Resilire: Achieving High Availability Through Virtual Machine Live Migration

Peng Lu

Dissertation 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

August 30, 2013
Blacksburg, Virginia

Keywords: High Availability, Virtual Machine, Live Migration, Checkpointing,
Load Balancing, Downtime, Xen, Hypervisor

Resilire: Achieving High Availability Through Virtual Machine Live Migration

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 virtual machine (VM) live 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.

This dissertation presents a set of VM live migration techniques, their implementations in the Xen hypervisor and Linux operating system kernel, and experimental studies conducted using benchmarks (e.g., SPEC, NPB, Sysbench) and production applications (e.g., Apache webserver, SPECweb). We first present a technique for reducing VM migration downtimes called FGBI. FGBI reduces the dirty memory updates that 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 on the dirty blocks to reduce the migration traffic. We implement FGBI in the Xen hypervisor and conduct experimental studies, which reveal that the technique reduces the downtime by 77% and 45% over competitors including LLM and Remus, respectively, with a performance overhead of 13%.

We then present a lightweight, globally consistent checkpointing mechanism for virtual cluster, called VPC, 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 Xen-based implementation and experimental studies reveal that VPC reduces the solo VM downtime by as much as 45% and reduces the entire VC downtime by as much as 50% over competitors including VNsnap, with a memory overhead of 9% and performance overhead of 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 experimental studies reveal that VM resumption time is reduced by an average of 57% and VM's unusable time is reduced by 73.8% over native Xen's resumption mechanism.

Traditional VM live migration mechanisms are based on hypervisors. However, hypervisors are increasingly becoming the source of several major security attacks and flaws. We present a mechanism called HSG-LM that does not involve the hypervisor during live migration. HSG-LM is implemented in the guest OS kernel so that the hypervisor is completely bypassed throughout the entire migration process. The mechanism exploits a hybrid strategy that reaps the benefits of both pre-copy and post-copy migration mechanisms, and uses a speculation mechanism that improves the efficiency of handling post-copy page faults. We modify the Linux kernel and develop a new page fault handler inside the guest OS to implement HSG-LM. Our experimental studies reveal that the technique reduces the downtime by as much as 55%, and reduces the total migration time by as much as 27% over competitors including Xen-based pre-copy, post-copy, and self-migration mechanisms.

In a virtual cluster environment, one of the main challenges is to ensure equal utilization of all the available resources while avoiding overloading a subset of machines. We propose an efficient load balancing strategy using VM live migration, called DCbalance. Differently from previous work, DCbalance records the history of mappings to inform future placement decisions, and uses a workload-adaptive live migration algorithm to minimize VM downtime. We improve Xen's original live migration mechanism and implement the DCbalance technique, and conduct experimental studies. Our results reveal that DCbalance reduces the decision generating time by 79%, the downtime by 73%, and the total migration time by 38%, over competitors including the OSVD virtual machine load balancing mechanism and the DLB (Xen-based) dynamic load balancing algorithm.

The dissertation's final contribution is a technique for VM live migration in Wide Area Networks (WANs), called FDM. In contrast to live migration in Local Area Networks (LANs), VM migration in WANs involve migrating disk data, besides memory state, because the source and the target machines do not share the same disk service. FDM is a fast and storage-adaptive migration mechanism that transmits both memory state and disk data with short downtime and total migration time. FDM uses page cache to identify data that is duplicated between memory and disk, so as to avoid transmitting the same data unnecessarily. We implement FDM in Xen, targeting different disk formats including raw and Qcow2. Our experimental studies reveal that FDM reduces the downtime by as much as 87%, and reduces the total migration time by as much as 58% over competitors including pre-copy or post-copy disk migration mechanisms and the disk migration mechanism implemented in BlobSeer, a widely used large-scale distributed storage service.

To my wife, parents, and little brother

Acknowledgments

I would first like to thank my advisor Dr. Binoy Ravindran for his guidance, inspiration, and patience during my research. When I experienced the hardest time during my study, I would not have been able to do the research and achieve learning without his help and support. His recommendations and instructions have enabled me to assemble and finish the dissertation effectively.

I would also like to thank my committee members: Dr. Robert P. Broadwater, Dr. Paul E. Plassmann, Dr. C. Jules White and Dr. Danfeng Yao, for their guidance and advice during my preliminary and defense exams. It is a great honor to have them serving in my committee. I am also grateful to Dr. Antonio Barbalace. We worked together in last two years, and he always shared invaluable advice and guidance.

In addition, many thanks to all my colleagues in Systems Software Research Group, who throughout my educational career have supported and encouraged me to believe in my abilities. They include: Dr. Bo Jiang, Dr. Bo Zhang, Dr. Roberto Palmieri, Dr. Alastair Murray, Junwhan Kim, Mohamed Saad, Alex Turcu, Mohammed El-Shambakey, Sachin Hirve, and all other friends in SSRG who have aided me throughout this endeavor.

Last but not least, thank all my family members for their love and support. I am grateful to my parents, who did their best in supporting my education and my life. They always had the constant belief and confidence on me during my journey chasing my dream. My little brother, Xiao Lu, although he didn't help a lot during my study. Actually he is still in high school right now. But thanks for coming to my life when I was twelve and thanks for always being there. Finally, my wife, Dr. Jinling Li, devoted her love and endured my grievance over the past years. Without her love, care and support, I could not have completed this dissertation. It is difficult to overstate my gratitude to her for being such a wonderful wife.

This dissertation is dedicated to all the people who helped me and are helping me all the way.

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 [43]. 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. Therefore, 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 [12], VMware [95], KVM [5]). 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 [99].

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 [12], VMware [95], KVM [5]) 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 [65, 75], 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 [37, 67, 76, 78, 104], 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).

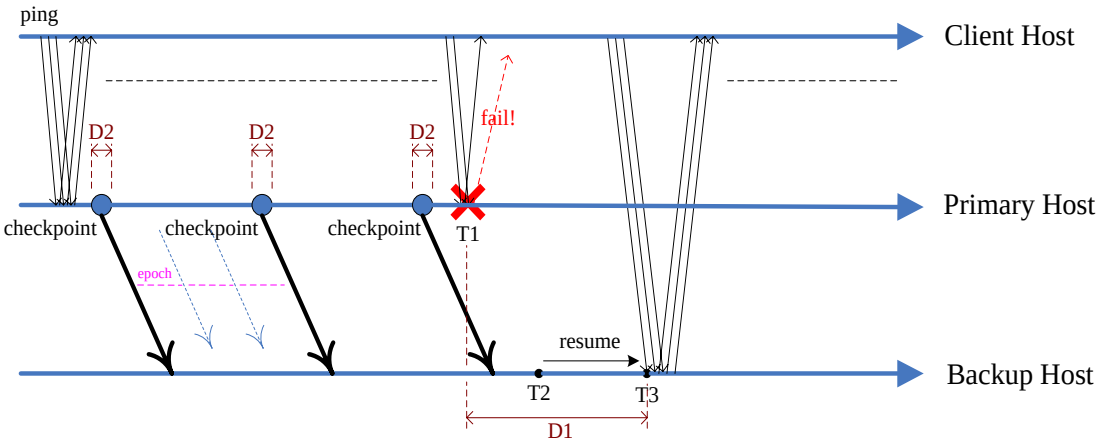


Figure 1.1: Primary-Backup model and the downtime problem (T_1: primary host crashes; T_2: client host observes the primary host crash; T_3: VM resumes on backup host; D_1 ($T_3 - T_1$): type I downtime; D_2: type II downtime).

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 [102]. 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 [4]) 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

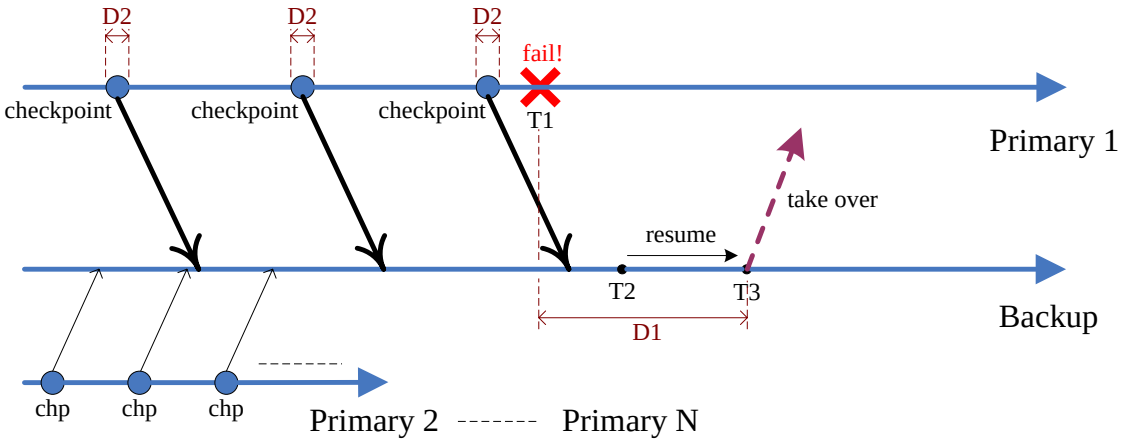


Figure 1.2: The downtime problem when checkpointing the VC. T_1: one of the primary VM fails; T_2: the failure is observed by the VC; T_3: VM resumes on backup machine; D_1 ($T_3 - T_1$): VC downtime; D_2: VM downtime.

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

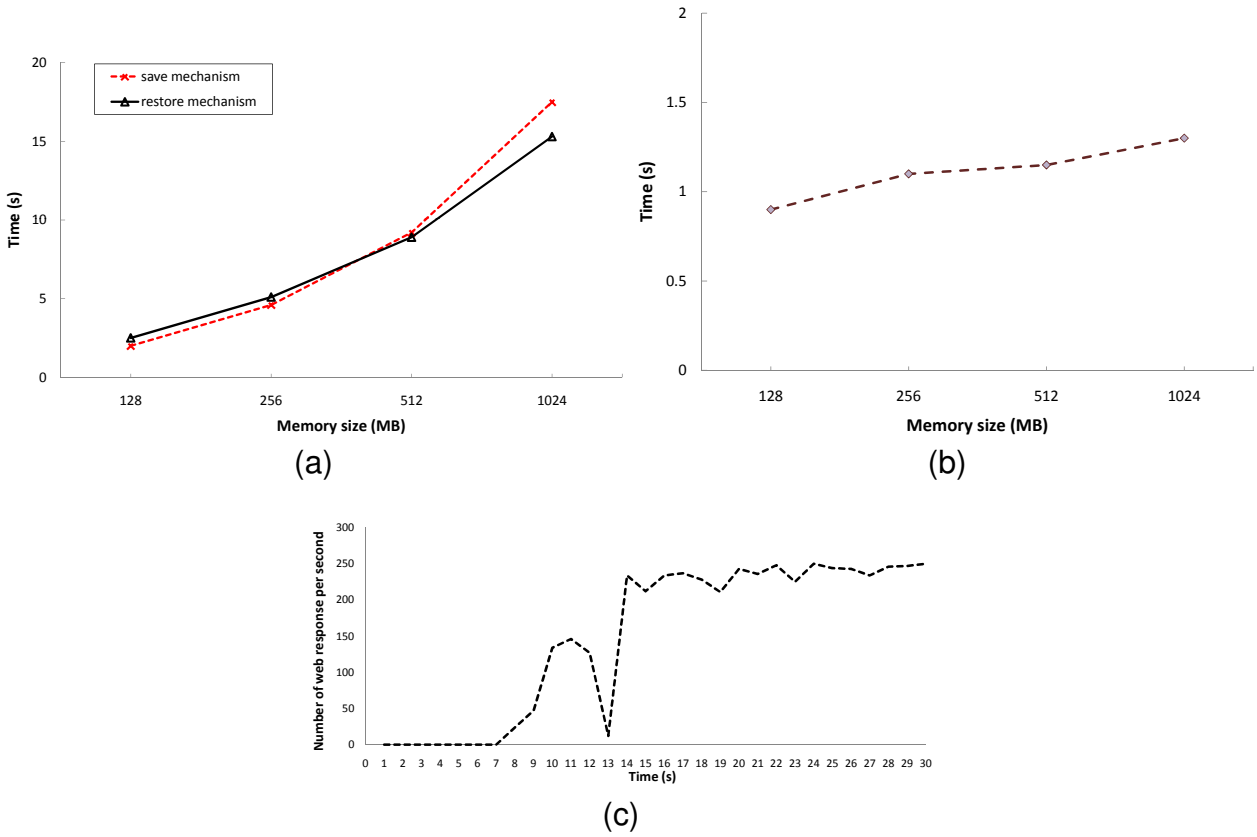


Figure 1.3: Comparison of VM resumption mechanisms.

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 Migration without Hypervisor

Most VMMs (e.g., Xen [12], VMware [95], KVM [5]) provide a hypervisor-based migration solution. However, relying on hypervisor in the migration has several disadvantages that have recently received significant attention [47, 90]. The main problem is that the hypervisor plays a key role throughout the entire migration process, and takes care of many system "chores" from the start to the end of the migration. This central role of the hypervisor has increasingly become the source of several major security concerns. Szefer *et al.* [90] show that if the hypervisor is attacked successfully by a malicious party, the attacker can easily inspect memory, expose confidential information, or even modify the software running in the VM. Note that an investigation of such security problems is outside the scope of this dissertation. However, the potential security problems caused by the hypervisor motivate us to explore VM live migration mechanisms that do not involve the hypervisor during the migration.

In an attempt to resolve these issues, a live migration mechanism *without* the hypervisor is needed. Besides addressing the security issues, the solution should also cover use-case scenarios that arise from common VM practice. One representative scenario is a data center setting, which offers an isolated VM for each user. In such a setting, a fully customized VM is usually created for each user, with custom environment and applications. In this scenario, there are several reasons why an administrator might decide to migrate the VMs, such as for load balancing and fault-tolerance purposes. In that case, what are the metrics that must be considered in providing high-availability service?

As discussed before, the downtime experienced by the user, i.e., the time interval during which the user cannot access the system, is clearly one important metric. Downtime must be as short as possible. Most hypervisor-based migration mechanisms provide minimal downtime. In addition to downtime, another important metric is the total waiting time to complete the migration, as it affects

the release of resources on both source and target hosts [38, 39]. For example, for load balancing, energy saving policies, hardware upgrades, etc., a running host must often be shut down. In such cases, all active VMs must be migrated to other hosts and resumed on the target machines before resources can be released on the source and target machines. Most previous works only focus on minimizing downtime. We propose to minimize total migration time, besides downtime.

1.5 Adaptive Live Migration to Improve Load Balancing

The virtualized networked environment, where multiple physical machines running VMs are interconnected via fast networking technologies, is able to increasingly provide on-demand computational services and high availability. One of its most interesting features is that the pool of available resources (CPU, virtual memory, secondary storage, etc.) can be shared between all the machines through the available interconnections. Several previous works [20, 41, 57] have observed that many resources, however, are actually unused for a considerable amount of the operational time. Therefore, load balancing is of great interest in order to avoid situations where some machines are overloaded while others are idle or under-used.

Traditionally, there are several ways to achieve load balancing in networked systems. One straightforward solution is static load balancing, which assigns applications to machines at the beginning. The efficiency of this strategy depends on the accuracy of the prior load prediction. A dynamic load balancing strategy can be exploited by migrating application processes at run-time among the different machines. A dynamic strategy is more efficient than limiting the applications to run on the machines where they were first assigned. However, it is usually complex to implement a process migration mechanism, because applications are often tightly bound to the host OS (e.g., file descriptors, sockets) as well as the underlying platform (e.g., device drivers, natively compiled code). Moreover, some types of processes may depend on shared memory for communication, which suffers from the problem of residual dependencies and causes further complications.

The limitations of process migration are overcome through VM live migration. Unlike process migration, VM live migration transfers the virtual CPU state, the emulated devices' state, and the guest OS memory data, which eliminates OS or platform dependencies. Also, VM live migration has several potential advantages for load balancing. A VM that hosts different applications on a heavily loaded machine can be migrated to a VMM on another machine that is idle, in order to exploit the availability of all the resources. Moreover, in contrast to stop-and-resume VM migration, VM live migration natively ensures minimal downtime and minimally interrupts the VM users' interaction. Therefore, a good load balancing strategy should also provide minimal downtime for users.

1.6 VM Disk Migration

In contrast to live migration in Local Area Networks (LANs), 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 memory state, all the disk data, including the I/O device state and the file system, must also be migrated, because they are not shared in both source and target machines. The disk data, in particular for I/O intensive applications, is usually very large (e.g., in the order of hundreds of GBs). Hence, LAN-based migration techniques that only migrate memory data (which is usually in the order of GBs) may not be suitable when applied to VM migration in WANs.

A straightforward approach to do this – explored in the past [49, 84, 97] – is to suspend the VM on the source machine, transfer its memory and local disk data (as a self-contained image file) over the network to the target machine, and resume the VM by reloading the memory and the file system. However, such stop-and-resume mechanisms suffer long downtime. In order to reduce the large amount of disk data that has to be transmitted, several optimization techniques have also been introduced – e.g., data compression during migration [44]; content-based comparison among memory and disk data [81]. However, these optimizations introduce either memory or computational overheads. Thus, it is necessary to develop new methods for migrating VMs and their potentially large file systems with minimal downtime and acceptable overhead.

To achieve fast VM live migration in WANs, the large amount of disk data that need to be transferred across the WAN must be reduced. Traditional LAN-based migration efforts (including our previous work [60,61,62,63,64]) use the checkpointing/resumption methodology to migrate memory data. Moreover, they also use an incremental checkpointing solution [78, 104] to reduce the updated memory data that need to be transferred during each migration epoch. The incremental checkpointing technique can also be used for disk migration to achieve small downtime, but the large data in memory and disk combined can still result in unacceptable total migration time. Therefore, a fast disk migration mechanism is needed.

1.7 Summary of Research Contributions

The dissertation makes the following research contributions:

We first focus on reducing VM migration downtime through memory synchronization and develop a technique called FGBI [63]. FGBI reduces the dirty memory updates that 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. FGBI also uses a hybrid compression mechanism for compressing the dirty blocks to reduce the migration traffic.

We implemented FGBI in the Xen VM and compared it with two state-of-the-art VM migration techniques including LLM [43] and Remus [19], on benchmarks including the Apache webserver [2] and applications from the NPB benchmark suite (e.g., EP program) [6]. While Apache is a network intensive application, NPB-EP is a memory intensive application, thereby covering a broad spectrum of workloads. 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, called VPC [64], 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 constructed a Xen-based implementation of VPC and conducted experimental studies. Our studies reveal that, compared with past VC checkpointing/migration solutions including VNsnap [46], VPC reduces the solo VM downtime by 45% for the NPB benchmark [6], and reduces the entire VC downtime by 50% for distributed programs (e.g., EP, IS) of the 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 overheads in 32, 64, and 128 VM environments were found to result in a speedup loss that is less than 14% for the NPB-EP benchmark.

The dissertation's third contribution is a VM resumption mechanism, called VMresume [62], 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. We implemented VMresume in Xen and conducted experimental studies. Our results reveal that VM resumption time is reduced by an average of 57% and VM's unusable time is reduced by 74% over native Xen's resumption mechanism.

Next, we present a mechanism called HSG-LM that does not involve the hypervisor during live migration [61]. The mechanism is implemented in the guest OS kernel to bypass the hypervisor during migration. HSG-LM exploits a hybrid strategy that reaps the benefits of both pre-copy and post-copy migration mechanisms, and uses a speculation mechanism to improve the efficiency of handling post-copy page faults. We implemented HSG-LM in the Linux kernel and conducted experimental evaluations using the Sysbench benchmark [49] and Apache webserver [2]. Our results show that HSG-LM incurs minimal downtime and short total migration time. In particular, HSG-LM reduces the downtime by 55% over post-copy migration mechanisms, and reduces the total migration time by 27% over pre-copy migration mechanisms.

In order to achieve fast and efficient load balancing in the VM environment, we propose a central-

ized load balancing framework based on history records, called DCbalance [60]. Instead of making predictions in advance, DCbalance uses previous load distribution records, and uses the record with similar load distribution as the current situation to generate the same migration decision. This reduces the latency for triggering migrations. Moreover, for non-memory-intensive workloads, DCbalance uses pre-copy migration, but compresses the dirty memory data before transmitting, reducing the downtime. For memory-intensive workloads, DCbalance uses post-copy migration to avoid transmitting the same memory page multiple times, yielding short total migration time. We implemented DCbalance in a Xen-based environment. Our evaluations show that the technique is faster than competitors including OSVD [55] and DLB [42], reducing the decision generating time by as much as 79% for the NPB benchmark [6]. Furthermore, it reduces the downtime by as much as 73% and the total migration time by as much as 38% for the Sysbench benchmark [49].

The dissertation's final contribution is a fast and storage-adaptive VM migration mechanism for WANs, called FDM. Using page cache, FDM identifies data that is duplicated between memory and disk, and thereby avoids transmitting the same data unnecessarily, reducing the downtime and total migration time. FDM also applies an adaptive migration method for different disk formats including two widely used images: raw and Qcow2. The key idea is to implement different optimizations for different image file types, in order to further reduce the total amount of data before each transmission, reducing both downtime and total migration time. We implemented FDM in Xen and conducted experimental studies using Sysbench [49]. The results reveal that FDM reduces the downtime by 87%, and reduces the total migration time by 58% over competitors including pre-copy or post-copy disk migration and the disk migration mechanism implemented in BlobSeer [71], a widely used VM image storage and deployment system.

1.8 Dissertation Organization

The rest of this dissertation proposal is organized as follows. Chapter 2 overviews the past and related work. Chapter 3 proposes our approach (FGBI) to achieve lightweight live migration for solo VM case, and presents our implementation and experimental evaluation. 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. Chapter 6 presents the design of HSG-LM and its optimization using speculative migration and the workload-adaptive design, reports our experimental environment, benchmarks, and evaluation results. Chapter 7 presents the design and implementation of our new load balancing strategy (DCbalance) which is based on Xen live migration, and reports our evaluation numbers. Chapter 8 proposes the fast and storage-adaptive migration mechanism (FDM), present the detailed implementation and summarize the evaluation results. We conclude the dissertation and identify future research directions in Chapter 9.

