Resource-Efficient Replication and Migration of Virtual Machines

by

Kai-Yuan Hou

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2015

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Peter M. Chen
Professor Jason N. Flinn
Professor Dawn M. Tilbury

© Kai-Yuan Hou 2015
All Rights Reserved

To my father.

ACKNOWLEDGEMENTS

I cannot say enough to thank my advisor, Professor Kang Shin; this thesis would not have been possible without him. He sets an example of continued pursuit of excellence, strong sense of responsibility and hard work. He has provided tremendous support all the years of my graduate study, and has shown me great patience and understanding in the difficult times. His advice on research, career and life has also benefited me immensely. I consider myself truly fortunate to have been his student, and would like to express my deep gratitude to him. I would also like to thank Professors Peter Chen, Jason Flinn and Dawn Tilbury, for serving on my thesis committee, proposing insightful questions and providing invaluable feedback for improving this thesis.

The work described in this thesis is not produced by my effort alone. HydraVM, described in Chapter III, was designed in collaboration with Dr. Arif Merchant, Dr. Mustafa Uysal and Dr. Sharad Singhal. The characterization of checkpoint compression methods described in Chapter II was conducted with help from Dr. Yoshio Turner and Dr. Sharad Singhal. Dr. Jan-Lung Sung contributed remarkably to application-assisted live migration and JAVMM, which are described in Chapter IV. I am truly grateful for these extraordinary researchers, whose insights greatly helped shape this thesis. I would like to extend a special thanks to Dr. Jan-Lung Sung for his kind encouragements and bold confidence in me. I also thank the Air Force Office of Scientific Research, HP Labs and Oracle Labs for supporting the research in this thesis.

I sincerely thank the past and present members of the Real-Time Computing Lab, for making such a supportive team and their friendship that goes beyond graduate studies. I

would like to mention a few in particular. Dr. Hai Huang provided the source code of his work, FS2, Free Space File System, and helped me understand the code, in my first year of graduate school; I started to learn about software systems since then. Dr. Howard Tsai offered help on virtually everything—from brainstorming research ideas to trouble-shooting when the network is down. It was also Howard who taught me the skills and attitude of being a system administrator. Dr. Pradeep Padala let me work with him in his AutoControl project; this broadened my exposure to virtualization technologies, upon which my thesis eventually dwelt. Xiaoen Ju has always been generous with his time, encouragements and insights in our conversations; I am grateful for having such a wonderful friend and colleague in the last few years of school.

My heartfelt thanks goes to my friends; they made my time at the University of Michigan preciously memorable. I am deeply, deeply indebted to my family; their love, understanding and support is what has kept me going. To my father, who has become a computer hobbyist, I dedicate this thesis.

Above all, I give thanks to God. He led me through valleys and provided for me. He has shown me incomparable grace and that He is without limits. I am humbled, and to Him I am forever grateful.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES xi

ABSTRACT xii

CHAPTER

I. Introduction 1

1.1 VM Replication for High-Availability 2
1.2 VM Live Migration 4
1.3 Thesis Overview 5
1.4 Thesis Organization 7

II. Tradeoffs in Compressing Virtual Machine Checkpoints in High-Availability Systems 9

2.1 Introduction 9
2.2 Checkpoint Replication 11
2.2.1 Preliminaries 11
2.2.2 Need for Checkpoint Compression 13
2.3 Evaluation Methodology 14
2.3.1 Framework 14
2.3.2 Metrics 15
2.4 Checkpoint Compression 17
2.4.1 Existing Techniques 17
2.4.2 Exploiting VM Similarity 18
2.5 Experimental Results 20
2.5.1 Workloads and Testbed 20

2.5.2 Traffic Reduction . . . 22
2.5.3 CPU and Memory Costs . . . 28
2.5.4 Checkpoint Transfer and Storage Time . . . 32
2.6 Discussions on Compression Method Selections . . . 37
2.7 Related Work . . . 38
2.8 Conclusions . . . 40

III. HydraVM: Memory-Efficient High-Availability for Virtual Machines . . . 41

3.1 Introduction . . . 41
3.2 HydraVM Design . . . 44
3.2.1 Storage-based VM High-Availability . . . 44
3.2.2 An Overview of HydraVM . . . 45
3.2.3 Advantages and Limitations . . . 46
3.3 VM Protection . . . 48
3.3.1 Checkpointing VM CPU and Memory State . . . 49
3.3.2 Checkpointing VM Disk State . . . 52
3.4 VM Recovery . . . 52
3.4.1 Slim VM Restore . . . 53
3.4.2 Fetching VM Pages On-demand . . . 54
3.4.3 Pre-fetching Nearby VM Pages . . . 55
3.5 Evaluation . . . 56
3.5.1 Testbed and Workloads . . . 56
3.5.2 Storage-based VM Protection . . . 57
3.5.3 Overheads of VM Protection . . . 60
3.5.4 Restoration of a Failed VM . . . 62
3.5.5 Operation of a Restored VM . . . 64
3.6 Discussions on Alternative Storage Architectures . . . 67
3.7 Related Work . . . 70
3.8 Conclusions . . . 72

IV. Application-Assisted Live Migration of Virtual Machines with Java Applications . . . 73

4.1 Introduction . . . 73
4.2 Related Work . . . 76
4.3 Application-Assisted Live Migration . . . 78
4.3.1 What Memory to Skip Migrating? . . . 78
4.3.2 Challenges and Design Principles . . . 78
4.3.3 A Generic Framework . . . 79
4.4 JAVMM: Java-Aware VM Migration . . . 85
4.4.1 Background on Java Heap Management . . . 86
4.4.2 Garbage in Java Heap . . . 86
4.4.3 JAVMM . . . 89

4.5 Evaluation 92
4.5.1 Experimental Setup 92
4.5.2 Progress of Migration 93
4.5.3 Performance of Migration 95
4.5.4 Impact of Young Generation Size 98
4.6 Discussions on Applications and Extensions 100
4.7 Conclusions 101

V. Conclusions 102

5.1 Thesis Contributions 102
5.2 Future Directions 103

BIBLIOGRAPHY 106

LIST OF FIGURES

Figure

2.1 The bandwidth requirements and total traffic of checkpoint replication during two minutes of workload execution. 13

2.2 The traffic reductions achieved for a single protected VM running different workloads. The ”sim-*” lines show the traffic reductions achieved by similarity compression using different chunk sizes and a single-interval hash table, and the ”sim2-*” lines show those achieved using a two-interval hash table. 22

2.3 The traffic reductions achieved by similarity compression for a HPC-C VM when the VM’s checkpoints are processed alone (Single-VM), together with other VMs’ checkpoints in a 4-VM HPC cluster (4-VM), and using a two-interval hash table (4-VM, 2-intv.) 26

2.4 The CPU cost of checkpoint replication for a single protected VM running different workloads. No compression is used in the baseline. Delta cache is 32MB. Similarity compression uses 256 byte chunks with a single-interval (Sim) and a two-interval (Sim2) hash table. 28

2.5 The memory cost of similarity compression using 256 byte chunks. 30

2.6 The checkpoint transfer time of a single protected VM running different workloads. 31

2.7 The ratio of the measured transfer time in Figure 2.6 to the configured checkpointing interval. 33

2.8 The storage time of each checkpoint of a single protected VM running different workloads. 34

2.9 The checkpoint transfer time of a HPC-C VM measured in our 4-VM HPC cluster and workload mixture scenarios. 34

2.10 The breakdown of the checkpoint transfer times of a HPC-C VM measured in our HPC clusters consisting of 1, 2 and 4 VMs. 36

3.1 The HydraVM system. 49

3.2 The VM pause time incurred for taking an incremental checkpoint. The legend is given in (configured checkpointing interval, fail-over image storage type). 60

3.3 The performance of the workloads when checkpointed periodically. The runtime of HPC-C without checkpointing (baseline) is 344 seconds. The baseline throughput of FFT is 6.1 ops/min. The baseline runtime of FFmpeg is 289.5 seconds. 61

3.4 The time required to bring up a failed VM from a HDD- and a SSD-based fail-over image storage. 62

3.5 The performance of the workloads under different conditions: *no protection* and not checkpointed, *protected* and configured to be checkpointed every second, and *restored* from a failure that occurs halfway through the workload executions. 64

3.6 Number of VM pages fetched to execute the last 50% of the workloads in a restored VM. 65

3.7 The storage I/Os incurred to demand fetch VM pages for FFmpeg after it resumes execution in a restored VM. Each dot represents an I/O, and the size of the dot represents the size of the I/O. 66

4.1 Live migration of a 2GB Xen VM running the Apache Derby database workload from SPECjvm2008. 74

4.2 A generic framework for application-assisted live migration. 80

4.3 An example of transfer bitmap updates. 82

4.4 The workflow of application-assisted live migration. 84

4.5 Java heap usage and GC behavior of sample workloads from SPECjvm2008 running in a 2GB VM; see Table 4.1 for workload descriptions. The Young generation of the Java heap is allowed to use at most 1GB memory. 87

4.6 An overview of JAVMM, which is built on our framework for application-assisted live migration. This is a zoom-in view of Figure 4.2 with JVM/Java application being the running application. 90

4.7 The workflow of JAVMM, with details of JVM's and our TI agent's actions to fulfill the requirements of an application assisting in migration shown in Figure 4.4. 91

4.8 Progress of migrating a VM running the compiler workload from SPECjvm2008. Each box represents a migration iteration; the width shows the duration and the area shows the amount of traffic sent. In (b), the second last iteration of JAVMM generates little network traffic while waiting for the workload to execute to a Safepoint (0.7 sec) and a minor GC to be done (0.1 sec). 93

4.9 Amount of memory processed when migrating a VM running the compiler workload from SPECjvm2008. In (b), the 4–10th iterations of JAVMM each process less than 2MB of dirty memory. 93

4.10 Performance of JAVMM and Xen live migration for workloads with different characteristics of Java heap usage. 96

4.11 Effect of VM migration on the throughput of running application, *i.e.*, the number of operations completed per second. Migration begins after the application runs for 300 seconds. 97

4.12 Performance of JAVMM and Xen live migration for Category 1 workloads with different size Young generations. 99

LIST OF TABLES

Table

2.1 The workloads used in our evaluation and their setup. 20

2.2 The average checkpoint sizes of each workload. 21

3.1 The size of the incremental checkpoints taken and the time required to send and store each checkpoint to the fail-over image storage during the execution of the workloads. 58

3.2 Amount of data loaded and the loading time incurred during fail-over (slim VM restore). 63

4.1 Description of the SPECjvm2008 workloads used in our experiments. . . 87

4.2 Workloads with different characteristics of Java heap usage and their experimental settings. 96

4.3 Workloads with high object allocation rates and their experimental settings. 98

ABSTRACT

Resource-Efficient Replication and Migration of Virtual Machines

by

Kai-Yuan Hou

Chair: Kang G. Shin

Continuous replication and live migration of Virtual Machines (VMs) are two vital tools in a virtualized environment, but they are resource-expensive. Continuously replicating a VM's checkpointed state to a backup host maintains high-availability (HA) of the VM despite host failures, but checkpoint replication can generate significant network traffic. Each replicated VM also incurs a 100% memory overhead, since the backup unproductively reserves the same amount of memory to hold the redundant VM state. Live migration, though being widely used for load-balancing, power-saving, etc., can also generate excessive network traffic, by transferring VM state iteratively. In addition, it can incur a long completion time and degrade application performance.

This thesis explores ways to replicate VMs for HA using resources efficiently, and to migrate VMs fast, with minimal execution disruption and using resources efficiently. First, we investigate the tradeoffs in using different compression methods to reduce the network traffic of checkpoint replication in a HA system. We evaluate gzip, delta and similarity compressions based on metrics that are specifically important in a HA system, and then suggest guidelines for their selection.

Next, we propose HydraVM, a storage-based HA approach that eliminates the unpro-

ductive memory reservation made in backup hosts. HydraVM maintains a recent image of a protected VM in a shared storage by taking and consolidating incremental VM checkpoints. When a failure occurs, HydraVM quickly resumes the execution of a failed VM by loading a small amount of essential VM state from the storage. As the VM executes, the VM state not yet loaded is supplied on-demand.

Finally, we propose application-assisted live migration, which skips transfer of VM memory that need not be migrated to execute running applications at the destination. We develop a generic framework for the proposed approach, and then use the framework to build JAVMM, a system that migrates VMs running Java applications skipping transfer of garbage in Java memory. Our evaluation results show that compared to Xen live migration, which is agnostic of running applications, JAVMM can reduce the completion time, network traffic and application downtime caused by Java VM migration, all by up to over 90%.

CHAPTER I

Introduction

Virtualization is used to create one or more Virtual Machines (VMs) that act like real machines in a physical host. It facilitates flexible partitioning and dynamic allocation of computing resources, and is widely used in computing environments of various kinds and scales.

In a virtualized environment, applications run in VMs, and multiple VMs may be consolidated in a single physical server. Server consolidation has in fact been the most common reason for using virtualization [66]. It is most useful when applications require a certain level of isolation, e.g., isolation of configurations, performances, faults, and so on, yet each of them does not need the full capacity of a single server. Running these applications in separate VMs on a single physical server enhances server utilization and reduces various operational costs, including management cost, power, space, etc.

However, server consolidation exacerbates the consequence of unexpected host failures. When VMs are consolidated, failure of a single host may bring down multiple VMs on the host and all applications running thereon, resulting in an unacceptable aggregate loss. As host failures are inevitable, even common in large systems [29, 94], maintaining highly available VMs despite the occurrences of host failure has become a crucial task. To achieve this, various approaches have been proposed to replicate VMs between hosts continuously throughout VMs' execution [20, 24, 32, 39, 44, 47, 93], but they incur high

resource costs, creating a tension between high-availability and resource-efficiency, both critical operational goals of a virtualized environment.

On the other hand, server consolidation increases the need for VM live migration, which is the ability to move a running VM from a physical host to another without disrupting the VM's execution. As co-located VMs' workload dynamics and resource demands change, live migration can be used to adjust placement of the VMs at runtime, to mitigate resource hotspots [98] or enhance VM performance [31]. It can also be used to achieve power savings [30, 40, 72]. While live migration has been implemented by many virtualization platforms [9, 11, 38, 73], it can perform poorly when the underlying network is a bottleneck; it not only incurs a high resource cost, but also takes a long time to complete and degrades running applications' performance.

The needs to achieve VM high-availability (HA) at reduced resource costs and to perform VM live migration efficiently despite a network bottleneck are the two main motivations behind this thesis. Recognizing these needs, this thesis explores ways to (1) replicate VMs for HA using resources efficiently; (2) migrate VMs fast, with minimal disruption to VM execution and using resources efficiently. Below, we discuss the motivations of the thesis in more detail, and then provide an overview of the thesis.

1.1 VM Replication for High-Availability

The simplest way to maintain highly available VMs despite the occurrences of physical host failure is to reboot the VMs brought down by a host failure in other healthy hosts automatically upon detection of the failure [21]. This approach is "stateless", since the restarted VM loses its runtime state before the failure.

To be able to resume a failed VM's execution from where it left off upon failure, various "stateful" HA approaches have been proposed. They create a *backup* VM in a separate host for each *primary* running VM, and synchronize a primary VM's runtime state to its backup in one of the following two ways: log-and-replay [20, 32] and checkpoint replication [24,

39, 44, 47, 93]. The backup VM stands by in the background until the primary VM fails, at which point it becomes active and takes over execution from the primary's state before the failure.

