across different machines, the failure of one VM can affect the states of other related VMs, and may sometimes cause them to also fail. For example, assume that we have two VMs, VM_a and VM_b, running in a VC. Say, VM_b sends some messages to VM_a and then fails. These messages may be correctly received by VM_a and may change the state of VM_a. Thus, when VM_b is rolled-back to its latest correct check-pointed state, VM_a must also be rolled-back to a check-pointed state before the messages were received from VM_b. In other words, all the VMs (i.e., the entire VC) must be check-pointed at globally consistent states.

The primary metric of high availability in VC is also *downtime*, as shown in Figure 1.2. Different from the type I and II downtimes defined in Section 1.1, when discussing the VM checkpointing. Two downtimes are of interest in the VC checkpointing: First is the VC downtime, which is the

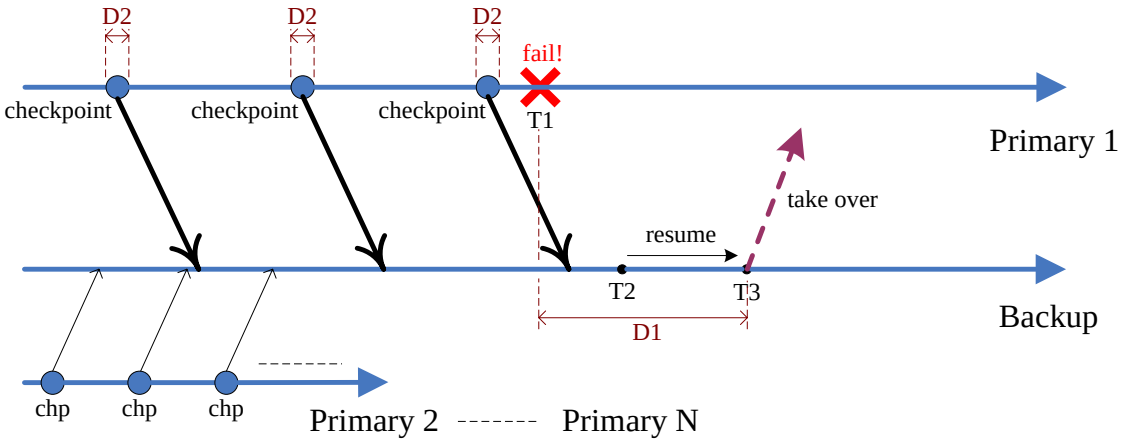


Figure 1.2: The downtime problem when checkpointing the VC. T_1: one of the primary VM fails; T_2: the failure is observed by the VC; T_3: VM resumes on backup machine; D_1 ($T_3 - T_1$): VC downtime; D_2: VM downtime.

time from when the failure was detected in the VC to when the VC (including all the VMs) resumes from the last check-pointed state on the backup machine. Second is the VM downtime, which is the time from when the VM pauses to save for the checkpoint to when the VM resumes. Obviously, saving a smaller checkpoint costs less time than saving a larger checkpoint. Thus, a lightweight checkpoint methodology directly reduces the VM downtime.

1.3 VM Resumption

From Figure 1.2, we also conclude that, for effective VM-level suspension and checkpointing, the hypervisor must be able to quickly resume the VM from a check-pointed state. Clients of network applications and other users are more inclined to suspend an idle VM if the latency magnitudes for resumption are in the order of seconds than minutes. The ability to quickly restore from a saved check-pointed image can also enable many other useful features, including fast relocation of VM, quick crash recovery, testing, and debugging.

Traditional VM resumption approaches can be classified in two categories. The first solution is to restore everything that is saved in the checkpoint, and then start VM execution. As the VM memory size dominates the checkpoint size, this solution works well for small memory sizes (e.g., MBs). However, VM resumption time significantly increases (e.g., 10s of seconds) when memory size becomes large (e.g., GBs). Figure 1.3a shows the time taken by native Xen's save and restore mechanisms as a function of memory size. We observe that Xen's save/restore times are in the order of multi-digit seconds when the memory size approaches 1GB.

In order to start the VM as quickly as possible, an alternate approach is to restore only the CPU

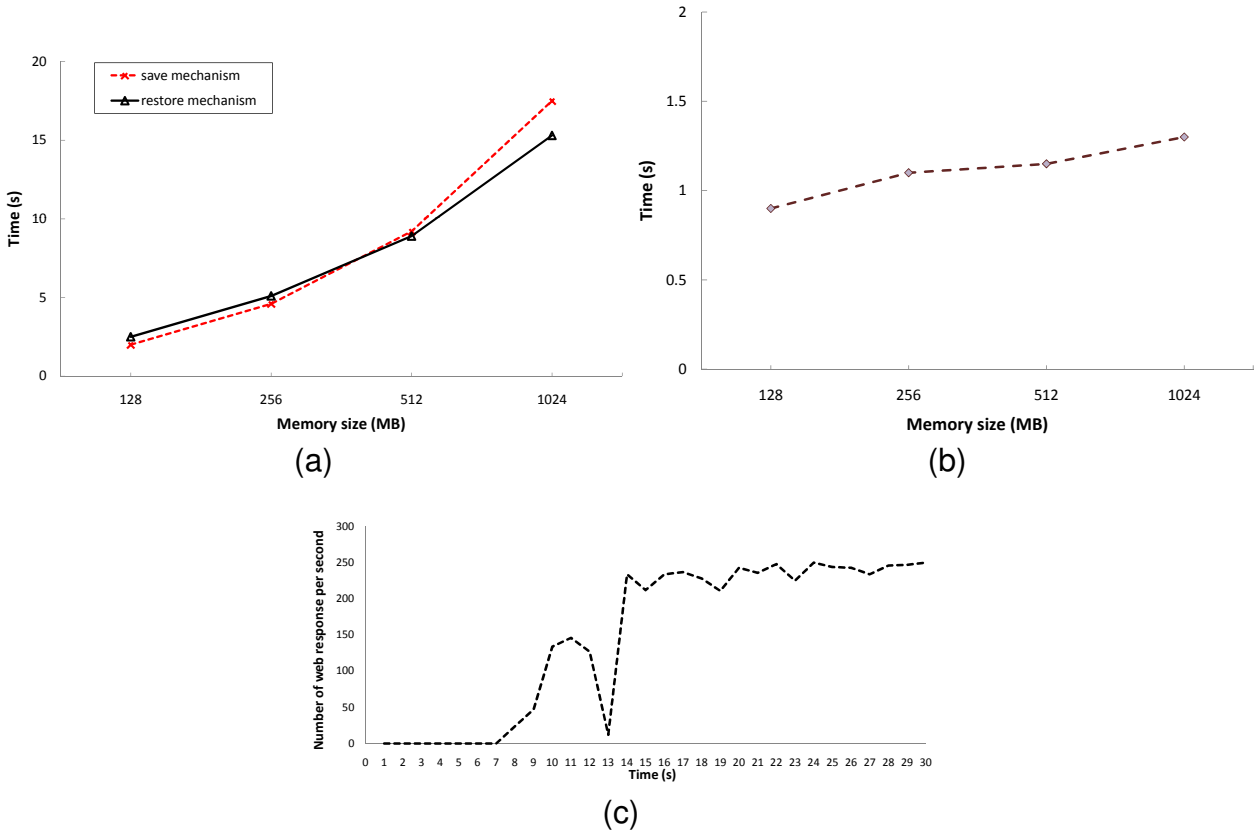


Figure 1.3: Several resumption mechanism comparison.

and device states that are necessary to boot a VM, and then restore the memory data saved in the checkpoint file after the VM starts. In this way, the VM can start very quickly. Figure 1.3b shows the VM resumption time under Xen when only the necessary CPU/device states are restored. We observe that, it takes ≈ 1.3 seconds to resume a VM with 1GB RAM.

Since the VM is still running when restoring the memory data, its performance would not be influenced by the VM memory size. However, with this approach, immediately after the VM starts, performance degrades due to cascading page faults, because there is no memory page loaded to use. Figure 1.3c shows the number of responses obtained per second for the Apache webserver running under Xen, after the VM has been restored using this approach. We observe that, the VM needs to wait for 14 seconds to resume normal activity."

Therefore, to further reduce the downtime, a check-pointed VM must be resumed quickly, while avoiding performance degradation after the VM starts.

1.4 Summary of Current Research and Contributions

The dissertation proposal makes the following research contributions:

We first focus on reducing VM migration downtimes through a memory synchronization technique, called FGBI. FGBI reduces the dirty memory updates which must be migrated during each migration epoch by tracking memory at block granularity. Additionally, it determines memory blocks with identical content and shares them to reduce the increased memory overheads due to block-level tracking granularity, and uses a hybrid compression mechanism among dirty blocks to reduce the migration traffic.

We implement FGBI in the Xen VM and compare it with two state-of-the-art VM migration techniques including LLM [23] and Remus [14], on benchmarks including the Apache webserver [2] and the applications from the NPB benchmark suite (e.g., EP program) [5]. The reason we choose these benchmarks is that Apache is a network intensive application, while NPB-EP is a memory intensive application. Our experimental results reveal that FGBI reduces the type I downtime by as much as 77% and 45%, over LLM and Remus, respectively, and reduces the type II downtime by more than 90% and 70%, compared with LLM and Remus, respectively. Moreover, in all cases, the performance overhead of FGBI is less than 13%.

We then present a lightweight, globally consistent checkpointing mechanism for VCs, which checkpoints the VC for immediate restoration after (one or more) VM failures. VPC predicts the checkpoint-caused page faults during each checkpointing interval, in order to implement a lightweight checkpointing approach for the entire VC. Additionally, it uses a globally consistent checkpointing algorithm, which preserves the global consistency of the VMs' execution and communication states, and only saves the updated memory pages during each checkpointing interval.

We construct a Xen-based implementation of VPC and conduct experimental studies. Our studies reveal that, compared with past VC checkpointing/migration solutions including VNsnap [25], VPC reduces the solo VM downtime by as much as ≈45%, under the NPB benchmark [5], and reduces the entire VC downtime by as much as ≈50%, when running the distributed programs (e.g., EP, IS) in NPB benchmark. VPC only incurs a memory overhead of no more than ≈9%. In all cases, VPC's performance overhead is less than ≈16%. Checkpointing overhead in 32, 64, and 128 VM environments was found to result in a speedup loss that is less than ≈14% under the NPB-EP benchmark.

The dissertation's third contribution is a VM resumption mechanism, called VMresume, that restores a VM from a (potentially large) checkpoint on slow-access storage in a fast and efficient way. VMresume predicts and preloads the memory pages that are most likely to be accessed after the VM's resumption, minimizing otherwise potential performance degradation due to cascading page faults that may occur on VM resumption. Our Xen-based implementation and experimentation reveals that VM resumption time is reduced by an average of $\approx 57\%$ and VM's unusable time is reduced by as much as $\approx 68\%$ over native Xen's resumption mechanism.

1.5 Summary of Proposed Post Preliminary-Exam Work

Building upon our current research results, we propose the following work:

Live VM migration without the hypervisor. Since the hypervisor manages all the hardware resources and isolates each VM from each other, it is a potential target for attacks, especially in shared infrastructures that allow untrusted parties to run VMs (e.g., cloud environments). One solution to this problem is to remove the hypervisor so as to eliminate the attack targeting the hypervisor. For example, [47] presents a non-hypervisor-based virtualization system. Critical services that are needed to enable guest VMs is then provided through a set of managers, including the core manager, and memory manager, etc. While the core manager provides the capability to start/stop and to suspend/resume the VM on each core, and the memory manager manages the physical/virtual memory for each guest OS. Live VM migration in no hypervisor-based virtualization environments, which will provide high availability with greater security, is an open problem. [1] This will be our first proposed research direction.

Since cores may be shared among different guest VMs, one responsibility of a traditional hypervisor is CPU scheduling. On the other hand, in non-hypervisor-based virtualized systems [47], only one VM runs on each CPU core, which eliminates the need for CPU scheduling. Since the core manager directly interacts with each core, via interrupts (e.g., inter-processor interrupt or IPI), to start/stop/suspend/resume the VM, the same mechanism can also be leveraged for live migration. Additionally, the memory manager can be leveraged to track which pages have been modified during each migration epoch. In this way, the memory differences between two continuous epoch could be recorded and transferred by periodically sending interrupts, which avoids having a hypervisor. Also, since each migration epoch is initiated by the core manager, no action performed by the guest VM requires any hypervisor management.

Live VM migration in WANs. In contrast to live migration in Local Area Networks (LANs), which has been the focus of our work so far, VM migration in Wide Area Networks (WANs) poses additional challenges. When migrating a VM in a LAN, the storage of both the source and target machines are usually shared by Network-attached storage (NAS) or Storage area network (SAN) media. Therefore, most data that needs to be transferred in LAN-based migration is derived from the run-time memory state of the VM. However, when migrating a VM in a WAN, besides the dirty memory, the I/O device state and the file system must also be migrated, because they are not shared in both source and target machines. File system sizes, in particular for I/O intensive applications, are usually large (e.g., in the order of several GBs). Hence, our current approach that only migrates memory data (which is usually in the order of MBs) may not scale.

A straightforward approach to do this, is to suspend the VM on the source machine, transfer its memory data and local disk data (in the image file) to the target machine, and resume the VM by reloading the memory and reconnecting to the file system, as done in the past [26, 45, 52]. How-

[1] In fact, [47] is the only non-hypervisor-based virtualization system that we are aware of. [47] propose a basic migration mechanism but does not support live VM migration.

ever, these stop/resume mechanisms suffer long downtime. In order to reduce the large amount of disk data, several optimization techniques have been introduced – e.g., data compression during migration [24]; content-based comparison among memory and disk data [44]. However, these optimizations introduce new overheads. Thus, our second proposed research direction is to develop scalable techniques for migrating VMs and their potentially large file systems in WANs with minimal downtime and acceptable overhead.

We propose to develop a record/replay mechanism for low downtime VM migration in WANs. At the beginning of VM migration, both source and target machines are initialized with identical VM set up. During the migration process, the running state of the source VM and its file system activities are kept synchronized with that of the target VM by recording the write activities on the (source) virtual file system on log files and replaying them on the (target) local file system. In contrast to past efforts, this approach therefore avoids transferring large amount of memory or disk data, except in the initial round.

1.6 Proposal outline

The rest of this dissertation proposal is organized as follows. Chapter 2 overviews the high availability at the VM level, and discuss past and related work on both VM and VC checkpointing mechanisms. Chapter 3 proposes our approach (FGBI) to achieve lightweight live migration for solo VM case, and presents our implementation and experimental evaluation compared with related efforts. Chapter 4 proposes an improved checkpointing mechanism (VPC) for the entire VC, presents the globally consistent checkpointing mechanism used in VPC, and summarize the evaluation results. Chapter 5 proposes a fast VM resumption mechanism (VMresume) based on a predictive checkpointing approach, and compare its performance with native Xen's resumption mechanism. We conclude the proposal in Chapter 6.