Chapter 2

Related Work

2.1 VM Live Migration: Overview

There are several non-live approaches to VM migration. Internet suspend/resume [85] focuses on saving/restoring computing state on anonymous hardware. Sapuntzakis *et. al.* [84] address user mobility and system administration by encapsulating the computing environment into capsules that to be transferred between distinct hosts. Schmidt *et. al.* [86] apply capsules, which are groups of related processes along with their IPC/network states, as the migration units. Similarly, Zap [73] uses process groups (pods) along with their kernel state as migration units. In all these solutions, execution is suspended and the applications within each VM do not make any progress.

To achieve high availability, currently there exist many virtualization-based live migration techniques [40, 58, 79, 103]. Two representatives are Xen live migration [17] and VMware VMotion [70], 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 minimal VM downtime, as the evaluation results in [30] show.

Besides the pre-copy mechanism such as Kemari [25, 52, 91, 92, 93], there are also other related works that focus on migration optimization [79, 103]. Post-copy based migration [38, 39] is proposed to address the drawbacks of pre-copy based migration. The experimental evaluation in [38] shows that the migration time using the post-copy method is shorter than the pre-copy method. However, its implementation only supports para-virtualized (PV) guests as the mechanism for trapping memory accesses and utilizes an in-memory pseudo-paging device in the guest. Since the post-copy mechanism requires a modified/patched guest OS, it is not as widely used as the pre-copy mechanism. Hines *et. al.* [38] also introduces the idea to combine pre-copy and post-

copy mechanism together. They propose an adaptive pre-paging method, which keeps track of the access pattern of the application.

2.2 Enhanced VM Live Migration

Remus [19] 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 [43]) 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 [59] (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.* [59] 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.

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 [17,94], 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 [17, 94] 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 [96] also illustrated the sharing potential in memory. CBPS was based on the compare-by-hash technique introduced in [34, 35]. 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 [89]. 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 [30]. It shows the huge potential of harnessing memory redundancy in VMs. However, Difference Engine also suffers from complexity problems when using the patching method because it needs additional modifications to Xen.

2.3 VM 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.* [65] 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.* [75] 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 [77] is an open-source, portable checkpointing tool for UNIX. It mainly focuses on performance optimization, and only supports single-threaded processes. Osman *et al.* [73] 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.

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 [68]. This technique is easy to implement, but incurs a large system downtime during checkpointing. Live VM migration is designed to avoid such large downtime ([33, 51, 53, 69, 98]). 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 [16] 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 [19] 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 [19] is approximately 50% for a checkpointing interval of $50ms$).

2.4 VC Checkpointing Mechanisms

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 [102], VCCP [72], and VNsnap [46]. VirtCFT provides high availability for virtual clusters by checkpointing individual VMs to additional backup hosts. VirtCFT adopts a two-phase commit coordinated-blocking algorithm [14] (assuming FIFO communication channels) as the global checkpointing algorithm. A checkpoint coordinator broadcasts checkpointing requests to all VMs and waits for two-phase acknowledgements. 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 non-blocking distributed snapshot algorithm, VNsnap [46] 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 [66]) 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%.

2.5 Load Balancing Using Live Migration

The general problem of the load balancing has been examined for decades, at many different levels using a variety of strategies [21, 28, 31]. A general motivation is to optimize the use of resources such as CPU, memory in a distributed computing environment. Traditionally process migration is used in a cluster system to share the work of heavily loaded processors with more lightly loaded processors. Process migration allows the system to take advantage of changes in process execution

or network utilization during the execution of a process. It can be performed either at the user level or the kernel level.

User level strategies, [56] allow dynamic migration of processes using checkpointing. [26] relies on the cooperation between the process and the migration subsystem to achieve migration. The problem in these user level implementations is that without kernel access, they are unable to migrate processes with location dependent information and interprocess communication. On the other side, in kernel level, such implementations [22, 82] allow the migration process to be done quicker and are able to migrate more types of processes. Compared with user level process migration, they provide better performance. Although kernel level process migration techniques are more efficient, they require modifications to the operating system kernel.

As virtualization becomes more and more prevalent, we can overcome these limitations in process migration by introducing the VM as a unit of load balancing. In the virtual world, all the applications/processes are running in the VM, so now it's possible to carry on the load balancing based on the whole-system replication [24,27,45,50,83,87]. Currently there exist many virtualization-based live migration techniques [40, 79]. With live migration, lots of new research has been proposed to improve the load balancing performance in cluster system. Some works [36,88,101] use prediction method to forecast future resource demands based on the current resource utilization. However, in order to get accurate prediction, these work needs to obtain and analyze the response performance all the time, which introduces overhead. Different from these works, our load balancing strategy doesn't make predictions in advance, instead, we always update and refer to the history record to assist generating the final decision. Some other works [13, 48, 99] focus on designing the algorithm to determine what and where to migrate and how much resource to allocate after the migration. However, they didn't consider the migration cost and the downtime and total migration time performance are not evaluated in experimental results.

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 2.2), 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 2.2), 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 [17, 34, 35, 94, 96] 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 [30] 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 [30]).

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 [44]. 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

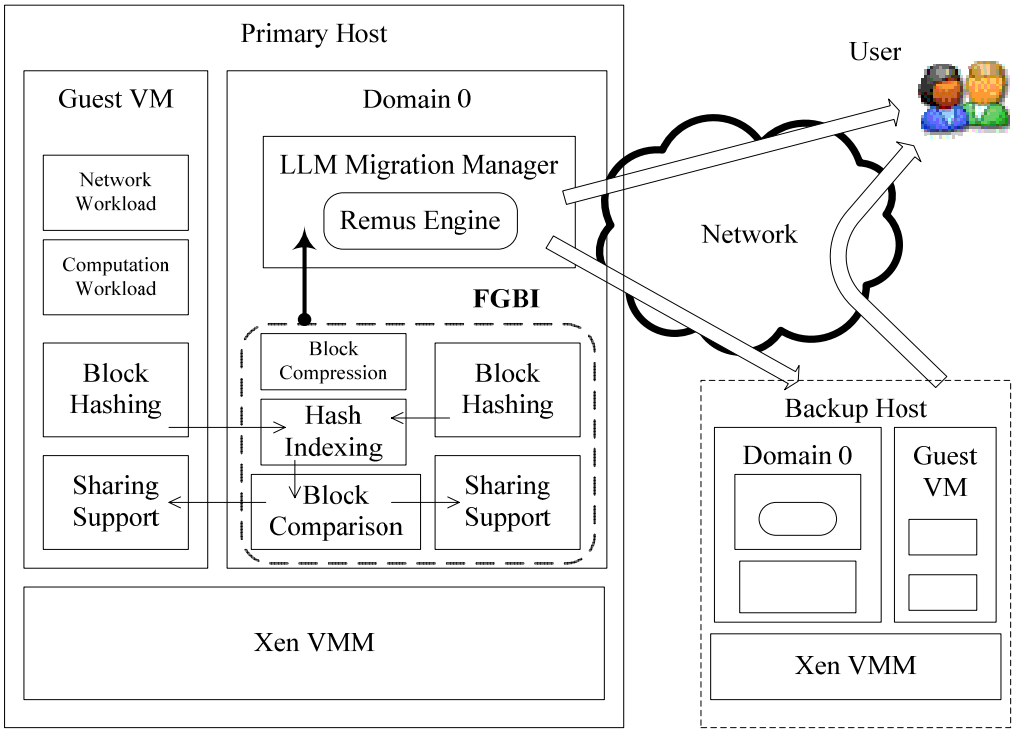


Figure 3.1: The FGBI architecture with sharing and compression support.

(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.* [23] 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 [10], 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:

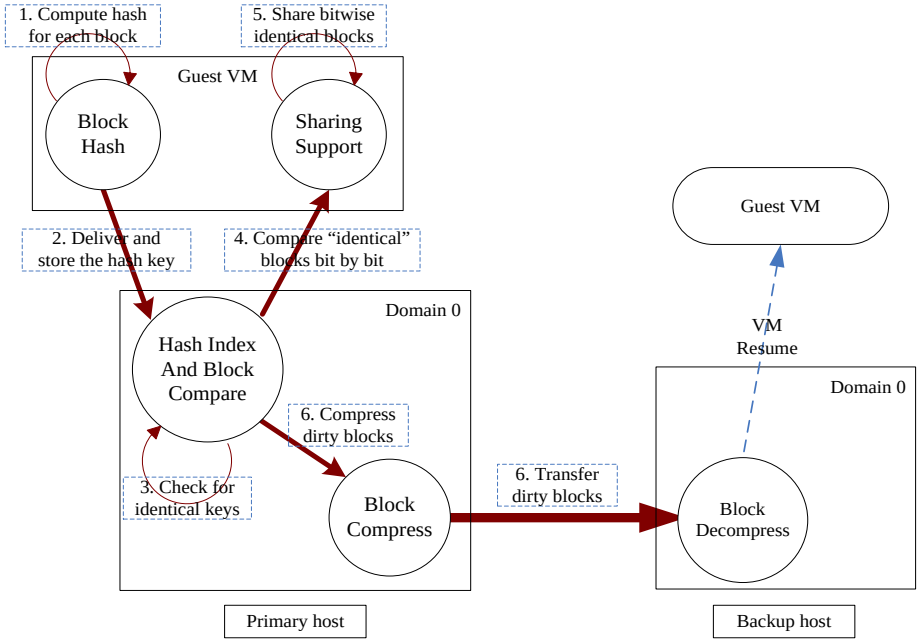


Figure 3.2: Execution flow of FGBI mechanism.

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.

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 [100], 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 [18]. 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 workload 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 [11].

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 [6], 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.

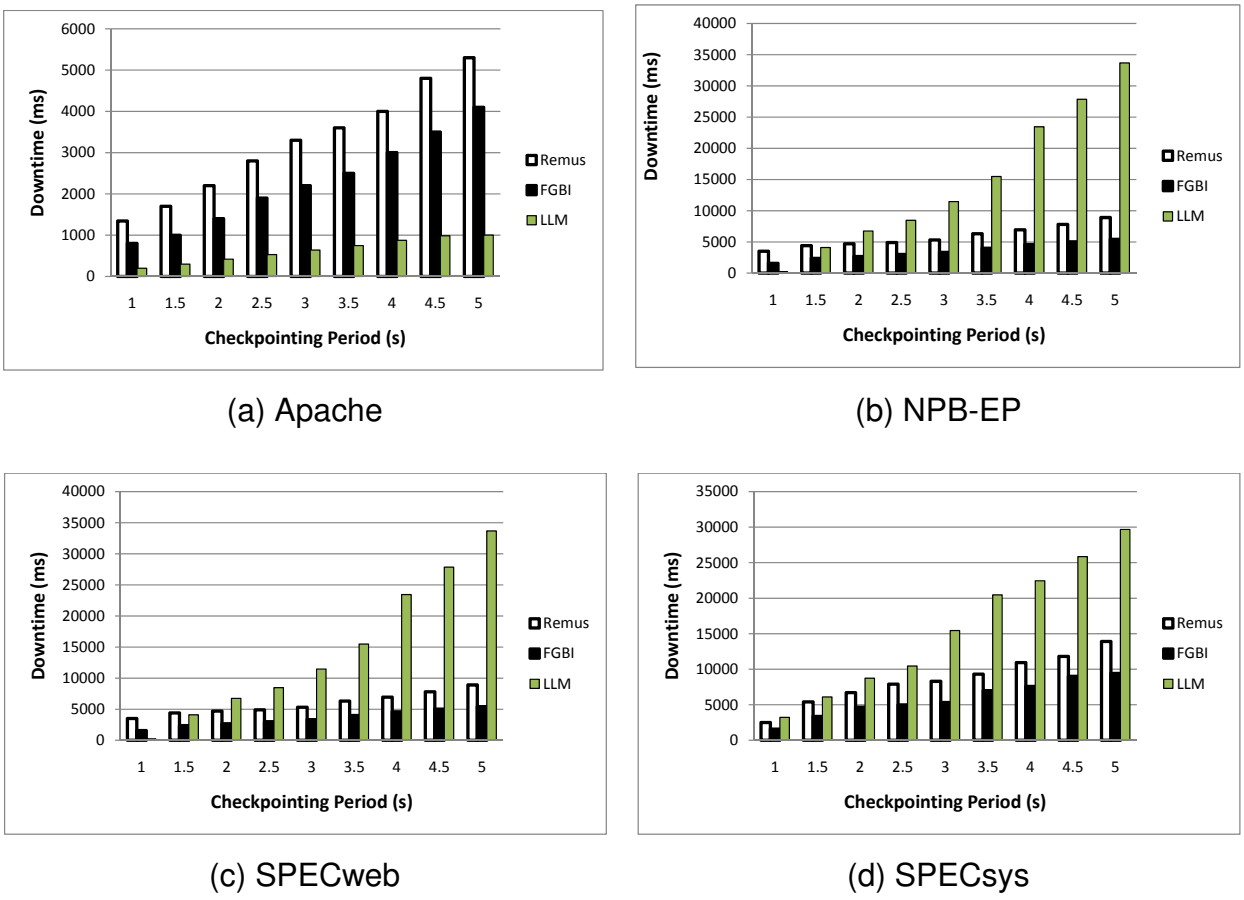


Figure 3.3: Type I downtime comparison under different benchmarks.

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 [11], 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.

3.2.2 Downtime Evaluations

Type I Downtime.

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.

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 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 [17]. 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 show that FGBI still outperforms both Remus and LLM.

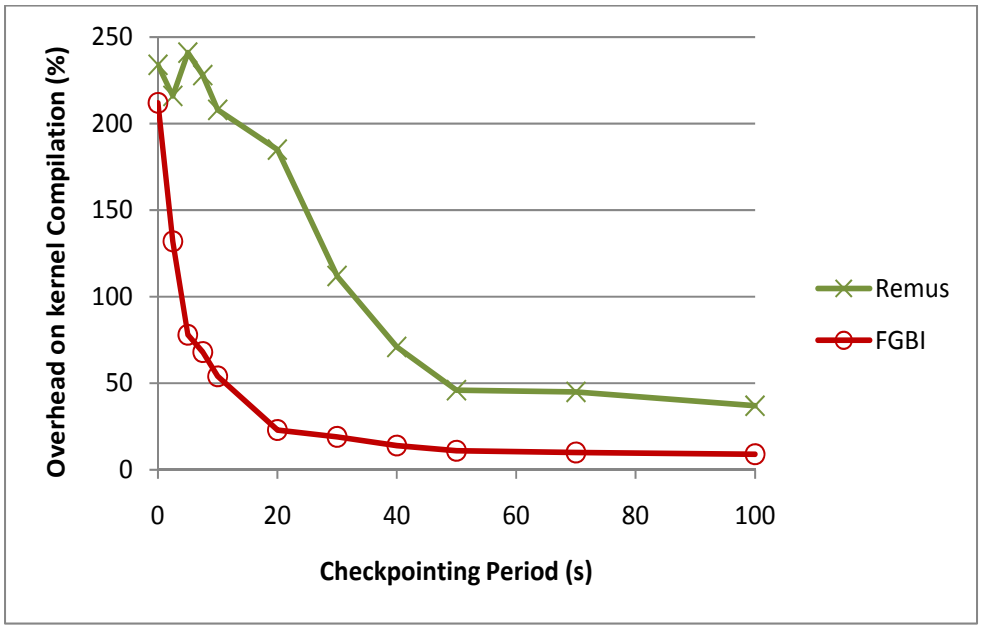


Figure 3.4: Overhead under Kernel Compilation.

3.2.3 Overhead Evaluations

Figure 3.4 shows the FGBI overhead under kernel compilation (compared with Remus). Note that, the overhead significantly changes only in the checkpointing period interval of [1,60] seconds, as shown in the figure. For checkpointing with shorter periods, the migration of system updates may take longer than a configured checkpointing period. Therefore, the kernel compilation time for these cases is almost the same with minor fluctuations. For checkpointing with longer periods, especially when it is longer than the baseline (i.e., kernel compilation without any checkpointing), a VM suspension may or may not occur during one compilation process. Therefore, the kernel compilation time will be closer to the baseline — no more than 12.5% overhead. Right in this interval, FGBI's overhead due to the suspension of guest VM is significantly lower than that of Remus. This is because, FGBI is implemented on LLM, which runs at a much lower frequency than Remus.

Figure 3.5 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. The checkpointing period interval is set as 2 seconds here. 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

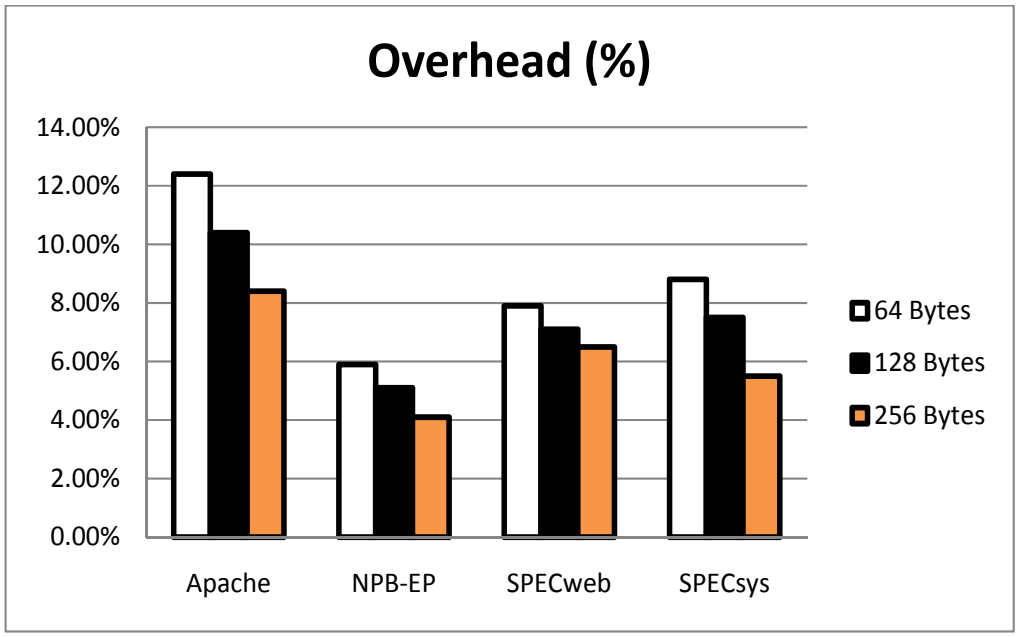


Figure 3.5: Overhead under different block size.

8.21%, on average.

In order to understand the respective contributions of the three proposed techniques (i.e., FGBI, sharing, and compression), Figure 3.6 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 3.6 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.

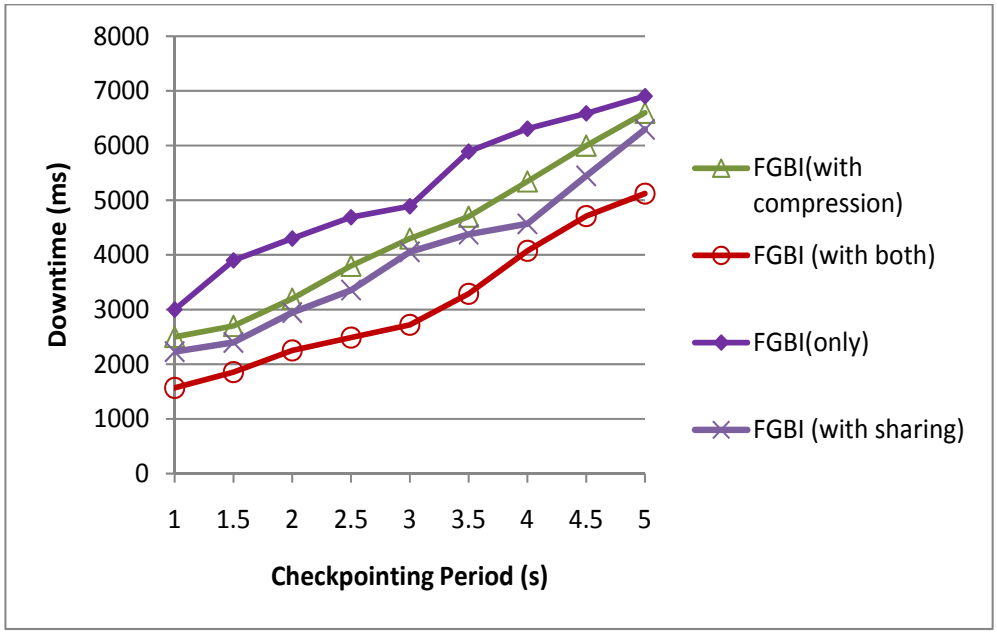


Figure 3.6: Comparison of proposed techniques.

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 [67,76], 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 [37]. 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)

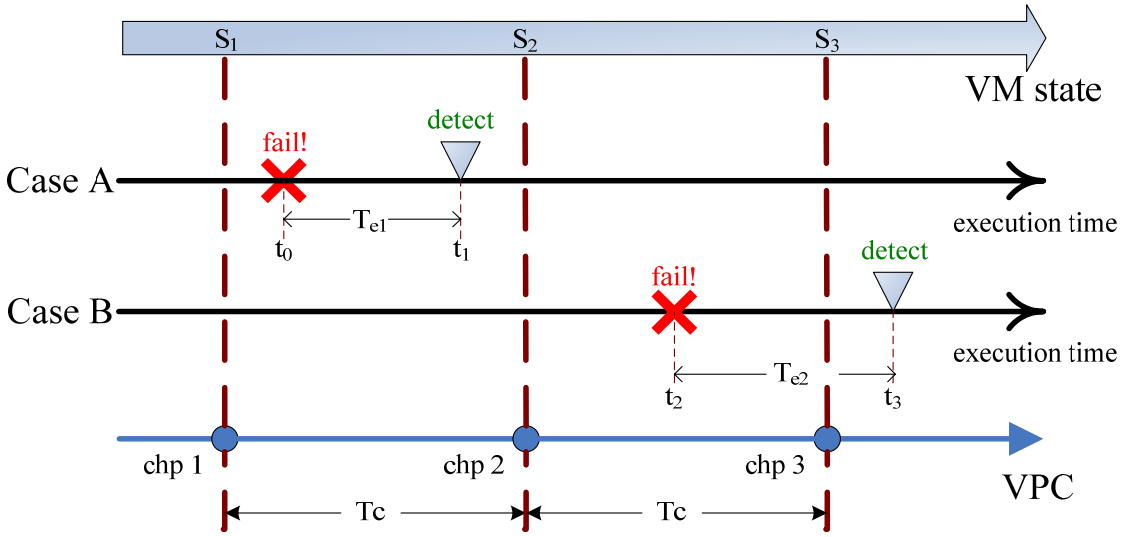


Figure 4.1: Two execution cases under VPC.