During normal operation of the primary, the aforementioned primary-backup synchronization approaches work as follows. Log-and-replay records the low-level events executed by the primary VM, e.g., instructions and interrupts, and replays them deterministically in the backup VM, to bring the backup to the same state as the primary. Checkpoint replication sends the primary's state to the backup directly and continuously, by sending a series of primary VM checkpoints. In the two approaches, checkpoint replication is more widely applicable to different hardware/software configurations in a virtualized environment; the performance of log-and-replay can degrade significantly for VMs configured with multiple virtual CPUs, since the shared memory communication between CPUs must be accurately tracked and replayed [45, 88].

By maintaining backup VMs, stateful HA approaches minimize the loss of VMs' completed work caused by failures, but at high resource costs. Though approaches based on checkpoint replication have a wide applicability, they can impose a heavy load on network resources, especially when frequent checkpointing is used to save the network packets of client-facing applications before sending the packets out. For example, replicating checkpoints for a single protected VM once every 25 ms can consume more than 3 Gb/s of network bandwidth. When multiple VMs are protected at the same time, even dedicated GbE links cannot provide the aggregate bandwidth required for checkpoint replication. With such prohibitive network requirements, replicating VM checkpoints for HA could potentially use up all available network resources, interfere with normal VM traffic, and degrade application performance. The network traffic of checkpoint replication needs to be reduced for real-world deployment of this technique, and this should be done taking into consideration any impact on protected VMs' HA properties and performances.

On the other hand, whether based on checkpoint replication or log-and-replay, existing

HA approaches use in-memory backups. The backup VM sits in the memory of a dedicated backup host, and reserves as much memory as its primary; the reserved memory space cannot be utilized by other running VMs. Therefore, using existing approaches, *each* VM pays a 100% memory overhead to achieve HA. The backup memory reservation is unproductive, since the backup VM does not contribute to workload execution and system throughput until the primary fails. The aggregate backup memory reservation made for a group of protected VMs can significantly and unproductively consume RAM, a rather expensive computing resource. Furthermore, inactively blocking host memory for backup VMs may hinder effective consolidation of active running VMs, and result in under-utilization of other host resources (e.g., CPU cores). An alternative HA approach using memory efficiently is thus needed, to reduce the overall memory requirement for supporting HA.

1.2 VM Live Migration

VM migration was first proposed to support user mobility [37], and the early migration systems moved a VM while the VM was suspended (*i.e.*, not executing) [61, 87]. Later, VM *live* migration was proposed to move a VM while the VM is (mostly) executing [38, 73]. Live migration has different usages than the early migration systems. It is intended to relocate VMs at runtime, and has primarily been used for achieving load-balancing, performance enhancements, and so on. This thesis focuses on live migration of VMs.

Various contemporary virtualization technologies support live migration, and most of them use a *pre-copy* approach [9, 11, 38, 73]. A pre-copy approach copies all the state of a VM to be migrated to the destination host *before* the VM starts to execute in the destination host. This is contrary to a *post-copy* approach [50], which copies a migrated VM's state from the source host *after* the VM starts execution in the destination. Pre-copy is more widely used mainly for reliability reasons. Should the destination host fail in the middle of migration of a VM, a pre-copy approach aborts the migration, and the VM remains running in the source host. In a post-copy approach, a failure of the destination host may lead to

a complete failure of the VM, since the VM has begun executing in the destination and its state in the source host is only partially valid.

A pre-copy approach works as follows. To migrate a VM with minimal disruption to its execution, while the VM continues to run in the source host, its memory pages are transferred to the destination host incrementally and iteratively. In the first iteration, all of the memory pages are sent; at each following iteration, only the pages dirtied during the previous iteration are sent. Ideally, dirty pages should be transferred faster than new pages get dirtied. As the iterations progress, the number of dirty pages pending transmission should decrease. In the last iteration, the VM is paused, but only for a short time, since only a small number of dirty pages remain to be sent. After this short pause, the VM resumes execution in the destination, and migration of the VM completes.

However, this ideal migration is not always achieved, since the underlying network can be a bottleneck. Under this condition, VM memory pages are dirtied faster than they can be transferred to the destination, and the number of dirty pages pending transmission cannot be reduced iteratively. This can result in sending a large number of dirty pages in each iteration, and the iterations can last long until the last one, during which the VM is paused. Consequently, the migration can take a long time to complete, create significant network traffic and cause a noticeable VM downtime, which leads to degradation of running applications' performance. For example, we have observed live migration of a 2GB database VM over a gigabit Ethernet to last for more than one minute, generate 7GB (over 3x the VM size) of network traffic, cause 8 seconds of VM downtime and degrade the database application's performance by more than 20%. It is crucial to enhance live migration to overcome the network bottlenecking problem, but previous enhancements (e.g., [38, 50, 54, 92]), mostly treating migrating VMs as black boxes, incur either high resource costs or application performance penalties.

1.3 Thesis Overview

The following statement summarizes this thesis:

VMs need to be replicated for high-availability (HA) against host failures at reduced resource costs, and migrated with minimal execution disruption even when network is a bottleneck. Approaches to reducing the network traffic of VM checkpoint replication in a HA system should be applied adaptively based on workload scenarios, and a memory-efficient HA alternative is feasible. Assistance from applications running in migrating VMs is useful for efficient VM live migration despite a network bottleneck.

This thesis consists of three parts to fulfill the above statement.

The first part of the thesis focuses on reducing the network traffic of VM checkpoint replication in a HA system. Checkpoint compression has been suggested to meet this purpose. While several compression methods are available, they have not been compared systematically when applied to VM checkpoints in the context of supporting HA. Therefore, we build a generic framework to evaluate compression methods based on metrics that are specifically relevant and important in a HA system. The primary objective of our evaluation is to quantify the tradeoffs between the effectiveness and overheads of different compression methods, and provide insights that could guide their selection. Using our framework, we evaluate and compare three compression methods: two existing approaches, gzip and delta compression, and a method we explore, called *similarity compression*; similarity compression applies redundancy elimination to VM checkpoints continuously at fine time granularities to reduce checkpoint traffic. Our evaluation shows that one can hardly find a single best compression solution to the problem of reducing checkpoint traffic. Based on the experimental results, we provide guidelines for applying the different compression methods according to the workload types and resource constraints in a HA system.

The second part of this thesis proposes a memory-efficient HA approach for VMs, called *HydraVM*, to reduce the cost of making backup memory reservation for VM protection. The primary objective of HydraVM is to provide stateful protection for VMs against

failures of their hosting machines *without* any backup memory reservation. Instead of maintaining a backup VM in a separate server, HydraVM keeps track of the runtime state of a protected VM in a *fail-over image* maintained in a networked, shared storage, which is commonly deployed in a virtualized environment to hold VM disks and facilitate VM management. Upon detection of a failure, HydraVM performs a *slim VM restore*, which loads only a small amount of critical VM state from the fail-over image to quickly bring a failed VM back alive. As the VM resumes execution, the VM state not yet loaded is supplied on-demand. Our experimental results show that HydraVM provides VM protection at a low overhead, and can recover a failed VM within 2.2 seconds. This memory-efficient, storage-based approach complements the HA toolbox currently available to system administrators with a cost-effective alternative.

Finally, this thesis proposes *application-assisted live migration*. We take a *white-box* approach to efficient VM live migration. Our approach leverages assistance from applications running in a migrating VM, and skips transfer of the VM's memory pages that need not be migrated for the applications to execute in the destination host. It reduces the amount of memory transfer during live migration, with the objective of migrating the VM fast, with minimal disruption to VM execution and using resources efficiently. We build a generic framework for application-assisted live migration, and then use the framework to build *JAVMM*, a system that migrates VMs running Java applications skipping transfer of garbage in Java memory. In JAVMM, Java Virtual Machine (JVM), the application-level VM that executes Java bytecode, is enabled to provide all the assistance needed for migration on behalf of Java applications; no modifications to Java applications are required. Our experimental results show that compared to Xen live migration, which is agnostic of applications running in migrating VMs, JAVMM can migrate a Java VM with up to more than 90% shorter completion time, less network traffic and shorter application downtime.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter II describes the framework we build for evaluating checkpoint compression methods, and discusses the experimental results of evaluating and comparing gzip, delta and similarity compressions using our framework, as well as the insights gained from the evaluation.

Chapter III presents HydraVM, a memory-efficient, storage-based HA approach for VMs. We discuss the rationale, advantages and limitations of a storage-based HA approach, and describe the design, implementation and evaluation of HydraVM.

Chapter IV describes application-assisted live migration. We first present a generic framework for VM live migration to use applications' assistance, and then describe the design, implementation and evaluation of JAVMM, built based on the framework to migrate Java VMs using JVM's assistance.

Chapter V summarizes the contributions and future directions of this thesis.

CHAPTER II

Tradeoffs in Compressing Virtual Machine Checkpoints in High-Availability Systems

2.1 Introduction

Continuous checkpoint replication is a prevalent approach to maintaining highly available VMs even in the case of host failures [24, 39, 44, 47, 93]. It periodically captures the state of a VM in checkpoints, and replicates the checkpoints to a backup host. If the physical host of the VM fails, the VM can be restored from the most recent checkpoint available in the backup. However, checkpoint replication protects VMs at the expense of significant network traffic. Large amounts of checkpoint data are transported over the network, especially when frequent checkpointing is used by client-facing, latency-sensitive applications to checkpoint network packets before sending them out. Reducing checkpoint replication traffic is crucial to using this technique in real-world HA systems.

One way of reducing checkpoint traffic is to "compress" checkpoints before sending them over the network. Checkpoint compression requires no modifications to a VM, and can be applied regardless of the applications running in the VM. It can be done by a general-purpose tool, such as gzip [5]. Alternatively, for each dirty memory page in a checkpoint, the bits that are actually changed (called the page *delta*) may be identified, and the delta is replicated instead of the full page [69, 80, 92, 97]. These compression methods are

available, but they have not been evaluated comparatively and systematically when applied to VM checkpoints in the context of supporting HA. There are few guidelines for selecting and using them under different workloads and operating conditions in a HA system.

The primary goal of this chapter is to quantify the tradeoffs between the effectiveness and overheads of various checkpoint compression methods, and provide insights that could guide their selection decisions. We compare three compression methods, including gzip, delta compression, and a method we explore, called *similarity compression*. Similarity compression finds and eliminates duplicate contents in VM checkpoints to reduce checkpoint traffic, exploiting the content redundancy in VM memory. Different from existing memory deduplication systems, which coalesce identical pages of co-located VMs and reduce host memory pressure [49, 58, 70, 96], similarity compression finds redundant contents in the *changed* set of VM pages, *i.e.*, the VM checkpoints, and much more frequently.

We evaluate the three compression methods using workloads chosen from types frequently seen in HA systems, including server workloads that constantly interact with external clients and long-running computation jobs. Our results show that one can hardly find a single best compression solution. gzip reduces checkpoint traffic substantially, but at a prohibitive CPU cost. It takes a long time to replicate and store each checkpoint when gzip is used, and this limits the applicability of gzip for interactive, latency-sensitive applications needing frequent checkpointing. Delta compression incurs a low CPU overhead and short checkpoint transfer times, but requires a cache larger than the average checkpoint size of a protected VM to achieve a reasonable traffic reduction. For workloads that touch large areas of memory rapidly, delta compression can consume hundreds of MBs of RAM for the cache.

Similarity compression eliminates redundant contents *within* checkpoints of the same VM (intra-VM similarity), and *between* checkpoints of different VMs on a host (inter-VM similarity). It is particularly effective for VM clusters running homogeneous workloads, such as High Performance Computing (HPC) clusters. Our results show that a non-trivial

amount of VM similarity exists in these environments, especially when VMs collaborate on a shared task set. However, in heterogeneous workload scenarios, limited VM similarity is found, so gzip and delta compression are better suited. Although similarity compression is suited for a smaller range of application scenarios compared to the other two methods, for suitable scenarios, it reduces checkpoint traffic effectively using both CPU and memory efficiently, and requires short checkpoint transfer time and storage time.

The contribution of this chapter is threefold. First, it explores similarity compression, and proposes its use in homogeneous workload scenarios. Similarity compression is a new application of the existing concept of redundancy elimination; it applies redundancy elimination to changed VM memory continuously at fine time granularities, and our evaluation quantifies the effectiveness of redundancy elimination in this specific case. Second, to our best knowledge, this chapter presents the first detailed evaluation and characterization of checkpoint compression methods in the context of supporting HA, considering gzip, delta and similarity compressions. Third, based on the evaluation results, this chapter suggests guidelines for selecting and using these compression methods for different workload types and resource constraints in a HA system.

The remainder of the chapter is organized as follows. Section 2.2 provides background on checkpoint replication. We describe our evaluation framework in Section 2.3 and the three compression methods evaluated in Section 2.4. Section 2.5 presents and analyzes our experimental results. We discuss the insights gained from our evaluation in Section 2.6. Section 2.7 describes related work, and the chapter concludes with Section 2.8.

2.2 Checkpoint Replication

2.2.1 Preliminaries

Checkpoint replication protects a VM from the failure of its physical host by sending checkpoints of the VM to a backup host continuously [24, 39, 44, 47, 93]; the backup

host is chosen so that it is isolated from the failure of the protected host. When protection begins, a full checkpoint containing every memory page and the CPU state of the protected VM is replicated to the backup. It is stored in the backup's RAM, and becomes the *fail-over image* of the protected VM.[1]

As the VM executes, *incremental* checkpoints are taken and replicated to the backup, usually at fixed time intervals (a pre-configured checkpointing frequency). An (incremental) checkpoint mainly consists of the VM pages dirtied during the last checkpointing interval. After all dirty pages in a checkpoint are replicated to the backup, their contents are stored in proper locations in the fail-over image according to the page indexes. The fail-over image is not updated as each dirty page is received, or it may become inconsistent and unusable for recovery if the protected host fails in the middle of sending a checkpoint. If a checkpoint takes longer than the configured checkpointing interval to replicate, the subsequent checkpoint is not taken when the next interval begins, but delayed until the on-going replication finishes. This way, checkpoints are not sent faster than they can be stored and made useful in the fail-over image.

Once a failure of the VM host is detected, the HA system initiates a fail-over. The failed VM is restored based on its fail-over image (in the backup's RAM) and a consistent disk state (in a shared storage), and resumes operation from the most recent checkpointed state in the backup host. In order to make this fail-over transparent to the VM's external clients, the HA system ensures that the clients never see "unprotected" VM state, *i.e.*, the state not yet backed up. Specifically, during normal operation of the VM, outgoing network packets are withheld until the checkpoint capturing the state from which the packets are generated is fully replicated to the backup. Therefore, HA systems commonly use checkpointing intervals of tens of milliseconds or even shorter, in order to checkpoint and release network packets very frequently and achieve reasonable performance for client-facing applications.

[1] VM disks are usually hosted in a shared storage accessible to all VM hosts. A VM disk state consistent with the VM's memory and CPU state in the fail-over image may be maintained by the storage system using copy-on-write techniques [12, 67, 84, 85].

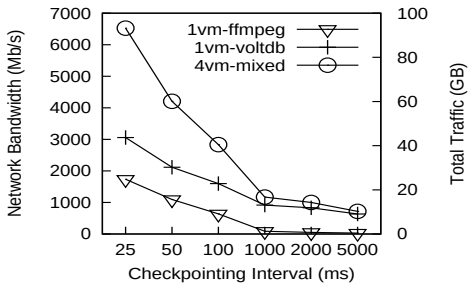


Figure 2.1: The bandwidth requirements and total traffic of checkpoint replication during two minutes of workload execution.