Chapter 2

Related Work

To achieve high availability, currently there exist many virtualization-based live migration techniques [22, 42, 56]. Two representatives are Xen live migration [12] and VMware VMotion [34], which share similar pre-copy strategies. During migration, physical memory pages are sent from the source (primary) host to the new destination (backup) host, while the VM continues running on the source host. Pages modified during the replication must be re-sent to ensure consistency. After a bounded iterative transferring phase, a very short stop-and-copy phase is executed, during which the VM is halted, the remaining memory pages are sent, and the destination VMM is signaled to resume the execution of the VM. However, these pre-copy methods incur significant VM downtimes, as the evaluation results in [17] show.

2.1 High Availability with VM

Remus [14] is now part of the official Xen repository. It achieves high availability by maintaining an up-to-date copy of a running VM on the backup host, which automatically activates if the primary host fails. Remus (and also LLM [23]) copies over dirty data after memory update, and uses the memory page as the granularity for copying. However, the dirty data tracking method is not efficient, as shown in [28] (we also illustrate this inefficiency in Section 3.1). Thus, our goal in this dissertation proposal is to further reduce the size of the memory transferred from the primary to the backup host, by introducing a fine-grained mechanism.

Lu *et al.* [28] used three memory state synchronization techniques to achieve high availability in systems such as Remus: dirty block tracking, speculative state transferring and active backup. The first technology is similar to our proposed method, however, it incurs additional memory associated overhead. For example, when running the Exchange workload in their evaluation, the memory overhead is more than 60%. Since main memory is always a scarce resource, the high percentage overhead is a problem. Different from these authors' work, we reduce memory overhead incurred by FGBI by integrating a new memory blocks sharing mechanism, and a hybrid compression

method when transferring the memory update.

To solve the memory overhead problems under Xen-based systems, there are several ways to harness memory redundancy in VMs, such as page sharing and patching. Past efforts showed the memory sharing potential in virtualization-based systems. Working set changes were examined in [12,49], and their results showed that changes in memory were crucial for the migration of VMs from host to host. For a guest VM with 512 MB memory assigned, low loads changed roughly 20 MB, medium loads changed roughly 80 MB, and high loads changed roughly 200 MB. Thus, normal workloads are likely to occur between these extremes. The evaluation in [12, 49] also revealed the amount of memory changes (within minutes) in VMs running different light workloads. None of them changed more than 4 MB of memory within two minutes. The Content-Based Page Sharing (CBPS) method [51] also illustrated the sharing potential in memory. CBPS was based on the compare-by-hash technique introduced in [18, 19]. As claimed, CBPS was able to identify as much as 42.9% of all pages as sharable, and reclaimed 32.9% of the pages from ten instances of Windows NT doing real-world workloads. Nine VMs running Redhat Linux were able to find 29.2% of sharable pages and reclaimed 18.7%. When reduced to five VMs, the numbers were 10.0% and 7.2%, respectively.

To share memory pages efficiently, recently, the Copy-on-Write (CoW) sharing mechanism was widely exploited in the Xen VMM [46]. Unlike the sharing of pages within an OS that uses CoW in a traditional way, in virtualization, pages are shared between multiple VMs. Instead of using CoW to share pages in memory, we use the same idea in a more fine-grained manner, i.e., by sharing among smaller blocks. The Difference Engine project demonstrates the potential memory savings available from leveraging a combination of page sharing, patching, and in-core memory compression [17]. It shows the huge potential of harnessing memory redundancy in VMs. However, Difference Engine also suffers from complexity problems when using the patching method. It needs additional modifications to Xen. We will present our corresponding mechanism and advantages over Difference Engine in Section 3.1.1.

Besides high availability systems such as Remus, LLM, and Kemari [48], which use the pre-copy mechanism, there are also other related works that focus on migration optimization. Post-copy based migration [21] is proposed to address the drawbacks of pre-copy based migration. The experimental evaluation in [21] shows that the migration time using the post-copy method is less than the pre-copy method, under SPECweb2005 and Linux Kernel Compile benchmarks. However, its implementation only supports PV guests as the mechanism for trapping memory accesses and utilizes an in-memory pseudo-paging device in the guest OS. Since the post-copy mechanism needs to modify the guest OS, it is not so much widely used as the pre-copy mechanism.

2.2 VM and VC Checkpointing Mechanisms

Checkpointing is a commonly used approach for achieving high availability. Checkpoints can be taken at different levels of abstraction. Application-level checkpointing is one of the most widely

used methods. For example, Lyubashevskiy *et al.* [30] develop a file operation wrapper layer with which a copy-on-write file replica is generated while keeping the old data unmodified. Pei *et al.* [38] wrap standard file I/O operations to buffer file changes between checkpoints. However, these checkpointing tools require modifying the application code, and thus, they are not transparent to applications.

OS-level checkpointing solutions have also been widely studied. For example, Libckpt [39] is an open-source, portable checkpointing tool for UNIX. It mainly focuses on performance optimization, and only supports single-threaded processes. Osman *et al.* [36] present Zap, which decouples protected processes from dependencies to the host operating system. A thin virtualization layer is inserted above the OS to support checkpointing without any application modification. However, these solutions are highly context-specific and often require access to the source code of the OS kernel, which increases OS-dependence. In contrast, VPC doesn't require any modification to the guest OS kernel or the application running on the VM.

Our work focuses on checkpointing at the VM-level. VM-level checkpointing can be broadly classified into two categories: stop-and-save checkpointing and checkpointing through live migration. In the first category, a VM is completely stopped, its state is saved in persistent storage, and then the VM is resumed [33]. This technique is easy to implement, but incurs a large system downtime during checkpointing. Live VM migration is designed to avoid such large downtimes — e.g., VMWare's VMotion [34], Xen Live Migration [12]. During migration, physical memory pages are sent from the source (primary) host to the destination (backup) host, while the VM continues to run on the source host. Pages modified during replication must be re-sent to ensure consistency. After a bounded iterative transferring phase, a very short stop-and-copy phase is executed, during which the VM is halted, the remaining memory pages are sent, and the destination hypervisor is signaled to resume the execution of the VM.

Disk-based VM checkpointing have also been studied. For example, efforts such as CEVM [11] create a VM replica on a remote node via live migration. These techniques employ copy-on-write to create a replica image of a VM with low downtime. The VM image is then written to disk in the background or by a separate physical node. However, disk-based VM checkpointing is often costly and unable to keep up with high frequency checkpointing (e.g., tens per second). Remus [14] uses high frequency checkpointing to handle hardware fail-stop failures on a single host with whole-system migration. It does this by maintaining a completely up-to-date copy of a running VM on the backup machine, which automatically activates if the primary machine fails. However, Remus incurs large overhead (overhead reported in [14] is approximately 50% for a checkpointing interval of $50ms$).

Multi-VM checkpointing mechanisms — our target problem space — have also been studied. The only efforts in this space that we are aware of include VirtCFT [55], VCCP [35], and VNsnap [25]. VirtCFT provides high availability for virtual clusters by checkpointing individual VMs to additional backup hosts. VirtCFT adopts a two-phase commit coordinated-blocking algorithm [10] (assuming FIFO communication channels) as the global checkpointing algorithm. A checkpoint coordinator broadcasts checkpointing requests to all VMs and waits for two-phase acknowledge-

ments. Since the checkpoint algorithm is FIFO-channel based, the network must be FIFO, which limits the scope of the work, or such channels must be emulated (e.g., using overlays), increasing overheads. Besides, as VirtCFT uses a checkpoint coordinator that communicates (several times) with each VM during a checkpoint period, the downtime is increased due to additional communication delays. VCCP also relies on reliable FIFO transmission to implement a blocking coordinated checkpointing algorithm. Due to its coordination algorithm, VCCP suffers from the overheads of capturing in-transit Ethernet frames and VM coordination before checkpointing.

Based on a nonblocking distributed snapshot algorithm, VNsnap [25] takes global snapshots of virtual networked environments and does not require reliable FIFO data transmission. VNsnap runs outside a virtual networked environment, and thus does not require any modifications to software running inside the VMs. The work presents two checkpointing daemons, called VNsnap-disk and VNsnap-memory. These solutions generate a large checkpoint size, which is at least the guest memory size. Also, VNsnap-memory stores the checkpoints in memory, which duplicates the memory, resulting in roughly 100% memory overhead. Additionally, their distributed snapshot algorithm (which is a variant of [31]) uses the "receive-but-drop" strategy, which would cause temporary backoff of active TCP connections inside the virtual network after checkpointing. The TCP backoff time is non-negligible and seriously affects the downtime. In Section 4.3.3, we experimentally compare VPC with VNsnap, and show VC downtime improvements by as much as 60%.

Chapter 3

Lightweight Live Migration for Solo VM

In this chapter, we first overview the integrated FGBI design, including some necessary preliminaries about the memory saving potential. We then present the FGBI architecture, explain each component, and discuss the execution flow and other implementation details. Finally we show our evaluation results by comparing with related work.

3.1 FGBI Design and Implementation

Remus and LLM track memory updates by keeping evidence of the dirty pages at each migration epoch. Remus uses the same page size as Xen (for x86, this is 4KB), which is also the granularity for detecting memory changes. However, this mechanism is not efficient. For instance, no matter what changes an application makes to a memory page, even just modify a boolean variable, the whole page will still be marked dirty. Thus, instead of one byte, the whole page needs to be transferred at the end of each epoch. Therefore, it is logical to consider tracking the memory update at a finer granularity, like dividing the memory into smaller blocks.

We propose the FGBI mechanism which uses memory blocks (smaller than page sizes) as the granularity for detecting memory changes. FGBI calculates the hash value for each memory block at the beginning of each migration epoch. Then it uses the same mechanism as Remus to detect dirty pages. However, at the end of each epoch, instead of transferring the whole dirty page, FGBI computes new hash values for each block and compares them with the corresponding old values. Blocks are only modified if their corresponding hash values do not match. Therefore, FGBI marks such blocks as dirty and replaces the old hash values with the new ones. Afterwards, FGBI only transfers dirty blocks to the backup host.

However, because of using block granularity, FGBI introduces new overhead. If we want to accurately approximate the true dirty region, we need to set the block size as small as possible. For example, to obtain the highest accuracy, the best block size is one bit. That is impractical, because

it requires storing an additional bit for each bit in memory, which means that we need to double the main memory. Thus, a smaller block size leads to a greater number of blocks and also requires more memory for storing the hash values. Based on these past efforts illustrating the memory saving potential (section 2.1), we present two supporting techniques: block sharing and hybrid compression. These are discussed in the subsections that follow.

3.1.1 Block Sharing and Hybrid Compression Support

From the memory saving results of related work (section 2.1), we observe that while running normal workloads on a guest VM, a large percentage of memory is usually not updated. For this static memory, there is a high probability that pages can be shared and compressed to reduce memory usage.

Block Sharing. Note that these past efforts [12, 18, 19, 49, 51] use the memory page as the sharing granularity. Thus, they still suffer from the "one byte differ, both pages cannot be shared" problem. Therefore, we consider using a smaller block in FGBI as the sharing granularity to reduce memory overhead.

The Difference Engine project [17] also illustrates the potential savings due to sub-page sharing, both within and across virtual machines, and achieves savings up to 77%. In order to share memory at the sub-page level, the authors construct patches to represent a page as the difference relative to a reference page. However, this patching method requires selected pages to be accessed infrequently, otherwise the overhead of compression/decompression outweighs the benefits. Their experimental evaluations reveal that patching incurs additional complexity and overhead when running memory-intensive workloads on guest VMs (from results for "Random Pages" workload in [17]).

Unlike Difference Engine, we use a straightforward sharing technique to reduce the complexity. The goal of our sharing mechanism is to eliminate redundant copies of identical blocks. We share blocks and compare hash values in memory at runtime, by using a hash function to index the contents of every block. If the hash value of a block is found more than once in an epoch, there is a good probability that the current block is identical to the block that gave the same hash value. To ensure that these blocks are identical, they are compared bit by bit. If the blocks are identical, they are reduced to one block. If, later on, the shared block is modified, we need to decide which of the original constituent blocks has been updated and will be transferred.

Hybrid Compression. Compression techniques can be used to significantly improve the performance of live migration [24]. Compressed dirty data takes shorter time to be transferred through the network. In addition, network traffic due to migration is significantly reduced when much less data is transferred between primary and backup hosts. Therefore, for dirty blocks in memory, we consider compressing them to reduce the amount of transferred data.

Before transmitting a dirty block, we check for its presence in an address-indexed cache of previously transmitted blocks (through pages). If there is a cache hit, the whole page (including this memory block) is XORed with the previous version, and the differences are run-length encoded

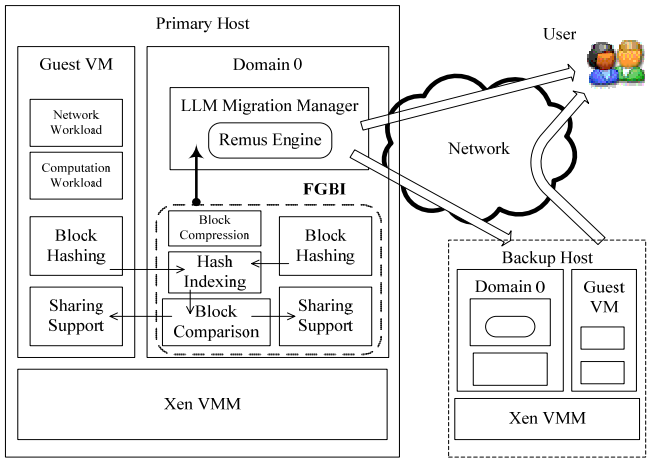


Figure 3.1: The FGBI architecture with sharing and compression support.

(RLE). At the end of each migration epoch, we send only the delta from a previous transmission of the same memory block, so as to reduce the amount of migration traffic in each epoch. Since smaller amount of data is transferred, the total migration time and downtime can both be decreased.

However, in the current migration epoch, there still may remain a significant fraction of blocks that is not present in the cache. In these cases, we find that Wilson *et. al.* [15] claims that there are a great number of zero bytes in the memory pages (so as in our smaller blocks). For this kind of block, we just scan the whole block and record the information about the offset and value of nonzero bytes. And for all other blocks with weak regularities, a universal algorithm with high compression ratio is appropriate. Here we use a general-purpose and very fast compression technique, zlib [7], to achieve a higher degree of compression.

3.1.2 Architecture

We implement the FGBI mechanism integrated with sharing and compression support, as shown in Figure 3.1. In addition to LLM, which is labeled as "LLM Migration Manager" in the figure, we add a new component, shown as "FGBI", and deploy it at both Domain 0 and guest VM.