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 [54,74,80] have shown that most system crashes occur due to transient failures. For example, in the Linux kernel [29], 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 [46] 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

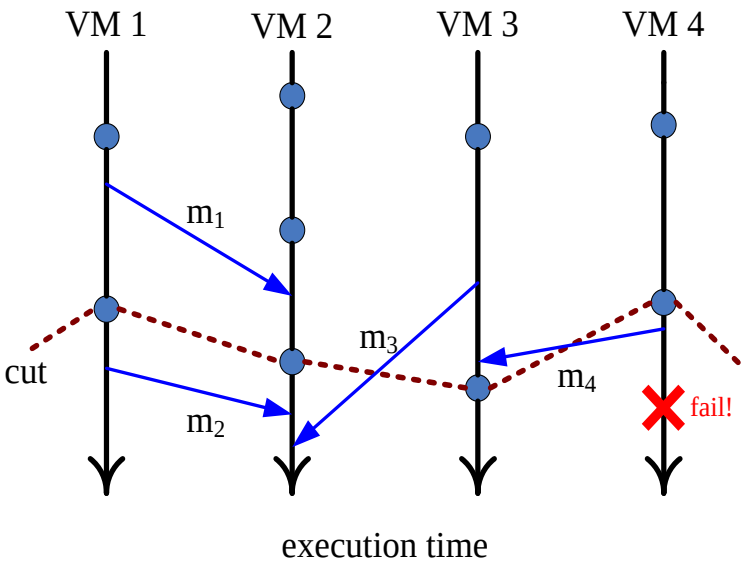


Figure 4.2: The definition of global checkpoint.

line.

2) The message's source state is above the cut line while the destination state is under the cut line, 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 [72, 102] 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 [66]. 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

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.

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 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 [19], LLM [43], and VNsnap [46]. 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 [6], 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 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 [6] 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 aver-

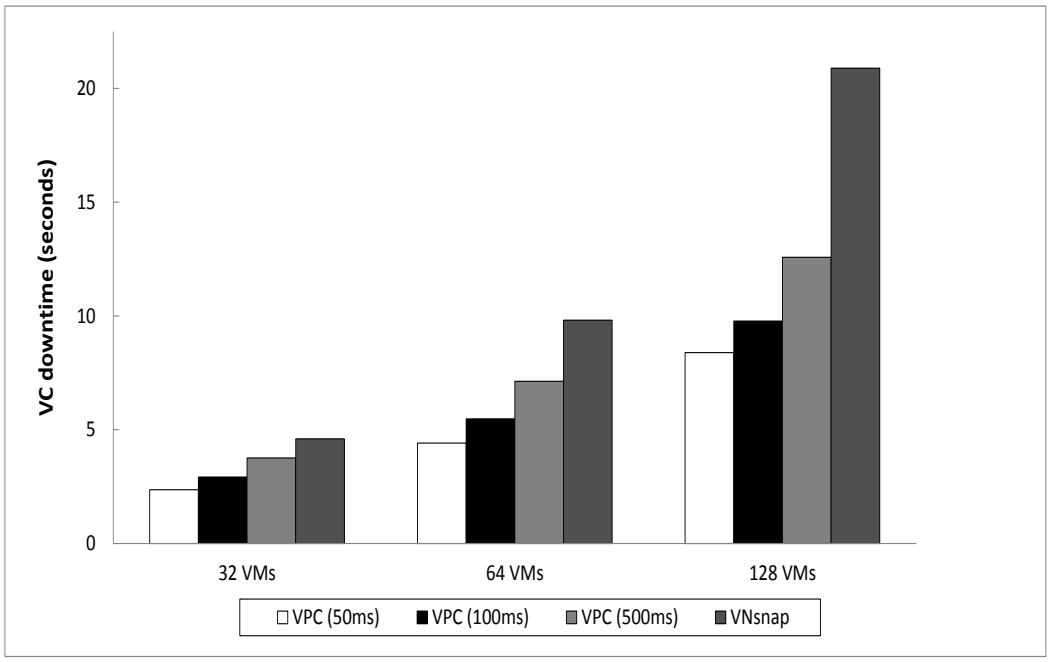


Figure 4.3: VC downtime under NPB-EP framework.

age 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 [8], 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);

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.

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.

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).

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 [6], 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 [19] deployed with one checkpointing interval ($50ms$), (4) LLM [43] 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

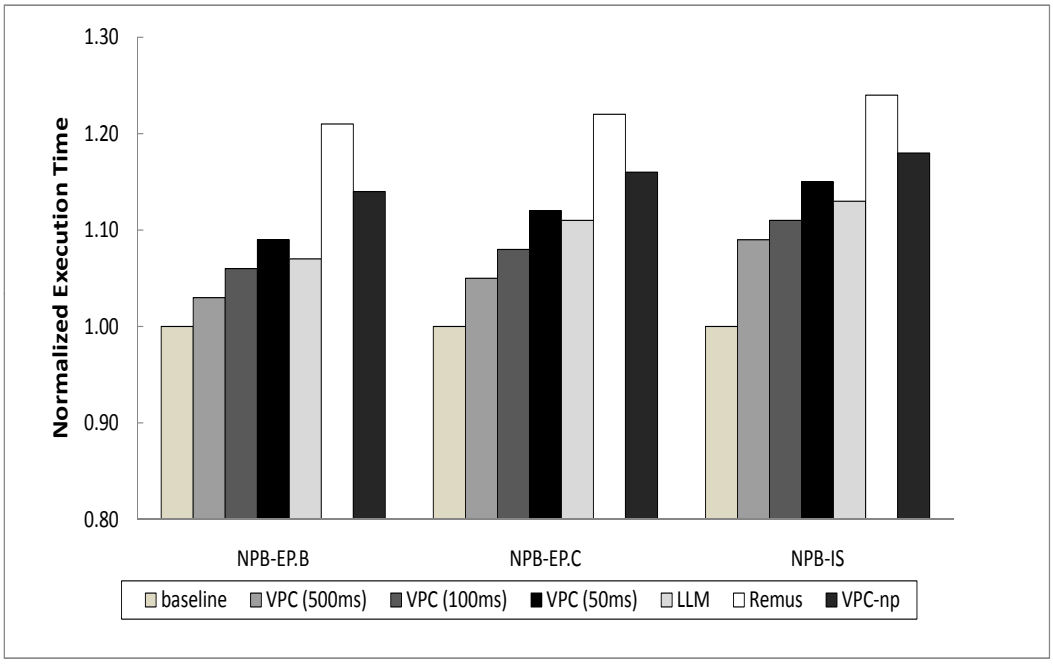


Figure 4.4: Performance overhead under NPB benchmark.

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

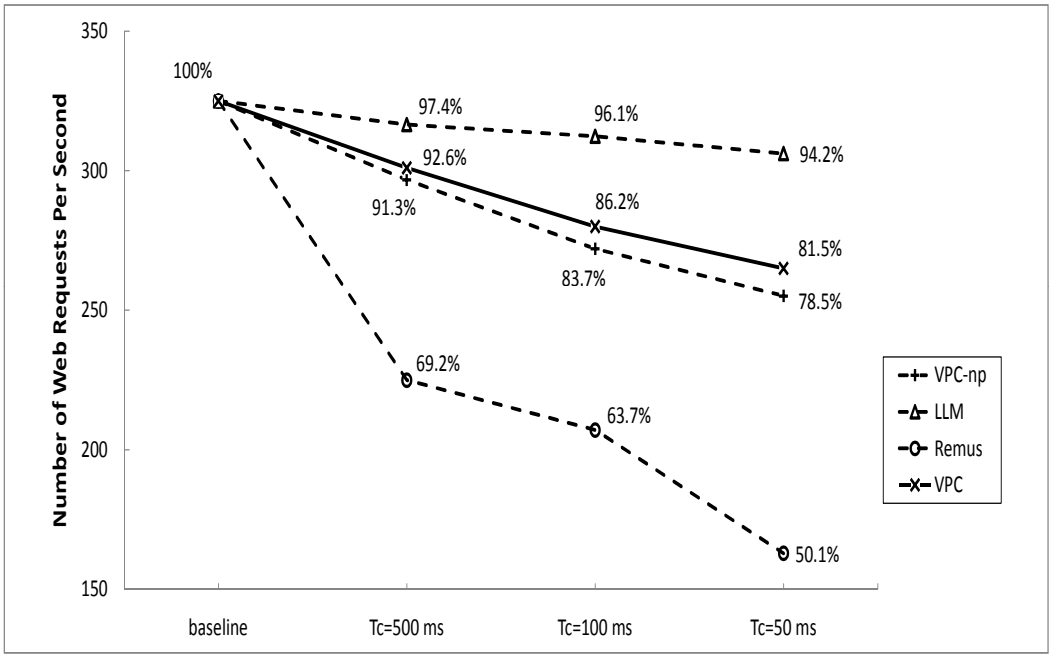


Figure 4.5: Impact of VPC on Apache web server throughput.

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 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 larger 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.

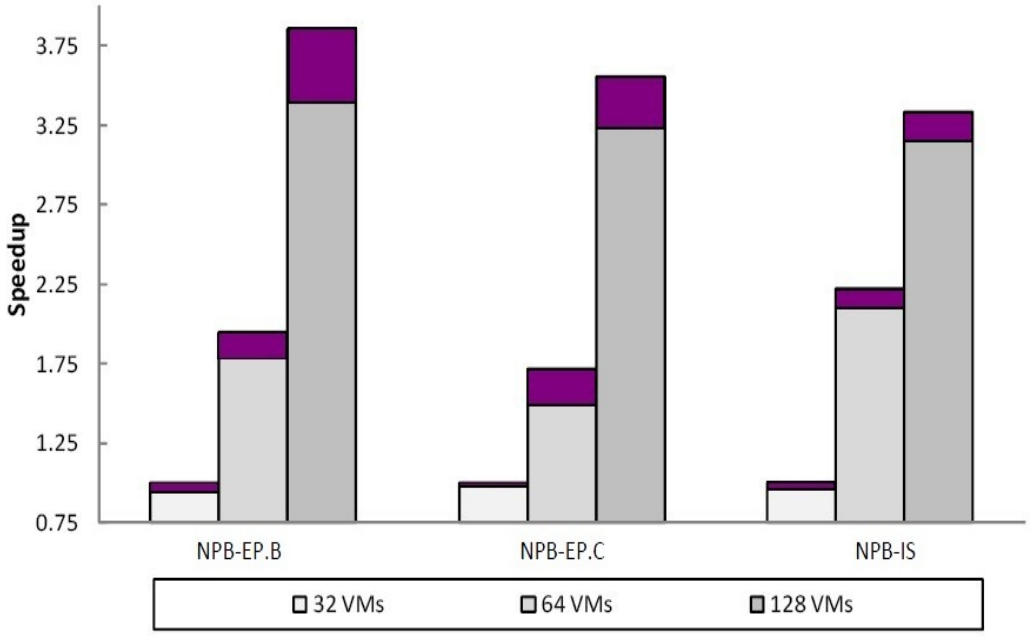


Figure 4.6: Checkpointing overhead under NPB-EP with 32, 64, and 128 VMs.

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%.

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 45% and reduces the entire VC downtime by as much as 50%, with a performance overhead that is less than 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

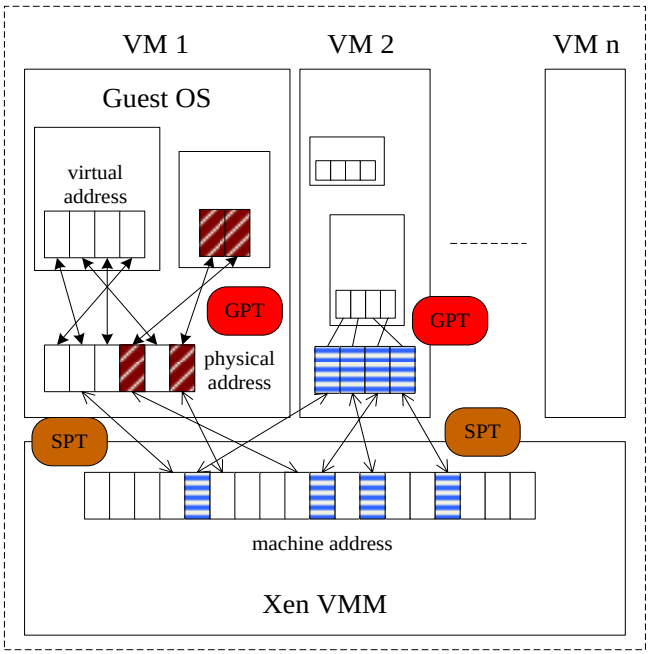


Figure 5.1: Memory model in Xen.

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

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.

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 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 [12]. 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 [17].

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

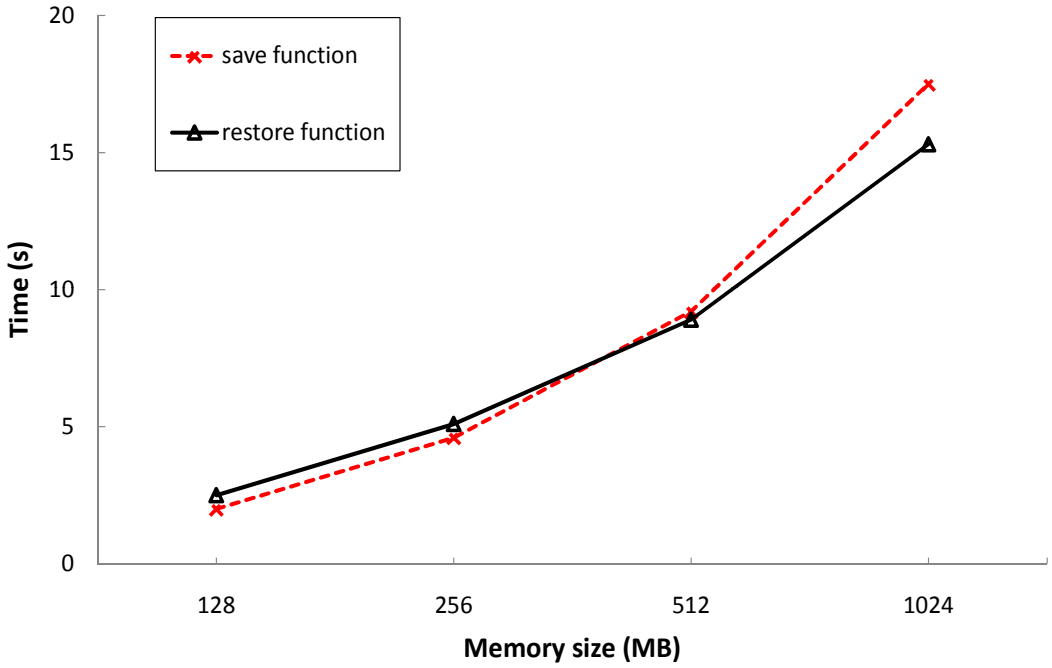


Figure 5.2: Native Xen's saving and restoring times.

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 hardware 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 [67, 78], 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 triggers 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."

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.

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 [37], 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.

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%.

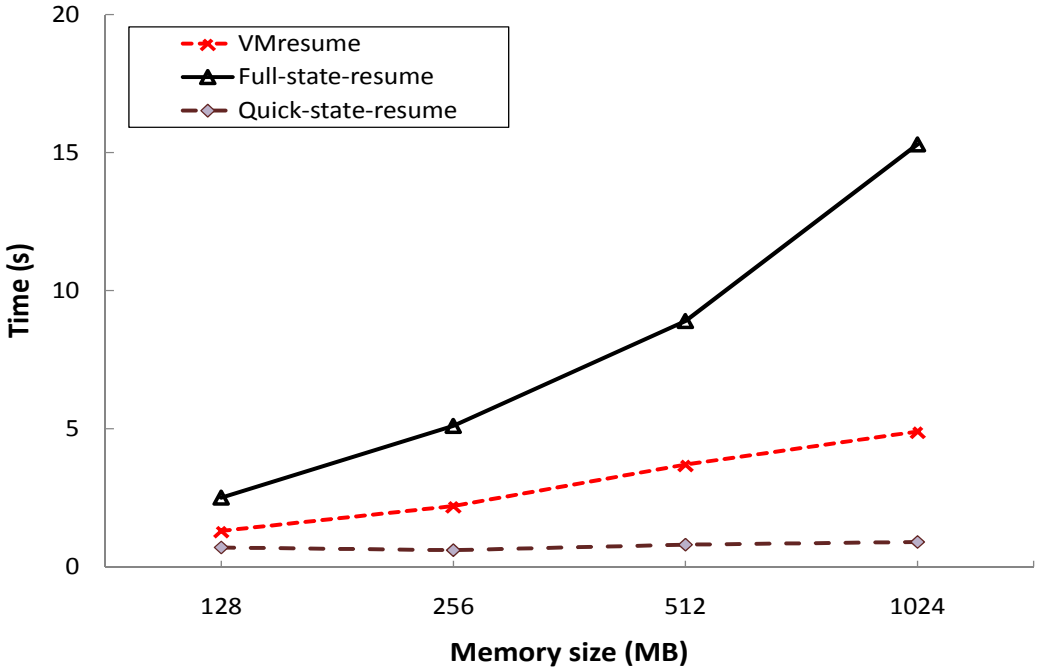


Figure 5.3: Time to resume a VM with diverse memory size under different resumption mechanisms.

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 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-

Figure 5.4: Performance after VM starts.

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 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 12 seconds to resume normal activity. Compared with Quick-state-resume, under VMresume, the VM endures performance degradation only for about 3.1 seconds, which reduces the VM's unusable time by as much as 73.8%.

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 57% and VM's unusable time is reduced by as much as 68% over native Xen's save/restore mechanism.

Chapter 6

Hybrid-Copy Speculative Guest OS Live Migration without Hypervisor

In this chapter, we first introduce the design of HSG-LM as well as the key methodology applied in HSG-LM. We then present the optimization of HSG-LM by integrating the speculative migration and the workload-adaptive design. Finally we show our evaluation results by comparing with other competitors.

6.1 Design and Implementation of HSG-LM

There are two main methods to migrate an execution entity (process, application or VM) from a source host to a target machine: stop-and-copy migration and live migration, as shown in Figure 6.1. Stop-and-copy migration requires that the running entity must be stopped on the source prior to copying its data and state to the target. After transferring data and state, the entity will resume on the new location. During the transfer period, the execution entity is no longer running, leading to a long waiting time, shown as "Downtime I" in Figure 6.1. On the other hand, live migration means replicating an executing entity (especially a VM, which is the focus of this paper) from one physical host (source) to another physical host (target), while it is continuously running. Figure 6.1 shows that only newly-updated states (e.g. memory pages) are transferred during each migration epoch, while the VM is still running and generating new updated states. Here, the downtime refers to the time from when the VM pauses on the source until it resumes on the target, shown as "Downtime II". A direct observation is that the Downtime II takes only a small portion of the total migration time and is shorter than Downtime I, meaning that the running applications in the VM are less influenced by the migration mechanism.

Thanks to the short downtime of live migration, it could be applied to facilitate proactive maintenance. VM live migration is widely used for load balancing, in which a workload is shared among VMs on different hosts, in order to optimize the utilization of available CPU resources. Because

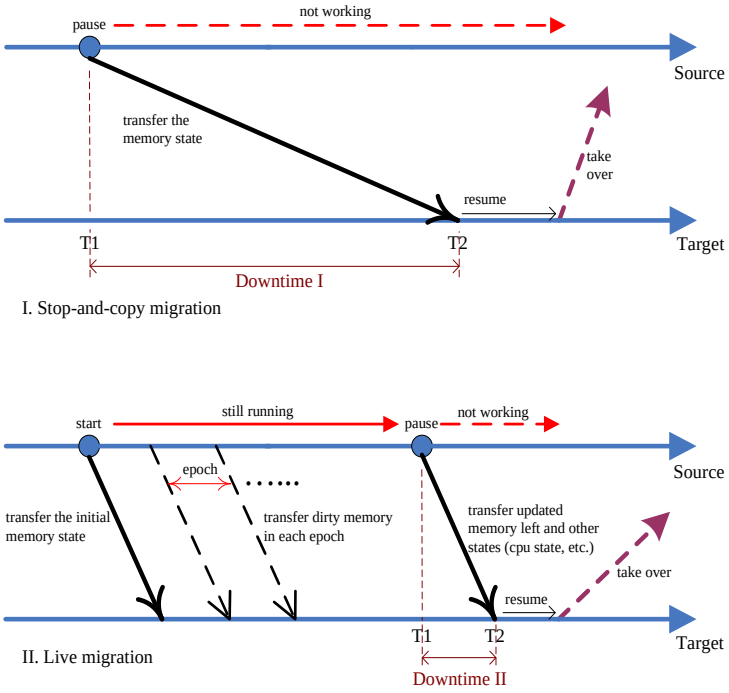


Figure 6.1: Comparison of two migration methods.

live migration is a more efficient mechanism compared with the stop-and-copy methodology, it is now widely supported by several virtualization environments [5, 12, 95], to move guest VMs from one hypervisor to another, usually with a short downtime of less than a second. When performing VM live migration, the hypervisor takes care of tracking the dirty memory pages and transferring them to the target. However, there are two main disadvantages in this migration mechanism (we discuss the first in Section 6.1.1 and the second in Section 6.1.2).