2.2.2 Need for Checkpoint Compression

We ran different workloads in VMs, and replicated VM checkpoints for two minutes of workload executions. Figure 2.1 shows the network bandwidth required and total traffic generated by checkpoint replication (see Table 2.1 for workload details). When checkpoints are replicated every 25 ms, protection of a media transcoding (ffmpeg) server and a database (voltdb) server uses more than 1700 Mb/s and 3000 Mb/s of network bandwidth, respectively. Even a dedicated GbE link cannot meet these requirements for protecting a single VM. When 4 VMs are protected concurrently, replicating checkpoints every 25 ms creates almost 100 GB of traffic in the network in only two minutes. To replicate these checkpoints at the configured checkpointing intervals, over 6500 Mb/s of network bandwidth is required. Even if a 10GbE link is available, it will soon become saturated with just a few more VMs to protect.

One may consider installing faster networks to meet the bandwidth requirement of checkpoint replication. However, higher-bandwidth links are expensive, and it is not always feasible to upgrade networks in existing infrastructures. Furthermore, as more and more VMs need protection, checkpoint replication traffic may eventually use up all available network resources and interfere with normal VM traffic, degrading application performance and users' experience with VMs. Therefore, reducing checkpoint traffic when providing VM protection is a must. This chapter conducts a systematic evaluation and

comparison of three compression methods for reducing checkpoint traffic, and provides guidelines for their selections in a HA system. Next, we describe how each compression method is evaluated.

2.3 Evaluation Methodology

2.3.1 Framework

To facilitate systematic comparison of multiple compression methods, we took an emulation approach for our evaluation. We built a framework consisting of a checkpoint sender (`emulhacp`) and a checkpoint receiver (`emulharcv`), which emulate the replication and storage of VM checkpoints in a HA system, as described in Section 2.2.1. `emulhacp` and `emulharcv` use multiple concurrent threads to emulate concurrent protection of multiple VMs. Different compression methods are implemented as modules inserted into the framework for evaluation.

To use this framework, we capture VM checkpoints *a priori* in a real HA system [52], and store them as individual files. `emulhacp` runs in a protected host. It reads a complete checkpoint from a file into a memory buffer, and from then operates on the buffer. It processes each dirty page using the compression method to be evaluated, and then sends the page to `emulharcv`, which runs in the backup. Once all dirty pages are received, `emulharcv` sends an ACK to `emulhacp`, and begins to decompress each page and store the page content to the fail-over image kept in RAM.

After `emulhacp` receives the ACK, it waits until the current checkpointing interval ends, and replicates a new checkpoint when the next interval begins. If an ACK is not received by the beginning of the next interval, `emulhacp` waits for the current checkpoint to be fully replicated, and immediately after an ACK is received, it sends a new checkpoint. In this case, `emulharcv` is receiving a new checkpoint while decompressing and storing the one just received at the same time.

We use double buffering in the backup: `emulharcv` uses at most two checkpoint buffers at the same time for each VM. If, by the time `emulharcv` receives the new incoming checkpoint in entirety, the previous checkpoint is not yet fully decompressed and incorporated into the fail-over image, the VM may not replicate more checkpoints. Checkpoint replication is resumed when the previous checkpoint is completely stored and its buffer released to receive another incoming checkpoint. Double buffering keeps any VM from using too much memory in the backup, and is especially useful when checkpoints take a non-trivial amount of time to store due to decompression.[2]

2.3.2 Metrics

We identify important metrics to consider when using checkpoint compression in a HA system, and evaluate them in our framework. For each compression method, we evaluate the **traffic reduction** achieved in each checkpointing interval, and the memory and CPU used in the protected and backup hosts to achieve such reduction. A compression method that uses excessive resources in the protected host can create a non-trivial interference with the normal operation of the protected VMs. The resource usage in the backup is also considered, since other active VMs may be running in the host (and backed up elsewhere) and their performances can be affected.

The **memory cost** of a compression method is evaluated by the average memory usage of its key data structures that enable page compression/decompression. **CPU cost** is evaluated by a per-page metric. We measure the total CPU time taken by `emulhacp` to compress and send checkpoints for all concurrently protected VMs. We then divide this time by the number of dirty pages processed, and obtain the average CPU time spent for each page. Likewise, we obtain the average CPU time taken by `emulharcv` to receive, decompress and store each page in the backup. These per-page metrics facilitate a fair comparison

[2] We use this simple scheme to facilitate evaluation and fair comparison of compression methods. Sophisticated flow control schemes may be devised to regulate checkpoint traffic in real HA systems, taking into account the checkpointing frequencies, compression methods and workloads used.

between different compression methods.

Besides consuming resources, compression affects the time required to replicate and store a checkpoint. We evaluate the **transfer time** of each checkpoint, which starts when `emulhacp` begins to send the checkpoint, and ends when an ACK for the checkpoint is received. Checkpoint transfer time consists of two components: the time to process/compress the dirty pages (processing time), and the time to send them over the network (sending time). Replicating compressed checkpoints reduces sending time, but performing compression lengthens processing time. The overall effect of compression on checkpoint transfer time must be quantified experimentally.

Checkpoint transfer time has a direct impact on achievable checkpointing frequency, which in turn affect the performance and HA property of a protected VM. Since a subsequent checkpoint may be replicated only after the on-going replication finishes, the actual (*elapsed*, not configured) checkpointing interval must be larger than the transfer time of a checkpoint. If a compression method incurs long transfer times, consecutive checkpoints must be separated by large intervals, thus the achievable checkpointing frequency is lowered. This can degrade application performance during normal operations, especially for latency-sensitive, server applications, since network packets are checkpointed and released infrequently. For computation jobs without external observers, low checkpointing frequency results in a greater loss of completed work upon a fail-over, since the VM has to resume execution from an earlier point in time.

The **storage time** of each checkpoint can also affect achievable checkpointing frequency and the performance of a protected VM. It is the time required to decompress all dirty pages and store their contents to the fail-over image. Without compression, checkpoint storage time is usually very short, since memory copying is fast. However, checkpoint storage time can be significantly lengthened when compression is used, if dirty page decompression involves complex steps or heavy computations. Checkpoint replication can become bottlenecked on slow storage of checkpoints: a new checkpoint may be delayed

waiting for a previous one to be completely stored so that the buffer space can be utilized, since a protected VM is not allowed to use unlimited memory in the backup.

2.4 Checkpoint Compression

Using the framework and metrics discussed in Section 2.3, we evaluate the three compression methods described below.

2.4.1 Existing Techniques

gzip is a commonly-used, general-purpose compression algorithm. Its application on checkpoint traffic was briefly discussed in [39] without a thorough evaluation. We implemented gzip using zlib [23] to evaluate it thoroughly and in comparison with other methods.

Delta compression identifies the parts in a dirty page that are changed when the page is written to, *i.e.*, the page delta, and replicates the delta to the backup instead of the entire page. It has been used in a few HA systems [69, 80].

Before sending a checkpoint, each dirty page is XOR'ed with its content in the last checkpointing interval. The outcome is compressed by RLE (Run-Length Encoding) [77], and the compression result is sent to the backup. To restore the page content in the backup, the RLE result is decoded, and the outcome is XOR'ed with the content of the page in the fail-over image; no extra memory copying is needed. Since keeping the prior content of every page incurs a 100% memory overhead, like previous work, we maintain a fixed-size cache of transmitted dirty pages. If a dirty page finds its prior content in the cache, delta compression is performed and the compression outcome is sent. On a cache miss, the entire page is replicated without compression. We implemented a LRU cache using a double linked list for efficient replacement and a lookup array to speed up queries. We use Basic Compression Library [2] for RLE.

2.4.2 Exploiting VM Similarity

Since VMs in a virtualized datacenter are often created from template images consisting of the same or similar operating systems and applications, they can load nearly identical kernel images and software binaries into memory, and read duplicate data from common files. Many systems use this content redundancy to enable page sharing [49, 58, 70, 96]. They detect duplicate contents in the entire memory space of co-located VMs, and coalesce identical pages in the same physical frame to save host memory.

We argue that not only may VMs be created from similar sources, even as they execute, various activities can keep *changing* their memory state in similar ways. For example, during maintenance, a group of VMs is updated with the same set of security patches at the same time. Also, in computing clusters, multiple VMs (and multiple processes in each VM) collaborate to finish computation-intensive tasks, each running the same application code and working on a common set of data. These VMs are often checkpointed at the same time intervals to gain a comparable level of protection, and the similar changes made to their memory may generate similar dirty pages in their checkpoints. We therefore explore **similarity compression**, which finds content redundancy in VM checkpoints and sends only one copy of the duplicate contents to reduce checkpoint traffic.

To detect content redundancy, we divide each dirty page into multiple *chunks*, and process checkpoints by chunks. *Unique chunks* are separated from *duplicate chunks*. A unique chunk contains a content different from any other chunks that have been processed. This content must be replicated to the backup in full. A duplicate chunk contains a content that is identical to at least one other chunk. Since the duplicate content can be found in another chunk that is already replicated (called a *reference chunk*), instead of sending the chunk again, we send a pointer to locate the reference chunk in the backup.

For similarity compression to be practically useful, two important requirements must be met: (1) unique and duplicate chunks must be separated quickly, and (2) the pointers sent for duplicate chunks must be small, yet contain enough information to restore the duplicate

contents in the backup. To meet these requirements, we build a hash table in the protected host. The hash table maps a chunk *content* to a chunk *location* that has the content. Chunk location is described by the VM to which the chunk belongs and the offset of the chunk in the checkpoint containing it. Chunk content is compactly represented and efficiently compared using the MD5 digest of the chunk. We also tested Rabin fingerprinting [79] over a sliding window, another commonly used method of detecting content redundancy, but found that to be much slower. For efficiency, we chose to detect duplicate fixed-size chunks by hashes.

In each checkpointing interval, the hash table is initially empty, and the checkpoints taken for concurrently protected VMs are processed together. For each chunk in the checkpoints, we compute its MD5 digest, and query the hash table by the digest. If the digest is not found in the hash table, the chunk content is sent to the backup, since this is a unique content that is not seen before. A new entry is inserted into the hash table to record the content and location of the chunk.

If the hash table lookup finds the chunk digest, we have a duplicate chunk, and the matching hash entry records the location of a replicated chunk with the same content, that is, the reference chunk. The location of the reference chunk is sent to the backup as a pointer, following which the duplicate content can be retrieved when incorporating the checkpoint into the fail-over image.[3] The reference chunk may belong to the same, or a different VM than the duplicate chunk does, upon detection of *intra-* and *inter-VM* similarity, respectively.

To increase the chance of finding and eliminating redundancy in checkpoint traffic, we also explore maintaining the hash table beyond interval boundaries. Instead of rebuilding the hash table in every checkpointing interval, we rebuild it every few intervals. With a multi-interval hash table, a duplicate chunk may contain the content of a chunk seen in a past interval. However, when a past chunk content is referenced, the referenced chunk in

[3]In our current implementation, each hash entry is 20 bytes. Chunk location is encoded in 4 bytes in the entry, and the remaining 16 bytes contain the MD5 digest of the chunk.

HPC-C [8]	A suite of 7 calculation-intensive benchmarks essential to long-running scientific jobs. We run HPC-C in a single VM and also in multiple VMs that collaborate via MPI. Each VM uses 512 MB RAM.
RUBiS [15]	An auction site benchmark modeled after ebay.com. We use a three-tier setup: a back-end database, a RUBiS server (Apache/PHP) and a client emulator. Each tier runs in a separate VM on a separate host. Our experiments use checkpoints of the RUBiS server (a 512 MB VM).
FFmpeg [3]	An open-source tool to "transcode" video/audio files, *i.e.*, to convert their codecs and formats. The transcoded media are usually fed to a real-time streaming service to fulfill requests from external clients. We run FFmpeg in a 512 MB VM.
VoltDB [22]	An in-memory database. We run it in a 1.75GB VM to support a TPC-C-like workload generated from a client VM running in a separate host. This OLTP workload simulates an order-entry environment for a business with multiple warehouses [18]. Our experiments use checkpoints of the VoltDB server.

Table 2.1: The workloads used in our evaluation and their setup.

the fail-over image may have been overwritten with a recent content. To ensure correct restoration of all duplicate contents, when a multi-interval hash table is used, a replicated (unique) chunk is cached in the backup until the next time the hash table is rebuilt and the chunk's content can no longer be referenced by other chunks.

2.5 Experimental Results

This section presents and analyzes our evaluation results of the three compression methods. Section 2.5.1 describes the workloads and testbed used in our evaluation. Section 2.5.2 evaluates the traffic reduction achieved by each compression method. Section 2.5.3 and Section 2.5.4 evaluate the resource and time overheads incurred by each method for achieving traffic reduction, respectively.

2.5.1 Workloads and Testbed

The computation tasks needing HA most are the ones that are not repeatable or prohibitively expensive to repeat after a failure occurs. These tasks include server workloads that constantly interact with external clients, and long-running computing jobs such as sci-

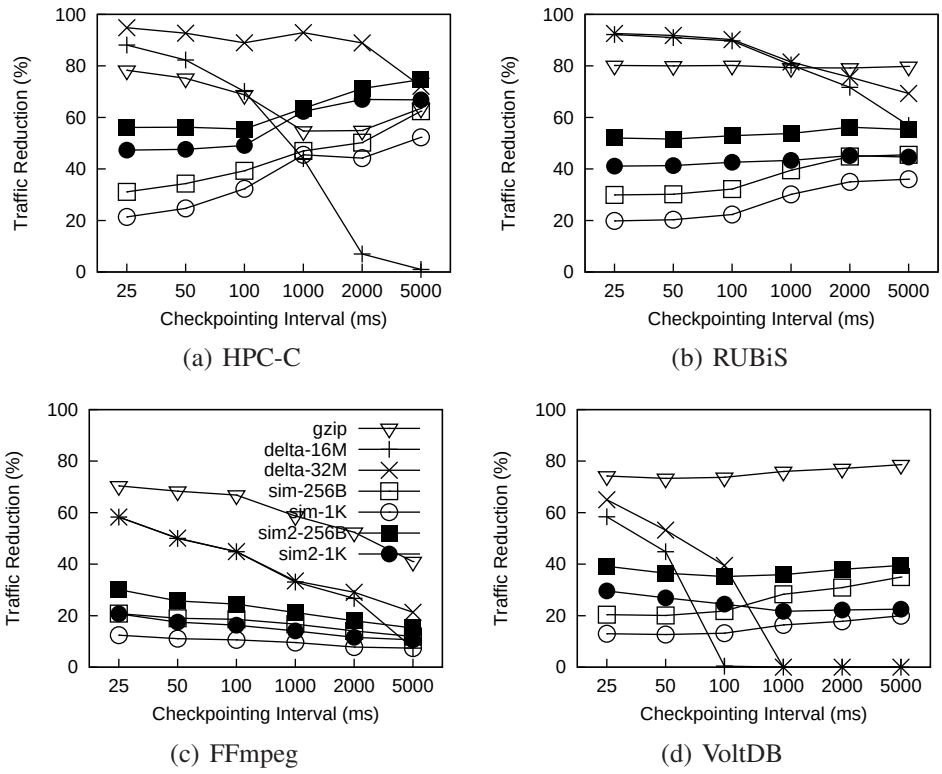
Checkpoint Sizes (MB)	Checkpointing Intervals (ms)					
	5000	2000	1000	100	50	25
HPC-C	19.3	18.2	13.1	3.1	2.1	1.7
RUBiS	13.0	11.4	8.2	4.4	4.1	3.7
FFmpeg	19.0	12.9	10.8	8.0	6.8	5.4
VoltDB	396.7	207.5	114.4	20.0	13.2	9.6

Table 2.2: The average checkpoint sizes of each workload.

entific computations. Our evaluation uses four different workloads of these types. Table 2.1 summarizes the workloads we use and their setup.