For easiness in presentation, we divide FGBI into three main components:

1) Dirty Identification: It uses the hash function to compute the hash value for each block, and identify the new update through the hash comparison at the end of migration epoch. It has three subcomponents:

Block Hashing: It creates a hash value for each memory block;

Hash Indexing: It maintains a hash table based on the hash values generated by the Block Hashing component. The entry in the content index is the hash value that reflects the content of a given

block;

Block Comparison: It compares two blocks to check if they are bitwise identical.

2) Block Sharing Support: It handles sharing of bitwise identical blocks.

3) Block Compression: It compresses all the dirty blocks on the primary side, before transferring them to the backup host. On the backup side, after receiving the compressed blocks, it decompresses them first before using them to resume the VM.

Basically, the Block Hashing component produces hash values for all blocks and delivers them to the Hash Indexing component. The Hash Indexing and Block Comparison components then check the hash table to determine whether there are any duplicate blocks. If so, the Hash Comparison component requests the Block Sharing Support component to update the shared blocks information. At the end of each epoch, the Block Compression component compresses all the dirty blocks (including both shared and not shared).

In this architecture, the components are divided between the privileged VM Domain 0 and the guest VMs. The VMs contain the Block Sharing Support component. We house the Block Sharing Support component in the guest VMs to avoid the overhead of using shadow page tables (SPTs). Each VM also contains a Block Hashing component, which means that it has the responsibility of hashing its address space. The Dirty Identification component is placed in the trusted and privileged Domain 0. It receives hash values of the hashed blocks generated by the Block Hashing component in the different VMs.

3.1.3 FGBI Execution Flow

Figure 3.2 describes the execution flow of the FGBI mechanism. The numbers on the arrows in the figure correspond to numbers in the enumerated list below:

1) Hashing: At the beginning of each epoch, the Block Hashing components at the different guest VMs compute the hash value for each block.

2) Storing: FGBI stores and delivers the hash key of the hashed block to the Hash Indexing component.

3) Index Lookup: It checks the content index for identical keys, to determine whether the block has been seen before. The lookup can have two different outcomes:

Key not seen before: Add it to the index and proceed to step 6.

Key seen before: An opportunity to share, so request block comparison.

4) Block Comparison: Two blocks are shared if they are bitwise identical. Meanwhile, it notifies the Block Sharing Support Components on corresponding VMs that they have a block to be shared. If not, there is a hash collision, the blocks are not shared, and proceed to step 6.

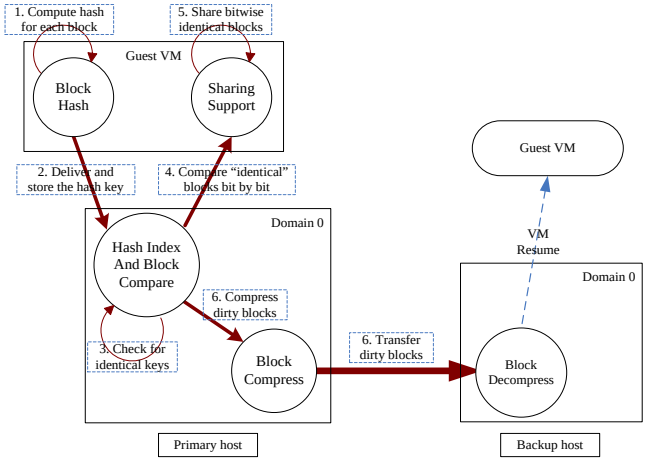


Figure 3.2: Execution flow of FGBI mechanism.

5) Shared Block Update: If two blocks are bitwise identical, then store the same hash value for both blocks. Unless there is a write update to this shared block, it doesn't need to be compared at the end of the epoch.

6) Block Compression: Before transferring, compress all the dirty blocks.

7) Transferring: At the end of epoch, there are three different outcomes:

Block is not shared: FGBI computes the hash value again and compares with the corresponding old value. If they don't match, mark this block as dirty, compress and send it to the backup host. Repeat step 1 (which means begin the next migration epoch).

Block is shared but no write update: It means that either block is modified during this epoch. Thus, there is no need to compute hash values again for this shared block, and therefore, there is no need to make comparison, compression, or transfer either. Repeat step 1.

Block is shared and write update occurs: This means that one or both blocks have been modified during this epoch. Thus, FGBI needs to check which one is modified, and then compress and send the dirty one or both to the backup host. Repeat step 1.

3.2 Evaluation and Results

We experimentally evaluated the performance of the proposed techniques (i.e., FGBI, sharing, and compression), which is simply referred to here as the FGBI mechanism. We measured downtime and overhead under FGBI, and compared the result with that under LLM and Remus.

3.2.1 Experimental Environment

Our experimental platform included two identical hosts (one as primary and the other as backup), each with an IA32 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz), and 3 GB RAM. We set up a 1 Gbps network connection between the two hosts, which is specifically used for migration. In addition, we used a separate machine as a network client to transmit service requests and examine the results based on the responses. We built Xen 3.4 from source [54], and let all the protected VMs run PV guests with Linux 2.6.18. The VMs were running CentOS Linux, with a minimum of services executing, e.g., sshd. We allocated 256 MB RAM for each guest VM, the file system of which is an image file of 3 GB shared by two machines using NFS. Domain 0 had a memory allocation of 1 GB, and the remaining memory was left free. The Remus patch we used was the nearest 0.9 version [13]. We compiled the LLM source code and installed its modules into Remus.

Our experiments used the following VM workloads under the Primary-Backup model:

Static web application: We used Apache 2.0.63 [2]. Both hosts were configured with 100 simultaneous connections, and repetitively downloaded a 256KB file from the web server. Thus, the network load will be high, but the system updates are not so significant.

Dynamic web application: SPECweb99 is a complex application-level benchmark for evaluating web servers and the systems that host them. This benchmark comprises a web server, serving a complex mix of static and dynamic page (e.g., CGI script) requests, among other features. Both hosts generate a load of 100 simultaneous connections to the web server [8].

Memory-intensive application: Since FGBI is proposed to solve the long downtime problem under LLM, especially when running heavy computational workloads on the guest VM, we continued our evaluation by comparing FGBI with LLM/Remus under a set of industry-standard workloads, specifically NPB and SPECsys.

1. *NPB-EP:* This benchmark is derived from CFD codes, and is a standard measurement procedure used for evaluating parallel programs. We selected the Kernel EP program from the NPB benchmark [5], because the scale of this program set is moderate and its memory access style is representative. Therefore, this example involves high computational workloads on the guest VM.

2. *SPECsys:* This benchmark measures NFS (version 3) file server throughput and response time for an increasing load of NFS operations (lookup, read, write, and so on) against the server over file sizes ranging from 1 KB to 1 MB. The page modification rate when running SPECsfs has previously been reported as approximately 10,000 dirty pages/second [8], which is approximately 40% of the link capacity on a 1 Gbps network.

To ensure that our experiments are statistically significant, each data point is averaged from twenty sample values. The standard deviation computed from the samples is less than 7.6% of the mean value.

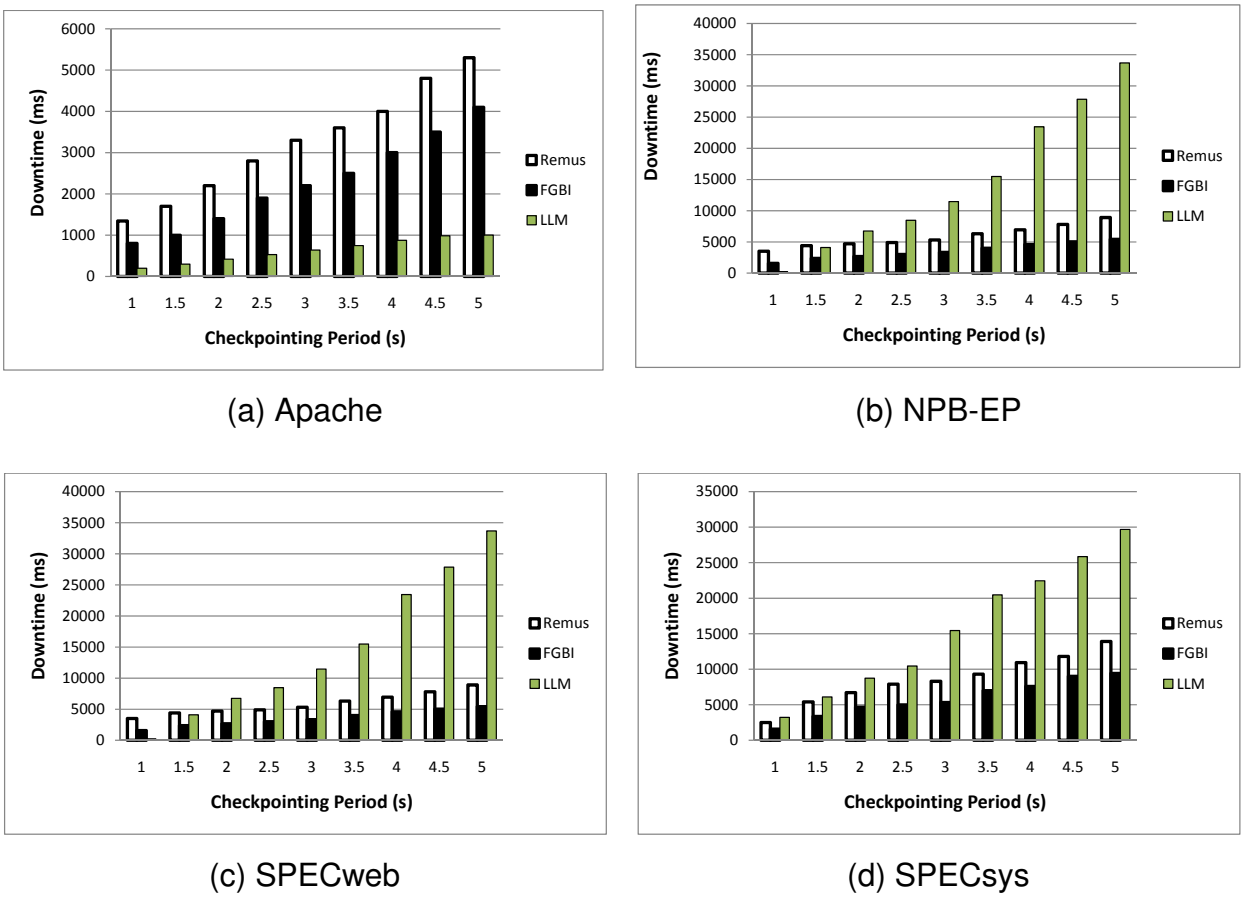


Figure 3.3: Type I downtime comparison under different benchmarks.

3.2.2 Downtime Evaluations

Type I Downtime.

Figures 3.3a, 3.3b, 3.3c, and 3.3d show the type I downtime comparison among FGBI, LLM, and Remus mechanisms under Apache, NPB-EP, SPECweb, and SPECsys applications, respectively. The block size used in all experiments is 64 bytes. For Remus and FGBI, the checkpointing period is the time interval of system update migration, whereas for LLM, the checkpointing period represents the interval of network buffer migration. By configuring the same value for the checkpointing frequency of Remus/FGBI and the network buffer frequency of LLM, we ensure the fairness of the comparison. We observe that Figures 3.3a and 3.3b show a reverse relationship between FGBI and LLM. Under Apache (Figure 3.3a), the network load is high but system updates are rare. Therefore, LLM performs better than FGBI, since it uses a much higher frequency to migrate the network service requests. On the other hand, when running memory-intensive applications (Figures 3.3b and 3.3d), which involve high computational loads, LLM endures a much longer downtime than

Application	Remus downtime	LLM downtime	FGBI downtime
idle	$64ms$	$69ms$	$66ms$
Apache	$1032ms$	$687ms$	$533ms$
NPB-EP	$1254ms$	$16683ms$	$314ms$

Table 3.1: Type II downtime comparison.

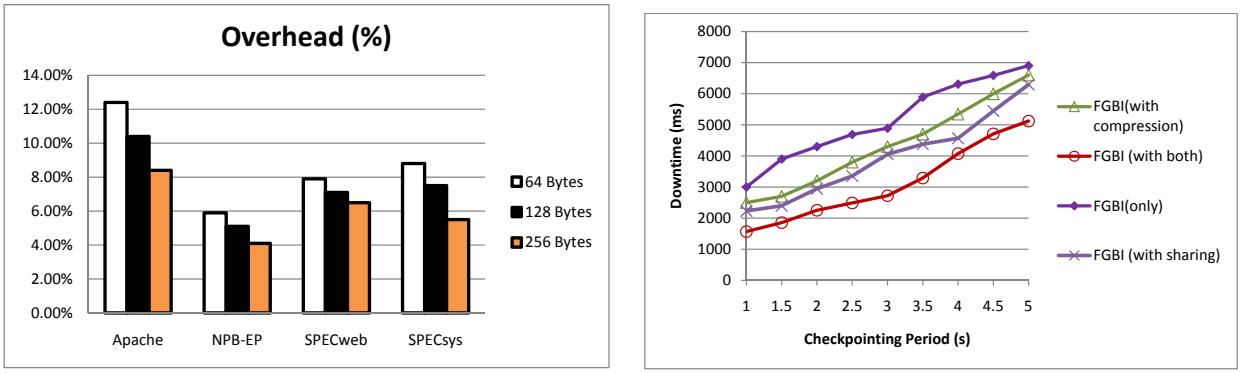
FGBI (even worse than Remus).

Although SPECweb is a web workload, it still has a high page modification rate, which is approximately 12,000 pages/second [12]. In our experiment, the 1 Gbps migration link is capable of transferring approximately 25,000 pages/second. Thus, SPECweb is not a lightweight computational workload for these migration mechanisms. As a result, the relationship between FGBI and LLM in Figure 3.3c is more similar to that in Figure 3.3b (and also Figure 3.3d), rather than Figure 3.3a. In conclusion, compared with LLM, FGBI reduces the downtime by as much as ≈77%. Moreover, compared with Remus, FGBI yields a shorter downtime, by as much as ≈31% under Apache, ≈45% under NPB-EP, ≈39% under SPECweb, and ≈35% under SPECsys.

Type II Downtime. Table 3.1 shows the type II downtime comparison among Remus, LLM, and FGBI mechanisms under different applications. We have three main observations: (1) Their downtime results are very similar for the idle run. This is because, Remus is a fast checkpointing mechanism and both LLM and FGBI are based on it. Memory updates are rare during the idle run, so the type II downtime in all three mechanisms is short. (2) When running the NPB-EP application, the guest VM memory is updated at a high frequency. When saving the checkpoint, LLM takes much more time to save huge dirty data caused by its low memory transfer frequency. Therefore, in this case FGBI achieves a much lower downtime than Remus (more than 70% reduction) and LLM (more than 90% reduction). (3) When running the Apache application, the memory update is not so much as that when running NPB, but memory update is more than that during the idle run. The downtime results shows that FGBI still outperforms both Remus and LLM.