6.1.1 Migration without Hypervisor

In traditional live migration, the hypervisor is responsible for keeping track of the dirty memory pages, as well as creating a new VM on the target host and coordinating the transfer of the VM state between the two hosts. However, one limitation comes from the fact that the migration process totally relies on the hypervisor throughout the entire migration phase. Specifically, in order to track the dirty memory pages, the hypervisor needs to record the dirty map, which is kept inside the host OS, translate each address, find the corresponding pages, and make the transfer. This hypervisor-central model, as stated in the Introduction, brings a potential security problem: if the hypervisor is attacked successfully by a malicious party, the attacker can easily inspect the memory, expose confidential information, modify the running software in the guest VM or even transfer the guest VM to a remote untrusted site.

An alternative is to perform live migration inside the guest OS. The migration mechanism can still apply the pre-copy procedure (like in [32]) as well as others, and all the tracking and transferring work of memory pages will be done by the guest OS itself, not by the hypervisor. However, this approach is complex to implement in existing operating system software. Difficulties arise in transferring all the consistent states from the running guest OS. Inside a running guest OS, the memory pages can be grouped into three categories while migrating:

1) user-space memory pages;

2) kernel-space memory pages;

3) migration-dependent memory pages (that hold the migration data structures).

For the last type of memory, because the guest OS is still running in order to transfer its final state, it generates new dirty memory pages through the migration mechanism itself. Therefore, the pages that are used by the guest OS to track and transfer other dirty pages are impossible to freeze and migrate. Hansen *et al.* [32] solve this problem by partitioning the final migration epoch into two stages: the pre-final stage and last stage. In the pre-final stage, they created a shadow buffer that is updated with the current dirty pages, and in the last stage, only the data in the shadow buffer is transferred to the target host by the hypervisor. They verified that a consistent final view of the guest VM may be achieved by performing this resend-on-write followed by a copy-on-write method. However, the new shadow buffer still creates new memory overhead. Moreover, the creation and other operations of the buffer are within the final epoch, which means that these operations increase the downtime as well. Instead of re-implementing this mechanism, and in order to avoid any further overheads, we apply a hybrid-copy migration methodology. After the necessary state of the VM is transferred to the target and the VM is resumed there, the memory pages of the source VM are always transferred per request from the target VM, and only the newest updated copy is fetched and copied during the migration. We present the details of our hybrid-copy in Section 6.1.2.

We set up a new page fault handler in the guest OS to track the dirty memory inside the OS itself (on the target host) and subsequently generate transfers. Therefore, the migration does not rely on the hypervisor to manage the memory pages anymore. For the new running VM, after resuming on the target host, all memory pages are set as read-only. Thus, if there is any write to a page, it will trigger a page fault. The page fault is then reported to the new page fault handler, and we log the change of this page to the dirty bitmap kept by the guest OS. The set of dirty pages can be implemented by using different data structures; here we choose a bit vector because it is easy to estimate the space requirements and memory overhead is minimal.

Note that when a page fault occurs, this memory page is set as writeable, but the page fault handler does not save the modified page immediately, because there may be another new write to the same page during the same interval. Instead, HSG-LM records the address of the faulting page in the dirty bitmap and removes the write protection from the page so that the application can proceed with the write. At the end of each migration, the dirty bitmap contains the address translation

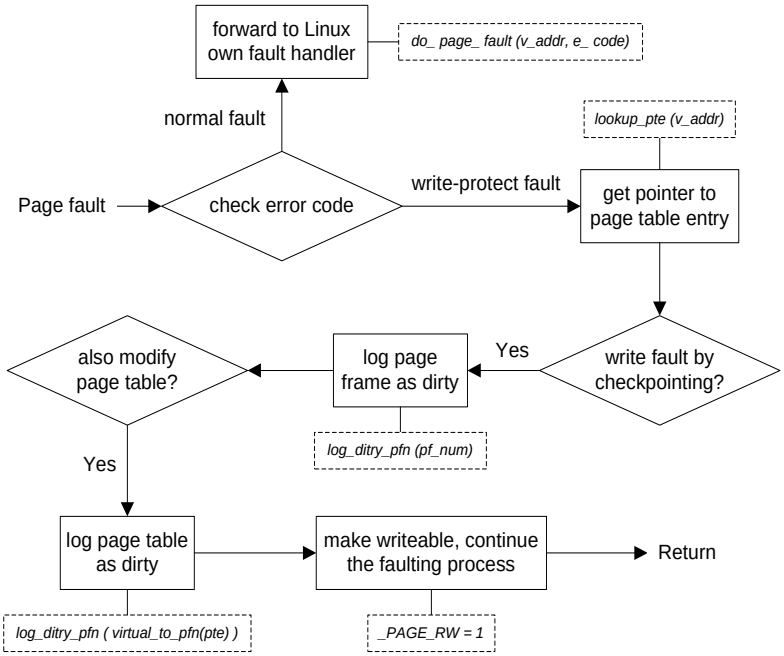


Figure 6.2: Workflow of the new page fault handler.

info for all the pages that were modified. The workflow of the new page fault handler is shown in Figure 6.2.

There is another advantage to installing a new page fault handler inside the guest OS. In traditional hypervisor-based migration, the hypervisor does not know what really occurs with the applications running because the isolation feature of the VM prevents all outside access. Therefore, compared with the page fault trap mechanism implemented in the hypervisor, our new handler is able to communicate with both kernel and user space, so as to make a more accurate speculative migration. We discuss the speculative migration in Section 6.2.

6.1.2 Hybrid-copy Migration

In the Introduction we report a significant use case that highlights two waiting times, both drawn in Figure 6.3, that are incurred during live migration: downtime and total migration time. As stated there, the downtime reflects the customer satisfaction but the total migration time matters as well, e.g., when dealing with operations management of a data center. Traditional VMM implementations [17, 70] exploit the pre-copy technique, which is organized in epochs and works as follows:

1) In the first migration epoch, the hypervisor on the source host starts a host thread that pre-copies all of the VM's memory pages to the target host while the VM is running.

2) At the end of each subsequent migration epoch, the hypervisor checks the dirty bitmap to determine which memory pages have been updated in this epoch. The hypervisor transfers the newly updated pages only. Meanwhile, the VM continues to run on source host.

3) When the pre-copy phase is no longer beneficial, the hypervisor suspends the VM on the source host, transfers the remaining data (newly updated pages, CPU register and device states, etc.) to the target host, and prepares to resume the VM there.

The traditional pre-copy migration mechanisms work well regarding the downtime measurement, which is usually less than 1 second. However, the downtime takes only a small fraction of the total migration time. From the pre-copy workflow presented above, we observe that if a memory page is frequently updated by some memory-intensive workload, its updated copy will be transferred every time during each migration epoch. Considering that a normal VM may be assigned up to several gigabytes of memory to run the guest OS, this will bring a consistent overhead throughout the entire migration, which leads to unacceptable total migration times. We show this in our evaluation of the original pre-copy mechanism in Section 6.3.3.

Instead of pre-copy, the post-copy migration mechanism can reduce the total migration time. Hines *et al.* [38] propose a basic post-copy framework via demand paging as follows:

1) During post-copy migration, the guest VM is first suspended on the source host until a minimal and necessary execution state (or checkpoint) of the VM (including CPU state, registers, and some non-pageable data structures in memory) has been transferred to the target.

2) Although the entire memory data is still on the source host and no memory pages have been transferred to the target, the VM is still resumed on the target host.

3) Whenever the resumed VM tries to access a memory page that has not been fetched, a page fault will be generated and redirected towards the source over the network, referred to as a network fault.

4) The source host will respond to these network faults by fetching the corresponding memory pages and transferring them to the target host.

In the pre-copy migration, the VM on the source host handles user requests throughout the entire migration process. As opposed to it, the post-copy migration mechanism delegates the user services' response to the VM on the target host. We present the downtime and total migration time for post-copy migration in Figure 6.3 as well. Both downtime and total migration time measurements are the same as that under pre-copy migration. However, with post-copy migration, when the VM starts to resume after a short downtime, it is actually unusable for users for a period of initial time because of the occurrence of too many page faults. This time period is shown as "resume time". In our evaluation, it is still counted in the downtime measurement because during the majority of the resume time, the users' normal activity is impaired.

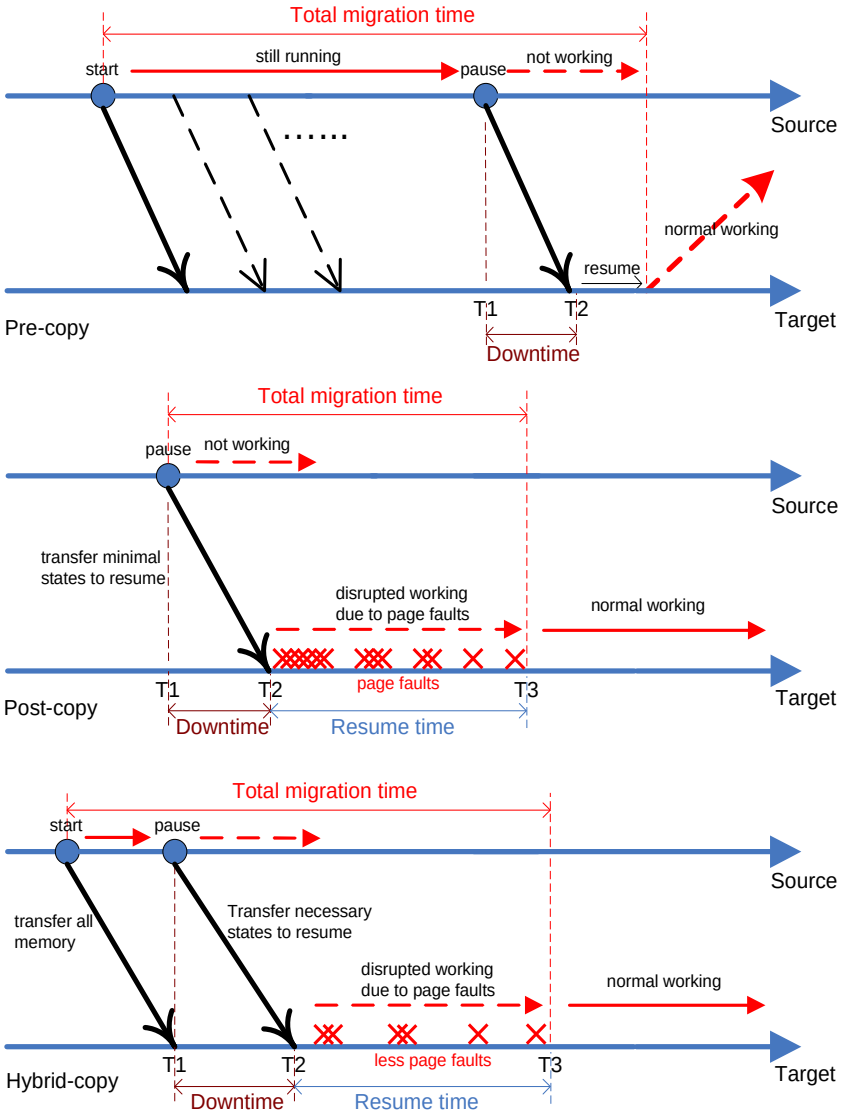


Figure 6.3: Downtime and total migration time measurements.

From Figure 6.3 and the above discussion, we observe that both pre-copy and post-copy migration mechanisms have their own benefits and limitations: short downtime but long total migration time in pre-copy migration, the contrary for post-copy migration, which leads to acceptable total migration times but incurs longer downtime (including resume time). The difference is shown in our evaluation of both original mechanisms in Section 6.3. To reap the benefits of both pre- and post-copy mechanisms and to overcome their limitations, our HSG-LM is the first that implements a hybrid-copy method (introduced in [38] as future work but there is no track of such work).

Our hybrid-copy migration performs a single round of classic pre-copy transfer at the beginning, that is, all the memory pages are copied from the source host to the target host while the VM is still running on the source. Then, following the post-copy approach, the VM is paused on the source host, a minimal set of necessary execution states is transferred to the target, and the VM is resumed there. As it performs a pre-copy round first, this hybrid mechanism eliminates a great number of the page faults that usually occur in the following post-copy round. Especially when running read-intensive workloads, HSG-LM incurs minimal downtime, as with the original pre-copy migration. On the other hand, HSG-LM also supports deterministic total migration time as with the original post-copy migration, especially for the write-intensive workloads, because it ensures that a memory page is transferred at most twice during the entire migration (all the numbers are further discussed in Section 6.3).

6.2 Speculative Migration

6.2.1 Pros and Cons of the Hybrid Design

In our early design, after the first pre-copy round where all memory pages are transferred from source host to target host, we let the VM resume on the target host after loading only the necessary CPU and device states. Then we restore the updated memory data that was modified by the running VM during the first pre-copy round. Whenever the VM needs to access a memory page which has not been updated, it retrieves the corresponding data from the source host and sets up the pages locally. The benefit of this solution is that the VM starts very quickly, and it always keeps running while restoring the memory data. Moreover, since the VM only needs to restore a small number of CPU and device states in order to start, its performance would not be reduced by large VM memory sizes.

Compared with pre-copy migration, we note that the hybrid approach has some disadvantages. With the pre-copy approach, after the VM starts, the VM works as well as it did before the migration occurred. However, with the hybrid-copy approach, the VM appears to be running almost immediately after restoring the basic CPU and device states, and with workloads with few writes, the penalty induced by page faults that must go out over the network is relatively small. However, write-intensive workloads may modify the memory pages with high frequency so that after the first round, most memory pages received by the target host will need to be updated again, so much of

the pre-cached memory data will not actually be usable. This will cause a significant number of network page faults to occur at the beginning, degrading VM performance. Note that the same problem happens in post-copy mechanism too, with an even longer resume time. Our experiments (Figure 6.9) show that using post-copy migration, for a guest VM with 1GB RAM, during the first 14 seconds, the VM runs too slowly to be useful. Almost all of this time was spent on restoring the updated memory data.

To reduce the total migration time, our goal is to run the VM on the target host as early as possible, but to avoid the performance degradation caused by page faults after the VM starts. Our approach is to first resume the VM by loading the necessary CPU/device states, and then when a page fault occurs, to determine the memory pages that have a high probability of being accessed in the near future and to restore these pages from the source host. In other words, for each page fault that occurs, we preload several additional memory pages. By doing so, we reduce the possibility of future page faults and thus reduce the overhead to handle them.

We determine the likely-to-be-accessed memory pages by the principle of spatial locality on virtual address: if a page is updated, its neighboring pages are likely to be updated in the near future. We can rely on our knowledge of a page's address to predict the upcoming memory accesses.

6.2.2 Speculation: Choose the Likely to be Accessed Pages

Our HSG-LM runs mostly in the guest OS: it is ready to run in a fully-virtualized environment, and we can directly exploit the kernel data structures of the virtualized guest OS. Knowing the data structure of the guest OS has the big advantage that for each page used by the operating system, we can easily determine which processes are using it. For each process, we can identify the neighboring pages to the faulting one. On top of that, we know the tasks that are currently running on the guest OS.

With these considerations in mind, and the fact that the bitmap of faulty pages is by design maintained in the guest OS, we developed a set of speculative methods aimed at improving the performance of our HSG-LM. The basic idea behind our techniques is to transfer not one page per dirty fault, but a bulk of pages (on-demand pre-paging), in order to decrease the number of network transfers between the source and the target host, shortening the waiting times in the migration. Also, in HPC, the network latencies are still an order of magnitude greater than for a multiprocessor interconnect, so reducing the number of network transfers will increase performance. To improve the user experience, we propose that the pages to be transferred cannot be just chosen randomly or like in [38] relying on the previous access pattern, from the dirty bitmap. Instead, we choose to transfer first the pages related to tasks that the user is interacting with that are not necessarily related to the previous access pattern. This turns out not to be straightforward but instead to require knowledge of the Linux kernel internals (task's memory related data structures) that we briefly summarize in the following for completeness. Figure 6.4 provides an illustrative sketch.

The Linux kernel, like every other operating system that runs on virtual memory, manages a per-

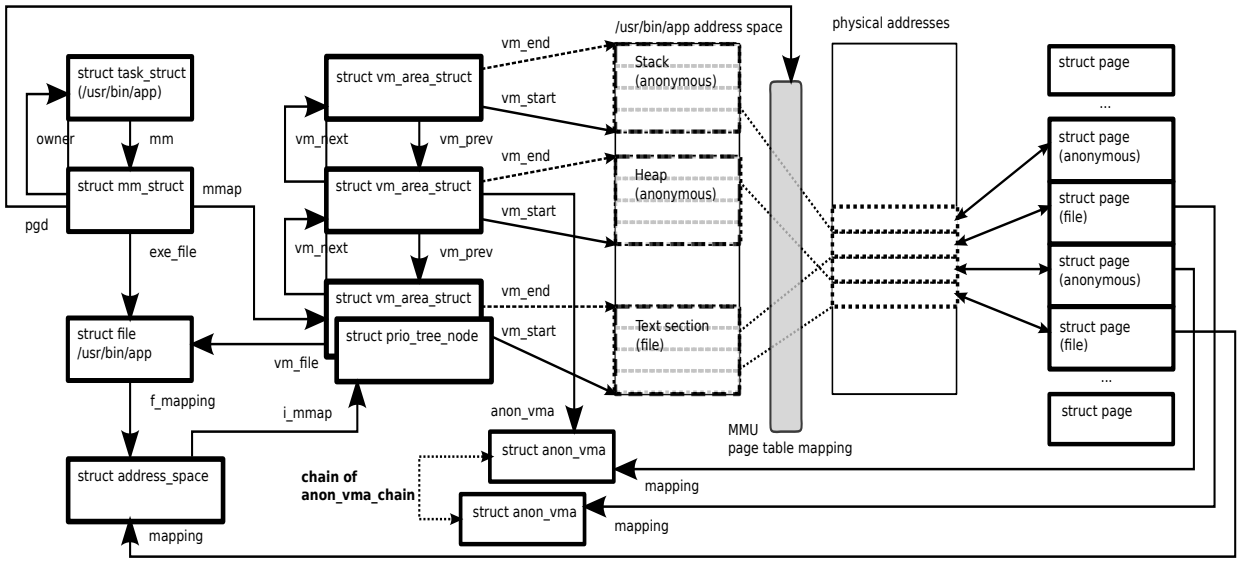


Figure 6.4: Linux kernel data structures involved in task's virtual to physical memory translation and inverse mapping.

task page table structure. The page table structures allow the processor's MMU to translate the addresses from virtual to physical. In Linux every hardware memory page has a corresponding `struct page` used by the system to "keeps track" of its usage and Linux implements `rmap` (reverse mapping) that makes use of the arrays of `struct page` to translate back from physical to virtual addresses. The Linux kernel maintains for each thread a task control structure (`struct task_struct`). For each process, there are mainly two types of pages that are interesting for our research: anonymous pages and file pages. File pages are memory pages that contain an entire file or part of a file that is normally resident on the hard drive. This file can be either an executable or library, or a data file. Anonymous pages contain the stack and the heap. From the `mm` field in a `struct task_struct`, it is possible to access the process' memory descriptor (`struct mm_struct`) that in the field `mmap` contains a linked list of virtual memory area descriptors (`vm_area_struct`). Each virtual memory area refers to a block of anonymous pages or a block of file pages.

We identify the following speculation techniques:

1) randomly choose a group of r linearly contiguous pages around the faulty page from the dirty bitmap, where r is less than a customizable threshold (RANDOM);

2) bulk copy all the dirty pages that belong to a memory-mapped file; a threshold can be specified to bound the number of pages per transfer (FILE);

3) bulk copy all the dirty pages that belong to an anonymous memory mapping (a single task VMA) (ANON);

4) same as the 3) but involves different VMAs; a threshold can again be specified (ANON_MULTI);

In our preliminary experiments, however, we found that a general VM live migration mechanism does not work well in all cases, for example, when dealing with memory read intensive workload, integrating speculation in the migration actually incurs longer downtime due to the overhead to locate the related pages. Therefore, we finally implement a workload-adaptive migration mechanism which could apply different migration methods based on the types of the workload. For such workloads which modify the memory in high frequency, the HSG-LM is triggered with speculation framework. Otherwise, the speculation is disabled in the HSG-LM and it will apply the hybrid-copy approach only. We evaluate the workload-adaptive design in Section 6.3.5

6.3 Evaluation and Results

6.3.1 Experimental Environment

Our preliminary experiments were conducted on a set of identical machines equipped with an x86 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz) and 4GB of RAM. We set up a 1Gbps network connection between the hosts and shared the file systems among all the machines in this LAN. We built Xen 3.4.0 and ran a modified guest VM with Linux kernel 2.6.18. The guest OS and the host OS (in Domain 0) ran CentOS Linux, with a minimum of services initially executing in the guest OS, e.g., `sshd`. Domain 0 has 1.5 GB of memory allocated, and the remaining memory was left free to be allocated for guest VMs. To ensure that our experiments are statistically significant, each data point is averaged from twenty samples. The standard deviation computed from the samples is less than 3.4% from the mean value.

We refer to our proposed migration design with speculation mechanism as HSG-LM. To evaluate the benefits of the speculative migration, we compare HSG-LM with our initial migration design without speculation, which we refer to as HSG-LM-ns. Our competitors include different checkpointing mechanisms including the original design of both pre-copy and post-copy migration mechanisms. To make a more clear comparison, we also implement an improved post-copy migration mechanism (post-copy-i) by integrating the pre-paging idea presented in [38]. In their paper, the pre-paging component is provided that the VM's pages are actively copied from the source to the target, instead of waiting for a fault to occur. They claim this could avoid most faults so as to reduce the waiting time. Furthermore, we compare all these results with a self-migration mechanism [32] which triggers the migration from the guest OS itself by following the similar pre-copy method. As opposed to our design, the self-migration mechanism follows the pre-copy strategy. We choose self-migration as another representative of pre-copy migration. While the original pre-copy implementation is included in Xen, the other competitors are not publicly available. Thus, we implemented prototypes for each mechanism and used them in our evaluation.