We run the workloads in VMs, take periodic checkpoints of the VMs for two minutes of workload execution, and store the checkpoints taken for repeated use in our various experiments. For each workload, we capture multiple series of checkpoints, each using a different checkpointing interval length. We use sub-second (25, 50 and 100 ms) checkpointing intervals to reflect those used in current HA systems [39], and also longer (1, 2 and 5 secs) intervals to explore a wider parameter space. For two minutes of workload execution, checkpointing every 5 seconds to every 25 ms generates 24 to 4800 checkpoints in each series. Table 2.2 summarizes the average checkpoint sizes of the different workloads. While in some cases individual checkpoints seem small, especially when taken at short intervals, sending these checkpoints frequently creates excessive network traffic in only two minutes, as discussed in Section 2.2.2.

All checkpoints are taken on HP Proliant BL465c blades, each with two dual-core AMD Opteron 2.2GHz CPUs, 4–8 GB RAM, one GbE NIC and two SAS 10K rpm disks. All our experiments are run on the same testbed. `emulhacp` and `emulharcv` run on top of Xen in the Domain-0 of two separate blades in the same LAN; this setup resembles the typical setup in a virtualized datacenter where protected and backup hosts are connected by an internal LAN for management operations. In each experimental run, `emulhacp` and `emulharcv` process complete series of checkpoints, and report average results over the checkpoints processed.



(a) HPC-C (b) RUBiS

(c) FFmpeg (d) VoltDB

Figure 2.2: The traffic reductions achieved for a single protected VM running different workloads. The "sim-*" lines show the traffic reductions achieved by similarity compression using different chunk sizes and a single-interval hash table, and the "sim2-*" lines show those achieved using a two-interval hash table.

2.5.2 Traffic Reduction

We first evaluate how each compression method reduces checkpoint traffic for a single protected VM. Figure 2.2 shows the traffic reductions achieved when different workloads are running in the protected VM. *gzip is effective for various workloads and checkpointing frequencies.* It reduces traffic by 40–80% in our evaluation.[4] Smaller traffic reductions are achieved for FFmpeg and HPC-C at 1-second and longer checkpointing intervals. These FFmpeg checkpoints contain large amounts of media contents that are already encoded by video/audio codecs, and hence are not compressed much further by gzip. The HPC-C checkpoints have many numerical values from the computation matrices of the workload.

[4]We use level 1 (fastest) compression for gzip. Our experiments with higher levels of compression show only 1-2% more reductions with a 4x CPU overhead for gzip.

The randomness of these values are not friendly to the gzip compression algorithm.

We evaluate delta compression starting with a 32 MB delta cache, which is large enough to store at least one complete checkpoint for most of the workloads and checkpointing frequencies we use. To test the sensitivity of traffic reduction to cache size, we also evaluate a smaller (16 MB) cache. The traffic reductions achieved by delta compression vary widely in our experiments, ranging from 0% to 92% for different workloads and checkpointing frequencies. *The effectiveness of delta compression is mainly impacted by how the delta cache is sized in relation to the size of the checkpoints. Our results suggest that to achieve more than 40% traffic reductions, the delta cache must be larger than the average checkpoint size of the workload under the checkpointing frequency used.*

A 16 MB cache is effective for RUBiS. When RUBiS is checkpointed at sub-second intervals, a 16 MB cache holds at least 5 consecutive checkpoints, and keeps a good history of the dirty page contents of the workload. The cache produces over 98% hit rates, letting almost all checkpointed pages be compressed before network transmission. In these cases, delta compression outperforms gzip and achieves up to 92% traffic reduction.

A 16 MB cache becomes ineffective at 1-second and longer checkpointing intervals for HPC-C. In these cases, using a 32 MB cache improves traffic reductions by an additional 41–81%, thanks to a 49–89% increase of cache hit rate. However, a 32 MB cache is still far from enough for VoltDB. At 1-second and longer intervals, the checkpoint traffic of VoltDB is not reduced at all. Each of these VoltDB checkpoints contains 100–400 MB of dirty page contents, much more than the delta cache can hold, and all cached dirty pages have to be replaced before enabling any compression. Thus, to reduce VoltDB checkpoint traffic reasonably in these cases, hundreds of MBs of RAM must be provisioned for the delta cache.

Given a sufficiently large cache, delta compression is more effective for workloads that modify memory pages by smaller areas. We observed that even though the cache is comparably effective for FFmpeg and RUBiS, producing over 97% hit rates for both workloads

at sub-second intervals, the checkpoint traffic of FFmpeg is reduced by 34–45% less than that of RUBiS. Our off-line analysis shows that, upon a cache hit, a dirty page of RUBiS is compressed by more than 80%, while FFmpeg's dirty page is compressed by less than 50%. This is because RUBiS modifies small areas in memory pages, but FFmpeg changes each page significantly by reading media contents in 4 KB blocks from disk to be transcoded; compressing FFmpeg's buffer pages by the changes from their past contents does not reduce data size effectively.

We evaluate similarity compression using 256 byte, 1 KB and 4 KB chunks to detect and remove redundancy in checkpoint traffic. For each workload, using 256 byte chunks always reduces traffic more effectively than using 1 KB chunks, which is, in turn, more effective than using 4 KB chunks. In our experiments concerning a single protected VM, similarity compression achieves smaller traffic reductions comparing to gzip and delta compression, ranging from 12% to 62% reduction using a single-interval hash table and 256 byte chunks. In these cases, similarity compression can only utilize intra-VM similarities—traffic is reduced by removing the duplicate checkpoint contents *within* each VM, since only one VM's checkpoints are processed at a time.

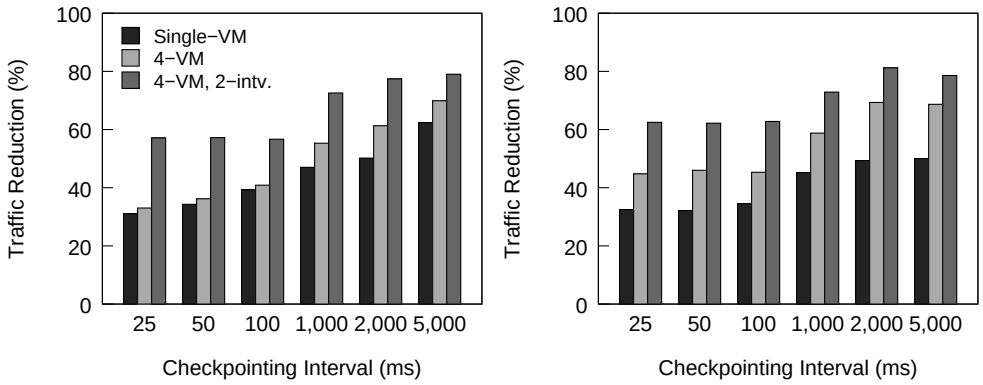
Even in the cases of only one protected VM, our results confirm and quantify the intuition that *similarity compression is particularly effective for workloads that have multiple components collaborating on a shared data set.* Similarity compression is particularly effective for HPC-C in the four workloads evaluated. Using only intra-VM similarity and a single-interval hash table, HPC-C checkpoint traffic is reduced by 47–62% at 1-second and longer intervals. We initially suspected that much of the reduction comes from the elimination of zero pages, generated upon memory allocations by the workload. An off-line analysis shows that less than 2.5% of these checkpointed pages contain all zeros. Thus, similarity compression reduces checkpoint traffic effectively for HPC-C by removing the non-zero workload data duplicated in the multiple processes spawned by the workload to collaborate on computation problems.

Maintaining the hash table across checkpointing interval boundaries improves the effectiveness of similarity compression, especially when checkpoints are taken at sub-second intervals. Using a multi-interval hash table, similarity compression not only removes the checkpoint contents duplicated in the current interval, but also those duplicated with the dirty page contents in the past intervals. Similarity compression achieves up to 25% additional traffic reduction by rebuilding the hash table every two intervals, rather than in every interval. Expanding the hash table further to cover 3 consecutive intervals yields another 10% more traffic reduction, compared to that achieved with a two-interval hash table.

A multi-interval hash table improves traffic reduction especially in short checkpointing intervals, where dirty pages are likely modified by a small degree—similarity compression thus finds and eliminates much more checkpoint redundancy when prior page contents are available in a multi-interval hash table for comparison. Our results also suggest that in these cases, expanding the *time coverage* of the hash table improves traffic reductions more effectively than refining the *granularity* of the hash table, *i.e.*, detecting redundancy by smaller chunks. As shown in Figure 2.2, at sub-second intervals, similarity compression reduces checkpoint traffic more effectively using 1 KB chunks with a two-interval hash table, comparing to using 256 byte chunks with a single-interval hash table.

2.5.2.1 Multiple Concurrently Protected VMs

We apply the compression methods to the checkpoint traffic of four VMs simultaneously, and evaluate the traffic reductions achieved in the following two scenarios: (S1) a HPC cluster, representative of a homogeneous workload environment, and (S2) A heterogeneous mixture of workloads. In S1, each of the four VMs runs an instance of HPC-C. To introduce realistic workload differences in S1, VM1 and VM2 work independently on different problem sets, and VM3 and VM4 collaborate on a third, larger set of problems. In S2, the four VMs run HPC-C, RUBiS, FFmpeg and VoltDB, respectively. In both scenarios, the VMs are co-located in a single protected host, and their checkpoints are captured



(a) An independent HPC-C (b) A collaborating HPC-C

Figure 2.3: The traffic reductions achieved by similarity compression for a HPC-C VM when the VM's checkpoints are processed alone (Single-VM), together with other VMs' checkpoints in a 4-VM HPC cluster (4-VM), and using a two-interval hash table (4-VM, 2-intv.)

at the same frequency.

The overall traffic reduction achieved in a multi-VM cluster roughly equals the average of the traffic reductions achieved for the individual VMs, weighted by the VMs' respective checkpoint sizes. This is a good estimate especially for gzip and delta compression, which process the checkpoint traffic of each VM independently. *Similarity compression is the only method in the three methods evaluated that may achieve additional traffic reductions when processing checkpoints of multiple VMs concurrently, since inter-VM similarity can be utilized*—dirty page contents duplicated across VM boundaries are also eliminated to further reduce checkpoint traffic.

To understand the effect of exploiting inter-VM similarity, we select a VM, and compare the traffic reductions achieved for this VM when its checkpoints are processed alone, versus together with other VMs' checkpoints in a cluster. This comparison is more reasonable than simply comparing the overall traffic reductions achieved in single- and multi-VM scenarios, since that observed in a multi-VM cluster is biased by the individual VMs' checkpoint sizes.

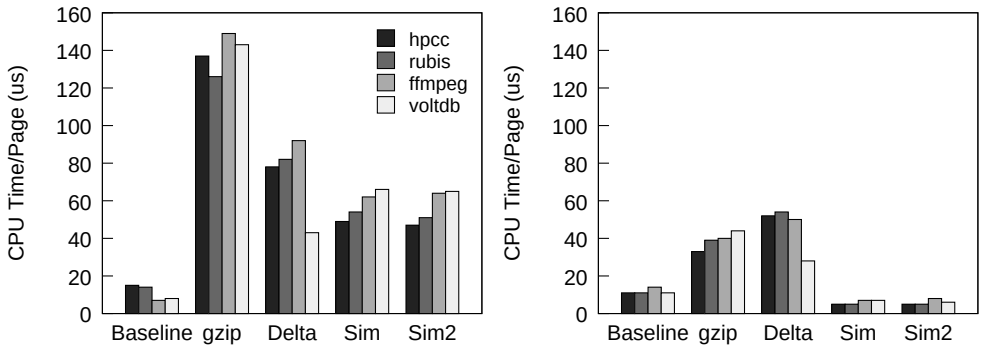
We would like to answer the following questions about inter-VM similarity: (1) How well does inter-VM similarity, when utilized, improve checkpoint traffic reductions in a

homogeneous workload environment? (2) In a homogeneous workload environment, does VM collaboration affect the level of inter-VM similarity that is present? (3) How well does inter-VM similarity and multi-interval hash tables, when used together, improve the effectiveness of similarity compression? (4) Is there any inter-VM similarity in a heterogeneous workload environment?

Figure 2.3(a) shows the traffic reductions achieved by similarity compression for VM1 in scenario S1, which works on a HPC problem set independently. When its checkpoints are processed together with the checkpoints of the other three VMs in the cluster, an additional 2–11% traffic reduction is achieved, compared to processing its checkpoints alone. Figure 2.3(b) shows the traffic reductions achieved for VM3 in scenario S1, which collaborates with another VM in the cluster on the same HPC problem set. Processing this VM's checkpoints with the other VMs' results in greater additional traffic reductions of 11–20% more; even in short, sub-second intervals, non-trivial amounts of inter-VM similarity and thus traffic reduction improvements are observed. Exploiting inter-VM similarity with a two-interval hash table, similarity compression reduces checkpoint traffic by 57–81% for these VMs, as effectively as gzip.

Similarity compression is effective for homogeneous workload environments, where non-trivial amounts of inter-VM similarity exist, especially when VMs collaborate on a common task set. However, limited similarity is found between VMs running heterogeneous workloads, for which similarity compression is a poor fit. We found in the workload mixture of scenario S2 that each VM's checkpoint traffic is hardly reduced further when their checkpoints are processed together, compared to processed independently. In the four VMs in S2, the greatest improvement on traffic reduction is observed for RUBiS to be only 6% more.

Note that the additional traffic reductions achieved by similarity compression in a multi-VM environment may be considered a lower-bound of the inter-VM similarity present in the environment, for two reasons. First, one copy of the duplicate contents must be sent over



(a) In the protected host (b) In the backup host

Figure 2.4: The CPU cost of checkpoint replication for a single protected VM running different workloads. No compression is used in the baseline. Delta cache is 32MB. Similarity compression uses 256 byte chunks with a single-interval (Sim) and a two-interval (Sim2) hash table.

the network, and is not counted in traffic reduction. Second, some contents are duplicated both across VM boundaries and within individual VMs, and are already removed when processing checkpoints of individual VMs independently. The elimination of these duplicates is thus not reflected in the additional traffic reduction gained by processing checkpoints of multiple VMs together.

2.5.3 CPU and Memory Costs

We now examine the resource requirements for each compression method to achieve the traffic reductions presented in Section 2.5.2.