3.2.3 Overhead Evaluations

Figure 3.4a shows the overhead during VM migration. The figure compares the applications' runtime with and without migration, under Apache, SPECweb, NPB-EP, and SPECsys, with the size of the fine-grained blocks varying from 64 bytes to 128 bytes and 256 bytes. We observe that in all cases the overhead is low, no more than 13% (Apache with 64 bytes block). As discussed in Section 3.1.1, the smaller the block size that FGBI chooses, greater is the memory overhead that it introduces. In our experiments, the smaller block size that we chose is 64 bytes, so this is the worst case overhead compared with the other block sizes. Even in this "worst" case, under all these benchmarks, the overhead is less than 8.21%, on average.



(a) Overhead under different block size. (b) Comparison of proposed techniques.

Figure 3.4: Overhead Measurements.

In order to understand the respective contributions of the three proposed techniques (i.e., FGBI, sharing, and compression), Figure 3.4b shows the breakdown of the performance improvement among them under the NPB-EP benchmark. It compares the downtime between integrated FGBI (which we use for evaluation in this Section), FGBI with sharing but no compression support, FGBI with compression but no sharing support, and FGBI without sharing nor compression support, under the NPB-EP benchmark. As previously discussed, since NPB-EP is a memory-intensive workload, it should present a clear difference among the three techniques, all of which focus on reducing the memory-related overhead. We do not include the downtime of LLM here, since for this compute-intensive benchmark, LLM incurs a very long downtime, which is more than 10 times the downtime that FGBI incurs.

We observe from Figure 3.4b that if we just use the FGBI mechanism without integrating sharing or compression support, the downtime is reduced, compared with that of Remus in Figure 3.3b, but it is not significant (reduction is no more than twenty percent). However, compared with FGBI with no support, after integrating hybrid compression, FGBI further reduces the downtime, by as much as 22%. We also obtain a similar benefit after adding the sharing support (downtime reduction is a further ≈26%). If we integrate both sharing and compression support, the downtime is reduced by as much as ≈33%, compared with FGBI without sharing or compression support.

3.3 Summary

One of the primary bottlenecks on achieving high availability in virtualized systems is downtime. We presented a novel fine-grained block identification mechanism, called FGBI, that reduces the downtime in lightweight migration systems. In addition, we developed a memory block sharing mechanism to reduce the memory and computational overheads due to FGBI. We also developed a dirty block compression support mechanism to reduce the network traffic at each migration epoch.

We implemented FGBI with the sharing and compression mechanisms and integrated them with the LLM lightweight migration system. Our experimental evaluations reveal that FGBI overcomes the downtime disadvantage of LLM by more than 90%, and of Xen/Remus by more than 70%. In all cases, the performance overhead of FGBI is less than 13%.

Chapter 4

Scalable, Low Downtime Checkpointing for Virtual Clusters

In this chapter, we first present the basic and improved checkpointing mechanisms used in our VPC design, explaining the reason why we implement a high frequency checkpointing mechanism. We then overview the consistency problem when designing a distributed checkpointing approach, and propose the globally consistent checkpointing algorithm used in the VPC design. Finally we show our evaluation results by comparing with previous VM and VC checkpointing mechanisms.

4.1 Design and Implementation of VPC

4.1.1 Lightweight Checkpointing Implementation

Since a VC may have hundreds of VMs, to implement a scalable lightweight checkpointing mechanism for the VC, we need to checkpoint/resume each VM with minimum possible overhead. To completely record the state of an individual VM, checkpointing typically involves recording the virtual CPU's state, the current state of all emulated hardware devices, and the contents of the guest VM's memory. Compared with other preserved states, the amount of the guest VM memory which needs to be check-pointed dominates the size of the checkpoint. However, with the rapid growth of memory in VMs (several gigabytes are not uncommon), the size of the checkpoint easily becomes a bottleneck. One solution to alleviate this problem is incremental checkpointing [32, 40, 41, 57], which minimizes the checkpointing overhead by only synchronizing the dirty pages during the latest checkpoint. A page fault-based mechanism is typically used to determine the dirty pages [20]. We first use incremental checkpointing in VPC.

We deploy a VPC agent that encapsulates our checkpointing mechanism on every machine. For each VM on a machine, in addition to the memory space assigned to its guest OS, we assign a

small amount of additional memory for the agent to use. During system initialization, we save the complete image of each VM's memory on the disk. To differentiate this state from "checkpoint," we call this state, "non-volatile copy". After the VMs start execution, the VPC agents begin saving the correct state for the VMs. For each VM, at the beginning of a checkpointing interval, all memory pages are set as read-only. Thus, if there is any write to a page, it will trigger a page fault. Since we leverage the shadow-paging feature of Xen, we are able to control whether a page is read-only and to trace whether a page is dirty. When there is a write to a read-only page, a page fault is triggered and reported to the Xen hypervisor, and we save the current state of this page.

When a page fault occurs, this memory page is set as writeable, but VPC doesn't save the modified page immediately, because there may be another new write to the same page in the same interval. Instead, VPC adds the address of the faulting page to the list of changed pages and removes the write protection from the page so that the application can proceed with the write. At the end of each checkpointing interval, the list of changed pages contains all the pages that were modified in the current checkpointing interval. VPC copies the final state of all modified pages to the agent's memory, and resets all pages to read-only again. A VM can then be paused momentarily to save the contents of the changed pages (which also contributes to the VM's downtime). In addition, we use a high frequency checkpointing mechanism (Section 4.1.2), which means that each checkpointing interval is set to be very small, and therefore, the number of updated pages in an interval is small as well. Thus, it is unnecessary to assign large memory to each VPC agent. (We discuss VPC's memory overhead in Section 4.3.4.)

Note that, this approach incurs a page fault whenever a read-only page is modified. When running memory-intensive workloads on the guest VM, handling so many page faults affects scalability. On the other hand, according to the principle of locality on memory accesses, recently updated pages tend to be updated again (i.e., spatial locality) in the near future (i.e., temporal locality). In VPC, we set the checkpointing interval to be small (tens to hundreds of milliseconds). So similarly, the dirty pages also follow this principle. Therefore, we use the updated pages in the previous checkpointing interval to predict the pages which will be updated in the upcoming checkpointing interval – i.e., by pre-marking the predicted pages as writable at the beginning of the next checkpointing interval. By this improved incremental checkpointing methodology, we reduce the number of page faults.

The page table entry (PTE) mechanism is supported by most current generation processors. For predicting the dirty pages, we leverage one control bit in the PTE: accessed (A) bit. The accessed bit is set to enable or disable write access for a page. Similar to our incremental checkpointing approach, for each VM, at the beginning of a checkpointing interval, all memory pages are set as read-only (accessed bit is cleared as "0"). Thus, if there is any write to a page, it will trigger a page fault, and the accessed bit is set to "1." However, unlike our incremental checkpointing approach, after the dirty pages in a checkpointing interval are saved in the checkpoint, we do not clear the accessed bits of these newly updated pages at the end of a checkpointing interval. Instead, the accessed bits of these pages are kept as writeable to allow write during the next interval. At the end of the next interval, we track whether these pages were actually updated or not. If they were not updated, their accessed bits are cleared, which means that the corresponding pages are set as read-only again. Our experimental evaluation shows that, this approach further reduces the (solo)

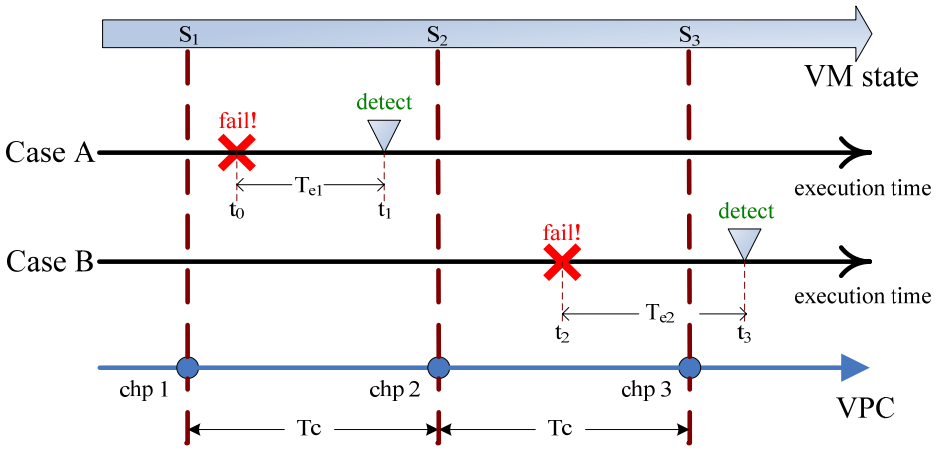


Figure 4.1: Two execution cases under VPC.

VM downtime (Section 4.3.2).

4.1.2 High Frequency Checkpointing Mechanism

VPC uses a high frequency checkpointing mechanism. Our motivation for this methodology is that, several previous fault injection experiments [27,37,43] have shown that most system crashes occur due to transient failures. For example, in the Linux kernel [16], after an error happens, around 95% of crashes occur within 100 million CPU cycles, which means that, for a 2 GHz processor, the error latency is very small (within $50ms$).

Suppose the error latency is T_e and the checkpointing interval is T_c. Thus, as long as $T_e \leq T_c$, the probability of an undetected error affecting the checkpoint is small. For example, if more than 95% of the error latency is less than T_e, the possibility of a system failure caused by an undetected error is less than 5%. Therefore, as long as T_c (application-defined) is no less than T_e (in this example, it is $50ms$), the checkpoint is rarely affected by an unnoticed error. Thus, this solution is nearly error-free by itself. On the other hand, if the error latency is small, so is the checkpointing interval that we choose. A smaller checkpointing interval means a high frequency methodology.

In VPC, for each VM, the state of its non-volatile copy is always one checkpointing interval behind the current VM's state except the initial state. This means that, when a new checkpoint is generated, it is not copied to the non-volatile copy immediately. Instead, the last checkpoint will be copied to the non-volatile copy. The reason is that, there is a latency between when an error occurs and when the failure caused by that error is detected.

For example, in Figure 4.1, an error happens at time t_0 and causes the system to fail at time t_1. Since most error latencies are small, in most cases, $t_1 - t_0 < T_e$. In case A, the latest checkpoint is $chp1$, and the system needs to roll-back to the state S_1 by resuming from the checkpoint $chp1$.

However, in case B, an error happens at time t_2, and then a new checkpoint $chp3$ is saved. After the system moves to the state S_3, this error causes a failure at time t_3. Here, we assume that $t_3 - t_2 < T_e$. But, if we choose $chp3$ as the latest correct checkpoint and roll the system back to the state S_3, after resumption, the system will fail again. We can see that, in this case, the latest checkpoint should be $chp2$, and when the system crashes, we should roll it back to the state S_2, by resuming from the checkpoint $chp2$.

VPC is a lightweight checkpointing mechanism, because, for each protected VM, the VPC agent stores only a small fraction of, rather than the entire VM image. For a guest OS occupying hundreds of megabytes of memory, the VPC checkpoint is no more than 20MB. In contrast, past efforts such as VNsnap [25] duplicates the guest VM memory and uses the entire additional memory as the checkpoint size. In VPC, with small amount of memory, we can store multiple checkpoints for different VMs running on the same machine. Meanwhile, as discussed in Section 1.2, the size of the checkpoint directly influences the VM downtime. This lightweight checkpointing methodology reduces VPC's downtime during the checkpointing interval. (We evaluate VPC's downtime in Section 4.3.2.)

4.2 Distributed Checkpoint Algorithm in VPC

4.2.1 Communication Consistency in VC

To compose a globally consistent state of all the VMs in the VC, the checkpoint of each VM must be coordinated. Besides checkpointing each VM's correct state, it is also essential to guarantee the consistency of all communication states within the virtual network. Recording the global state in a distributed system is non-trivial because there is no global memory or clock in a traditional distributed computing environment. So the coordination work must be done in the presence of non-synchronized clocks for a scalable design.

We illustrate the message communication with an example in Figure 4.2. The messages exchanged among the VMs are marked by arrows going from the sender to the receiver. The execution line of the VMs is separated by their corresponding checkpoints. The upper part of each checkpoint corresponds to the state before the checkpoint and the lower part of each checkpoint corresponds to the state after the checkpoint. A global checkpoint (consistent or not) is marked as the "cut" line, which separates each VM's timeline into two parts.

We can label the messages exchanged in the virtual network into three categories:

1) The state of the message's source and the destination are on the same side of the cut line. For example, in Figure 4.2, both the source state and the destination state of message m_1 are above the cut line. Similarly, both the source state and the destination state of message m_2 are under the cut line.

2) The message's source state is above the cut line while the destination state is under the cut line,

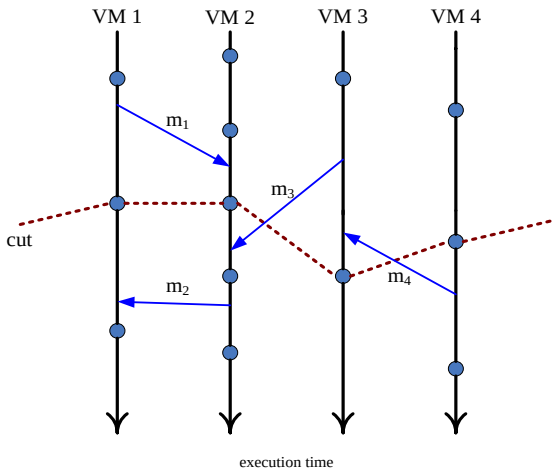


Figure 4.2: The definition of global checkpoint.

like message m_3.

3) The message's source state is under the cut line while the destination state is above the cut line, like message m_4.

For these three types of messages, we can see that a globally consistent cut must ensure the delivery of type (1) and type (2) messages, but must avoid type (3) messages. For example, consider the message m_4 in Figure 4.2. In VM3's checkpoint saved on the cut line, m_4 is already recorded as being received. However, in VM4's checkpoint saved on the same cut line, it has no record that m_4 has been sent out. Therefore, the state saved on VM4's global cut is inconsistent, because in VM4's view, VM3 receives a message m_4, which is sent by no one.