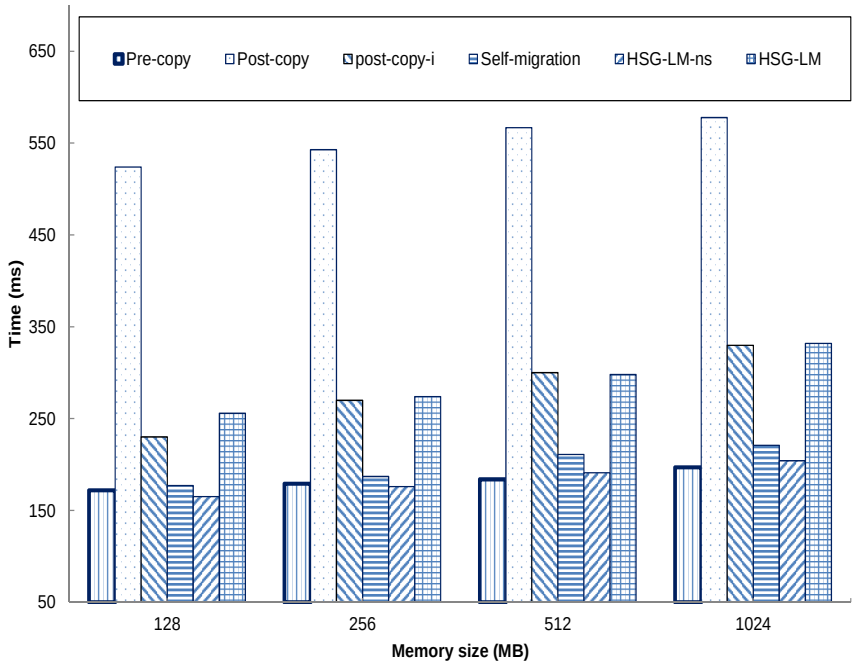


Figure 6.5: Downtime comparison under read intensive workload.

6.3.2 Downtime

We evaluate the performance for migrating VMs running two types of memory-intensive workloads: read-intensive and write-intensive memory operations. We use the Sysbench [9] online transaction processing benchmark, which consists of a table containing up to 4 million entries. We perform either read or write transactions on the Sysbench database to evaluate the performance of all the migration mechanisms. The experiment is conducted on the guest VM with assigned memory from 128 MB to 1GB, in order to investigate the impact of memory size on downtime, total migration time and other performance characteristics. Although this range of memory may seem moderate given the high quantity of RAM available in today's data centers, we found it to be reasonable for understanding the trends.

We first consider the measurement of the VM downtime. The definition of downtime in the Introduction works well under the pre-copy migration mechanism because after VM resumption, the user could resume normal activity immediately. However, when using the post-copy strategy, after the VM has been resumed on the target host, it is actually unusable by the users for an initial period of time due to excessive page faults, so the resume time should be counted into the downtime measurement as well. When measuring the downtime in our experiment, we start measuring the elapsed time when the VM is stopped on the source host, and stop when the resumed VM is fully usable by the users. Figure 6.5 shows the downtime results for the Sysbench-read benchmark for six migration mechanisms with four different sizes of assigned guest memory.

We can make observations regarding the downtime measurements. First, the downtime results of the pre-copy migration mechanisms (including self-migration) are short, while the original post-copy migration incurs the longest downtime. This is because memory updates are rare during the Sysbench-read workload runs; there are relatively few dirty memory pages left in its final migration epoch. On the other hand, by using post-copy migration, there are no memory pages restored in the resumed VM, so thousands of page faults occur, and the resumed VM needs to go back to the source host to fetch the corresponding pages, leaving the VM unusable and leading to a much longer downtime. The improved post-copy mechanism could significantly reduce the downtime by fetching the pages actively instead of on-demand, but this additional process does introduce new overhead, which still leads to longer downtime than pre-copy mechanism.

Second, the downtime results under the HSG-LM design (including HSG-LM-ns) are very similar to those under pre-copy mechanisms. Sysbench-read is a memory read intensive workload. Thus, the guest VM memory is updated at a low frequency. Because we apply a hybrid strategy in the HSG-LM design, it first runs a pre-copy round to transfer all the memory pages and then follows the post-copy strategy to resume the VM on the target host. Memory updates are rare when running this workload, meaning that most memory pages are already restored in the resumed VM after the first round. Therefore, there would not be so many page faults in the HSG-LM mechanism compared with the original post-copy mechanism, and we do see a downtime reduction of roughly 55%.

Finally, compared with our original design without speculation (HSG-LM-ns), HSG-LM incurs longer downtime. As for both HSG-LM-ns and HSG-LM, the downtime depends on the time that is used to handle the page faults after the VM has been resumed on the target host. As the Sysbench-read benchmark only generates minimal dirty memory, page faults rarely occur, so spatial locality does not hold in this case. However, the HSG-LM mechanism will still perform speculative migration, finding neighboring pages of the faulted page and transferring them. This migration is unnecessary and increases the transfer time. Therefore, HSG-LM-ns incurs a smaller downtime than HSG-LM.

Figure 6.6 shows the downtime results in the same cases as Figure 6.5, except running a memory write intensive workload (Sysbench-write benchmark) which updates the guest memory with high frequency. The findings are mostly similar to the previous ones about Figure 6.5, but there are two contrary observations in Figure 6.6:

The downtime results under the HSG-LM-ns design are very similar to the post-copy mechanism results, meaning that the downtime is not as good as that under the pre-copy migration mechanisms. As the guest VM memory is updated at high frequency, after the first pre-copy round in our original design, most of the memory pages restored in the resumed VM have been updated again, so there would still be thousands of page faults and the pre-copy round would not do anything to help.

Another observation is that HSG-LM incurs a shorter downtime than HSG-LM-ns. Although in the HSG-LM mechanism the memory pages restored in the resumed VM need to be updated again after the pre-copy round, the hope is that the speculative migration will transfer the bulk of the required memory pages when one page fault occurs, reducing the frequency of network page faults as well

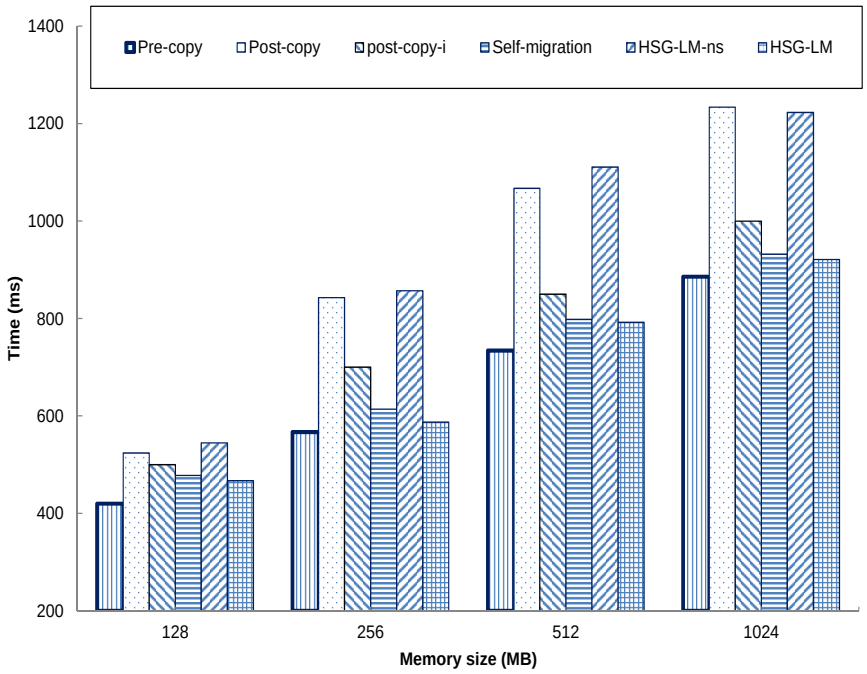


Figure 6.6: Downtime comparison under write intensive workload.

as the total time to make the transfer. From Figure 6.6, we observe that the HSG-LM mechanism achieves downtime performance comparable to that of pre-copy migration mechanisms.

6.3.3 Total Migration Time

We also measured the total migration time, which is from when the migration is triggered to when the resumed VM is fully usable by users. Figure 6.7 shows the total migration time results when running the Sysbench-read workload. We firstly observe that the results of the pre-copy migration mechanisms (including self-migration) are longest, while the original post-copy migration mechanism incurs the shortest downtime. All migration mechanisms need to migrate all the memory pages at least once, either by pre-copying or by page fault handling. However, the pre-copy migration mechanism needs another two rounds, one to determine to stop the pre-copy migration, followed by the final round which transfers the updated memory (although rare) and the necessary CPU and device states. On the other hand, the post-copy migration only needs one round to fetch and transfer all the memory pages based on the page faults, so it achieves a better total migration time.

Second, the total migration time using the HSG-LM design (including HSG-LM-ns) is between that of the pre-copy and the post-copy mechanisms. Because we apply a hybrid strategy in the HSG-LM design, it first runs a pre-copy round to transfer all the memory pages and then follows the post-copy strategy to resume the VM on the target host. With this strategy, there are two

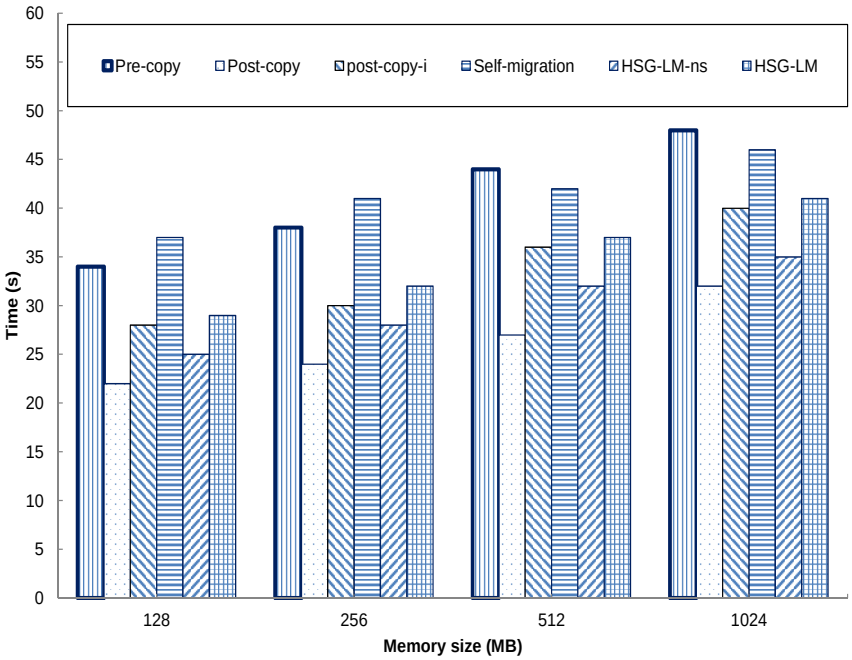


Figure 6.7: Total migration time comparison under read intensive workload.

migration-related rounds in total, and compared to the pre-copy migration mechanisms, the total migration time reduction is roughly 27%.

Finally, the comparison between HSG-LM and HSG-LM-ns is similar to that when measuring the downtime. The reason is also the same: since HSG-LM performs speculative migration, it finds neighboring pages of the faulted page and transfers them in bulk. This transfer is unnecessary and incurs longer transfer time. Therefore, as for the total migration time when running memory read intensive workloads, HSG-LM-ns performs better than HSG-LM.

Similarly, Figure 6.8 shows the total migration time results in the same case as Figure 6.7, except running a memory write intensive workload (Sysbench-write benchmark) that updates the guest memory with high frequency. An obvious observation is that the post-copy migration mechanisms and our hybrid migration mechanisms perform much better than the pre-copy migration mechanism. This is because all the memory pages are only transferred once under post-copy migration mechanism, or at most twice under our hybrid migration. However, under pre-copy migration, if one memory page is frequently updated by the workload, it will be transferred in the corresponding round as long as it's marked as dirty. Therefore, a huge number of memory pages may be transferred several times in the pre-copy migration, leading to a long total migration time. Also, we could observe that HSG-LM performs better than HSG-LM-ns when running a memory write intensive workload, the same as when measuring the downtime. This again verifies that our speculative migration works well when the running application updates the memory in some way, which is more like practical workloads.

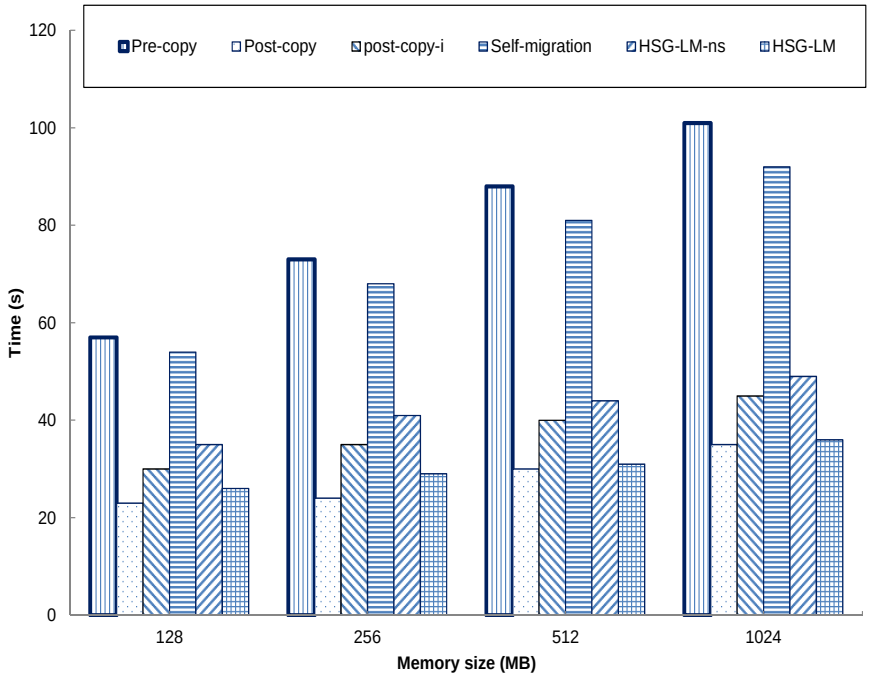


Figure 6.8: Total migration time comparison under write intensive workload.

6.3.4 Performance Degradation after Resumption

Figure 6.9 shows the performance degradation after the VM starts. We configure the VM with 1GB RAM, and first run the Apache web server [2] on it. We set a client on another server that makes requests to load web pages, one immediately after another, simultaneously via a 1Gbps LAN, under four mechanisms (Pre-copy, Post-copy, post-copy-i, HSG-LM). Note that we did not include Self-migration evaluation in Figure 6.9, which exhibits very similar results as Pre-copy migration because it also follows the pre-copy approach during migration. Note that we did not measure the resume time directly because it's hard to define the exact ending time. As an alternative measurement, because 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 resumes. We set a threshold that all the throughput numbers under the threshold are recognized as performance degradation.

We observe that under pre-copy migration mechanism, the VM starts to work immediately after resuming, with no obvious performance degradation. However, the trade-off with the pre-copy migration mechanism is that it takes a long total migration time (tens of seconds) to transfer the memory page, which is especially bad when running memory write intensive workload. On the other hand, with the post-copy migration mechanism, 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 its normal activity. The improved post-copy migration performs better, but it still suffers degradation due

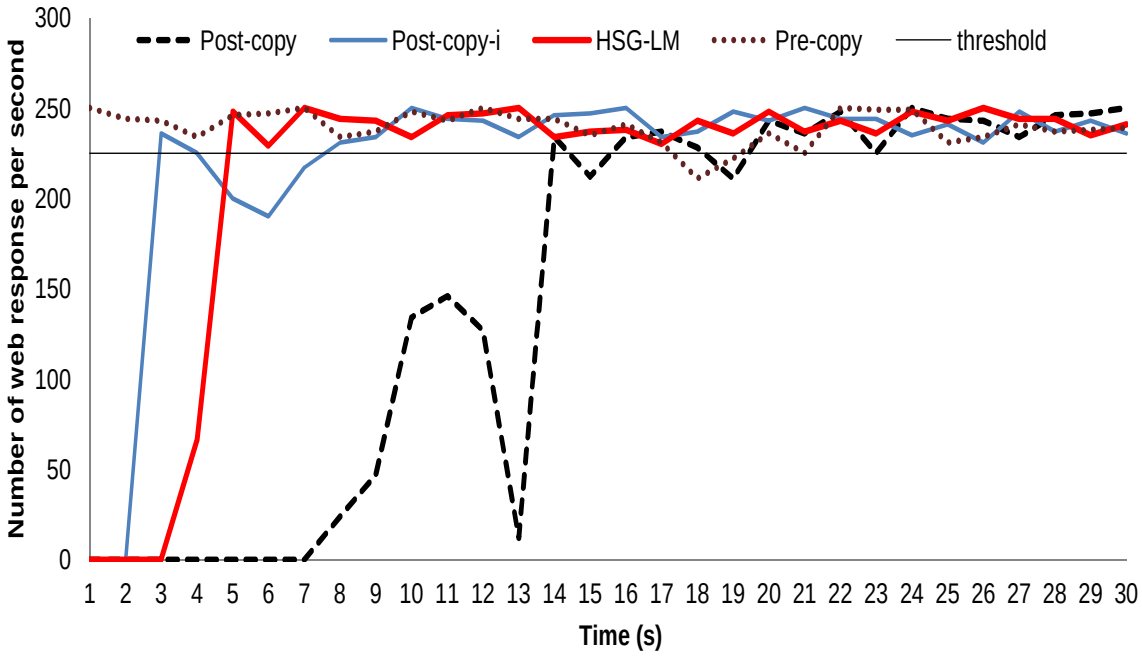


Figure 6.9: Performance degradation after VM resumes.

to the page faults at the beginning period. Compared with post-copy migration, under HSG-LM, the VM endures performance degradation for only about 4.5 seconds, which reduces the VM's unusable time by as much as 68%.

6.3.5 Workload-Adaptive Evaluation

Our final experiment evaluates the effectiveness of our workload-adaptive migration in Table 6.1. Using Sysbench [9], we set mixed transactions in the benchmark, e.g., 25% read transactions and 75% write transactions and understand the trends under different migration mechanisms. We still assign 1GB RAM for the guest VM. The key in our workload-adaptive design is whether to trigger the speculation or not, when dealing with different types of workloads. In this evaluation, there are two extreme cases regarding the workload, one with totally read transactions and another with totally write transactions. We observe that in the downtime evaluation, the original pre-copy migration shows the best performance since it's designed to provide the minimal downtime. Nevertheless, HSG-LM always has the second shortest downtime in all cases, which means HSG-LM is an adaptive approach for both memory read and write intensive workloads, as well as the workload with mixed read and write transactions. We observe the similar result regarding the total migration downtime, i.e. the original post-copy migration leads to the best performance while HSG-LM is the second best in all cases.

Sysbench	100%Read,0%Write	75%R,25%W	50%R,50%W	25%R,75%W	0%R,100%W
Downtime(ms)					
Pre-copy	179	331	564	765	866
Post-copy	522	604	789	979	1250
post-copy-i	313	444	647	891	1003
Self-migration	198	392	601	821	923
HSG-LM	**184**	**360**	**591**	**808**	**919**
Total migration time(s)					
Pre-copy	47	62.1	71.7	94,8	103.9
Post-copy	30.7	31.1	32.3	34.3	35.5
post-copy-i	40	41.5	44	46.6	49.2
Self-migration	45.1	58	69.9	88	92.8
HSG-LM	**34.5**	**34.8**	**35.4**	**36.1**	**37.2**

Table 6.1: Downtime and total migration time comparison under Sysbench with mixed operations.

6.4 Summary

We present a novel live migration technique called HSG-LM, which also aims to provide short waiting time to whoever is responsible for triggering the VM migration. HSG-LM is implemented in the guest OS kernel in order to not rely on the hypervisor throughout the entire migration process. HSG-LM exploits a hybrid strategy that reaps the benefits of both pre-copy and post-copy mechanisms. Furthermore, HSG-LM integrates a speculation mechanism that improves the efficiency of handling post-copy page faults. From our evaluation on different real-world workloads (Sysbench, Apache, etc.), the results show that HSG-LM incurs minimal downtime as well as short total migration time. Moreover, compared with competitors, HSG-LM reduces the downtime by up to 55%, and reduces the total migration time by up to 27%.

Chapter 7

Adaptive Live Migration to Improve Load Balancing in Virtual Machine Environment

In this chapter, we first present the design and implementation of our load balancing mechanism, as well as the load balancing algorithm. We then presents the optimization of DCbalance by integrating the workload-adaptive live migration design. Finally we show our evaluation results by comparing with other competitors.

7.1 Architecture

Out prototype includes several machines which are linked by network. We choose one machine as the control node and let it run nothing but the load balancing strategy. The other machines are used as the computational nodes with several VMs running on. The entire architecture is shown as Figure 7.1.

The load balancing mechanism is centralized: the control node runs the cloud management software that controls all the computational nodes running the VMs. The control node has three main components:

1) Load Collection: To make a correct decision, the control node has to know the current load status of all the computational nodes. This is achieved by each computational node sending its own info to the control node (through its Load Monitor) periodically.