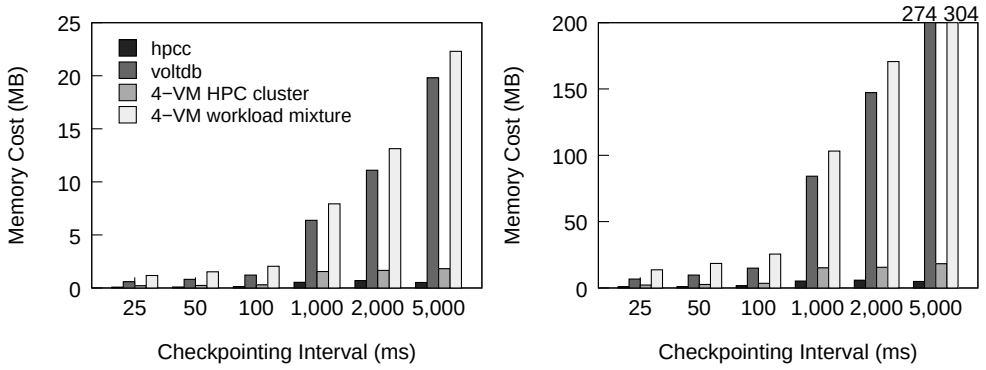
Figure 2.4 shows the per-page CPU cost of the compression methods measured in the protected and backup hosts. We show the measurements taken at 100 ms checkpointing intervals for the single protected VMs running the different workloads; the measurements in our other experimental cases show consistent trends. In our experiments, checkpoint replication for each VM can use one CPU core in the protected and backup hosts, respectively. The CPU usage in the protected host is generally larger than that in the backup host, as shown in Figure 2.4.

While reducing checkpoint traffic effectively, gzip requires significant CPU for compres-

sion. In the protected host, gzip typically shows an above 80% CPU usage. It incurs the largest CPU cost in the three methods evaluated. It also uses more CPU when compressing less effectively: gzip achieves lower traffic reductions at higher CPU costs for FFmpeg and HPC-C compared to the other workloads. While gzip can operate at a minimal memory cost—a 4 KB buffer per VM to contain the compression outcome as checkpoints are processed page-by-page, its high CPU usage can greatly impact the protected VMs' normal operation, especially when checkpoints of multiple protected VMs are processed concurrently.

Delta compression uses less CPU for compression than gzip. It typically shows a 70–80% CPU usage in the protected host. Though it uses fixed-size caches of 16MB and 32MB in our experiments, we note that *for workloads that dirty large areas of memory rapidly, delta compression can use excessive RAM in the protected host to cache transmitted dirty pages for effective compression.* VoltDB is an example of such workloads. At 1-second and longer intervals, delta compression cannot reduce VoltDB's checkpoint traffic at all using 16MB and 32MB caches. To achieve reasonable traffic reductions in these cases, up to 400MB of memory is required for the delta cache, since as discussed in Section 2.5.2, delta compression needs a cache larger than the average checkpoint size to reduce checkpoint traffic by more than 40%. If a few such VMs are protected at the same time, the total memory consumption of delta compression can grow quickly and create memory pressure in the protected host.

In the three methods evaluated, similarity compression uses CPU most efficiently. Typically, it shows a 50–60% CPU usage in the protected host. Higher CPU costs are incurred when checkpoints are processed by smaller chunks, since more MD5 digests are computed for each dirty page. But even when using 256 byte chunks (computing 16 digests per page), its CPU cost is the lowest of the three methods evaluated. Similarity compression also uses little CPU in the backup host, since it "decompresses" checkpoints simply by copying duplicate contents from reference chunks, requiring no additional computations. In some



(a) A single-interval hash table (b) A two-interval hash table

Figure 2.5: The memory cost of similarity compression using 256 byte chunks.

cases, it even uses less CPU than receiving and storing uncompressed checkpoints in the backup (the baseline), thanks to the reduced checkpoint data sizes.

Similarity compression requires memory in the protected host for the hash table. The hash table installs one entry for each unique chunk processed, so it uses more memory as more unique chunks are recorded. Figure 2.5(a) shows the memory cost of similarity compression with a single-interval hash table. In the four workloads, similarity compression incurs the smallest memory costs for HPC-C and the largest memory costs for VoltDB. It uses up to 20 MB of memory to record the chunk contents of the large VoltDB checkpoints in the hash table. For the other three workloads, it incurs low memory overheads, ranging from 95 KB to 1.3 MB when using 256 byte chunks, and less (8–350 KB) when 1 KB and 4 KB chunks are used.

Similarity compression uses memory more efficiently in the presence of greater VM similarity, since fewer unique chunks need to be stored in the hash table. In our 4-VM HPC cluster (scenario S1 in Section 2.5.2.1), it uses less than 2 MB of memory at all times to process the checkpoints of all four VMs concurrently with a single-interval hash table, and is able to reduce checkpoint traffic effectively thanks to the significant intra- and inter-VM similarity present in the cluster. However, in the workload mixture of scenario S2, it uses up to 22 MB of memory with most entries in the hash table recording the unique contents of the large VoltDB checkpoints.

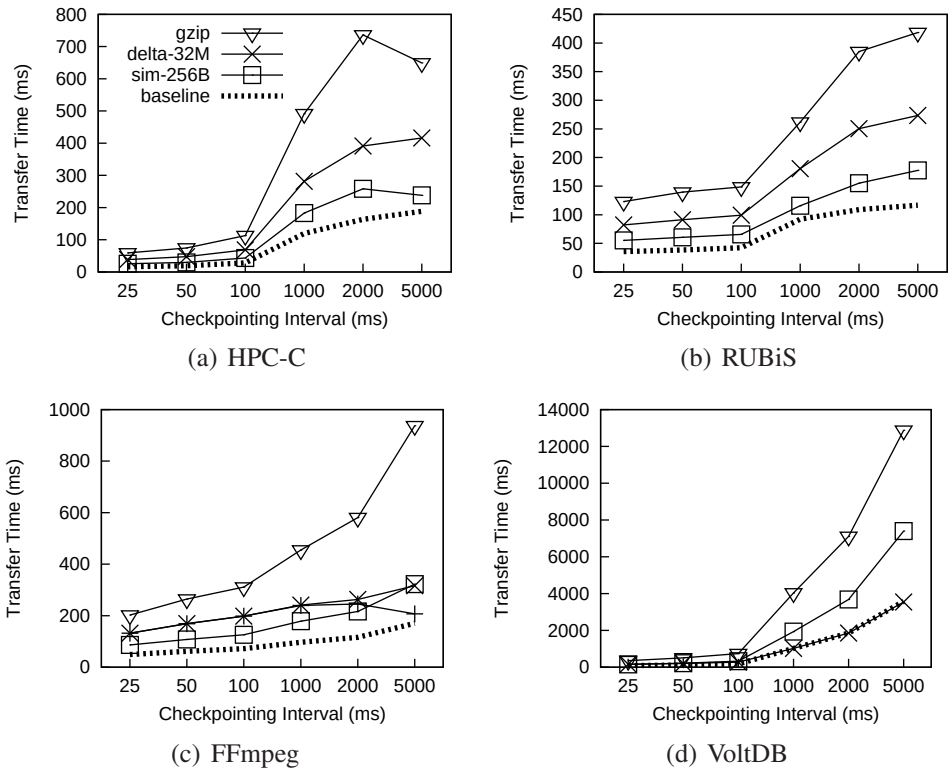


Figure 2.6: The checkpoint transfer time of a single protected VM running different workloads.

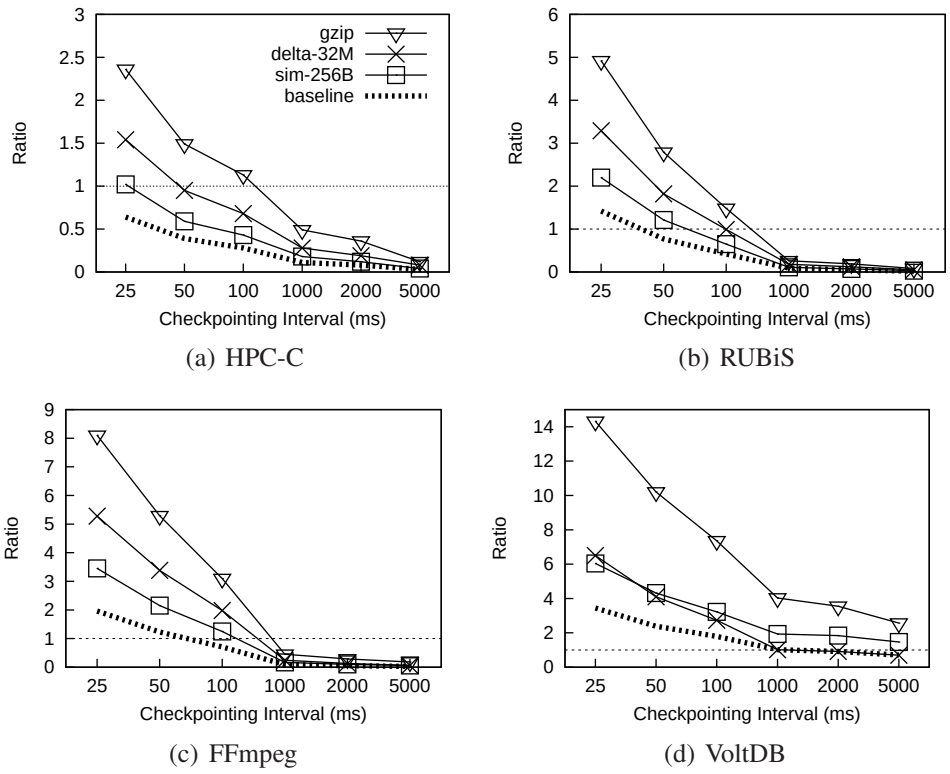
Larger memory costs are incurred when similarity compression uses a two-interval hash table, as shown in Figure 2.5(b), since additional memory in the backup is required to cache the unique chunks in the previous interval for possible references; this cache constitutes the main part of the memory consumptions. In the HPC cluster of scenario S1, using a two-interval hash table improves traffic reduction—similarity compression reduces checkpoint traffic by up to 81%, at a moderate memory cost of 2–18 MB. However, in a heterogeneous workload environment, like the workload mixture in scenario S2, a multi-interval hash table may not be suitable. Since limited VM similarity exists in such an environment, a multi-interval hash table may likely incur a large memory overhead by caching unique checkpoint contents in the backup, without effectively improving traffic reduction.

2.5.4 Checkpoint Transfer and Storage Time

In addition to consuming resources, checkpoint compression affects the time taken to transfer and store each checkpoint. Figure 2.6 shows the average checkpoint transfer time incurred by a single protected VM running the different workloads. For all workloads, transfer time is the shortest when compression is not used (the baseline), ranging from 16 to 188 ms except for VoltDB; VoltDB checkpoints are larger and each takes much longer to finish replication.

In these experiments concerning a single protected VM, uncompressed VM checkpoints are usually replicated within each configured interval, unless the intervals are 50 ms and shorter. Take FFmpeg as an example. When checkpointed every 50 and 25 ms, an uncompressed checkpoint takes an average of 61 and 49 ms, respectively, to replicate. Since each checkpoint does not finish replication within the configured interval time, subsequent checkpoints must be delayed, and the configured checkpointing frequencies (one checkpoint every 50 and 25 ms) are not achieved. For VoltDB, even lower frequency of checkpointing every 1 second cannot be achieved.

Compression lengthens the checkpoint transfer time incurred by the single protected VMs, and gzip requires the longest transfer times in the three methods evaluated. Figure 2.7 plots the ratio of the measured transfer time in Figure 2.6 to the configured checkpointing interval. A ratio greater than 1 indicates that the elapsed time between two consecutive checkpoints is larger than the configured interval length, and the configured checkpointing frequency is not achieved. Even though gzip sends only 20–60% of the original checkpoint data over the network, it incurs transfer times up to 14x longer than the configured checkpointing interval, due to performing compression. As a result, for VoltDB, none of the checkpointing frequencies evaluated can be achieved, and for all other workloads, checkpointing every 100 ms and higher frequencies are no longer feasible when gzip is used. The long checkpoint transfer times required by gzip can limit its applicability for server applications that are very sensitive to network latencies.



(a) HPC-C (b) RUBiS

(c) FFmpeg (d) VoltDB

Figure 2.7: The ratio of the measured transfer time in Figure 2.6 to the configured checkpointing interval.

Similarity compression requires the shortest checkpoint transfer times for the single protected VMs, as well as the shortest checkpoint storage times in the three methods evaluated. It takes 1.5x less time than delta compression to replicate each checkpoint of the protected VM using 256 byte chunks and a single-interval hash table. Even shorter transfer times are required when processing checkpoints by larger chunks and a multi-interval hash table. It also stores each replicated checkpoint to the fail-over image as fast as if compression were not used, incurring almost the same amount of checkpoint storage time as in the baseline cases, as Figure 2.8 shows.

Checkpoint decompression of gzip and delta compression takes a non-trivial amount of time to complete, resulting in prolonged checkpoint storage times and lowering the checkpointing frequency actually achieved. In our experiments, gzip and delta compression incur up to 10x and 13x longer storage times compared to similarity compression, respectively.

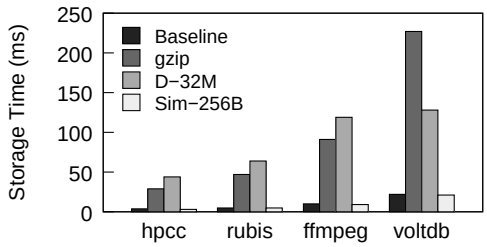
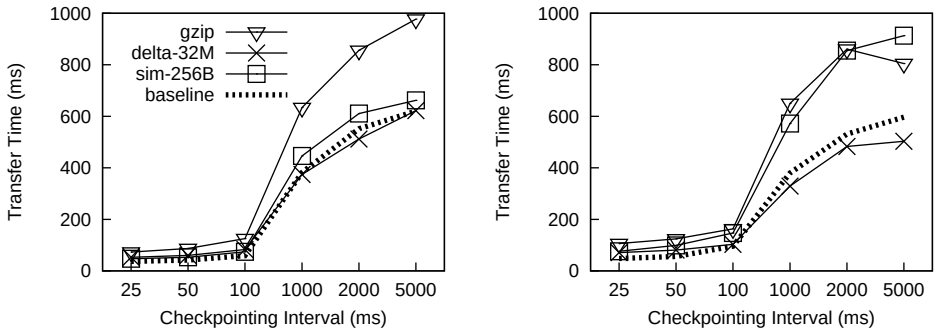


Figure 2.8: The storage time of each checkpoint of a single protected VM running different workloads.



(a) In a HPC cluster (b) In a workload mixture

Figure 2.9: The checkpoint transfer time of a HPC-C VM measured in our 4-VM HPC cluster and workload mixture scenarios.

We observed that using the two methods, replication of a subsequent checkpoint is often delayed by the previous checkpoint not being completely stored and releasing its buffer in time for a new incoming checkpoint, especially when checkpoints are configured to be replicated at sub-second intervals. The checkpointing frequencies actually achieved thus decrease.

2.5.4.1 Transfer Time in Multi-VM Environments

To understand how checkpoint transfer time varies when multiple VMs are concurrently protected, we consider the transfer times incurred by a HPC-C VM in our multi-VM experimental scenarios S1 and S2, the HPC cluster and workload mixture, as shown in Figure 2.9.