4.2.2 Globally Consistent Checkpointing Design in VPC

Several past approaches [35, 55] require FIFO channels to implement globally consistent checkpointing. There are several limitations in these approaches such as the high overheads of capturing in-transit Ethernet frames and VM coordination before checkpointing. Therefore, in VPC design, we modify a distributed checkpointing algorithm for non-FIFO channels [31]. For completeness, we summarize Mattern's algorithm here. This algorithm relies on vector clocks, and uses a single initiator process. At the beginning, a global snapshot is planned to be recorded at a future vector time s. The initiator broadcasts this time s and waits for acknowledgements from all the recipients. When a process receives the broadcast, it remembers the value s and acknowledges the initiator. After receiving all acknowledgements, the initiator increases its vector clock to s and broadcasts a dummy message. On the receiver's side, it takes a local snapshot, sends it to the initiator, and increases its clock to a value larger than s. Finally, the algorithm uses a termination detection scheme for non-FIFO channels to decide whether to terminate the algorithm.

We develop a variant of this classic algorithm, as the basis of our lightweight checkpointing mechanism. As illustrated before, type (3) messages are unwanted, because they are not recorded in any source VM's checkpoints, but they are already recorded in some checkpoints of a destination VM. In VPC, there is always a correct state for a VM, recorded as the non-volatile copy in the disk. As explained in Section 4.1.2, the state of the non-volatile copy is one checkpointing interval behind the current VM's state, because we copy the last checkpoint to the non-volatile copy only when we get a new checkpoint. Therefore, before a checkpoint is committed by saving to non-volatile copy, we buffer all the outgoing messages in the VM during the corresponding checkpointing interval. Thus, type (3) messages are never generated, because the buffered messages are unblocked only after saving their information by copying the checkpoint to the non-volatile copy. Our algorithm works under the assumption that the buffering messages will not be lost or duplicated. (This assumption can be overcome by leveraging classical ideas, e.g., as in TCP.)

In VPC, there are multiple VMs running on different machines connected within the network. One of the machines is chosen to deploy the VPC Initiator, while the protected VMs run on the primary machines. The Initiator can be running on a VM which is dedicated to the checkpointing service. It doesn't need to be deployed on the privileged guest system like the Domain 0 in Xen. When VPC starts to record the globally consistent checkpoint, the Initiator broadcasts the checkpointing request and waits for acknowledgements from all the recipients. Upon receiving a checkpointing request, each VM checks the latest recorded non-volatile copy (not the in-memory checkpoint), marks this non-volatile copy as part of the global checkpoint, and sends a "success" acknowledgement back to the Initiator. The algorithm terminates when the Initiator receives the acknowledgements from all the VMs. For example, if the Initiator sends a request (marked as rn) to checkpoint the entire VC, a VM named VM_1 in the VC will record a non-volatile copy named "vm1_global_rn". All of the non-volatile copies from every VM compose a globally consistent checkpoint for the entire VC. Besides, if the VPC Initiator sends the checkpointing request at a user-specified frequency, the correct state of the entire VC is recorded periodically.

4.3 Evaluation and Results

4.3.1 Experimental Environment

Our experimental testbed includes three multicore machines as the primary and backup machines (the experiment in Section 4.3.7 uses more machines as it deploys hundreds of VMs). Each machine has 24 AMD Opteron 6168 processors (1.86GHz), and each processor has 12 cores. The total assigned RAM for each machine is 11GB. We set up a 1Gbps network connection between the machines for experimental studies. We used two machines as the primary machines, and used the third as the backup machine. To evaluate the overhead and throughput of VPC (Sections 4.3.5 and 4.3.6), we set up the VC environment by creating 16 guest VMs (allocated 512MB RAM for each guest VM) on one primary machine, and 24 guest VMs (allocated 256MB RAM for each guest VM) on the other primary machine. We built Xen 3.4.0 on all machines and let all the guest

Application	VPC	VPC-np	VNsnap
idle	$55ms$	$57ms$	$53ms$
Apache	$187ms$	$224ms$	$267ms$
NPB-EP	$179ms$	$254ms$	$324ms$

Table 4.1: Solo VM downtime comparison.

VMs run PV guests with Linux 2.6.31. Each Domain 0 on the three machines has a 2GB memory allocation, and the remaining memory was left for the guest VMs to use. All the physical machines and the VMs were connected with each other based on the bridging mechanism of Xen.

We refer to our proposed checkpointing design with page fault prediction mechanism as VPC. In Sections 4.3.2, 4.3.4, 4.3.5, and 4.3.6, to evaluate the benefits of VPC, we compare VPC with our initial incremental checkpointing design without prediction, which we refer to as VPC-np.

Our competitors include different checkpointing mechanisms including Remus [14], LLM [23], and VNsnap [25]. Remus uses checkpointing to handle hardware fail-stop failures on a single host with whole-system migration. The Lightweight Live Migration or LLM technique improves Remus's overhead while providing comparable availability. While Remus and LLM implementations are publicly available, VNsnap is not. Thus, we implemented a prototype by using the distributed checkpointing algorithm in VNsnap and used that implementation in our experimental studies.

4.3.2 VM Downtime Evaluation

Recall that there are two types of downtime in the VC: VM downtime and the VC downtime. We first consider the case of solo VM to measure the VM downtime. The solo VM case is considered, as it is a special case of the virtual cluster case, and therefore gives us a baseline understanding of how our proposed techniques perform.

As defined in Section 1.2, the VM downtime is the time from when the VM pauses to save for the checkpoint to when the VM resumes. Table 4.1 shows the downtime results under VPC, VPC-np, and VNsnap daemon for three cases: i) when the VM is idle, ii) when the VM runs the NPB-EP benchmark program [5], and iii) when the VM runs the Apache web server workload [2]. The downtimes were measured for the same checkpointing interval, with the same VM (with 512MB of RAM) for all three mechanisms.

Several observations are in order regarding the downtime measurements. First, the downtime results of all three mechanisms are short and very similar for the idle case. This is not surprising, as memory updates are rare during idle runs, so the downtime of all mechanisms is short and similar.

Second, when running the NPB-EP program, VPC has much less downtime than the VNsnap

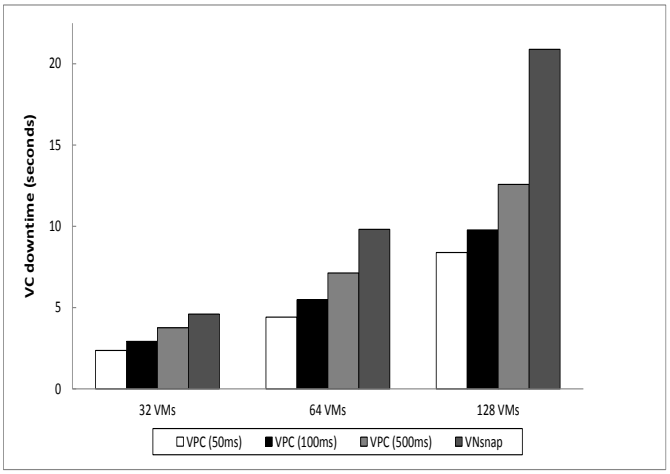


Figure 4.3: VC downtime under NPB-EP framework.

daemon (reduction is roughly 45%). This is because, NPB-EP is a computationally intensive workload. Thus, the guest VM memory is updated at high frequency. When saving the checkpoint, compared with other high-frequency checkpointing solutions, the VNsnap daemon takes more time to save larger dirty data due to its low memory transfer frequency.

Third, when running the Apache application, the memory update is not so much as that when running NPB. But the memory update is significantly more than that under the idle run. The results show that VPC has lower downtime than VNsnap daemon (downtime is reduced by roughly 30%).

Finally, compared with VPC-np, VPC also has less downtime when running NPB-EP and Apache (reduction is roughly 30% and 17%, respectively). As for both VPC-np and VPC, the downtime depends on the amount of checkpoint-induced page faults during the checkpointing interval. Since VPC-np uses an incremental checkpointing methodology, and VPC tries to reduce the checkpoint-induced page faults, VPC incurs smaller downtime than VPC-np.

4.3.3 VC Downtime Evaluation

As defined in Section 1.2, the VC downtime is the time from when the failure was detected in the VC to when the entire VC resumes from the last globally consistent checkpoint. We conducted experiments to measure the VC downtime under 32-node (VM), 64-node, and 128-node environments. We used the NPB-EP benchmark program [5] as the distributed workload on the protected VMs. NPB-EP is a compute-bound MPI benchmark with a few network communications.

To induce failures in the VC, we developed an application program that causes a segmentation failure after executing for a while. This program is launched on several VMs to generate a failure, while the distributed application workload is running in the VC. The protected VC is then rolled back to the last globally consistent checkpoint. A suite of experiments were conducted with VPC

deployed at 3 different checkpointing intervals ($500ms$, $100ms$, and $50ms$). We ran the same workloads on VNsnap daemon. We note that in a VC with hundreds of VMs, the total time for resuming all the VMs from a checkpoint may take up to several minutes (under both VPC and VNsnap). In the presentation, we only show the downtime results from when the failure was detected in the VC to when the globally consistent checkpoint is found and is ready to resume the entire VC.

Figure 4.3 shows the results. From the figure, we observe that, in the 32-node environment, the measured VC downtime under VPC ranges from 2.31 seconds to 3.88 seconds, with an average of 3.13 seconds; in the 64-node environment, the measured VC downtime under VPC ranges from 4.37 seconds to 7.22 seconds, with an average of 5.46 seconds; and in the 128-node environment, the measured VC downtime under VPC ranges from 8.12 seconds to 13.14 seconds, with an average of 11.58 seconds. The corresponding results from VNsnap are 4.7, 10.26, and 22.78 seconds, respectively. Thus, compared with VNsnap, VPC reduces the VC downtime by as much as 50%.

Another observation is that the VC downtime under VPC slightly increases as the checkpointing interval grows. Since we didn't consider the resumption time from the checkpoint, when VPC is deployed with different checkpointing intervals, the VC downtime is determined by the time to transfer all the solo checkpoints from primary machines to the backup machine. Therefore, a smaller checkpoint size incurs less transfer time, and thus less VC downtime. The checkpoint size depends on the number of memory pages restored. Therefore, as the checkpointing interval grows, the checkpoint size also grows, so does the number of restored pages during transfer.

4.3.4 Memory Overhead

In VPC, each checkpoint consists of only the pages which are updated within a checkpointing interval. We conducted a number of experiments to study the memory overhead of VPC's checkpointing algorithm at different checkpointing intervals. In each of these experiments, we ran four workloads from the SPEC CPU2006 benchmark [6], including:

1) perlbench, which is a scripting language (stripped-down version of Perl v5.8.7);

2) bzip2, which is a compression program (modified to do most work in memory, rather than doing I/O);

3) gcc, which is a compiler program (based on gcc version 3.2); and

4) xalancbmk, which is an XML processing program (a modified version of Xalan-C++, which transforms XML documents to other document types).

For both VPC-np and VPC designs, we measured the number of checkpoint-induced page faults (in terms of the number of memory pages) in every checkpointing interval (e.g., $T_c = 50ms$) of each experiment duration.

T_c	perlbench	bzip2	gcc	xalancbmk
$VPC - np : 50ms$	684	593	991	1780
$VPC - np : 100ms$	1389	1231	2090	2824
$VPC - np : 500ms$	5345	5523	5769	5428
$VPC : 50ms$	826	737	1041	2227
$VPC : 100ms$	1574	1411	2349	3572
$VPC : 500ms$	6274	5955	6813	7348

Table 4.2: VPC checkpoint size measurement (in number of memory pages) under SPEC CPU2006 benchmark.

Table 4.2 shows the results. We observe that for both designs, the average checkpoint sizes are very small: around 2.00% of the size of the entire system state when the checkpointing interval is $50ms$. For example, when VPC-np is deployed with a checkpointing interval of $50ms$, the average checkpoint size is 1012 memory pages or 3.9MB, while the size of the entire system state during the experiment is up to 65,536 memory pages (256MB). The maximum checkpoint size observed is less than 7MB (1780 pages when running the xalancbmk program), which is less than 3% of the entire system state size. When the checkpointing interval is increased to $100ms$, all checkpoints are less than 3,000 pages, the average size is 1883.5 pages (7.36MB, or 2.9% of the entire memory size), and the maximum checkpoint size is about 11MB (2824 pages when running the xalancbmk program). When the checkpointing interval is increased to $500ms$ (i.e., two checkpoints in a second), we observe that all the checkpoint sizes are around 5500 pages, the average size is 5516.25 pages (21.55MB, or 8.4% of the entire memory size), and the maximum checkpoint size is about 22.5MB (5769 pages when running the gcc program). Thus, we observe that, the memory overhead increases as the checkpointing interval grows. This is because, when the interval increases, more updated pages must be recorded during an interval, requiring more memory.

Another observation is that, the checkpoint size under VPC is larger than that under VPC-np. This is because, under VPC, as we improve VPC-np with page faults prediction, it generates more "fake" dirty pages in each checkpointing interval. For example, we pre-make the updated pages in the first checkpointing interval as writable at the beginning of the second checkpointing interval. Thus, at the end of the second checkpointing interval, there are some fake dirty pages, which are actually not updated in the second checkpointing interval. Therefore, besides the dirty pages which are actually updated in the second interval, the checkpoint recorded after the second checkpointing interval also includes these pages which are not updated in the second interval but still set as dirty because of the prediction mechanism. Note that although VPC generates a slightly larger checkpoint, it incurs smaller downtime than VPC-np (Section 4.3.2).

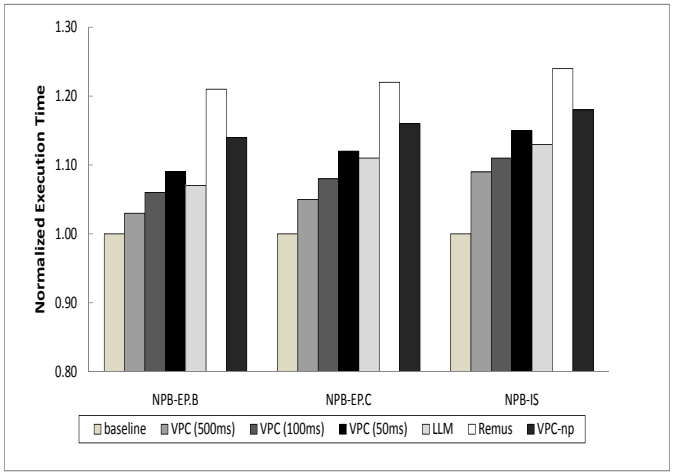


Figure 4.4: Performance overhead under NPB benchmark.

4.3.5 Performance Overhead

We measured the performance overhead introduced into the VC by deploying VPC. We chose two distributed programs in the NPB benchmark [5], and ran them on 40 guest VMs with VPC deployed. NPB contains a combination of computational kernels. For our experimental study, we chose to use the EP and IS programs. EP is a compute-bound MPI benchmark with a few network communications, while IS is an I/O-bound MPI benchmark with significant network communications.

A suite of experiments were conducted under the following cases: (1) a baseline case (no checkpoint), (2) VPC deployed with 3 different checkpointing intervals ($500ms$, $100ms$, and $50ms$), (3) Remus [14] deployed with one checkpointing interval ($50ms$), (4) LLM [23] deployed with one checkpointing interval ($50ms$), and (4) VPC-np deployed with one checkpointing interval ($50ms$).