2) Decision Maker: This is the key part of the load balancing strategy. Based on the load information from the Load Collection component, the Decision Maker will generate the migration decision by either referring to the history records or proposing new migration case. The framework of making the load balancing decision will be detailed discussed in Section 3.2

3) Migration Trigger: Because we implement an adaptive migration mechanism which will apply

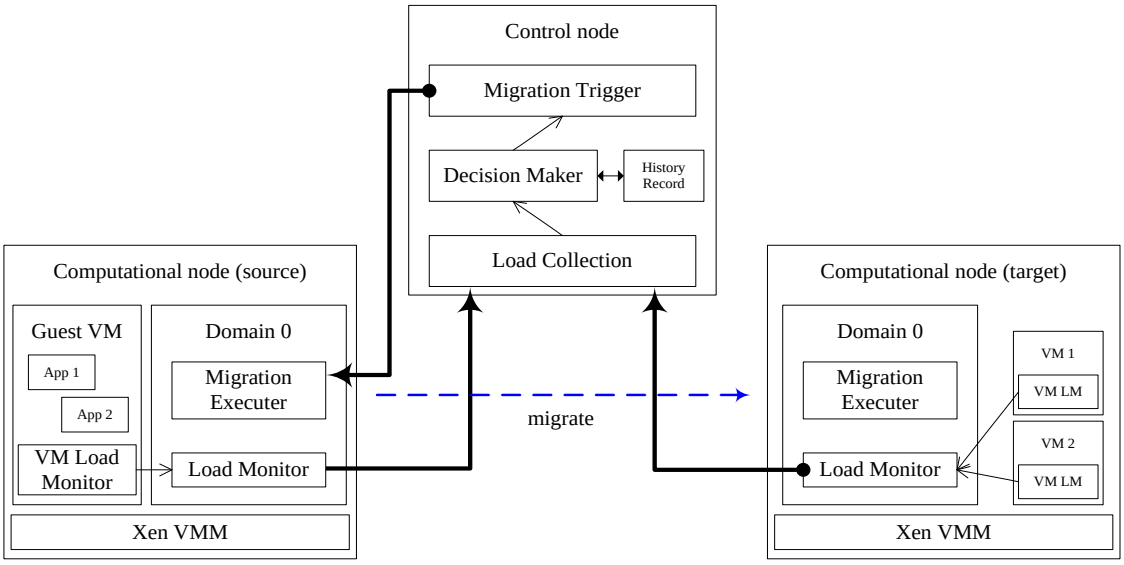


Figure 7.1: The architecture of DCbalance.

specific method for different kinds of application. Therefore, after getting results from Decision Maker, the Migration Trigger will further determine which migration method will be used to carry out this decision. Then it will cooperate with Migration Executer on the computational node to finish the migration.

On the other side, to complete the load balancing strategy, there are also two components implemented on the computational nodes:

1) Load Monitor: Each computational node has this Load Monitor which not only monitors the resource usage on the physical machine, but also the information on all VMs running on this machine. Because in this paper we only focus on CPU load-balancing and memory load-balancing, so the Load Monitor mainly collect CPU and memory utilization records.

2) Migration Executer: It get migration decision from Migration Executer on control node and complete the migration as required. As our work is based on Xen, this component is placed in the trusted and privileged Domain 0 because it needs to schedule the migration outside the guest VM (which is in Domain U).

7.2 The Framework to Generate Load Balancing Decision

From a global point of view, when a node is heavy-loaded while others are light-loaded, the imbalance problem emerges and it's time to generate a load balancing decision. The dynamic load balancing strategy continuously attempts to reduce the differences among all the nodes, by migrat-

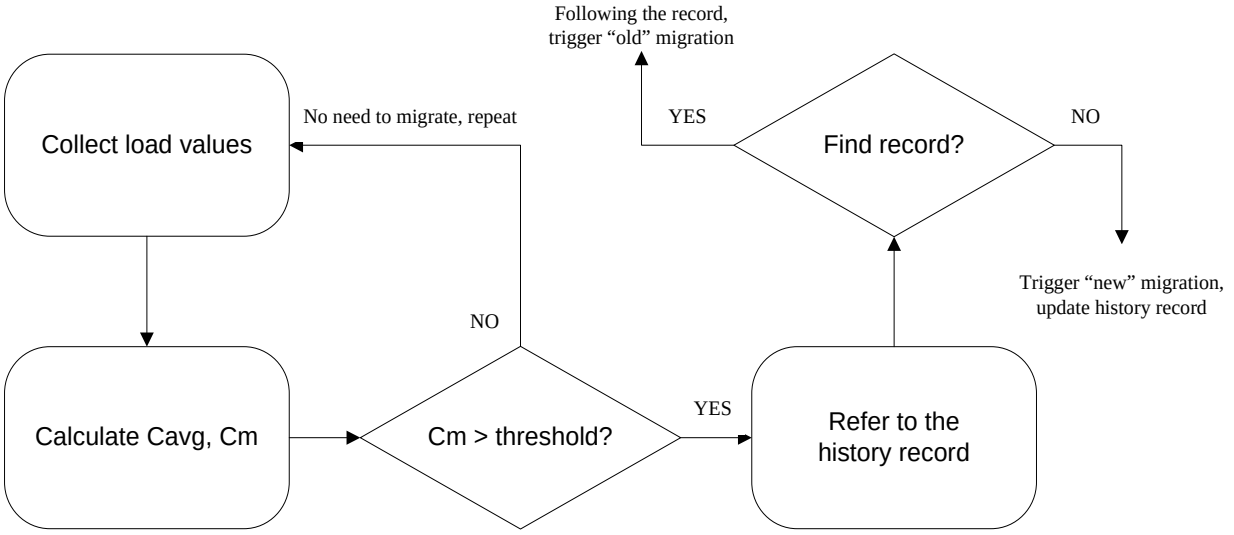


Figure 7.2: Execution flow of DCbalance.

ing VM from heavy-loaded to light-loaded nodes. Figure 7.2 breakdowns the process into steps:

7.2.1 Collect the Load Values on Each Computational Node

In this paper we focus on CPU load-balancing and memory load-balancing of different nodes in a VM environment. We model each node as a Linux box, which provides different interfaces to live record CPU and memory load information for node and VMs.

There are two ways to measure the load value at user level, by either using `sysinfo` system call (`/usr/include/sysinfo.h`) or reading the information residing in the `/proc` file system. For CPU information we check the total `jiffies` (1/100ths of a second) that the processor spent on each VM and the physical machine as well, and compute the utilization percentage for each part. For memory information we record the page operations per second, for example, the number of the page faults requiring a page to be loaded into memory from disk.

7.2.2 Determine Whether to Trigger the Live Migration

After the control node collected the load values from all computational nodes the total and average utilizations of all the machines involved in the load balancing are computed. We adopt a threshold based strategy for deciding when virtual machines should be migrated between the nodes. The VM live migration will be triggered if C_m, the mean value of the sum of the maximum and minimum distances (respectively $C_{diff_{max}}$ and $C_{diff_{max}}$) from the average utilization (C_{avg}), is greater than a threshold T.

$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} C_i$
$C_{diff_i} = C_i - C_{avg}$
$C_{diff_{max}} = \max_{i\in\{1,..,n\}} C_{diff_i}$
$C_{diff_{min}} = \min_{i\in\{1,..,n\}} C_{diff_i}$
$C_m = (C_{diff_{max}} + C_{diff_{min}})/2$

Figure 7.3: C_i is the total CPU utilization of the i^{th} computational node, $i \in 1, .., n$.

7.2.3 Schedule the Live Migration by Checking the Load Balancing History Record

Whenever a migration is triggered the control node checks the history record for a similar CPU utilizations scenario, i.e. a similar load distribution on the computational nodes. Note that the same C_{avg} doesn't ensure the same CPU utilizations scenario. If a previous record exists, we can schedule the VM live migration by choosing the same source and destination nodes. If several similar records exist, we just follow the latest record and schedule the migration. Otherwise if we can't find such a record, meaning that the current situation is totally new, then the nodes with C_i value close to $C_{diff_{max}}$ are used as sources and the nodes with C_i value close to $C_{diff_{min}}$ are used as destinations of the live migration. After the migration, we also add this situation as a new entry and add into the history record.

7.3 Workload Adaptive Live Migration

Differently from previous works, we aim to provide load balancing with minimal downtime by virtual machine live migration. We focus on effective balancing the usage of two kinds of resources: the CPU and memory. In our preliminary experiments, however, we found that a general VM live migration mechanism doesn't work well in all cases, for example, when dealing with memory-intensive application. Therefore, we implement a workload-adaptive migration mechanism.

7.3.1 For General Application Load Balancing

By "general" we mean that there is no memory-intensive workload running in the guest VM, for example, when running the Apache benchmark [2]. Our proposal is implemented by exploiting the pre-copy technique [17, 70], however, we compress the dirty memory data before transmitting. Compressed dirty data takes less time to be transferred through the network. In addition, network traffic due to migration is significantly reduced when less data is transferred between two computational nodes. The technique is summarized as following.

1) In the first migration epoch, on the source node, all the memory pages of the selected VM are

transmitted to the target node while the VM is still running.

2) For subsequent migration epochs, at the end of each migration epoch, the mechanism checks the dirty bitmap to determine which memory pages have been updated in this epoch. Then only the newly updated pages are transmitted. The VM continues to run on the source node during these epochs.

3) Before transmitting in every epoch, the dirty data is checked for its presence in an address-indexed cache of previously transmitted 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 (RLE). Only the delta from a previous transmission of the same memory data is transmitted.

4) For the memory data which is not present in the cache, we apply a general-purpose and very fast compression technique, zlib [10], to achieve a higher degree of compression before transmitting.

5) When the pre-copy phase is no longer beneficial, the VM is stopped on the source node, the remaining data (newly updated pages, CPU registers and device states, etc.) is transmitted to the target node, and prepares to resume the VM there.

7.3.2 For Memory-Intensive Application Load Balancing

When dealing with the memory-intensive applications that update memory with high frequency, the above solution doesn't work because it leads to unacceptable overhead. Assume a memory page is frequently updated by some memory-intensive workloads, its updated copy will be compressed and transferred every time during each migration epoch. It would become worse considering that a normal VM may be assigned up to several gigabytes of memory to run the guest OS. Therefore, for load balancing workload characterized by memory-intensive applications, we propose another migration design showed as following.

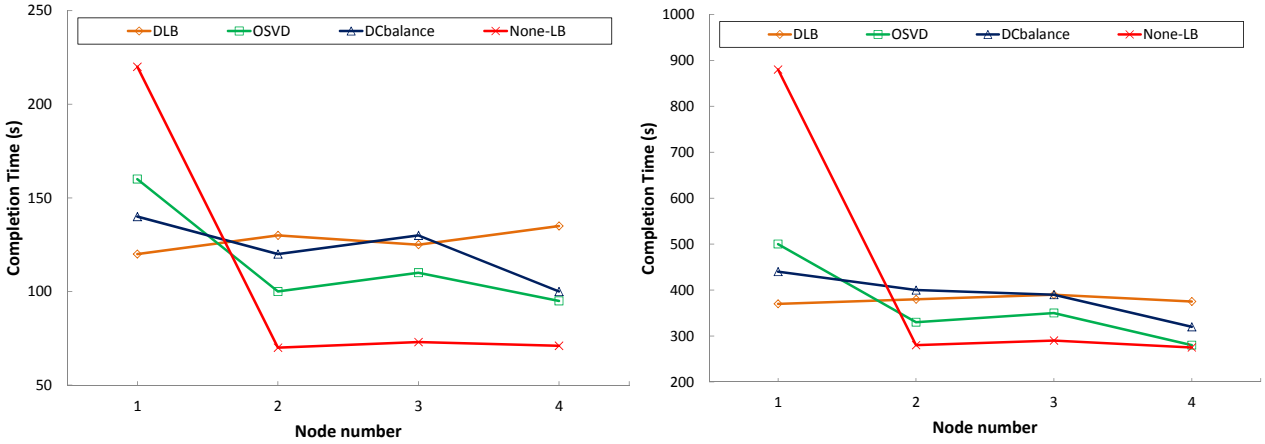
1) Same as previous design, initially all the memory pages of the selected VM are compressed and transmitted from source to the target node while VM is still active.

2) Then the guest VM is suspended on the source node until a minimal and necessary execution state (or checkpoint) of the VM (including CPU state, registers, and some non-pageable data structures in memory) has been transmitted to the target and resumed there.

3) The VM is also resumed on the source node.

4) On the target node, whenever the resumed VM tries to access a memory page that has not been updated, a page fault will be generated and redirected towards the source over the network. The source node will respond to these faults by fetching the corresponding memory pages, compress and transmit them to the target node.

5) The compress method is the same as in the previous design.



(a) Running NPB-EP.C program (b) Running NPB-EP.D program

Figure 7.4: Workload completion time comparison.

When the VM applications are mostly characterized by memory-intensive workload, this design ensures that each memory page is compressed and transmitted at most twice. Additionally, if the workload is read-intensive most memory data is transmitted once.

7.4 Evaluation and Results

Our experimental environment includes five machines, one is used as the control node, and the other four machines (computational nodes) are running the guest VMs. Each computational node contains a 2.3 GHz CPU with 4GB or 6GB RAM. All machines are connected by 1 Gigabit Ethernet switch. Our load balancing strategy and migration mechanisms are based on Xen(3.4.0). The operating system (installed on each of the five machines and guest VMs) is Centos 5.2 with the Linux 2.6.18.8 kernel. We refer to our workload-adaptive live migration mechanism as *DCbalance*. Our competitors include another load balancing mechanism, called OSVD [55], which is also based on VM live migration and integrates performance prediction mechanism. Another competitor, referred to as DLB, implements a dynamic load balancing algorithm [42] which is deployed on Xen's original live migration mechanism. We split the performance evaluation of our load balancing strategy and migration mechanism in the following.

7.4.1 Load Balancing Strategy Evaluations

For such evaluation we run the MPI version of the NPB benchmark [6], we present here the results about the EP program, which is a compute-bound benchmark with few network communications [6]. Initially we let each computational node run the same number of VMs. The total

Table 7.1: Workload percentage after migration and triggering latency comparison.

LB strategy	#1 Load	#2 Load	#3 Load	#4 Load	Triggering latency
DLB	24%	25%	26%	25%	2.2s
OSVD	27%	35%	17%	21%	3.1s
DCbalance	28%	26%	23%	21%	0.63s
DCbalance-nh	29%	25%	27%	19%	1.09s

workload is divided and assigned to all the VMs on the 4 nodes. We let *node1* execute 50% of the total workload while the remaining 3 nodes execute the rest (16.6% each). Obviously *node1* is overloaded. Each node run 8 VMs with 256MB assigned as guest memory. And initially the workload assigned to each node is balanced as the other VMs on the same node. We use 10% as the threshold T in this experiment.

We first compare the three load balancing strategies with the case that no load balancing strategy is used. Figure 7.4a shows the completion times when running class C problem size in NPB-EP benchmark, "none-LB" is the case in which no load balancing strategy is applied to the workload. The x-axis is node numbers while the y-axis is the workload completion time in seconds. As said before, initially each node has 8 VMs on average and *node1* is overloaded with half of the total workload. Without loading strategy, *node1* took close to 220 seconds to complete the workload while all other nodes took only about 70 seconds each. When applying any of the three load balancing strategies, however, all the nodes exhibit relatively consistent completion times because the VM(s) on the overloaded node were already migrated to other nodes. We observe that compared with other published load balancing algorithms, DCbalance achieves similar performance. Figure 7.4b shows the same evaluation but all the test cases are running larger problem. Our load balancing strategy can still provide comparable performance when dealing with the larger problem.

We also compare the triggering latency of each load balancing strategy, i.e. the elapsed time between the collection of load information and the time when the decision is generated. We compute the workload percentage as well as the VMs number on each node after the migration finished. We keep the recently 20 load balancing entries as the history record. To evaluate the benefits of the history record, we add another method, which is our initial load balancing design without storing the history record, which we refer to as DCbalance-nh. Table 7.1 shows the results under four load balancing strategies. From the numbers we observe that the DLB algorithm outperforms the others but it did need longer time to generate the decision. The OSVD system incurs even more time due to the prediction overhead, and also we can observe that its final decision is not so balanced. Compared with these two strategies, the main benefit of our strategy is the short time needed for finding the solution, reducing the triggering latency by as much as 79% compared with OSVD system. Besides we also observe that it is an efficient method as the migration decision is comparably fair. Besides, from the difference between DCbalance and DCbalance-nh we conclude that by referring the history record, we can quickly generate the migration decision.

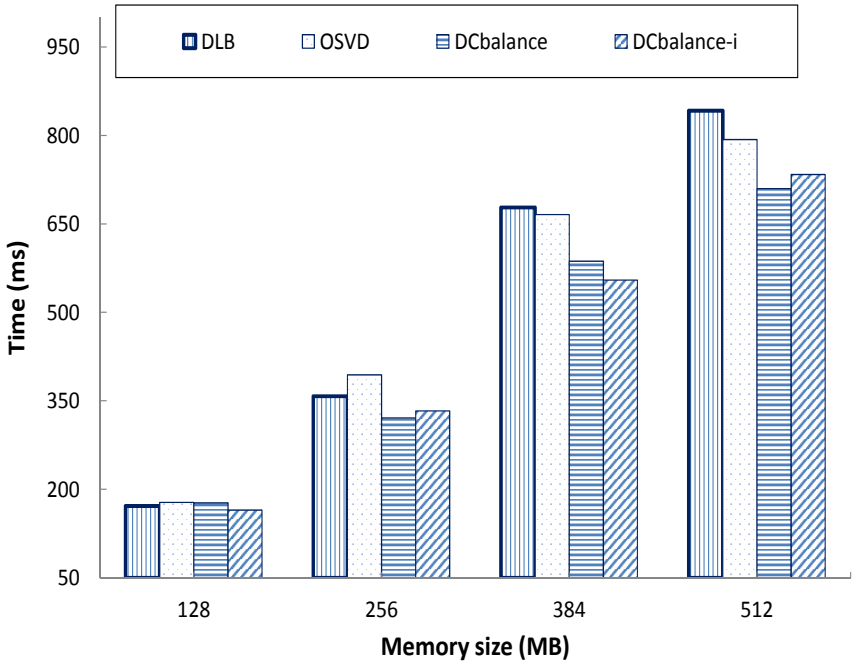


Figure 7.5: Downtime comparison under Apache.

7.4.2 Migration Mechanism Evaluations

We evaluate the performance of migrating VMs running two types of workloads: Apache [2] as a "general" workload and Sysbench [9] as a "memory-intensive" workload (e.g., 25% read transactions and 75% write transactions). We let each node run 4 VMs with the same assigned memory size, we conduct the evaluation varying the guest memory size from 128MB to 512MB, in order to investigate the impact of memory size on the downtime and other performance characteristics. To verify the effectiveness of our adaptive migration mechanism, we evaluate both our proposed migration solutions on the same test case. We refer to our proposed migration design for general application still as DCbalance but refer to its improved version for the memory-intensive application as DCbalance-i.

Figure 7.5 shows the downtime results for the Apache benchmark for four migration mechanisms with three different sizes of assigned guest memory. We observe that all four mechanisms incurs minimal downtime (within 1 second), however, because our proposed mechanism further reduces the dirty memory data by pre-compression before migration, it leads to lower downtime numbers. As the assigned guest memory goes up, the gap becomes larger, with a 17% reduction under the 512MB case. Another observation is that there is no obvious difference between the results of DCbalance and DCbalance-i. This is because the DCbalance-i is specially improved for memory-intensive applications but the Apache benchmark doesn't update the memory with high frequency.

Figure 7.6 shows the downtime results in the same cases as Figure 7.5, except running a memory-

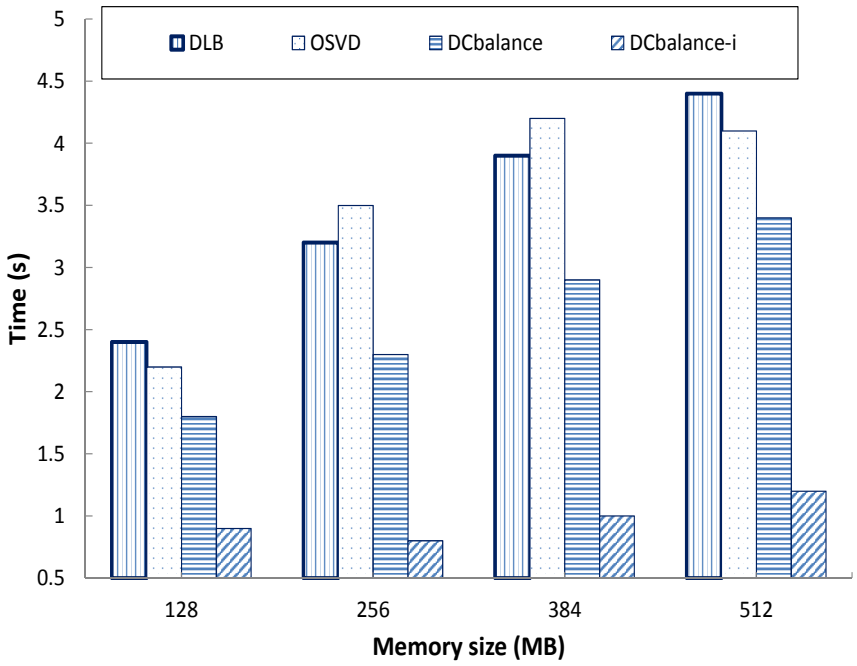


Figure 7.6: Downtime comparison under Sysbench.

intensive workload (Sysbench benchmark) which updates the guest memory with high frequency. The findings are very different from the previous ones in Figure 7.5. The DCbalance still performs better than DLB and OSVD which incur several seconds, it reduces the downtime by roughly 35%. However, the DCbalance-i is the best mechanism which maintains the downtime results around 1s in all cases. The downtime reduction is up to 73% compared with DLB and OSVD.

Under the same cases shown above, we also measure the total migration time, which is from when the migration is triggered to when the resumed VM is fully usable by users. Figure 7.7 and 7.8 shows the total migration time results for the Apache benchmark and Sysbench benchmark, for four migration mechanisms with three different sizes of assigned guest memory. Because downtime is also part of total migration time and our proposed mechanism incurs smaller downtime as shown in Figure 7.5 and 7.6, the findings of total migration time are mostly similar to the previous ones. We observe that our proposed mechanism reduce the total migration time in DLB and OSVD by up to 33% and 38%.