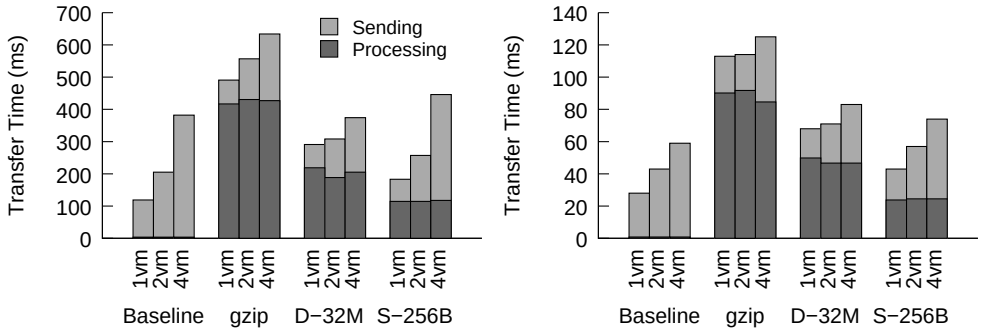
While gzip consistently requires long transfer times, those incurred by similarity com-

pression show an interesting variation. As discussed above, similarity compression requires the shortest transfer times in the three methods evaluated for the single protected VMs running different workloads. This observation remains true in our various experiments with a 2-VM HPC cluster. However, as the number of VMs in the HPC cluster increases to 4 (scenario S1), in some cases, the transfer times incurred by delta compression become the shortest. In the workload mixture (scenario S2), similarity compression even takes as long as gzip to replicate each checkpoint.

These results suggest that *similarity compression may lose the advantage of requiring small checkpoint transfer times as the number of concurrently protected VMs increases, and we have found that this is the case when the target environment is bottlenecked on network bandwidth, but not CPU*, as we will discuss below. Our testbed happens to represent such a computing environment.

Checkpoint transfer time consists of processing time, during which dirty pages are compressed, and sending time, during which the pages are transferred over the network. Compression lengthens processing time while reducing sending time, and the two components vary by different degrees for different compression methods. A computation-intensive method can greatly increase processing time, while an effective method greatly reduces sending time. The resource availability in the target environment also impacts the two components in different ways: abundant CPU improves processing time, while faster network improves sending time.

In an environment with limited network bandwidth (assuming abundant CPU), sending time plays a more important role than processing time in the variation of checkpoint transfer time. Using an effective method like gzip, less dirty data is sent for each protected VM, and as the number of VMs increases, sending time grows more gently. By contrast, since similarity compression reduces traffic less effectively, with more VMs concurrently protected, sending time increases more rapidly. As a result, similarity compression may begin to require longer transfer times than the other methods when the network becomes



(a) 1sec intervals (b) 100ms intervals

Figure 2.10: The breakdown of the checkpoint transfer times of a HPC-C VM measured in our HPC clusters consisting of 1, 2 and 4 VMs.

saturated.

We verify this reasoning by breaking down the transfer time measurements into processing time and sending time. We estimate the (elapsed) processing time by the average CPU time spent for each checkpoint. This estimate is reasonable, since there is minimal CPU sharing in our testbed: when `emulhacp` spawns four threads to replicate checkpoints for 4 VMs in parallel, CPU is almost always available to the threads in our server, which has 4 physical cores. We then consider the remainder of the transfer time as sending time, which is a lower-bound of the time required to send the dirty data (the part not overlapping with processing time).

Figure 2.10 shows the breakdowns of the transfer times incurred for a HPC-C VM measured in a HPC cluster consisting of 1, 2, and 4 VMs; we show the breakdowns in cases of 1 second and 100 ms checkpointing intervals as an example. While similarity compression requires the shortest processing time, as the number of VMs increases, its sending time grows by a larger degree comparing to gzip and delta compression. Comparing Figure 2.10(a) with Figure 2.10(b), the sending time of similarity compression also increases more significantly in the cases of longer checkpointing intervals, where a larger amount of checkpoint data is generated in the network in each interval. Therefore, similarity compression requires longer transfer times than delta compression when the network is saturated to a certain degree. As the amount of checkpoint traffic keeps increasing, similarity compres-

sion may take as long as gzip, or even longer, to replicate each checkpoint.

2.6 Discussions on Compression Method Selections

From our evaluation, one can hardly find a single best compression solution, and compression methods should be selected based on the workload types and resource constraints in the target environment.

Our results suggest that similarity compression is the most suitable for VM clusters running homogeneous workloads, especially when VMs in the cluster collaborate on a shared task set. In these scenarios, similarity compression achieves satisfactory traffic reductions using both CPU and memory efficiently, and it becomes even more effective with a multi-interval hash table. For example, using a two-interval hash table in our 4-VM HPC cluster yields up to 32% more traffic reduction than a single-interval hash table would achieve, without incurring an undue resource overhead. The additional traffic reduction also helps shorten checkpoint transfer time further; similarity compression regains the advantage of requiring the shortest transfer time in the three methods evaluated with a two-interval hash table in our HPC cluster. Note, however, that maintaining the hash table across many intervals may not be necessary, since the additional traffic reductions gained decrease with increasing resource costs.

For other workload scenarios, especially a heterogeneous mixture of workloads, gzip and delta compression are better suited. These methods are effective for a wider range of workload types than similarity compression. When using them, the resource availability in the target environment must be considered.

To use gzip, the protected host must have abundant CPU to support checkpoint compression in addition to normal operation of protected VMs and applications. Since gzip generally reduces checkpoint traffic effectively at a high CPU cost, it is bottlenecked on checkpoint compression rather than transmission. If processing checkpoints of multiple VMs creates severe CPU contention in the protected host, the usefulness of gzip can de-

grade. In such a case, not only are VM operations in the protected host affected, each checkpoint of the protected VMs also takes longer to replicate. Consequently, the VMs must be checkpointed at lower frequencies, and this can potentially make gzip unusable for VMs running latency-sensitive server applications.

For delta compression to be effective, sufficient memory must be available and provisioned for the delta cache. Our results show minimal traffic reductions when the delta cache is smaller than the average checkpoint size of the protected VM. It is thus critical to properly size the delta cache according to the memory access behavior of the target workloads and the checkpointing frequencies used. A proper cache size may be determined by profiling the target workloads *a priori*. Alternatively, the cache may first be over-provisioned and then dynamically adjusted as the workloads execute.

2.7 Related Work

This chapter examines the prohibitive network requirements of checkpoint replication. Two types of approaches have been proposed to reduce checkpoint replication traffic in HA systems. One reduces the amount of VM state to protect/checkpoint; the other checkpoints the full VM state, but reduces the amount of data sent for each checkpoint taken.

RemusDB [69], a highly available database system in VM, uses the first approach. It does not checkpoint clean disk buffers, and "de-protects" certain data structures in the database system that can be regenerated after a failure. Therefore, it creates smaller checkpoints and less replication traffic in the network. However, it requires in-depth understanding of the applications running in a protected VM to identify data structures that can be safely de-protected. The VM and applications must also be instrumented to recover uncheckpointed state after a fail-over.

Checkpoint compression exemplifies the second type of approach. Compression is generally applicable regardless of the applications in the VMs, and requires less system instrumentation; hence we focus on compression in our study presented in this chapter. The

authors of [39] briefly discussed compressing checkpoints by gzip and delta compression, although a thorough evaluation was not included. RemusDB [69] and SecondSite [80], designed for database HA and datacenter disaster recovery, respectively, use delta compression in their systems. They both find page delta by XOR, and compress the delta by RLE, like evaluated in this chapter. Lu *et al.* propose fine-grained dirty region tracking (FDRT) [64], which shares the same concept of delta compression. FDRT divides each dirty page into fixed-size regions, and finds the regions that are actually modified by comparing content hashes. It then replicates the modified regions to the backup instead of the entire page. While delta compression has been used in HA systems and during live VM migration [92, 97], our study compares it with other types of compression methods to understand when it is best suited for use.

Similarity compression exploits memory content redundancy to reduce checkpoint replication traffic. In a broader context, content redundancy is widely used for storage deduplication to improve I/O performance [59, 78, 81], and in network infrastructures to improve network capacity and end-to-end application performance [27, 28]. Content redundancy in VM memory is often utilized to share identical pages and reduce host memory pressure [49, 70, 96]. Different from these systems, which find and coalesce redundant pages in the entire memory space of co-located VMs, similarity compression finds redundant data in dirtied memory pages. It also detects redundancy much more frequently than these systems as well as VM migration systems that use redundancy elimination [82, 87, 97].

Gerofi *et al.* presents another way to utilize VM similarity and reduce checkpoint replication traffic [48]. Instead of finding "identical" data in dirty pages, the authors find memory areas that are "similar" to the dirty pages, and send only the differences between the dirty pages and these memory areas over the network; their approach is currently applied to each VM independently. The same idea has also been used to reduce VM migration traffic [42, 99]. Note that similarity compression examines the content similarity in the memory space of live, executing VMs, in contrast to systems like VMFlock [25], which

utilize the similarity in VM disks to reduce the traffic incurred for migrating static VM disk images.

2.8 Conclusions

Reducing checkpoint traffic is crucial to using checkpoint replication for maintaining VM availability. Although checkpoint compression methods are available, they have not been thoroughly evaluated and compared in the context of supporting HA. In this chapter, we conducted a detailed characterization of three checkpoint compression methods, gzip, delta compression, and similarity compression, based on their effectiveness and overheads. Our evaluation uncovered the different strengths and weaknesses of each method, and provided guidelines for selecting and using these methods based on the workload types and resource constraints in a HA system. The evaluation framework developed in this chapter is generic, and can be used to evaluate other compression methods.

CHAPTER III

HydraVM: Memory-Efficient High-Availability for Virtual Machines

3.1 Introduction

To restore a failed VM and resume its execution from where it left off in the failure, existing approaches create a backup VM in a separate host for each primary running VM, and synchronize a primary VM's runtime state to its backup by replaying VM instructions [20, 32] or replicating VM checkpoints [24, 39, 44, 47, 93]. The backup stands by in the background until a failure of the primary occurs, at which point it becomes active and takes over execution from the primary's state before the failure.

Using backup VMs, existing HA approaches reduce the amount of completed work lost upon failures, but at a high resource cost. In particular, each backup VM reserves as much memory as its primary. This reserved memory space cannot be used by other VMs even though the backup remains inactive during normal VM operations. Making backup memory reservation is expensive for several reasons. First, each VM that desires HA support incurs a 100% memory overhead without any increase in its system throughput. Second, the aggregate backup memory reservation made for a group of protected VMs can significantly and unproductively consume RAM, a scarce and expensive resource. Third, as many-core processors become pervasive, memory is increasingly the resource bottleneck

of VM consolidation [49]. Inactively blocking host memory for backup VMs may hinder effective consolidation of active running VMs, and result in under-utilization of other host resources (e.g., CPU cores) and degradation of the overall resource efficiency in a virtualized environment.

In this chapter, we propose *HydraVM*, a memory-efficient HA approach for VMs. The primary objective of HydraVM is to protect VMs against failures of their hosting machines *without* any backup memory reservation. Instead of creating a backup VM, HydraVM keeps track of the runtime state of a protected VM in a *fail-over image* maintained in a networked, shared storage, which is commonly deployed in a virtualized environment to hold VM disks and facilitate VM management and migration. In case of a failure, HydraVM quickly restores the VM from the fail-over image and resumes its execution.

HydraVM keeps track of VM runtime state by taking incremental VM checkpoints, as existing HA systems do [24, 39, 44, 47, 93]. Different from existing systems, which store the checkpoints taken in an in-memory backup VM, HydraVM consolidates the checkpoints in a shared storage. HydraVM utilizes inexpensive storage, and frees up expensive RAM for better usage. A failed VM can be recovered in any host that has sufficient resources, since HydraVM maintains the VM state needed by fail-over in a shared storage instead of a dedicated backup host. However, fast VM recovery is challenging for HydraVM, since the fail-over state must be loaded from the shared storage. To bring a failed VM back alive quickly, HydraVM performs a *slim VM restore*, which loads only a small amount of critical VM state from the fail-over image to restore a VM, and activates the execution of the restored VM immediately. As the VM executes, the VM state not yet loaded is supplied on-demand.

HydraVM was not intended to provide "hot mirroring" of VM state as existing approaches that use in-memory backup VMs do, since each VM checkpoint takes longer to store on a storage device than in memory. With hot mirroring, existing approaches buffer, and checkpoint, VM network outputs before releasing them, to provide transparent failure

recovery for client-facing applications. However, even with hot mirroring, latency-sensitive applications can suffer from significant performance degradations due to network buffering [39, 44]. HydraVM does not buffer network outputs for protected VMs with its "warm mirroring" of VM state, and therefore, does not provide transparent recovery for server applications. We make this tradeoff to maintain fail-over VM state in inexpensive storage, and provide a cost-effective HA alternative for applications which do not require network buffering, including long-running computation jobs and distributed applications that run in a VM cluster. These applications benefit from the protection of HydraVM at a reduced resource cost without requiring any modification.

We implemented HydraVM based on Xen, and evaluated it using workloads that may benefit from HydraVM. Our results show that HydraVM provides VM protection at a low overhead, and can recover a failed VM within 2.2 seconds. While the individual techniques used in HydraVM have been applied to different problems, the contribution of this chapter is a new combination of the techniques that solves the real-world problem of providing resource-efficient HA support for VMs and the demonstration of the applicability of this solution. Our aim is not to replace existing HA approaches that use in-memory backups—they are needed especially for client-facing applications to benefit from transparent failure recovery, but rather to complement the HA toolbox currently available with a cost-effective alternative.

The remainder of this chapter is organized as follows. Section 3.2 discusses the design rationale, and provides an overview of HydraVM. Section 3.3 and Section 3.4 detail the VM protection and recovery mechanisms in HydraVM, respectively. Section 3.5 evaluates HydraVM experimentally. Section 3.6 discusses alternative storage architectures that HydraVM may take advantage of, especially in large environments. Section 3.7 summarizes related work, and the chapter concludes with Section 3.8.

3.2 HydraVM Design

The design of HydraVM is based on a key observation that existing HA approaches incur a high cost by maintaining backup VMs in memory. It is our primary objective to reduce backup memory reservation and provide a memory-efficient HA alternative.

3.2.1 Storage-based VM High-Availability

One may consider downsizing backup memory reservation by memory ballooning [96]. Ballooning is designed for adjusting VM memory allocation at runtime, and requires the VM to cooperate. It is very challenging for a backup VM to exercise its balloon driver, since the backup may not be operational, and its state changes must follow that of the protected VM very closely.

Another way to reduce the overhead of backup memory reservation is to make the memory usable for other actively running VMs, *i.e.*, to time-share a backup VM's memory space with other VMs in the host. To do this, a host-wide paging system may be needed to schedule physical page frames between hosted VMs. Unfortunately, since this extra level of paging can introduce performance anomalies due to unintended interactions with the paging system in the guest OS [96], current platform virtualization technologies, such as Xen and VMware ESX, either do not support, or do not prefer host-wide paging. It is thus difficult to let other running VMs use the memory reserved by a backup.

A third alternative is to page out backup VMs [34], reclaim their memory and re-distribute to active VMs. This approach appears feasible, since a backup VM remains inactive until the protected VM fails. However, to be able to take over execution from where the protected VM left off in the event of a failure, the backup needs to be swapped in very frequently to synchronize with the execution state of the protected VM. This overhead is non-trivial, and can offset the resource benefits gained by swapping the backup out.

HydraVM "pages out" the backup VM to *eliminate* the unproductive reservation of backup memory, and shortcuts the state synchronization process by updating the backup's

on-disk pages without paging them back in. In other words, HydraVM takes a *storage-based* approach. Instead of reserving memory in an additional server, we "maintain" the backup VM in a stable storage. For each protected (primary) VM, HydraVM maintains in a networked, shared storage the state needed by fail-over, based on which a backup VM can be quickly restored and activated to take over execution when the primary fails. The storage system holding the VM fail-over states is assumed to be fail-independent of and accessible from the physical servers that host VMs.