A given program executes with the same input across all experiments. To test the performance accurately, we repeated the experiment with each benchmark five times. The execution times were measured and normalized, and are shown in Figure 4.4. The normalized execution time is computed by dividing the program execution time with the execution time for the corresponding baseline case.

We first measured the benchmarks' runtime when they were executed on the VMs without any checkpointing functionality. After this, we started the VPC agents for each protected VM, and measured the runtime with different checkpointing intervals. We chose EP Class B program in NPB benchmark and recorded its runtime under different situations. Besides, we also chose EP Class C in NPB benchmark (the Class C problems are several times larger than the Class B problems) to see how VPC performs if we enlarge the problem size. Finally, we tested some extreme cases with I/O intensive applications. We chose the IS program in NPB benchmark, a NPB benchmark with significant network communications, and excluding floating point computations. We

ran the same benchmarks on Remus, LLM, and VPC-np, following the same manner as on VPC.

Figure 4.4 shows the results. We observe that, for all programs running under VPC, the impact of the checkpoint on the program execution time is no more than 16% (the normalized execution times are no more than 1.16), and the average overhead is 12% (the average of the normalized execution times is 1.12), when VPC is deployed with $50ms$ checkpointing interval. When we increase the checkpointing interval to $100ms$, the average overhead becomes 8.8%, and when we increase the checkpointing interval to $500ms$, the average overhead becomes 5.4%. Thus, we observe that the performance overhead decreases as the checkpointing interval grows. Therefore, there exists a trade-off when choosing the checkpointing interval. In VPC, checkpointing with a larger interval incurs smaller overhead, while causing a longer output delay and a larger checkpoint size. (This also means larger memory overhead, confirming our observation in Section 4.3.4.)

Another observation is that VPC incurs lower performance overhead compared with other high-frequency checkpointing mechanisms (Remus and VPC-np). The reason is that memory access locality plays a significant role when the checkpointing interval is short, and VPC precisely reduces the number of page faults in this case. LLM also reduces the performance overhead, but it is a checkpointing mechanism only for solo VM. In our experiment, we deployed 40 VMs in the VC, and from Figure 4.4, we observe that the overhead of VPC is still comparable to that in the solo VM case under LLM.

4.3.6 Web Server Throughput

We conducted experiments to study how VPC affects Apache web server throughput when the web server runs on the protected guest system. We configured the VC with 40 guest VMs, and let all Apache web clients reside on the two multicore machines and each protected VM to host one client. The clients request the same load of web pages from the server, one request immediately after another, simultaneously via a 1Gbps LAN.

We measured the web server throughput with VPC deployed at different checkpointing intervals ($500ms$, $100ms$, and $50ms$). As the same load of web requests are processed in these experiments, the measured throughput can be compared for evaluating the impact of VPC's checkpoint on the throughput. To enable a comparison, we conducted the same experiment under Remus, LLM, and VPC-np. Figure 4.5 shows the measured server throughput as a function of checkpointing intervals. The percentages indicated along the data points on the graph represent the ratio of the throughput measured with the checkpoint deployed to the throughput in the baseline case (when no checkpointing mechanism is deployed).

From Figure 4.5, we observe that the throughput is reduced by 19.5% when a checkpoint is taken 20 times per second (so the interval equals $50ms$). When the checkpointing interval is increased to $100ms$, the throughput is reduced by 13.8%. And, when the checkpointing interval is increased to $500ms$, the throughput is reduced by only 7.4%. Therefore, we observe that the server throughput (performance overhead) increases with higher checkpoint frequency, which also confirms our

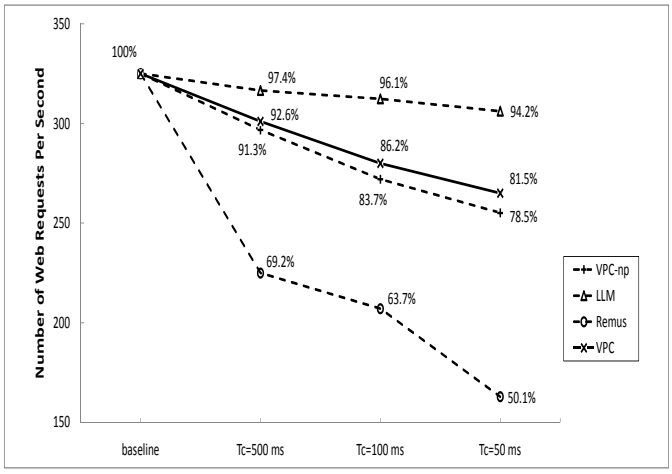


Figure 4.5: Impact of VPC on Apache web server throughput.

observation in Section 4.3.5.

Another observation is that under the same checkpointing interval, VPC achieves higher throughput than Remus. In the worst case ($50ms$ as the checkpointing interval), Remus's overhead is approximately 50%, while that of VPC is approximately 20% of the throughput. VPC also has performance improvements over VPC-np, especially in cases with smaller checkpointing intervals (the gap between the two curves keeps increasing and is much larger for intervals of $100ms$ and $50ms$ in Figure 4.5). The best results are for LLM, whose overhead is roughly 10% of the throughput in all cases because it handles the service requests from clients at high frequency, and as we explained in Section 4.3.5, LLM only targets the solo VM case. Since there are multiple VMs (40 nodes) running under VPC, VPC's throughput reduction is acceptable.

4.3.7 Checkpointing Overhead with Hundreds of VMs

We also conducted experiments to measure the total execution time when running the NPB-IS and NPB-EP (class B and C) distributed applications under 32-node (VM), 64-node, and 128-node environments. These results help provide insights into the scalability of VPC. Figure 4.6 depicts the speedup of the execution time on 64 and 128 nodes with respect to that on 32 nodes. The figure also shows the relative speedup observed with and without checkpointing. The lightly colored regions of the bars represent the normalized execution time of the benchmarks with checkpointing. The aggregate value of the light and the dark-colored portions of the bars represent the execution time normalized to the equivalent runtime without checkpointing. Hence, the dark-colored regions of the bars represent the loss in speedup due to checkpointing/restart. From the figure, we observe that, under all benchmarks, the speedup with checkpointing is close to that achieved without checkpointing. The worst case happens under the NPB-EP (class B) benchmark with 128 VMs deployed, but still, the speedup loss is less than 14%.

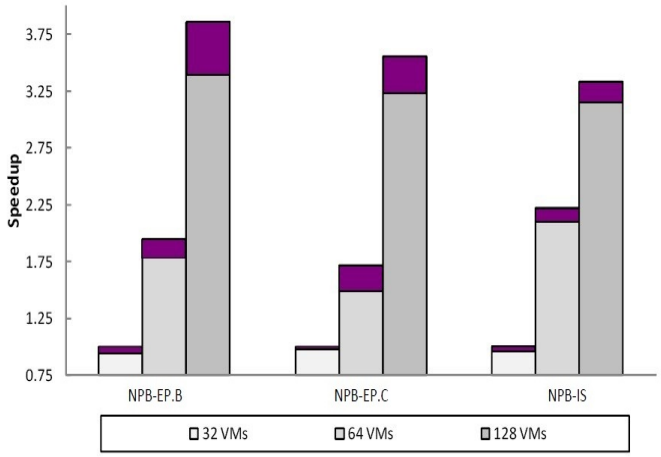


Figure 4.6: Checkpointing overhead under NPB-EP with 32, 64, and 128 VMs.

4.4 Summary

We present VPC, a lightweight, globally consistent checkpointing mechanism that records the correct state of an entire VC, which consists of multiple VMs connected by a virtual network. To reduce both the downtime incurred by VPC and the checkpoint size, we develop a lightweight in-memory checkpointing mechanism. By recording only the updated memory pages during each checkpointing interval, VPC reduces the downtime with acceptable memory overhead. By predicting the checkpoint-caused page faults during each checkpointing interval, VPC further reduces the downtime. In addition, VPC uses a globally consistent checkpointing algorithm (a variant of Matten's snapshot algorithm), which preserves the global consistency of the VMs' execution and communication states. Our implementation and experimental evaluations reveal that, compared with past VC checkpointing/migration solutions including VNsnap, VPC reduces the solo VM downtime by as much as $\approx$45% and reduces the entire VC downtime by as much as $\approx$50%, with a performance overhead that is less than $\approx$16%.

Chapter 5

Fast Virtual Machine Resumption with Predictive Checkpointing

In this chapter, we first overview the Xen's memory model and its basic save/restore mechanism, including some necessary preliminaries about the potential improvement. We then present the VMresume mechanism, explain the design of the predictive checkpointing approach and other implementation details. Finally we show the improvement of the resumption time and performance overhead in VMresume over native Xen's save/restore mechanism.

5.1 VMresume: Design and Implementation

5.1.1 Memory Model in Xen

Since VMresume is based on Xen's memory virtualization technique (in particular Xen's SPT mode), we first overview that for completeness.

In a system without virtualization, there are two types of addresses recognized in an OS: virtual address and physical address. A virtual address is the reference of a memory object in a process address space; a physical address provides the information of where the memory object is actually located in the physical memory. The processor translates a virtual address to the corresponding physical address, and accesses the correct object in physical memory via a memory management unit (MMU) in the processor. The MMU consults a page table, maintained by the OS, for address translation.

With Xen virtualization, there are three types of addresses: machine address, physical address, and virtual address. Address translation under Xen involves two levels of mapping: mapping between machine and physical address, and mapping between physical and virtual address, as shown in Figure 5.1. For the first level mapping, since several VMs maybe running on the machine and Xen

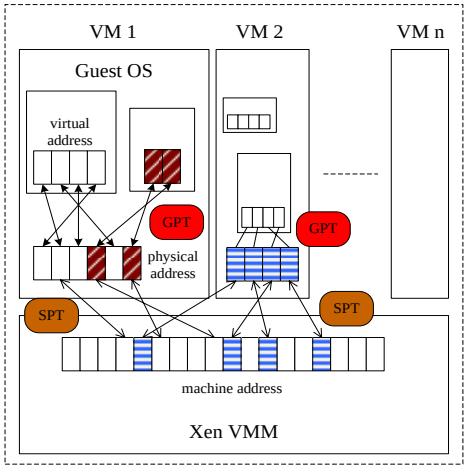


Figure 5.1: Memory model in Xen.

ensures isolation, each VM needs a continuous view of memory address from 0 to n, recognized as the physical address. Each VM uses a global table to map the physical address to the machine address. Outside all the VMs, the VMM manages the available memory pages (recognized as machine address) on the physical machine. VMM allocates these machine memory addresses, and maps them to a VM's physical addresses, according to the VM's need for memory. Similar to traditional OS address translation, contiguous physical address space in Xen may map to non-contiguous machine address space.

In addition, there is the second level mapping inside each VM, which relies on guest OS's page tables to translate between physical address and virtual address. However, in x86 architectures, MMU only supports one-level mapping. Thus, additional software implementation is needed to attain the two-level mapping. There are several solutions to accomplish memory virtualization, such as Shadow Page Mode, Direct Paging Access Mode, etc. However, some solutions such as Direct Paging Access require modifying the guest OS kernel. To achieve full virtualization (which means that no modifications to the guest OS kernel are needed), Xen uses the Shadow Page Table (SPT) mechanism.

With the SPT mechanism, a separate page table is created for the VM, and for each VM, the guest OS also maintains its own page table, called the guest page table (or GPT). The GPT is not used directly by the VMM or the hardware. On the other hand, a guest OS does not have access to the SPT either. The GPT in the guest OS does not directly map the virtual address to the machine address, but to the physical address instead. VMM synchronizes the SPT with the GPT so that the two page tables are consistent at all times. Therefore, any virtual address of a process (here it's the virtual address) in the guest system has the correct physical address in the GPT and the correct machine address in the SPT with the same information in the two tables.

The SPT allows the VMM to track all writes performed to the pages in the GPT. When a process in the guest OS wants to update the GPT, it makes a hypercall (mmu_update), which transfers

VM memory assigned (MB)	Checkpoint size (MB)
128	128.7
256	258.2
512	515.9
1024	1031.3

Table 5.1: Checkpoint sizes for different memory sizes.

control to the VMM. The VMM then checks that the update does not violate the isolation of the guest VM. If it violates the memory constraints, the guest OS is denied write access to the page table. If it does not violate any constraint, it is allowed to complete the write operation and update the GPT. Meanwhile, the VMM synchronizes the SPT entries accordingly. Thus, there is a performance penalty caused by the synchronization, which keeps the SPT and the GPT up-to-date. Moreover, extra memory is needed to save the SPT. However, compared with other solutions, the SPT mechanism is one of the most effective solutions for the two-level address mapping problem. Xen implements the SPT in a transparent fashion. Therefore, the guest OS is unaware of the SPT's existence and supports full virtualization.

5.1.2 Checkpointing Mechanism

VMresume is based on the Xen VMM [9]. To provide a basic checkpointing mechanism, Xen relies on its hypervisor to implement two commands, `xm save` and `xm restore`. `xm save` stores the current state of the guest VM in an on-disk file (checkpoint), from which the VM can be resumed on the same machine or on another machine (after sharing or transferring the on-disk file). `xm restore` resumes the checkpoint on its current machine and restarts the VM based on the state when it was saved. Besides as a checkpointing mechanism for high availability, the save and restore commands are also widely used for other purposes such as VM relocation and live migration [12].

The performance of both functions is directly related to user experience, since a long wait time is not acceptable for users after issuing commands. Therefore, it is important to save and restore a VM in a quick and efficient way. To evaluate native Xen's saving and restoring mechanism, we conducted an experiment to measure the time for saving and restoring a VM, and also the final size of its checkpoint. We built Xen 3.4.0 on the host machine and let the guest VM run PV guests with Linux 2.6.31. The Domain 0 on the machine was allocated 2GB, and the guest VM was allocated with different memory sizes (from 128MB to 1GB). The results are shown in Table 5.1 and Figure 5.2.