7.5 Summary

We propose a centralized load balancing framework based on past migration cases. Instead of making prediction in advance, we refer to the history record to help scheduling the VM migration. Our strategy is proved to be a fast and efficient load balancing approach. The reduction in

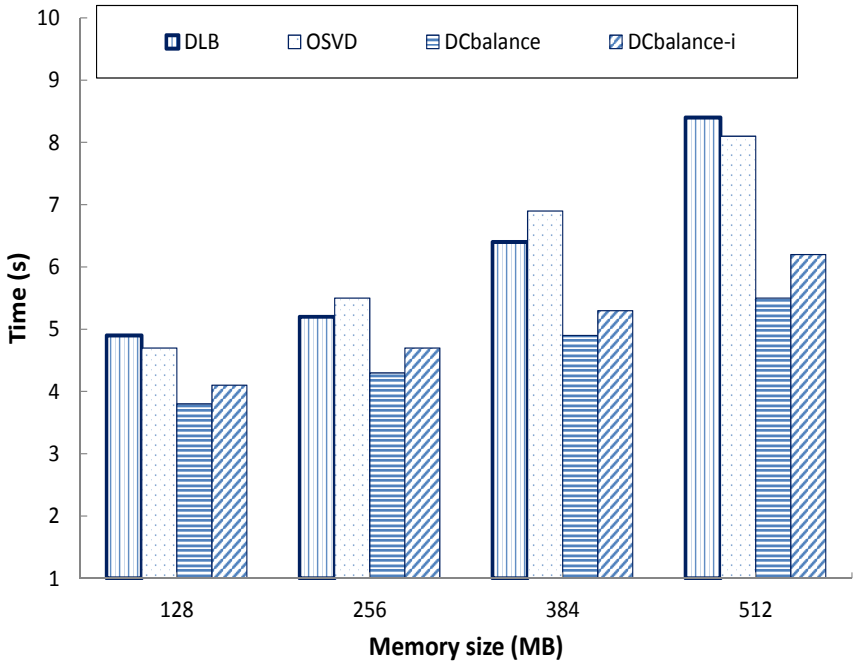


Figure 7.7: Total migration time comparison under Apache.

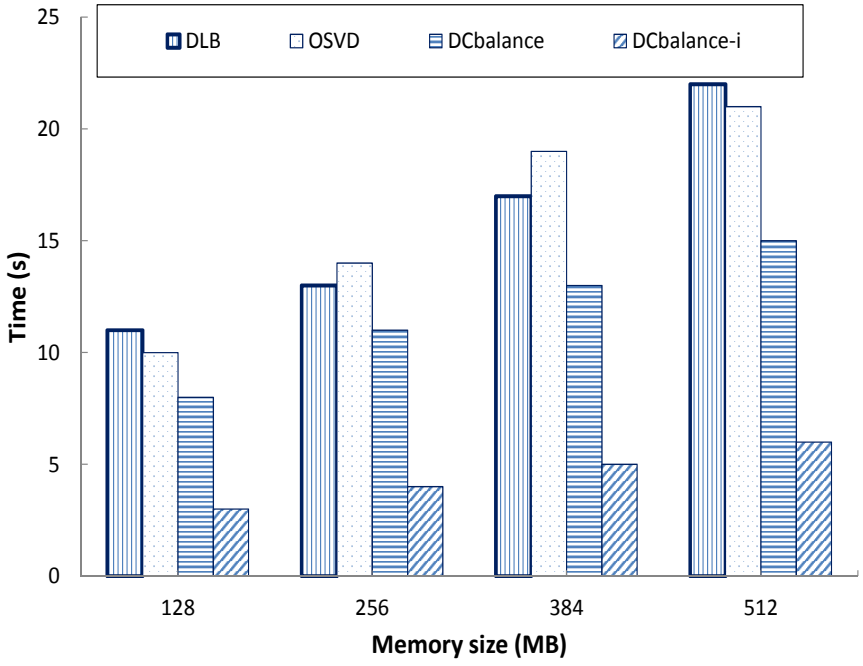


Figure 7.8: Total migration time comparison under Sysbench.

the decision generating time is up to 79% compared with competitors. Moreover, we apply an workload-adaptive migration mechanism to provide the minimal downtime requirement for different kinds of application, so as to improve the user experience. Results show that our proposed migration mechanism reduces the downtime by up to 73%, and reduces the total migration time by up to 38% compared with competitors.

Chapter 8

Fast and Storage-Adaptive Disk Migration

In this chapter, we first present the design and implementation of FDM, explaining the benefits from using page cache. We then propose optimization of FDM by integrating the storage-adaptive live migration design. Finally we show our evaluation results by comparing with pre-copy or post-copy disk migration mechanisms, as well as a widely used VM image storage and deployment system.

8.1 Live Migration Using Page Cache

Traditional VM live migration techniques developed for LAN interconnected environments only consider data residing in memory during the migration, because they assume that the source and target machines share all the disk data, i.e., through Network Attached Storage (NAS). To transfer the dirty memory, they apply an incremental checkpointing mechanism, that is, only transfer the newly dirty memory data in each migration epoch. The incremental checkpointing can also be used for disk migrations. With incremental checkpointing, the VM's virtual file system on the disk needs to be split into multiple chunks with the same size as the memory page, while 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 such as compression or choosing smaller size chunks for optimizations. However, for I/O intensive applications that write to the disk at high frequency, the number of dirty chunks during each epoch may not be small. A large amount of data due to file system modifications may need to be transferred.

Therefore, we need to improve our previous incremental checkpointing, in order to further reduce the data which needs to be transferred in each migration epoch. The idea is to identify the duplication between the memory and disk, so if the duplicated data has already been migrated during the memory migration, we discard sending these data again while migrating the disk. The traditional

way to identify physical memory pages with identical contents includes using hash page's content as well as the disk chunk's content and comparing the their hash values. If they share the same hash value, then we can make a byte-by-byte comparison to ensure the duplication exists.

However, the hashing method needs to reserve enough memory space to store the hash values, which leads to non-trivial overhead. Moreover, it also costs time to compute the hash value during each duplication identification, which will increase the downtime. Instead, we refer to the page cache [15] provided by the OS to facilitate fast access to the disk. The page cache resides in memory and is kept by the OS kernel to buffer data that read from or be written to disk. It is designed to offset the performance gap between the memory and disk, since accessing memory is faster. In modern operating systems the majority of the available memory (i.e., memory not in use by the kernel or any active application) is allocated to the page cache.

As opposed to the traditional VM live migration mechanisms which only schedule the dirty memory transfer, our new migration mechanism includes both memory and disk migration in every migration epoch, similar to the pre-copy way. Moreover, we prevent unnecessary transmission by identifying the duplicated data between memory and disk by relying on the page cache. The performance enhancements are achieved by realizing that the page cache will contain the same information both on disk as well as in memory. At the end of each migration epoch, the newly updated data in the page cache will be regarded as part of the entire dirty memory and will be transferred to target host. As these data in page cache also has duplication stored on the disk, it is not necessary to transfer these data again when migrating the disk data. Therefore, the unmodified pages in the page cache and the ones that have the same content in memory and on disk, are discarded from the migration.

The unmodified pages are read from disk at the target. The modified pages are sent over the network. This entails a solution where the source machine records the unmodified page frame number (PFN) with the corresponding disk sector in a data structure, called a pfn_to_sec map in order to know what PFN has its corresponding data on disk. The PFNs included in the pfn_to_sec are marked as read-only in their page table entries. If a page is modified during this migration epoch, and its PFN is in the pfn_to_sec map, the data on this page is only to be transferred over the network once. The other dirty data on either memory or disk is sent iteratively to the target in the same fashion as pre-copy does with the memory, in order to keep the downtime as low as possible. Note that we can't just send the pfn_to_sec map instead of transferring the real dirty data. This is because we can't share any storage in our mechanism so all the data in both memory and disk needs to be transferred at least once during the entire migration.

Although this pre-copy methodology leads to minimal downtime, it also has a drawback as other pre-copy methods. As we discussed in previous chapters, the downtime takes only a small fraction of the total migration time. In the pre-copy migration, we observe that if a memory page or disk chunk is frequently updated by some memory-intensive or I/O-intensive workload, its updated copy will be transferred every time during each migration epoch. Considering that a normal VM may be assigned up to several gigabytes of memory and even larger disk to run the guest OS, this will bring a consistent overhead throughout the entire migration, which leads to unacceptable total

migration times.

In order to reduce the total migration time when running I/O-intensive workloads on the VM, we keep recording how many times each disk chunk was written as long as the VM is active on the source host. If a chunk was written more times than a predefined threshold, we mark this chunk as dirty and avoid transmitting it to the target during the subsequent migration epoch. Moreover, if there are other dirty memory page updates to the pfn_to_sec map and claims to modify this chunk in subsequent migration epoch, this dirty memory page will not be transmitted either. Instead, for all these disk chunks which are updated more than threshold times, we transmit their final state in the last migration epoch. This ensures that each chunk is transferred no more than threshold times to the target host so as to reduce the total migration time.

8.2 Storage-Adaptive Live Migration

Besides the previous discussion that addresses how to reduce the transmitted data resident in both memory and disk storage, we propose to further reduce the network traffic generated by the migration when disks are implemented by different storage options. Our mechanism is based on Xen, which supports many different storage options, each of which has its own benefits and limitations. They can be separated into two categories: file based and device based. Compared to device based storage, file based storage has the advantage of being simple, easy to move, mountable from the host OS with minimal effort, and easy to manage. It used to be slow, but as the blktap driver (especially the blktap2 [3]) advents, the speed is no longer a problem. The blktap2 driver provides disk I/O interface on user-level. Using its associated libraries, blktap2 allows virtual block devices presented to VMs to be implemented in user space and to be backed by different formats, e.g. raw image. Compared with device based storage, the key benefit of blktap2 is that it makes it easy and fast to manipulate.

Blktap2 currently supports different disk formats [3], including:

1) Raw Images which can be deployed on both partitions and in image files.

2) Fast sharable RAM disk between VMs. Note that this format requires modification of the guest kernel, in order to be supported by cluster-based filesystem.

3) VHD, which involves snapshots and sparse images.

4) Qcow2, which also involves snapshots and sparse images.

In our preliminary experiments, however, we found that a general disk migration mechanism does not work well in all cases, for example, when dealing with raw format, a straightforward encoding method actually incurs shorter downtime due to the there are many unused chunks in raw image file. Therefore, we finally implement a storage-adaptive migration mechanism which could apply different migration methods based on the types of the disk formats.

In our prototype, we develop an adaptive live migration mechanism for both raw and Qcow2 formats. A raw image is simple to create and can be manipulated using standard Linux tools. It can also be mounted directly via the loopback device. With a strictly local installation, raw image is probably the best choice because it is easy to manipulate. After a raw image is created, when a tool needs to copy it, the tool will need to read the "logical" size of the image. That is, if you create a 10GB image with only 1GB of actual data, the copy operation will need to read 10GB of data. Because the content of a raw image is sparse, i.e., a great percentage of the image file does not contain real data, we propose to use Run-length encoding (RLE) [7] to compress all the dirty chunks on the source host before transmitting to the target host. Note that we initialized the full raw image at 0 in order to take advantage of RLE.

On other hand, a Qcow2 image has a smaller file size, so it can be efficiently copied over the network. It requires special tools to manipulate and can only be mounted using the qemu-nbd server. However, Qcow2 provides a key advantage over raw disk images: the format only stores data that has been written to the image. The practical impact is that the "logical" size of the file is the same as the physical size. Following the same example above, a 10GB Qcow2 image with only 1GB of data would only require reading/writing 1GB of data when copying the image. Therefore, there are no free chunks, and it does not hamper correctness because no assumptions on the contents of free chunks can be made. Moreover, Qcow2 supports copy-on-write so that the image only represents the latest changes. If there are several memory pages in the page cache which update the same disk chunk, the VM restoration needs to follow the same order as each memory page that is transmitted to the target host. We apply zlib based compression [10] to compress the data before transmission.

Our final mechanism is summarized as follows:

1) In the first migration epoch, on the source host, all the memory pages of the selected VM are transmitted to the target host while the VM is still running.

2) Still on the source host, all the disk data of the selected VM are also transmitted to the target host while the VM is still running. For different disk formats, these data are handled in the corresponding way (RLE, zlib based compression) before transmission.

3) For subsequent migration epochs, at the end of each migration epoch, the mechanism checks the dirty bitmap to determine which memory pages have been updated in this epoch. Then only the newly updated pages are transmitted. The VM continues to run on the source host during these epochs.

4) For those dirty pages with corresponding PFN existing in the pfn_to_sec map, they are scheduled to be flushed to disk in the near future but this activity may not occur in this migration epoch. Because these pages are already transmitted to the target host, there is no need to transmit the corresponding disk data again, even if its status is marked as dirty.

5) For all of the disk chunks which are updated more than threshold times, the mechanism marks each chunk as dirty and avoids transmitting it to the target during the subsequent migration epoch.

Instead, these chunks are only transmitted in the last migration epoch.

6) When the pre-copy phase is no longer beneficial, the VM is stopped on the source node, the remaining data (newly updated memory pages and disk chunks, CPU registers and device states, etc.) is transmitted to the target node, and the VM is prepared to resume on the target host.

8.3 Evaluation and Results

8.3.1 Experimental Environment

Our preliminary experiments were conducted on a set of identical machines equipped with an x86 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz) and 4GB of RAM. We built Xen 4.0 and ran a modified guest VM with Linux kernel 2.6.18. The guest OS and the host OS (in Domain 0) ran CentOS Linux, with a minimum of services initially executing in the guest OS, e.g., `sshd`. Domain 0 has 1.5 GB of memory allocated, and the remaining memory was left free to be allocated for guest VMs. To ensure that our experiments are statistically significant, each data point is averaged from twenty samples. The standard deviation computed from the samples is less than 6.4% from the mean value.

We refer to our proposed migration design with speculation mechanism as FDM. To evaluate the benefits of the storage-adaptive migration, we compare FDM with our initial migration design using only page cache, which we refer to as FDM-i. Our competitors include different checkpointing mechanisms including the original design of both pre-copy and post-copy migration mechanisms. To make a more clear comparison, we compare all these results with an open-source VM image storage and deployment system, called BlobSeer [71], which proposes a high performance incremental block storage migration. As opposed to our design, BlobSeer relies on cloud middleware to share the same base image among different VMs. As BlobSeer currently only supports KVM, we implemented its idea based on Xen so as to make a direct comparison with our prototypes under the same environment.

8.3.2 Downtime Evaluation

We evaluate the performance for migrating VMs running I/O-intensive workloads. We use the Sysbench [9] online transaction processing benchmark, which consists of a table containing up to 4 million entries. We perform either read or write transactions on the Sysbench database to evaluate the performance of all the migration mechanisms. The experiment is first conducted on the guest VM with assigned memory of 1.5GB, but with different disk size from 512 MB to 2GB, in order to investigate the impact of disk size on downtime, total migration time and other performance characteristics. Secondly we evaluate again with the opposite configuration: 1GB fixed disk size, with different memory size from 512 MB to 2GB. Although this range of memory or disk selected

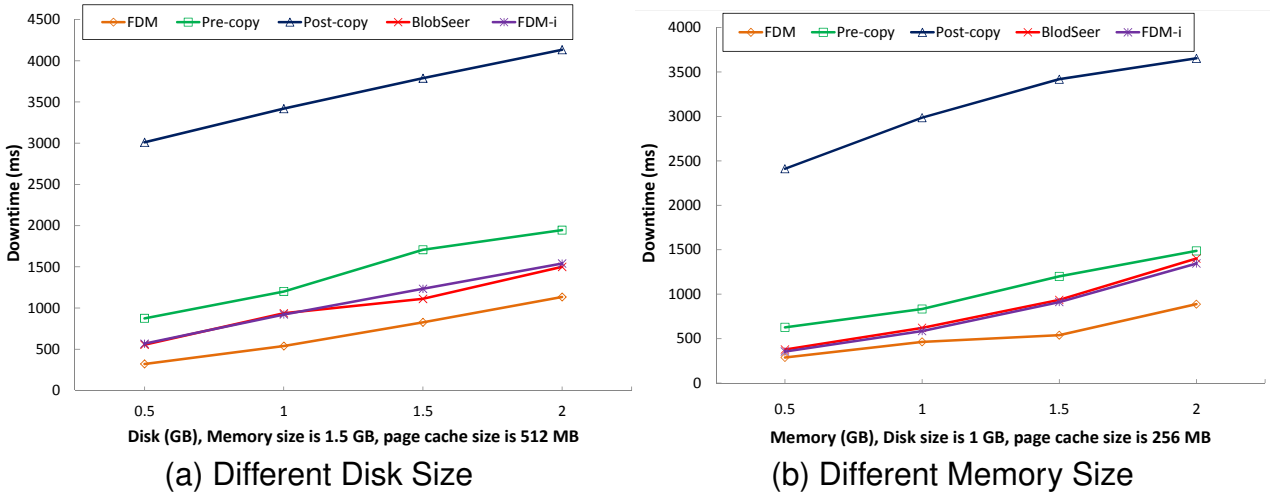


Figure 8.1: Downtime comparison for raw image.

in our experiment may seem moderate given the high quantity of RAM available in today's data centers, we found it to be reasonable for understanding the trends.

Since BlobSeer needs to rely on cloud middleware to achieve its best performance, in our evaluation of BlobSeer, we set up a 1Gbps network connection between the source and target hosts and shared part of its file systems (used as base image) among all the machines in this LAN. Note that we didn't share any storage together in the evaluation of pre-copy, post-copy, or our FDM system because our motivation is to provide fast disk migration across the WAN.

We first consider the measurement of the VM downtime. The definition of downtime in the Introduction works well under the pre-copy migration mechanism. After VM resumption, the user could resume normal activity immediately. One exception is using the post-copy strategy. After the VM has been resumed on the target host, it is actually unusable by the users for an initial period of time due to excessive page faults, so the resume time should be counted into the downtime measurement as well. When measuring the downtime in our experiment, we start measuring the elapsed time when the VM is stopped on the source host, and stop when the resumed VM is fully usable by the users. Figure 8.1a and Figure 8.1b show the downtime results for the Sysbench I/O-intensive benchmark for five migration mechanisms with different memory and disk configuration, and all disks use raw image.

We can make several observations regarding the downtime measurements.

1) The downtime results of the pre-copy migration mechanisms are short compared with the post-copy migration which incurs the longest downtime. This is due to the design motivation of each technique. In pre-copy, because memory updates are rare during the I/O intensive workload runs; there are relatively few dirty memory pages left. It only needs to transmit the dirty disk chunks in the final migration epoch. On the other hand, by using post-copy migration, there are no memory

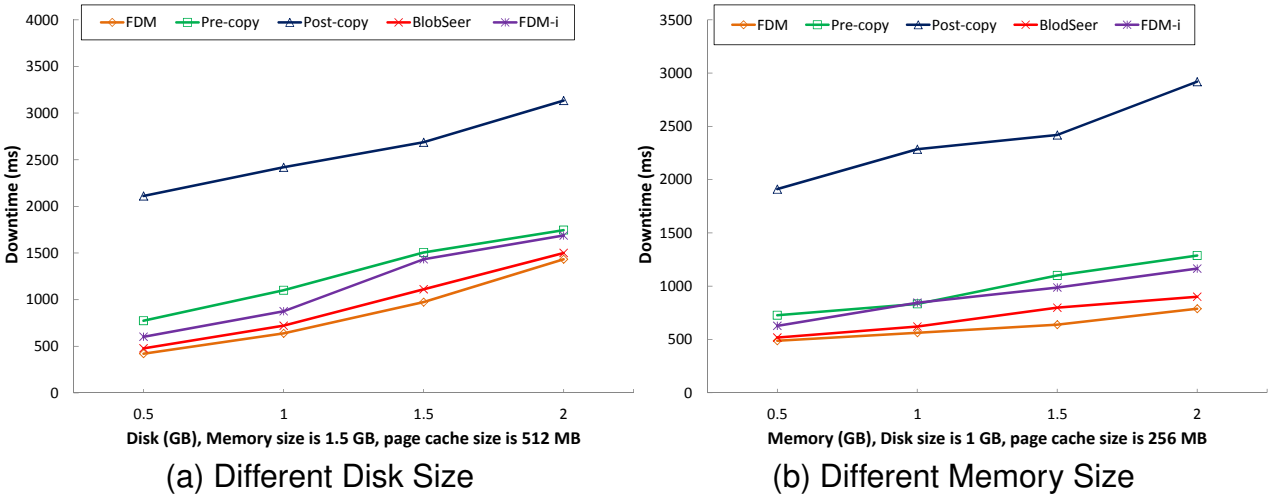


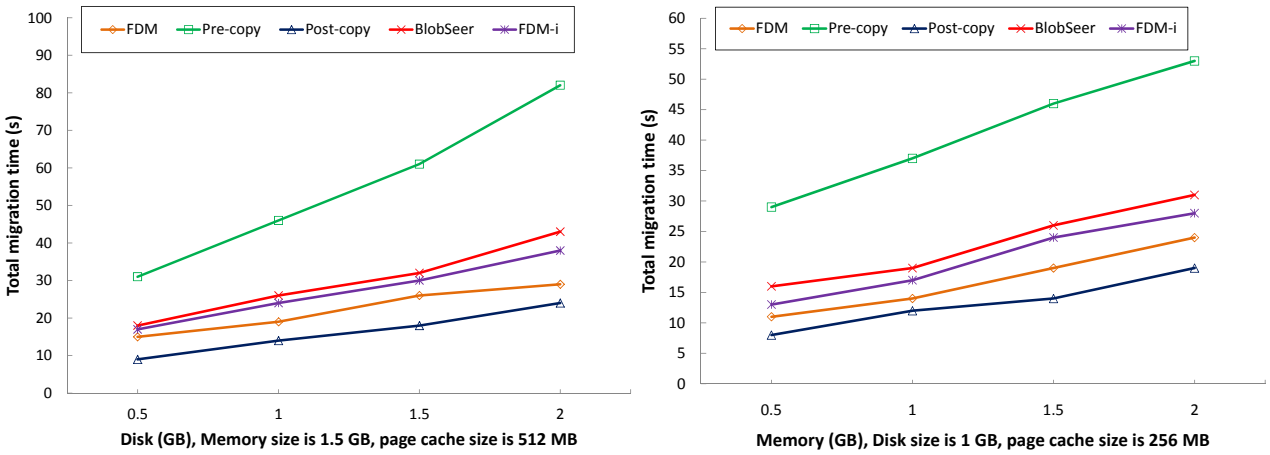
Figure 8.2: Downtime comparison for Qcow image.

pages or disk chunks restored in the resumed VM, so thousands of page faults occur, and the resumed VM needs to cross the network, reach the source host and fetch the corresponding pages. This makes the VM unusable at the beginning of restoration and leads to a much longer downtime.