Rather than adding a new shared storage into a virtualized environment for the sole purpose of providing VM protection, HydraVM utilizes the one that is already installed—the shared storage commonly deployed as a central store for VM images. This shared storage may be provisioned in various ways in practice, to deliver the aggregate capacity and combined throughput of a federation of devices and meet the I/O demands of all VMs running in the environment. For example, it may be a shared block storage accessed via a Fibre Channel or iSCSI-based storage area network (SAN), or a cluster filesystem. To ensure undisrupted accesses to VM data, this storage is usually built with redundant connections to and from the VM hosts, as well as storage-level reliability mechanisms against device and controller failures. HydraVM depends on these properties to assume the reliability and availability of the shared storage, and focuses on maintaining VM availability in the face of VM host failures.

3.2.2 An Overview of HydraVM

HydraVM is designed to provide VM protection from fail-stop host failures. It does not attempt to recover VM execution from non-fail-stop conditions caused by configuration errors, software bugs, and so on.

HydraVM has two operation modes, *protection* and *recovery*. We give a brief overview of HydraVM below, and detail its two operation modes in Section 3.3 and Section 3.4, respectively.

During normal operation of a primary VM, HydraVM operates in the protection mode. HydraVM maintains a backup copy of the primary VM state in a shared storage. As the primary executes, the backup state is periodically synchronized with the changing state in the VM, such that in the event of a primary host failure, the VM can be recovered based on its backup state without losing much of the completed work. Like existing approaches [24, 39, 44, 47, 93], HydraVM replicates checkpoints of the primary VM to update the backup VM state. Unlike previous systems, HydraVM consolidates and stores the checkpoints in storage, without making any backup memory reservation.

Upon detection of a failure of the primary host, HydraVM switches to the recovery mode and responds to the failure. A restoration host with available memory must be selected, either from the stand-by hosts, or from the surviving hosts. HydraVM then initiates a fail-over to restore the failed primary VM in the restoration host. HydraVM performs a "slim" VM restore, which loads only a small amount of critical VM state from the shared storage to instantiate and restore the VM. It activates the restored VM immediately to take over execution from the most recent runtime state recorded before the failure. As the VM executes, the state not loaded during fail-over is supplied on-demand.

We implemented the protection and recovery mechanisms of HydraVM based on Xen (version 3.3.2) as management commands to be invoked via the `xm` interface. Currently, HydraVM does not implement custom-built mechanisms to detect host failures and select restoration hosts, and assumes to cooperate with and be informed by existing failure detectors (e.g., [83]) and cluster resource managers (e.g., [19, 100]) of these decisions.

3.2.3 Advantages and Limitations

The storage-based HydraVM approach offers several advantages. First, by using inexpensive storage to maintain the VM state necessary for fail-over in place of expensive RAM, the hardware costs of providing HA support are reduced. Second, the memory reserved by inactive backup VMs are freed up for better usage. They may be added to active

VMs for performance enhancements, or used to create new VMs and consolidate existing ones more effectively. Third, since HydraVM maintains the fail-over states in a shared storage instead of dedicated backup machines, in the event of a failure, the affected VMs may be recovered in any physical host that has access to the storage. This allows fail-over to any host that currently has sufficient spare memory and other resources. This ability is critical given the highly variable utilization of hosts in a virtualized environment. Finally, using HydraVM, a relatively small number of spare hosts needs to be provisioned in anticipation of failures, instead of a large number of hosts passively synchronized with the protected VMs, since a single host can now back up or protect many more VMs, as long as they do not all fail together.

However, maintaining fail-over state in storage also causes limitations for HydraVM, mainly because persistent storage devices are orders of magnitudes slower than RAM. When maintaining backup VMs in memory, the primary and backup can be synchronized quickly and frequently, or kept in lock-step, even though significant overheads are incurred [32, 39, 44]. In HydraVM, each synchronization of the primary and backup VM state takes much longer to finish on storage devices. HydraVM thus has to replicate VM checkpoints and update backup VM state much less frequently than approaches using in-memory backup VMs.

Previous approaches use in-memory backup VMs and frequent checkpointing mainly to enable network buffering. Network buffering is the act of withholding outgoing network packets of the primary VM until the checkpoint that captures the state from which the packets are generated is fully replicated and acknowledged by the backup. Frequent checkpointing is necessary, since network outputs must be released frequently after each checkpoint to not incur unacceptably long delays. However, even when checkpoints are taken and replicated as frequently as 10–40 times per second, latency-sensitive applications still suffer from significant performance degradations [39, 44].

Network buffering ensures that any exposed state is recoverable if a failure occurs, and

is needed when the protected VM interacts with external clients and transparent failure recovery is required. However, for most client-facing applications, any storage-based HA approach does not replicate checkpoints frequently enough to sustain reasonable application performance with network buffering. HydraVM currently does not buffer network outputs for protected VMs, and does not provide transparent recovery for server applications constantly interacting with external clients. Without cooperation from the end-user or programmer during recovery, server applications should be protected by in-memory backup VMs with network buffering, even though at larger application and resource overheads.

HydraVM makes this design tradeoff to maintain the state needed by VM fail-over in storage, instead of RAM, and provide a cost-effective HA alternative for application scenarios in which frequent checkpointing and network buffering are not needed. HydraVM protects long-running computing jobs which can be prohibitively expensive to repeat after a failure occurs, such as scientific computation and simulation, at a reduced resource cost and without requiring any modification to the application. In a cluster, HydraVM can work with distributed snapshot algorithms [46] to provide coordinated protection and recovery for VMs running distributed and multi-tiered applications. Alternatively, HydraVM can work with synchronized clocks to checkpoint and protect multiple VMs in a cluster concurrently.

3.3 VM Protection

During normal execution of a primary VM, HydraVM maintains a backup copy of the VM state in a shared storage, and updates the backup state continuously as the primary runs.

There are two main approaches to primary-backup state synchronization. Log-and-replay records all instructions and non-deterministic events executed by the primary VM, and replays them deterministically to generate an identical state in the backup [20, 32]. This approach is not suitable for HydraVM, since by maintaining the backup state in a storage,

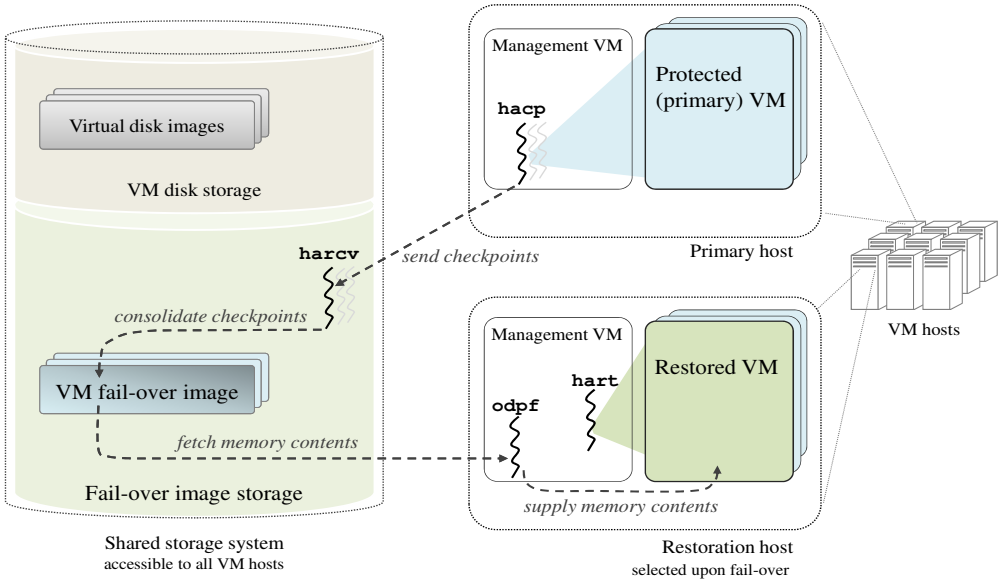


Figure 3.1: The HydraVM system.

we essentially page out the entire backup VM, and to replay primary VM execution would require frequently bringing pages of the backup VM back in. Therefore, HydraVM uses the other approach, checkpoint replication [24, 39, 44, 47, 93], which captures the entire state of the primary VM in checkpoints and replicates them to the backup.

3.3.1 Checkpointing VM CPU and Memory State

Figure 3.1 illustrates the architecture of HydraVM. For each primary VM, HydraVM runs a checkpointing daemon, called `hacp`, to enforce the protection of the VM. `hacp` first creates a *fail-over image* for the VM by taking a full VM checkpoint and replicating it to the fail-over image storage; this full checkpoint is taken only once at the beginning of the protection. The fail-over image contains the backup state of the VM, based on which the VM can be restored in case a failure occurs. Specifically, a fail-over image contains (1) the configuration of the VM describing the VM's allocated resources, virtual devices, pages shared with the hypervisor, and so on, (2) the VM's virtual CPU state, and (3) all memory pages of the VM, laid out sequentially in the order they appear in the VM's memory space.

In our implementation, (1) and (2) contain only 60 KB of data, so the size of the fail-over image is just slightly larger than the memory size of the VM.

As the VM continues executing, `hacp` takes incremental checkpoints periodically. Each incremental checkpoint captures the *changes* of the VM's CPU and memory state since the last checkpoint taken. We implemented `hacp` by modifying the stop-and-copy stage of Xen's live migration support [38]. `hacp` uses Xen's shadow page tables to find the memory pages modified in each checkpointing interval, by putting the execution of a primary VM in *log-dirty* mode. In this mode of operation, Xen maintains a private (shadow) copy of the VM's page tables, in which all pages of the VM are marked read-only. A write to a page generates a fault and traps into Xen, so Xen can record the page in a dirty bitmap.

At the end of each checkpointing interval, `hacp` pauses the VM momentarily to capture a consistent set of changed VM state while the VM is not executing. It finds the dirty memory pages based on the dirty bitmap, and copies their contents along with the CPU state of the VM to a buffer. It then resets the dirty bitmap to record for the next checkpoint, and un-pauses the VM. While the VM continues executing, `hacp` sends the buffer containing the checkpointed VM state to the fail-over image storage.

HydraVM replicates each checkpoint asynchronously, overlapping checkpoint replication with VM execution as existing HA systems do [39, 44, 47]. Unlike previous systems, which store checkpoints in server memory, HydraVM writes checkpoints to storage. A checkpoint may be stored as an individual patch file, or merged with the fail-over image in the shared storage. The first approach may store the checkpoint faster with sequential I/Os, but checkpoint patches can consume storage space very rapidly as the VM continues running. In our experiments, we observed that a workload can dirty 10–40MB of memory per second. Using these results as an example, protection of a VM can generate 36–144GB of checkpoint data per hour, and potentially use up an entire disk per day.

The checkpoint patches need to be consolidated to reclaim storage space and to generate a recent, consistent VM state for fail-over. However, merging checkpoint patches is very

time-consuming. Each patch file must be read from disk, and the dirty page contents written to different locations in the fail-over image according to the page indexes. Our experience suggests that merging only 80 checkpoints or so, each containing about 30MB of data, takes almost 4 minutes to complete. For a workload that dirties 30MB of memory per second, to merge all its checkpoints when restoration of the VM is required would result in an unacceptably long fail-over time. Even if a daemon runs to merge a few checkpoints periodically, the I/Os incurred can interfere with the storage of new incoming checkpoints, and offset the speed benefit of writing checkpoints sequentially as patches.

Therefore, HydraVM takes the other approach, to consolidate an incoming checkpoint into the fail-over image, and keep the image updated with the latest checkpointed VM state, ready for use by VM restoration. For each primary VM, HydraVM runs a checkpoint receiver daemon, `harcv`, in the fail-over image storage. `harcv` receives a checkpoint in its entirety, and applies all state changes included in the checkpoint to the fail-over image. Note that `harcv` does not write to the fail-over image while receiving the checkpoint, nor does `hacp` update the fail-over image directly by sending checkpointed VM state in network file or block I/Os. Otherwise, the fail-over image may become inconsistent and unusable for recovery, if the primary host fails in the middle of transmitting a checkpoint.

`harcv` commits all checkpoint writes to the storage, releases the buffer cache used, and sends an ACK back to `hacp`, confirming the successful completion of the checkpoint. After receiving the ACK, `hacp` waits until the current checkpointing interval ends, and takes a checkpoint again. If a checkpoint takes a longer time than the configured checkpointing interval to replicate, `hacp` waits until the on-going replication finishes and an ACK is received, and then starts the next checkpoint. This way, checkpoints are not taken faster than they can be stored and made useful (in the fail-over image).

3.3.2 Checkpointing VM Disk State

To correctly restore the primary VM in the event of a failure, a VM disk state that is consistent with its CPU and memory state in the fail-over image is required. The focus of HydraVM is to keep track of and recover VM runtime state in a memory-efficient manner, and we currently use a simple technique to meet the requirement of checkpointing virtual disk state. Our system hosts VM disks under LVM [12], and uses the snapshot functionality of LVM to capture the disk state at the time of an incremental checkpoint.

LVM snapshots can be taken very quickly and at a low overhead. It is implemented by performing copy-on-write at block level. Note that HydraVM is not designed for, nor confined to, the use of LVM. File system snapshots, such as those supported in ZFS [85], BTRFS [84], Ext3cow [76], to name a few, can also be used. It is most ideal to integrate HydraVM with a storage infrastructure that is designed to support efficient creation of virtual disk snapshots; one good example of such a storage system is Parallax [67].

3.4 VM Recovery

For prior approaches that use in-memory backup VMs, when a failure occurs, all VM states needed for fail-over are in place in the memory of the backup VM. Ideally, fail-over can be accomplished simply by switching on backup VM execution. However, quick restoration of a failed VM is challenging in HydraVM, since the fail-over image is kept on a networked stable storage, and must be loaded into the memory of the selected restoration host before the VM can resume execution from the last checkpointed state.

Loading the entire fail-over image of a VM can take an unacceptably long time. It takes 20–40 seconds to load a single fail-over image of 1–2GB from a hard drive and send it over a GbE link to the restoration host in our experiments, and it will take even longer to load a larger fail-over image or multiple images at a time. Loading large amounts of VM state from the fail-over image can also make VM restoration a highly variable

procedure, especially when restoring multiple VMs concurrently: even a small VM which may individually restore acceptably can require an unpredictably long fail-over time, due to the aggregate network and storage traffic incurred. It is therefore important to reduce the time and traffic incurred for VM fail-over.

3.4.1 Slim VM Restore

HydraVM performs a "slim" restore for a failed VM to quickly instantiate the VM in the restoration host based on a minimal set of information in the fail-over image. The execution of the VM is activated immediately after the instantiation, without waiting for the fail-over image to be fully loaded.

The restoration agent `hart` is invoked in the restoration host, and is responsible for performing the fail-over. `hart` first loads the VM configuration from the fail-over image, which describes the allocated resources and virtual devices of the failed VM. Based on this information, `hart` reserves sufficient memory in the host, constructs a VM container, establishes communication channels between the VM container and the hypervisor, and creates the virtual devices used by the VM. We recycled Xen's code for VM restore to implement this stage of slim restore.

Although sufficient memory is reserved in the restoration host when `hart` creates the VM container, physical page frames that constitute the memory area are not yet designated when the memory reservation is made. `hart` loads the VM page tables from the fail-over image. It walks through the page table entries, allocates and assigns physical page frames in the restoration host to VM pages. This establishes the mappings between guest page frame numbers (GPFNs), which are the page indexes in the VM's private view of its contiguous memory space, and machine frame numbers (MFNs), which are the host-dependent physical frame numbers in the restoration host. The VM page tables are updated to contain correct references to the allocated memory of the VM in the restoration host.