From Table 5.1, we observe that the checkpoint size almost always equals the memory size assigned to the VM. Besides, from Figure 5.2, we observe that, as the VM memory size increases, the time taken by the `xm save` command increases linearly. This is because, to completely save the state of an individual VM, the virtual CPU's state, the current state of all emulated hard-

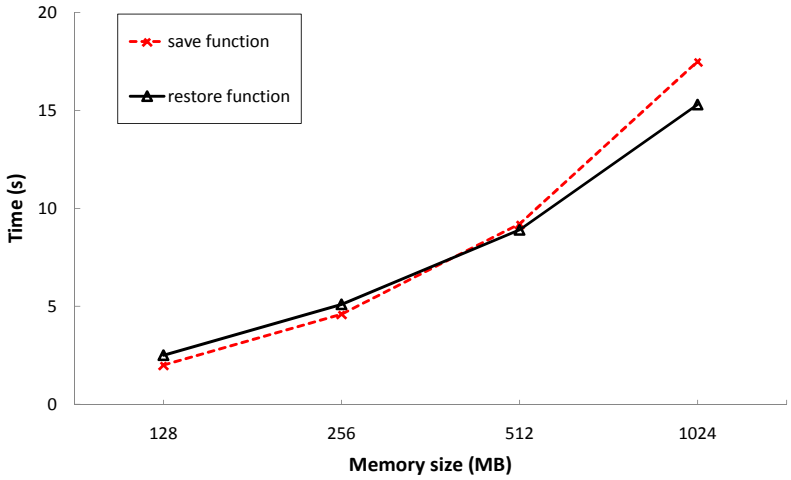


Figure 5.2: Native Xen's saving and restoring times.

ware devices, and the contents of the VM memory must be recorded and saved in the checkpoint file. Compared with other CPU/device states, the VM memory which needs to be check-pointed dominates the checkpoint size, and thus the time spent on saving memory. Therefore, as the VM memory size increases, both the checkpoint size (Table 3.1) and the save time (Figure 5.2) increase linearly.

However, with the rapid growth of memory assigned to a VM (several gigabytes are not uncommon now), the checkpoint size (relative to the saving time) easily becomes a bottleneck. One solution to this problem is incremental checkpointing [32, 40, 41, 57], which minimizes the checkpointing overhead by only synchronizing the dirty pages during the latest checkpointing interval. Note that, incremental checkpointing is not our contribution. We focus on resuming the VM after checkpointing finishes. Our contribution is a fast mechanism to resume the VM from the checkpoint saved on the local disk or a shared storage (storage access time is anyway longer than memory access time).

5.1.3 Resumption Mechanism

When resuming a saved VM from the checkpoint file stored on a slow-access storage, the states saved in the checkpoint must be retrieved. The saved states include the virtual CPU state, the emulated devices' state, and the contents of the VM memory. Usually, most data saved in the checkpoint comes from the VM memory contents. Thus, a straightforward way to resume the check-pointed VM is to restore all the saved memory data first, and then retrieve the saved CPU and device data. Without the necessary CPU and device states, the VM cannot start until all its memory data have been retrieved and all its previous memory pages have been set up. Currently, Xen uses this methodology to resume a check-pointed VM.

We can conclude that the same problem of the checkpointing mechanism (discussed in Section 5.1.2) also exist in the resumption mechanism: as the amount of VM memory contents dominate the saved

data in the checkpoint file, when the memory assigned to VM increases, the time spent on restoring its saved data would quickly become the bottleneck. As shown in Figure 5.2, as the VM memory size increases, the time taken by the `xm restore` command also increases linearly. It works well for small memory (e.g., with a VM memory size of 128MB), but the resumption time significantly increases when retrieving gigabyte data from the checkpoint file (e.g., tens of seconds in the 1GB case).

If the first solution, which restores memory data before the CPU and device data is not effective, how about restoring these data in the reverse order? That is, we let the VM boot first after loading only the necessary CPU and device states, and then restore the memory data saved in the checkpoint file after the VM starts. Whenever the VM needs to access a memory page which hasn't been loaded, it retrieves the corresponding data from the on-disk checkpoint file and sets up the page(s). The benefit of this solution is that the VM starts very quickly, and it always keeps running while restoring the memory data. Moreover, since in this way, the VM only needs to restore a small amount of CPU and device states to start, its performance would not be influenced by the VM memory size.

However, compared with the first approach, the second approach has disadvantages. With the first approach, after the VM starts (although it may take tens of seconds or even several minutes), the VM works as well as that before checkpointing. In contrast, with the second approach, the VM appears to be running after restoring the CPU and device states. However, whenever it accesses a memory page which has not been restored, an immediate page fault occurs. The current execution must then be paused by the hypervisor, the memory page restored from the checkpoint file, and then resumed. Since the VM doesn't restore any memory data at first, significant number of page faults occur at the beginning, degrading VM performance. Our experiments (Section 5.2) show that, for a VM with 1GB RAM, during the first 10 seconds, the VM runs too slow to be useful. Almost all of these seconds were spent on restoring the needed memory data.

To reap the benefits of both resumption solutions and to overcome their limitations, VMresume uses a hybrid resumption mechanism. Our goal is to run the VM as early as possible, but avoid the performance degradation caused by page faults after the VM starts. Our basic idea is to first determine the memory pages that have a high possibility to be accessed during the initial period after the VM starts, restore these pages from the checkpoint file, and then boot the VM by loading the necessary CPU/device states. By preloading the likely-to-be-accessed memory pages, we ensure that after the VM starts, there wouldn't be as much page faults as in the second approach. And, because we don't preload all the memory data saved in the checkpoint before restoring the CPU/device states, we ensure that the VM starts earlier, compared with the first approach.

Now, how to determine the likely-to-be-accessed memory pages? By the principle of temporal locality, recently updated pages tend to be updated in the near future. Therefore, we could rely on the knowledge of recent memory access activities to predict the upcoming memory access activities. Suppose we get an updated checkpoint and we want to resume the VM from the latest checkpoint. By the temporal locality principle, those memory pages accessed during the latest checkpointing interval would have the highest likelihood to be accessed during the initial period.

Therefore, when receiving the checkpoint file during a checkpointing interval, we keep a record of the recently accessed memory pages during that interval, and use that record to predict the memory pages that are likely to be accessed after resuming the VM. This requires a predictive checkpointing mechanism, which we now discuss.

5.1.4 Predictive Checkpointing Mechanism

We develop an incremental checkpointing mechanism in VMresume. During system initialization, VMresume saves the complete image of VM memory and CPU/device states to on-disk file, which is the VM's initial checkpoint. Then, it checkpoints the VM at a fixed frequency. At the beginning of a checkpointing interval (i.e., the time interval between the previous checkpoint and the next checkpoint), all memory pages are set as read-only. Thus, if there is any write to a page, it trigger a page fault. By leveraging Xen's shadow-paging feature, VMresume controls whether a page is read-only and traces whether a page is dirty. When there is a write to a read-only page, a page fault is triggered and reported to the VMM, and that page is set as writeable. VMresume then adds the address of the faulting page to the list of changed pages and removes the write protection from the page so that the application can proceed with the write. At the end of the interval, the list of changed pages contains all the pages that were modified in that interval. VMresume copies the state of all modified pages to the checkpoint, and resets all pages to read-only again.

With the help of this incremental checkpointing mechanism, we can easily find all the write-accessed memory pages during the latest checkpointing interval. These write-accessed pages are likely to be accessed after the VM resumption. However, usually write-accessed pages are only a small portion of the pages likely to be accessed after the VM resumption. Besides, there are more memory pages which are only read-accessed during the same checkpointing interval. The read-accessed pages are not recorded by our checkpointing mechanism, but they must also be preloaded when resuming the VM to reduce potential page faults on those pages. Now, how can we trace and record these read-accessed pages?

One solution is to consider the knowledge provided by the guest OS kernel. For each memory page assigned to it, the guest OS kernel knows exactly whether a page is accessed or not by keeping track of the current status of each page frame. The state information of a page frame is kept in a page descriptor, and all page descriptors are stored in the mem_map array. Each page descriptor has a usage reference counter (`_count` with type `atomic_t`) for the corresponding page. If it is set to -1, the corresponding page frame is free, and that page is not accessed during the current execution of the guest OS. If the reference count is larger than -1, then that page is treated as used (i.e., accessed). Although all the physical memory pages are allocated to the guest OS by the Xen hypervisor, their reference counter information collected within the VM cannot be delivered outside in a synchronous mode. The Xen hypervisor cannot obtain the runtime page descriptor array mem_map from the guest OS kernel because the VM is still running. Since we propose the resumption mechanism to run in the privileged Domain 0, it works outside the target VM. To obtain the access information from the guest OS kernel, the VM must be paused and the

page descriptor array must be copied to Domain 0 (e.g., by `memcpy`). Therefore, this approach generates new memory copy overhead and is not efficient, since checkpointing may happen at high frequency (e.g., two times per second as in our experiment).

An alternate approach to avoid this performance overhead is to leverage the page table entry (PTE) mechanism, which is supported by most current generation processors. VMresume uses this approach for predicting the accessed pages. It leverages PTE's control bit: accessed (A) bit. The accessed bit is set to 1 if the page has been accessed; 0 if not. Besides using the read/write (R/W) bit to track dirty memory pages, VMresume also uses the accessed bit to record all the accessed pages. For each VM, at the beginning of a checkpointing interval, all memory pages are set as read-only and not accessed (both R/W and A bits are cleared). Thus, if there is any write to a page, it will trigger a page fault, and both the R/W and A bits are set to 1. Besides, if a page is read-accessed, its accessed bit is also set to 1.

Therefore, at the end of a checkpointing interval, for the pages updated (i.e., write-accessed) in the interval, VMresume saves them to the checkpoint file and clears their R/W bit. For the pages whose A bit is set to 1, VMresume determines whether they were read-accessed or write-accessed. If they were write-accessed, then they are already saved in the checkpoint file. If they were only read-accessed, then VMresume keeps a record of these read-accessed pages for the prediction purpose when resuming the VM from the corresponding checkpoint file. It is unnecessary to save the content of the read-accessed pages, because they are not updated during the checkpointing interval (actually these pages should be already saved during the previous checkpointing interval, or should be saved in the initial checkpoint file if they were never updated since the VM's start).

When resuming the VM, VMresume first reloads all the write-accessed pages (which are newly saved to the checkpoint), as well as other likely-to-be-accessed pages by tracing the record of the read-accessed pages. Then the VM is started by restoring the CPU and device states. Since all the memory pages preloaded by the resumption mechanism are actually saved by the checkpointing mechanism for prediction purpose, we call the checkpointing mechanism as "predictive checkpointing."

5.2 Evaluation and Results

5.2.1 Experimental Environment

Our preliminary experiments were conducted on one machine with an IA32 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz) and 4 GB RAM. We built Xen 3.4.0 from source [20], and ran paravirtualized guest VM with Linux kernel 2.6.18. The guest OS and the privileged domain OS (in Domain 0) ran CentOS Linux, with a minimum of services initially executing, e.g., `sshd`. Domain 0 had a memory allocation of 2 GB, and the remaining memory was left free to be allocated for guest VM.

VM memory (MB)	128	256	512	1024
Xen-c time (s)	2.11	4.63	9.18	17.45
VMr-c time (s)	0.047	0.064	0.119	0.182
Xen-c size (MB)	128.7	258.2	515.9	1031.3
VMr-c size (MB)	4.33	6.21	9.87	14.14

Table 5.2: Comparison between VMresume' (shown as VMr-c) and Xen's (shown as Xen-c) checkpointing mechanisms.

We implemented VMresume as well as the two approaches in Section 5.1.3, which we call "Full-state-resume" (which restores memory state, CPU state, and device state) and "Quick-state-resume" (which restores only the necessary CPU and device states). To ensure that our experiments are statistically significant, each data point is averaged from ten samples. The standard deviation computed from the samples is less than 3.4% of the mean value.

5.2.2 Resumption Time

We first compare the performance of VMresume's incremental checkpointing approach with Xen's checkpointing mechanism. Note that Xen's checkpointing mechanism saves the complete state of an individual VM (including CPU/device states and all contents of VM memory) into the checkpoint file. The results are shown in Table 5.2. The same checkpointing frequency (500ms) is used for evaluating the performance of both mechanisms.

From Table 5.2, we observe that Xen's checkpointing mechanism needs several seconds to checkpoint a VM. In contrast, VMresume's checkpointing time is in the order of tens to hundreds of milliseconds. This is due to VMresume's incremental checkpointing approach, which does not save all the memory data during each checkpointing interval. Instead, it only saves the memory data updated in the current interval, which requires the VM to be paused only for a short time. Compared with Xen, VMresume reduces checkpointing time by at least 98%.

We also measure the size of each VM checkpoint file with different memory assigned, and observe that the VM checkpoint size can be significantly reduced by the incremental checkpointing approach. Compared with Xen's checkpointing mechanism, VMresume reduces the checkpoint file size by at least 97%.

Next, we measure the resumption time from when triggering the restore function to when the VM starts. Here, the VM is respectively assigned with different memory sizes. The comparison results are shown in Figure 5.3. We observe that Quick-state-resume's resumption time is the shortest. This is because, it does not preload any memory pages, but only restores the necessary CPU and device states, and then immediately starts the VM. Even though Quick-state-resume's resumption time is the shortest, it does not restore any memory pages. Thus, greater amount of page faults

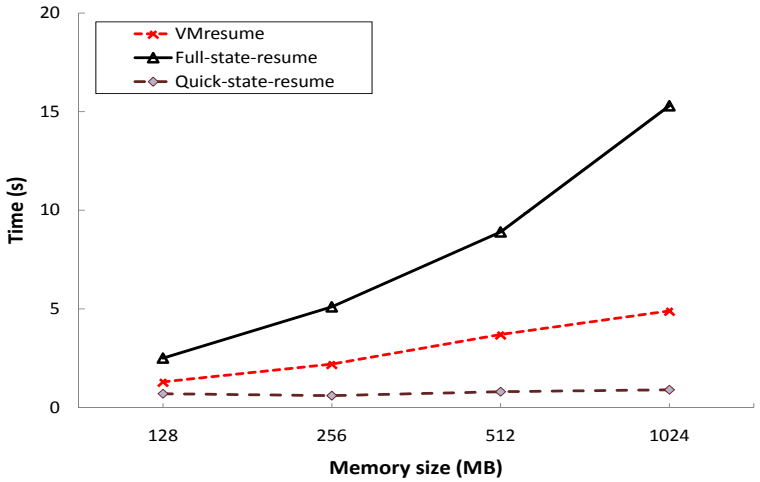


Figure 5.3: Time to resume a VM with diverse memory size under different resumption mechanisms.

would be generated during the initial period after the VM starts, causing the VM to be unusable. (We show this in Figure 5.4.)

Ignoring the non-practical Quick-state-resume design, from Figure 5.3 we observe that VMresume mechanism has great performance improvement on the resumption time, compared with Full-state-resume design. VMresume reduces the time to restore a VM by an average of 67.3%. The reason is because that in VMresume, it doesn't need to preload all the memory pages as the Full-state-resume does, instead, it only restore the most likely accessed pages recorded during the predictive checkpointing. As the most likely write-accessed pages are stored in the checkpoint file, and from Table 5.2 we observe that the checkpoint file is much smaller than the whole memory assigned, so the resumption time in VMresume is dominated by the time to restore the most likely read-accessed pages. As shown in Figure 5.3, when the VM memory size increases, the resumption time in VMresume also increases, meaning that more read-accessed pages need to be preloaded.