2) The downtime result under the BlobSeer is better than the traditional pre-copy method. The main reason is because in BlobSeer, it uploads the unmodified part of the disk data into the cloud as a base image. Then every machine which could access the cloud service is able to download the base image. So it only needs to transmit the modified data to the target host during the migration.

3) Our original FDM design (as FDM-i) is competitive. We have similar results as BlobSeer but we apply a total different method to achieve this. We use page cache to identify the duplication on both memory and disk so we reduce the amount of data which needs to be transmitted in the last migration epoch. Compared with our original design (FDM-i), FDM incurs shorter downtime. This is because we further apply a storage-adaptive migration on our original design. For raw image, we use the RLE to compress all the dirty chunks, which further reduces the amount of data in each transmission. Therefore, FDM incurs a shorter downtime than FDM-i. Compared to the worst case, post-copy migration mechanisms, the downtime reduction is roughly 87%.

Figure 8.2a and Figure 8.2b show the downtime results in the same cases as Figure 8.1a and Figure 8.1b, except using Qcow2 as the disk format. The findings are mostly similar to the previous observation. Even though FDM-i does not perform as well as BlobSeer, FDM results are still competitive.



(a) Different Disk Size (b) Different Memory Size

Figure 8.3: Total Migration Time comparison for raw image.

8.3.3 Total Migration Time Evaluation

We also measured the total migration time, which is from when the migration is triggered to when the resumed VM is fully usable by users. Figure 8.3a and Figure 8.3b show the total migration time results when running the Sysbench I/O intensive workload, using raw as the disk format. We can make several observations regarding the total migration time measurements.

1) The total migration time of pre-copy migration mechanisms (including self-migration) are longest, while the original post-copy migration mechanism incurs the shortest downtime. Because in the experiment, no storage is shared in any mechanisms, all migration mechanisms need to migrate all the memory pages at least once, either by pre-copying or by post-copying. However, the pre-copy migration mechanism may need to transmit a memory page or dirty chunk several times if it's modified during this migration epoch. On the other hand, the post-copy migration only needs one round to fetch and transfer all the memory pages and disk data based on the page faults, so it achieves a better total migration time.

2) The total migration time using the FDM design (including FDM-i) is between that of the pre-copy and the post-copy mechanisms. Because we set a threshold during the migration, if a disk chunk was written more times than this predefined number, FDM marks this chunk and avoids transmitting it to the target during the subsequent migration epoch. Therefore, for all these disk chunks which are updated more than threshold times, we transmit their final state in the last migration epoch. This ensures that each chunk is transferred no more than threshold times to the target host so as to reduce the total migration time. Compared to the worst case pre-copy migration mechanisms, the total migration time reduction is roughly 58%.

3) Finally, the comparison between FDM and FDM-i is similar to that when measuring the down-

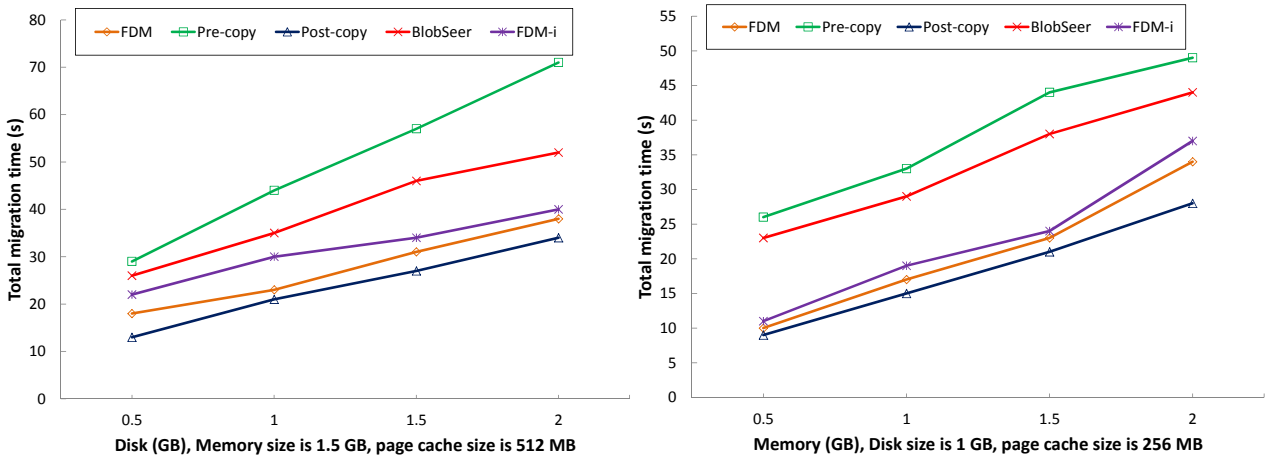


Figure 8.4: Total Migration Time comparison for Qcow image.

time. The reason is also the same: since FDM performs further compression on the transmitted data in raw image, the smaller amount of data reduces the migration time in each migration epoch. Therefore, as for the total migration time when running I/O intensive workloads, FDM performs better than FDM-i.

Figure 8.4a and Figure 8.4b show the total migration time results in the same cases as Figure 8.3a and Figure 8.3b, except using Qcow2 as the disk format. The findings are mostly similar to the previous observation. The reduction from Figure 8.4a and Figure 8.4b is not so obvious as Figure 8.3a and Figure 8.3b, meaning that for FDM, it gets more benefits from the adaptive migration method for raw image than that for Qcow2 image.

8.3.4 Evaluation of Page Cache Size

We also evaluate the relation between the downtime and size of the OS page cache. We assign fixed memory and disk resources in all cases, while measuring the downtime for different migration mechanism. From Figure 8.5 we observe that as the size of the page cache increases, the downtime becomes shorter. The reason is due to the fact that both memory and disk have fixed size, so if the page cache becomes larger, it can identify more duplicated data between the memory and disk. Therefore, FDM can mark more unnecessary data which is unnecessary to transmit in the last migration epoch, and finally, reduce the downtime further.

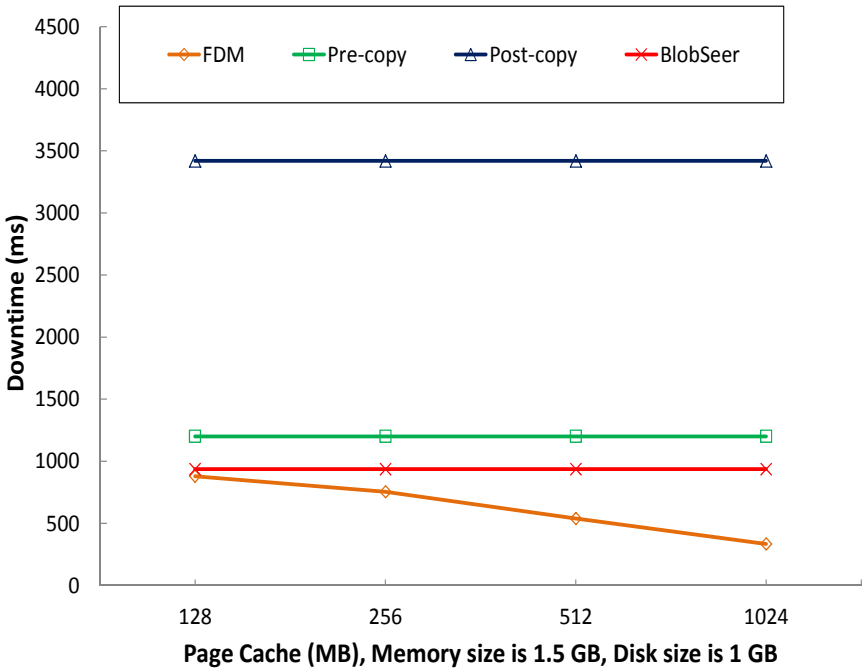


Figure 8.5: Downtime with different page cache size.

8.4 Summary

We propose a fast and storage-adaptive live migration to collaborative migrate memory and disk data with short downtime and total migration time. We use page cache to identify the duplication between memory and disk, so as to transmit these data only once during each migration epoch. Furthermore, we apply different techniques such as encoding and compression for different disk formats, in order to further reduce the downtime and total migration time. Results show that our proposed migration mechanism reduces the downtime by up to 87%, and reduces the total migration time by up to 58% compared with competitors.

Chapter 9

Conclusions and Future Work

This dissertation presents a set of techniques that provide high availability through VM live migration, their implementation in the Xen hypervisor and the Linux operating system kernel, and experimental studies conducted using a variety of benchmarks (e.g., SPEC, NPB, Sysbench) and production applications (e.g., Apache webserver). The techniques include:

- a novel fine-grained block identification mechanism called FGBI;
- a lightweight, globally consistent checkpointing mechanism called VPC;
- a fast VM resumption mechanism called VMresume;
- a guest OS kernel-based live migration technique that does not involve the hypervisor for VM migration called HSG-LM;
- an efficient live migration-based load balancing strategy called DCbalance; and
- a fast and storage-adaptive migration mechanism called FDM.

FGBI 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 that 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 [43], Remus [19]) show that FGBI (augmented with the optimizations) significantly reduces the downtime with acceptable performance overhead.

VPC 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. VPC also ensures the VC's global consistency, thanks to its underlying distributed snapshot algorithm.

VMresume predicts the most likely to be accessed memory pages during a checkpointing interval for fast VM resumption from checkpoints stored on slow-access storage. 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 suffer 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.

HSG-LM does away with the hypervisor for VM live migration using a hybrid pre-copy and post-copy technique and a guest OS kernel-based implementation. Our evaluations of HSG-LM with and without speculations on the pages likely to be accessed in the future revealed that, the technique is a good compromise between pre-copy and post-copy mechanisms developed in the past. Furthermore, HSG-LM is the only technique that provides short downtime and short total migration time, on the average. By speculating on the pages likely to be requested in the future and migrating them together in a single transfer, we improved our original design.

DCbalance's strategy of using the history record to help schedule VM migration was found to be an efficient load balancing strategy. Our evaluation shows that using history records, DCbalance accelerates the load balancing decision process. DCbalance is able to achieve minimal downtime for different kinds of applications by using different migration approaches including encoding and compression. The evaluation also reveals that DCbalance is effective as a workload-adaptive migration mechanism.

FDM's design, implementation, and evaluation revealed that, it is possible to track the similarity between memory state and disk data, and that, it is fast and efficient to identify duplication by relying on page cache. Once duplication has been identified, significant reduction on downtime and total migration time can be achieved by ensuring that the dirty data is transmitted only once during each migration epoch. Furthermore, we observed that for different disk formats, it is better to apply different optimizations, in order to achieve the biggest improvement. For example, for the raw image file, further encoding is efficient to reduce the total amount of the transmitted data, whereas for the Qcow2 image file, zlib-based compression is a good choice. FDM was found to be an effective storage-adaptive solution because it applies different techniques to handle different cases, in order to achieve the best performance.

9.1 Summary of Contributions

Our research contributions are summarized as follows:

- We developed a novel fine-grained block identification mechanism called FGBI, to track and transfer the memory updates efficiently, by reducing the total number of dirty bytes that need to be transferred from the primary to the backup host. FGBI enhances LLM's performance by overcoming its downtime disadvantage, especially for applications with memory-intensive workloads. We integrate memory block sharing support with FGBI to reduce the newly introduced memory and computation/comparison overheads. In addition, we also support a hybrid compression mechanism for compressing the dirty memory blocks to further reduce the migration traffic in the transfer period. Working together with these two optimization techniques, FGBI migrates a solo VM with minimal downtime.

- We developed VPC, a lightweight checkpointing mechanism. For solo VM, VPC checkpoints only the updated memory pages instead of the whole image during each checkpointing interval. The small checkpoint leads to minimal downtime with acceptable memory overhead. By predicting the checkpoint-caused page faults during each checkpointing interval, VPC further reduces the solo VM downtime than traditional incremental checkpointing approaches. We developed a variant of Mattern's distributed snapshot algorithm in VPC, in order to capture global snapshots of the VC that preserve the consistency of the VMs' execution and related communication states.

- We developed VMresume, a hybrid resumption mechanism, which quickly resumes a checkpointed VM, while avoiding performance degradation after the VM starts. VMresume augments incremental checkpointing with a predictive mechanism, which predicts and preloads the memory pages that are most likely to be accessed after resumption.

- We developed HSG-LM, a hybrid migration mechanism inside the guest OS kernel, which bypasses the hypervisor throughout the entire migration. HSG-LM's combination of pre-copy and post-copy techniques and speculative strategy for predicting the pages likely to be requested in the future significantly reduces the downtime and total migration time.

- In order to achieve fast and efficient load balancing in the VM environment, we proposed a centralized load balancing framework based on history records, called DCbalance. Instead of making predictions in advance, DCbalance uses history records to help schedule VM migration. Moreover, DCbalance uses a workload-adaptive approach to minimize downtime for different kinds of applications.

- We developed a fast and storage-adaptive migration mechanism, called FDM, to transmit both memory state and disk data with short downtime and total migration time. Using page cache, FDM identifies data that is duplicated between memory and disk, and thereby avoids transmitting the same data unnecessarily. FDM also uses an adaptive migration method for different disk formats including raw and Qcow2 image types.

9.2 Future Research Directions

There are several directions for future research.

Our work on HSG-LM opens future research directions on how to adaptively select when to speculate and when not to speculate, as well as selecting what kind of speculation technique to use based on the application's workloads. It may also be possible to further reduce downtime by giving higher priority in the transmission to page-faulted pages amongst the prefetched pages. Moreover, compressing memory pages before transmission e can further reduce the memory overhead.

Future work directions on load balancing strategies exploiting live migration exist. Adding the knowledge of the network topology to our DCbalance algorithm and pairing the current implementation with a network monitor (sniffer) will enable DCbalance to make network-aware mapping decisions in order to reduce the network traffic due to migrations. A further problem worth to consider, it's how to handle faulty control nodes in a DCbalance setup, i.e. which fault tolerant techniques we can apply in order to make DCbalance work also in the case of faults.

Integrating and evaluating other disk image formats into our FDM storage-adaptive design will increase the value of our research. For different disk image formats, it's possible to develop different optimization to further reduce the total amount of the disk data to be transferred across LAN. Moreover, our research shows the potential to design a novel disk image file format which will be more suitable to be migrated in a WAN environment.

Bibliography

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

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

[3] Blktap2. http://wiki.xen.org/wiki/Blktap2.

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

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

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

[7] Run-length encoding. http://en.wikipedia.org/wiki/Run-length_encoding.

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

[9] Sysbench benchmark. http://sysbench.sourceforge.net.

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

[11] 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.

[12] 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.

[13] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, 2007.

[14] B.S. Boutros and B.C. Desai. A two-phase commit protocol and its performance. volume 0, page 100, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[15] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[16] 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.

[17] 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.

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

[19] 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.

[20] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.

[21] Sagar Dhakal, Majeed M. Hayat, Jorge E. Pezoa, Cundong Yang, and David A. Bader. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):485–497, April 2007.

[22] Fred Douglis and John Ousterhout. Transparent process migration: design alternatives and the sprite implementation. *Softw. Pract. Exper.*, 21(8):757–785, July 1991.

[23] 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.

[24] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[25] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 550–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] D. Freedman. Experience building a process migration subsystem for unix. In *USENIX Winter'91*, pages 349–356, 1991.

[27] Laura Grit, David Irwin, Aydan Yumerefendi, and Jeff Chase. Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, VTDC '06, pages 7–, Washington, DC, USA, 2006. IEEE Computer Society.

[28] D. Grosu, A.T. Chronopoulos, and Ming-Ying Leung. Load balancing in distributed systems: an approach using cooperative games. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10 pp–, 2002.

[29] 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, 2004.

[30] 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.

[31] Anna Hać. Load balancing in distributed systems: a summary. *SIGMETRICS Perform. Eval. Rev.*, 16(2-4):17–19, February 1989.

[32] Jacob Gorm Hansen and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[33] Eric Harney, Sebastien Goasguen, Jim Martin, Mike Murphy, and Mike Westall. The efficacy of live virtual machine migrations over the internet. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, VTDC '07, pages 8:1–8:7, New York, NY, USA, 2007. ACM.

[34] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

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

[36] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Proceedings of the 11th IFIP/IEEE INM*, IM'09, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press.

[37] 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.

[38] 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.

[39] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Reactive consolidation of virtual machines enabled by postcopy live migration. In *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, VTDC '11, pages 11–18, New York, NY, USA, 2011. ACM.

[40] 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.

[41] Mei Hui, Dawei Jiang, Guoliang Li, and Yuan Zhou. Supporting database applications as a service. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 832–843, Washington, DC, USA, 2009.

[42] Parveen Jain and Daya Gupta. An algorithm for dynamic load balancing in distributed systems with multiple supporting nodes by exploiting the interrupt service. *International Journal of Recent Trends in Engineering*, 1(1):232–236, 2009.

[43] Bo Jiang, Binoy Ravindran, and Changsoo Kim. Lightweight live migration for high availability cluster service. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, pages 420–434, Berlin, Heidelberg, 2010. Springer-Verlag.

[44] 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.

[45] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 14–24, New York, NY, USA, 2006. ACM.

[46] 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.

[47] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[48] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373 –381, april 2006.

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

[50] Sanjay Kumar, Vanish Talwar, Vibhore Kumar, Parthasarathy Ranganathan, and Karsten Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 127–136, New York, NY, USA, 2009. ACM.

[51] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[52] John R. Lange and Peter A. Dinda. Transparent network services via a virtual traffic layer for virtual machines. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 23–32, New York, NY, USA, 2007. ACM.

[53] Kien Le, Ricardo Bianchini, Jingru Zhang, Yogesh Jaluria, Jiandong Meng, and Thu D. Nguyen. Reducing electricity cost through virtual machine placement in high performance computing clouds. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 22:1–22:12, New York, NY, USA, 2011. ACM.

[54] 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.

[55] Zhihong Li, Wei Luo, Xingjian Lu, and Jianwei Yin. A live migration strategy for virtual machine based on performance predicting. In *Proceedings of the 2012 International Conference on Computer Science and Service System*, CSSS '12, pages 72–76, Washington, DC, USA, 2012. IEEE Computer Society.

[56] Michael Litzkow and Marvin Solomon. Mobility. chapter Supporting checkpointing and process migration outside the UNIX kernel, pages 154–162. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[57] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative automated cloud resource orchestration. pages 26:1–26:8, 2011.

[58] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 101–110, New York, NY, USA, 2009. ACM.

[59] Maohua Lu and Tzi cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 534–543, 2009.

[60] Peng Lu, Antonio Barbalace, Roberto Palmieri, and Binoy Ravindran. Adaptive live migration to improve load balancing in virtual machine environment. In *Proceedings of the 1st International workshop: Federative and interoperable cloud infrastructures, with 19th Euro-Par*, 2013.

[61] Peng Lu, Antonio Barbalace, and Binoy Ravindran. Hsg-lm: Hybrid-copy speculative guest os live migration without hypervisor. In *Proceedings of the 6th Annual International Systems and Storage Conference*, SYSTOR '13, 2013.

[62] Peng Lu and Binoy Ravindran. Fast virtual machine resumption with predictive checkpointing (technical report), 2012.

[63] Peng Lu, Binoy Ravindran, and Changsoo Kim. Enhancing the performance of high availability lightweight live migration. In *Proceedings of the 15th international conference on Principles of Distributed Systems*, OPODIS'11, pages 50–64, Berlin, Heidelberg, 2011. Springer-Verlag.

[64] Peng Lu, Binoy Ravindran, and Changsoo Kim. Vpc: Scalable, low downtime checkpointing for virtual clusters. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 203–210, 2012.

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

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

[67] 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.

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

[69] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 23–32, New York, NY, USA, 2007. ACM.

[70] 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.

[71] Bogdan Nicolae and Franck Cappello. A hybrid local storage transfer scheme for live migration of i/o intensive workloads. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 85–96, New York, NY, USA, 2012. ACM.

[72] H. Ong, N. Saragol, K. Chanchio, and C. Leangsuksun. Vccp: A transparent, coordinated checkpointing system for virtualization-based cluster computing. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, 2009.

[73] 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.

[74] 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, 2007.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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, 2005.

[81] 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.

[82] Ellard T Roush. The freeze free algorithm for process migration. Technical report, Champaign, IL, USA, 1995.

[83] P. Ruth, Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 5–14, 2006.

[84] 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.

[85] Mahadev Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, D.R. O'Hallaron, Ajay Surie, A. Wolbach, J. Harkes, A. Perrig, D.J. Farber, M.A. Kozuch, C.J. Helfrich, P. Nath, and H.A. Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. *Internet Computing, IEEE*, 11(2):16–25, 2007.

[86] Brian Keith Schmidt. *Supporting ubiquitous computing with stateless consoles and computation caches*. PhD thesis, Stanford, CA, USA, 2000. AAI9995279.

[87] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 53:1–53:12, Piscataway, NJ, USA, 2008. IEEE Press.

[88] Ying Song, Yaqiong Li, Hui Wang, Yufang Zhang, Binquan Feng, Hongyong Zang, and Yuzhong Sun. A service-oriented priority-based resource scheduling scheme for virtualized utility computing. In *Proceedings of the 15th HiPC*, HiPC'08, pages 220–231, Berlin, Heidelberg, 2008. Springer-Verlag.

[89] 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.

[90] 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.

[91] 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.

[92] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees de Laat, Joe Mambretti, Inder Monga, Bas van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Gener. Comput. Syst.*, 22(8):901–907, October 2006.

[93] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 254–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[94] 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.

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

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

[97] 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.

[98] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 121–132, New York, NY, USA, 2011. ACM.

[99] 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.

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

[101] Jing Xu, Ming Zhao, José Fortes, Robert Carpenter, and Mazin Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, September 2008.

[102] 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.

[103] 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.

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