5.2.3 Performance Comparison after VM Resumption.

Our final experiment evaluates performance after the VM starts. We configure the VM with 1GB RAM, and run the Apache web server [2] on it. We set a client on another machine, requesting the same load of web pages, one request immediately after another, simultaneously via a 1Gbps LAN, under all three mechanisms (Full-state-resume, Quick-state-resume, and VMresume). Since the same load of web requests are processed in this experiment, the measured throughput (i.e., number of web responses received per second) can be compared for evaluating the impact of page faults on the VM performance during the initial period after the VM starts.

Figure 5.4 shows the results. We observe that under Full-state-resume, the VM starts to work immediately after booting, with no obvious performance degradation. However, the trade-off in

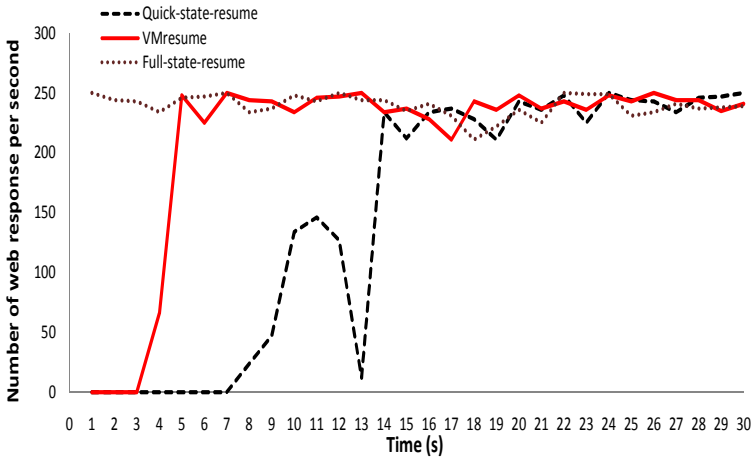


Figure 5.4: Performance after VM starts.

Full-state-resume is that, before the VM starts, it costs a long time (several tens of seconds) to restore each memory page. On the other hand, under Quick-state-resume, the VM starts the fastest, but it suffers performance degradation for a long time – i.e., it needs to wait for 14 seconds to resume normal activity. Compared with Quick-state-resume, under VMresume, the VM endures performance degradation only for about 4.5 seconds, which reduces the VM's unusable time by as much as 67.8%.

5.3 Summary

We present a VM resumption mechanism, called VMresume, which quickly resumes a checkpointed VM, while avoiding performance degradation after the VM starts. Our key idea is to augment incremental checkpointing with a predictive mechanism, which predicts and preloads the memory pages that are most likely to be accessed after resumption. Our preliminary experimentation is promising: VM resumption time is reduced by an average of $\approx 57\%$ and VM's unusable time is reduced by as much as $\approx 68\%$ over native Xen's save/restore mechanism.

Chapter 6

Conclusions, and Proposed Post Preliminary-Exam Work

One of the primary bottlenecks on achieving high availability in virtualized systems is downtime. We presented a novel fine-grained block identification mechanism, called FGBI, that reduces the dirty memory traffic during each migration epoch. The design, implementation, and experimental evaluation of FGBI shows that by recording and transferring the dirty memory at a finer granularity, the total data which needs to be transferred during each epoch is significantly reduced. However, FGBI introduces memory overhead. To reduce that overhead, we proposed two optimization techniques: a memory sharing mechanism and a compression mechanism. Our implementation and experimental comparison with state-of-the-art VM migration solutions (e.g., LLM [23], Remus [14]) shows that FGBI (augmented with the optimizations) significantly reduces the downtime with acceptable performance overhead.

We also presented VPC, a lightweight, globally consistent checkpointing mechanism that records the correct state of an entire VC, which consists of multiple VMs connected by a virtual network. The design, implementation, and experimental evaluation of VPC shows that by using a high frequency checkpointing mechanism and recording only the updated memory pages during each (smaller) checkpointing interval, the downtime can be reduced with acceptable memory overhead. Additional reduction in the downtime can be obtained by predicting the checkpoint-caused page faults during each checkpointing interval. Based on the lightweight design under VPC, the global consistency of the VC is ensured by modifying and implementing a classic distributed snapshot algorithm.

Reducing the time for resuming a VM from a checkpoint on slow-access storage, especially when checkpoint sizes are large (e.g., GBs), is an important problem for a truly effective checkpointing mechanism. We presented VMresume, a hybrid resumption mechanism which predicts the most likely to be accessed memory pages during a checkpointing interval. VMresume's design, implementation, and experimental evaluation confirms that, predicting such pages and preloading them is effective for reducing VM resumption time. We also find that there is a tradeoff between

resumption time and the performance after the VM starts. Previous solutions either achieve fast resumption but suffers performance degradation, or have full performance recovery but suffers a long resumption time. Our evaluation shows that VMresume maintains a reasonable balance between these two endpoints of the tradeoff spectrum.

6.1 Post Preliminary-Exam Work

We propose two major research directions for post-exam work: 1) live VM migration in virtualized environments without a hypervisor and 2) live VM migration in wide area networks (WANs).

6.1.1 Live VM Migration without Hypervisor

Because of its central role in the virtualized system, the hypervisor is a potent target for attacks, especially in shared infrastructures which allow multiple parties to run VMs (e.g., in today's cloud environment). An untrusted hypervisor may influence the guest VM's normal activity. Therefore, if the VM could interact with hardware directly instead of through the hypervisor, it could removes the attack surface of the hypervisor and thus eliminates security vulnerabilities for the entire system.

In non-hypervisor-based virtualized systems [47], each VM runs directly on the hardware without an underlying active hypervisor. However, we cannot remove management functionality completely. For example, the functionality provided by the system manager (e.g., interruption mechanism), core manager (e.g., boot OS), and memory manager (e.g., allocation and garbage collection) are still needed. Nevertheless, in such systems, unlike traditional virtualization architectures, once a VM starts, it have direct access to devices. Thus, the guest OS does not interact with any management software running on the physical machine. Additionally, the virtualization layer's functionality (e.g., CPU schedule, hypervisor call) is not needed while the VM is running.

However, in order to implement an efficient live migration mechanism, we need to start and stop a VM on demand. A straightforward migration methodology, which just stops the execution of the VM, captures the entire state of the VM, and transfers the entire state to the target machine would suffer unacceptable downtime. In order to minimize downtime, optimizations using iterative approaches (also used in our previous work) have been developed [12, 29]. They track the dirty memory during each epoch and synchronize the guest VM's memory state on both source and target machines. Building upon such optimizations, we propose to develop efficient techniques to start and stop VMs.

An example way to do this is by leveraging the system manager and the core manager components. To start a VM, the system manager will store the VM's image on the local disk and will zero out the VM's memory. After initialization, the manager will issue a start interrupt on the processor core on which the VM will boot. A core manager will initialize memory and I/O mapping, and

will boot the guest OS and the VM. To stop a VM, the system manager will issue a stop interrupt to the VM's executing processor core. The core manager will save the disk image of the VM, clears and un-maps the memory and I/O, and transitions the core into the idle state.

Efficient VM migration also requires techniques for suspending and restoring VM. The techniques are similar to VM starting/stopping, and hence could be implemented by leveraging our proposed steps for doing so with minimal modification. For example, when suspending a VM, instead of zeroing out the VM's memory when starting the VM, we must maintain the current state of the VM's memory. This could be done with assistance from the memory manager, for example.

6.1.2 Live VM Migration in WANs

Previous VM live migration mechanisms [12, 34] demonstrate their effectiveness in local area networks (LAN), but only work in the traditional clusters that all the physical machines are located in the same room and connected by LAN. The VM migration mechanisms in a wide area network (WAN) environment is a more interesting problem for modern data centers and cloud computing platforms, as their physical machines are usually located across different places, cities or even countries.

To achieve fast live migration in WANs, the large amount of the disk data which needs to be transferred across the WAN must be reduced. Traditional LAN-based migration efforts (including our previous research) use the checkpointing/resumption methodology to migrate memory data. Moreover, they also use an incremental checkpointing solution [20, 32, 40, 41, 57] to reduce the updated memory data that needs to be transferred during each migration epoch.

Incremental checkpointing can also be done for WAN migrations, especially when transferring the file system data. With incremental checkpointing, however, the VM's virtual file system on the disk needs to be split into multiple chunks with the same size, and a complete copy of the file system must be transferred from the source to the target machines during the initialization round. Then during each migration epoch, if there is any write to a chunk on the source side, that chunk will be marked as "dirty" and must be transferred to the target machine. Moreover, we can use similar techniques as in the FGBI design for optimizations, such as choosing smaller size chunks, which will approximate the true dirty disk data more accurately. However, for I/O intensive applications that write to the disk in high frequency, the number of dirty chunks during each epoch may not be small. A large amount of data due to file system modifications must still be transferred across the WAN.

An alternate approach that we propose is to develop a record/replay mechanism, in order to reduce the network traffic due to transferring the dirty disk data from the source to the target machine. Such a mechanism will have several phases: initialization, recording, replaying, and finalization.

In the initialization phase, the necessary system states (CPU and device states, VM configuration information) of the VM on the source machine is transferred to the target machine. After that, the VM starts to run on the source machine, and the initial state of the virtual file system is transferred

to the target machine. This is followed by a recording phase. In this phase, during each migration epoch, the write activities on the virtual file system are recorded in a log file and transferred to the target machine, instead of transferring potentially large dirty disk data chunks.

This is followed by the replaying phase. On the target machine, upon receiving the log file from the source, the VM replays the write activities to its local file system, following the log file. Thus, the running state of the VMs on the source and the target machines are kept synchronized.

During the final migration epoch, the VM on the source machine is stopped and the remaining state of the VM and the newly generated log file are transferred to the target machine. After the last log file has been replayed, the virtual file system on the target machine should be an identical replica of that on the source machine. Note that, during the entire migration process, the VM is still running on the source machine, and the process is transparent to the applications.

Bibliography

[1] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/.

[2] The apache http server project. http://httpd.apache.org/.

[3] Google app engine - google code. http://code.google.com/appengine/.

[4] Kvm: Kernel based virtual machine. www.redhat.com/f/pdf/rhev/DOC-KVM.pdf.

[5] Nas parallel benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[6] Spec cpu2006. http://www.spec.org/cpu2006/.

[7] Zlib memory compression library. http://www.zlib.net.

[8] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W. Moore, and Andy Hopper. Predicting the performance of virtual machine migration. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:37–46, 2010.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[10] B.S. Boutros and B.C. Desai. A two-phase commit protocol and its performance. In *Database and Expert Systems Applications, 1996. Proceedings., Seventh International Workshop on*, pages 100 –105, sep 1996.

[11] K Chanchio, C Leangsuksun, H Ong, V Ratanasamoot, and A Shafi. An efficient virtual machine checkpointing mechanism for hypervisor-based hpc systems. In *Proceedings of High Availability and Performance Computing Workshop*, 2008.

[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[13] B. Cully, G. Lefebvre, N. Hutchinson, and A. Warfield. Remus source code. `http://dsg.cs.ubc.ca/remus/`.

[14] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[15] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *ISCA '05:Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors. In *Dependable Systems and Networks, 2004 International Conference on*, pages 887 – 896, june-1 july 2004.

[17] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53:85–93, October 2010.

[18] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, pages 3–3, 2003.

[19] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. `http://infohost.nmt.edu/~val/review/hash2.pdf`.

[20] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1558–1562, New York, NY, USA, 2005. ACM.

[21] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[22] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *CLUSTER '07:Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 11–20, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Bo Jiang, Binoy Ravindran, and Changsoo Kim. Lightweight live migration for high availability cluster service. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, New York, NY, USA, 2010.

[24] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[25] A. Kangarlou, P. Eugster, and Dongyan Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 524 –533, 2009.

[26] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 40 – 46, 2002.

[27] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[28] M. Lu and T. Cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems Networks*, pages 534–543, 2009.

[29] Peng Lu, Binoy Ravindran, and Changsoo Kim. Enhancing the performance of high availability lightweight live migration. In *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin / Heidelberg, 2011.

[30] Igor Lyubashevskiy and Volker Strumpen. Fault-tolerant file-i/o for portable checkpointing systems. *J. Supercomput.*, 16:69–92, May 2000.

[31] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18:423–434, August 1993.

[32] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 41–54, New York, NY, USA, 2008. ACM.

[33] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000.

[34] Michael Nelson, Beng Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[35] H. Ong, N. Saragol, K. Chanchio, and C. Leangsuksun. Vccp: A transparent, coordinated checkpointing system for virtualization-based cluster computing. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[36] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.

[37] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 211 –216, july 2007.

[38] Dan Pei. Modification operation buffering: A low-overhead approach to checkpoint user files. In *IEEE 29th Symposium on Fault-Tolerant Computing*, pages 36–38, 1999.

[39] James Plank, James S. Plank, Micah Beck, Micah Beck, Gerry Kingsley, Gerry Kingsley, Kai Li, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–223, 1995.

[40] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[41] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Softw. Pract. Exper.*, 29:125–142, February 1999.

[42] A. Feldmann R. Bradford, E. Kotsovinos and H. Schioeberg. Live wide-area migration of virtual machines including local persistent state. In *VEE'07: Proceedings of the third International Conference on Virtual Execution Environments*, pages 169–179, San Diego, CA, USA, 2007. ACM Press.

[43] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243 – 254, march 2005.

[44] Pierre Riteau, Christine Morin, and Thierry Priol. Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011*, volume 6852 of *Lecture Notes in Computer Science*, pages 431–442. Springer, 2011.

[45] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, December 2002.

[46] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. Fast live cloning of virtual machine based on Xen. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 392–399, Washington, DC, USA, 2009. IEEE Computer Society.

[47] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 401–412, New York, NY, USA, 2011. ACM.

[48] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshii Moriai. Kemari: Virtual machine synchronization for fault tolerance using DomT (technical report). `http://wiki.xen.org/xenwiki/Open_Topics_For_Discussion?action=AttachFile&do=get&target=Kemari_08.pdf`, 2008.

[49] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.

[50] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[51] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[52] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Grible. Constructing services with interposable virtual hardware. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[53] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[54] XenCommunity. Xen unstable source. `http://xenbits.xensource.com/xen-unstable.hg`.

[55] Minjia Zhang, Hai Jin, Xuanhua Shi, and Song Wu. Virtcft: A transparent vm-level fault-tolerant system for virtual clusters. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 147 –154, dec. 2010.

[56] Ming Zhao and Renato J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC '07:Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, pages 5:1–5:8, New York, NY, USA, 2007. ACM.

[57] Weiming Zhao, Zhenlin Wang, and Yingwei Luo. Dynamic memory balancing for virtual machines. *SIGOPS Oper. Syst. Rev.*, 43:37–47, July 2009.