Subsequently, `hart` loads several VM data structures that are critical to restoring VM

execution, for example, the running virtual CPU, wall-clock time, and a few pages shared with the hypervisor. Finally, `hart` loads the virtual CPU state. With the above information loaded from the fail-over image storage and a consistent virtual disk state in the VM disk storage, `hart` restores the VM, reconnects the virtual devices to the restored VM, and switches on VM execution.

3.4.2 Fetching VM Pages On-demand

Immediately after the restored VM begins to execute, its memory space is partially populated. Only the small set of VM pages loaded by slim restore is placed in the memory space of the restored VM and is ready for use; no data pages of the VM are loaded during fail-over.

As the restored VM executes and accesses its memory, valid contents must be supplied for the execution to proceed. In HydraVM, memory references made by the restored VM are intercepted by the hypervisor. If a page accessed is not yet present in the memory space of the VM, the hypervisor requests on the VM's behalf that the page content be fetched from the fail-over image.

VM memory references may be intercepted in different ways. One approach is to mark all resident pages in the VM as "not-present" in their page table entries when loading VM page tables during slim restore, so that when the VM accesses a page, it generates a fault and traps into the hypervisor. However, this approach requires significant modification in the guest kernel to work. Our implementation uses Xen's support of shadow page tables instead. Before finishing slim restore, `hart` puts the operation of the restored VM in *shadow* mode. The shadow copy of the VM page tables are initially empty, and therefore, the first access to each VM page traps into Xen upon a fault; the shadow entries are filled in as Xen handles these faults during VM execution.

When the restored VM accesses a page that is not yet loaded from the fail-over image, its execution is temporarily paused, and the content of the page is requested. A page

fetching daemon, called `odpf`, runs in the restoration host to service such a request. Based on the GPFN of the requested page, `odpf` loads the page content from the fail-over image storage into the memory space of the restored VM. Once the page is brought in, the VM is un-paused and continues executing.

By requesting memory contents on-demand, no unnecessary pages are transferred for the restored VM, and the network and storage traffic incurred at the beginning stage of VM recovery is greatly reduced. The remainder VM pages can be pushed from the fail-over image storage at a later time to fully populate the memory space of the restored VM when the system is relatively lighter loaded. Although the execution of the restored VM is inevitably interrupted by page fetches, we use page pre-fetching to reduce the frequency of such interruptions, as we will next describe. Interruption to VM execution becomes minimal once the working set of the VM is brought into memory.

3.4.3 Pre-fetching Nearby VM Pages

Memory references often exhibit spatial locality. When servicing a page fetch request, it may be beneficial to also fetch pages that are adjacent to the one requested, in anticipation of the VM's future needs. However, two factors must be considered for page pre-fetching. First, pre-fetching can incur additional storage I/O overhead. Second, getting more pages than the one requested can increase the request service time, keeping the VM paused and waiting for the requested page longer.

In HydraVM, pre-fetching adjacent pages means reading additional blocks sequentially in a fetch I/O, or issuing additional sequential I/Os. Since sequential I/Os can be performed efficiently on storage devices, page pre-fetching will likely incur more benefit than overhead. In our prototype implementation, `odpf` accesses the fail-over image over NFS [36], to take advantage of NFS' *asynchronous* read-ahead buffering for page pre-fetching. As greater memory reference locality is detected (upon detection of greater sequentiality in fetch I/Os), NFS pre-fetches more pages in the proximity of the requested pages asyn-

chronously, without blocking the operation of `odpf` or increasing request service times. Pre-fetched pages are brought into the page cache memory of the restoration host in parallel with, but not on the critical path of, the page fetch requests issued by the restored VM. When the VM accesses a pre-fetched page, the content becomes available instantly with minimal interruption to VM execution.

3.5 Evaluation

We evaluate HydraVM addressing the following questions:

- What type of VM protection does HydraVM provide without making any backup memory reservation?
- How much overhead is incurred for protection with HydraVM?
- When a host failure is detected, how quickly does HydraVM bring a failed VM back up?
- How does a fail-over performed by HydraVM affect VM operation?

3.5.1 Testbed and Workloads

We ran all experiments on a testbed consisting of four HP Proliant BL465c blades in the same LAN. The four blades are used as the primary host, restoration host, VM disk storage, and fail-over image storage in HydraVM, respectively. Each blade is equipped with two dual-core AMD Opteron 2.2GHz processors, 4–8GB RAM, two Gigabit Ethernet cards, and two 146GB SAS 10K rpm hard disks.

The VM under test is configured with 1G memory, one virtual CPU and one virtual NIC. Its virtual disks are hosted in the VM disk storage under LVM, and mounted in the primary and restoration hosts via NFS. The VM under test and HydraVM use separate network interfaces: all VM traffic, including its disk traffic, go through one NIC of the

hosts, while the VM checkpoint and restoration traffic incurred by HydraVM go through the other NIC.

We evaluate HydraVM on two types of fail-over image storage, a hard disk drive (HDD) and a SSD; we replaced one disk in the blade used as the fail-over image storage with an Intel 80GB X25M Mainstream SATA II MLC SSD. HydraVM writes to and reads from the VM fail-over image at the granularity of an OS page in its protection and recovery modes, respectively. Running experiments with both HDD and SSD helps us understand how these small and largely random I/Os are performed on different storage media types.[1]

As discussed in Section 3.2.3, HydraVM is suitable for protecting long-running computation jobs and cluster applications. We used three workloads of these types in our evaluation. HPC-C [8] is a suite of 7 benchmarks essential for long-running scientific jobs. These benchmarks stress floating point computation and the memory subsystem. All of them are executed in our experiments. SPECjvm2008 [16] is a benchmark suite for evaluating the performance of a Java runtime environment. We executed the FFT benchmark of this suite for 10 minutes in each experimental run.[2] FFmpeg [3] is an open-source media transcoding tool. We used it to convert a 124MB MP4 file to AVI format in our experiments.

Unless otherwise mentioned, all measurements reported are the averages of at least four runs of each experiment. All bar graphs show 90% confidence intervals. The VM under test is rebooted between each experimental run.

3.5.2 Storage-based VM Protection

We ran the three workloads in a VM, and configured `hacp` to take checkpoints of the VM periodically throughout the execution of the workloads. Table 3.1 summarizes the average size of each checkpoint taken, the time required to replicate a checkpoint over the network, the time required to store a checkpoint in the fail-over image and the storage

[1] In all experiments, the virtual disks of the VM under test were hosted on a hard disk in the VM disk storage.

[2] Specifically, we used scimark.fft.large, which computes Fast Fourier Transform and is designed to stress the memory subsystem by using a dataset large enough to not fit within a standard L2 cache.

HPC-C						
Configured checkpointing interval (sec)	1		2		5	
Fail-over image storage	HDD	SSD	HDD	SSD	HDD	SSD
Actual checkpointing interval (sec)	1.0	1.0	2.0	2.0	5.0	5.0
Checkpoint size (MB)	10.2	10.2	12.7	12.6	14.3	14.4
Checkpoint sending time (ms)	93	93	117	116	131	133
Checkpoint storage time (ms)	724	476	808	572	987	665
Checkpoint storage throughput (MB/s)	14.1	21.4	15.7	22.1	14.4	21.6
FFT						
Configured checkpointing interval (sec)	1		2		5	
Fail-over image storage	HDD	SSD	HDD	SSD	HDD	SSD
Actual checkpointing interval (sec)	1.7	1.6	2.2	2.2	5.0	5.0
Checkpoint size (MB)	38.7	38.7	40.8	40.8	50.8	51.0
Checkpoint sending time (ms)	349	349	368	368	458	459
Checkpoint storage time (ms)	1335	1274	1442	1359	1939	1711
Checkpoint storage throughput (MB/s)	29.0	30.4	28.3	30.0	26.2	29.8
FFmpeg						
Configured checkpointing interval (sec)	1		2		5	
Fail-over image storage	HDD	SSD	HDD	SSD	HDD	SSD
Actual checkpointing interval (sec)	1.3	1.0	2.0	2.0	5.0	5.0
Checkpoint size (MB)	11.0	10.3	13.1	12.9	20.5	20.8
Checkpoint sending time (ms)	102	95	121	120	189	191
Checkpoint storage time (ms)	1108	607	1123	716	1488	1088
Checkpoint storage throughput (MB/s)	9.9	17.0	11.7	18.1	13.8	19.1

Table 3.1: The size of the incremental checkpoints taken and the time required to send and store each checkpoint to the fail-over image storage during the execution of the workloads.

throughput achieved, under different checkpointing frequencies and storage types.

The checkpoint storage throughput achieved is dependent on both the workload in the checkpointed VM and the storage type. In our experiments, checkpoints are written to a HDD at a rate of 10–29MB/s. In the three workloads, higher throughputs (26–29MB/s) are observed for the storage of FFT checkpoints. While FFT touches more memory pages in each checkpointing interval, resulting in larger checkpoints than the other two workloads, the I/Os incurred for storing its checkpoints have a higher degree of sequentiality and are carried out much more efficiently. In our experiments, HDD even achieves a performance comparable to SSD when storing FFT checkpoints.

However, in most cases, SSD still handles checkpoint storage more efficiently. Our results show that for HPC-C and FFmpeg, checkpoints are stored faster on SSD by 7MB/s than HDD. While SSD generally incurs much lower access latencies without any mechanical parts, its write performance can be affected by the need to erase blocks before reusing them, and is also workload-dependent. In our experiments, the checkpoint storage throughputs achieved on SSD range from 17MB/s to 30MB/s. Write performance tends to degrade with increasing randomness in the I/Os, due to the impact of internal fragmentation and increased overhead of garbage collecting clean blocks [68].

With these storage throughputs, all three workloads have their checkpoints committed to storage within each checkpointing interval when checkpointed every 5 seconds, whether using HDD or SSD in the fail-over image storage. When checkpoints are taken at shorter intervals, it becomes more challenging to finish writing each checkpoint before the interval ends. For example, when FFmpeg is configured to be checkpointed every second, each of its checkpoints takes an average of 0.1 second to be transmitted over the network, and another 1.1 seconds to be written on a HDD. The total time taken to replicate and commit a FFmpeg checkpoint to the fail-over image storage thus exceeds the 1-second configured interval.

`hacp` does not start the next checkpoint until the on-going one is committed to storage. As a result, many FFmpeg checkpoints are taken after the configured interval has passed. The delays result in discrepancies between the *configured* checkpointing interval (1 second) and the *actual* elapsed time between consecutive checkpoints (1.3 seconds on average). If the fail-over image storage uses a SSD, it takes only 0.6 second on average to store each checkpoint, and the configured checkpointing frequency of one checkpoint per second can actually be achieved for FFmpeg. However, SSD does not always meet the I/O demand of achieving the configured checkpointing frequency. In our experiments, when FFT is configured to be checkpointed every 1 and 2 seconds, checkpoints are delayed and actually taken every 1.6 and 2.2 seconds, even when storing on a SSD, since these checkpoints each

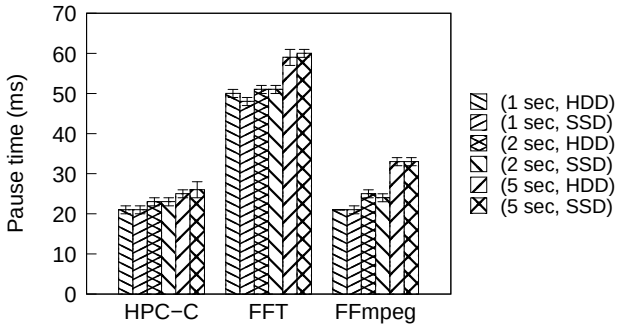


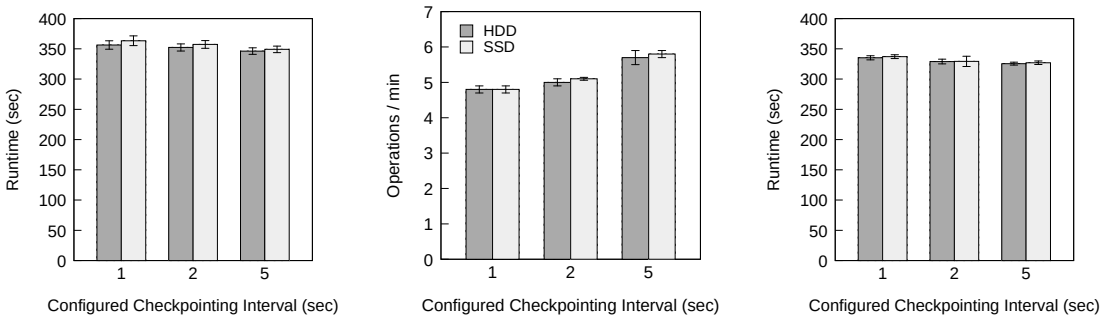
Figure 3.2: The VM pause time incurred for taking an incremental checkpoint. The legend is given in (configured checkpointing interval, fail-over image storage type).

contain a large number of dirty pages to be stored.

3.5.3 Overheads of VM Protection

HydraVM protection affects the operation of a protected VM, mainly because the VM must be paused periodically for taking incremental checkpoints. Figure 3.2 shows the VM pause time incurred for each checkpoint. The pause times are in the order of tens of milliseconds, and we observed that the time taken to copy dirty pages to a buffer for transmission is the main part. Therefore, pause time increases with the number of dirty pages in a checkpoint. For each workload, longer pause times are incurred at larger checkpointing intervals. In the three workloads, FFT incurs the longest pause times, since it has larger checkpoints than the other two workloads.

Figure 3.3 shows the performance of the workloads running in a protected VM periodically paused and checkpointed throughout the execution of the workloads. Our results show that HydraVM provides VM protection with a moderate impact on application performance. When checkpointed every 5, 2, and 1 seconds, the execution of HPC-C takes 1–6% longer time to finish comparing to the baseline execution, for which no checkpoints are taken and no protection is provided. As for FFT, the throughput achieved (operations completed per minute) drops by 7–22% when protected. FFmpeg takes 12–16% longer time to finish transcoding the media file under protection.



(a) HPC-C (b) FFT (c) FFmpeg

Figure 3.3: The performance of the workloads when checkpointed periodically. The runtime of HPC-C without checkpointing (baseline) is 344 seconds. The baseline throughput of FFT is 6.1 ops/min. The baseline runtime of FFmpeg is 289.5 seconds.

Although when checkpointed at larger intervals, workload execution is paused for a longer time in each interval, pauses happen less frequently, and hence the overall impact on workload performance is smaller. Workload overhead increases as checkpointing interval becomes shorter. Taking checkpoints at shorter intervals provides a higher level of VM protection, since if a failure occurs, the VM rolls back to a more recent point in time and loses a smaller amount of completed work. However, this benefit is gained at the cost of a larger loss of application performance during normal operation, since VM execution is interrupted more frequently.

HydraVM provides VM protection with efficient resource usages. `hacp` and `harcv`, the daemons responsible for checkpoint replication, use less than 10% of the CPUs in the protected host and fail-over image storage, respectively. The daemons each need a transmission buffer to send and receive checkpoints one-by-one. In our experiments, each daemon uses no more than 80MB of memory, which is enough for transmitting the largest checkpoint of the workloads we use. The memory consumption of HydraVM is significantly less than that required by approaches maintaining in-memory backup VMs, which, in this case, would use 1G RAM for the backup VM in addition to the transmission buffers.

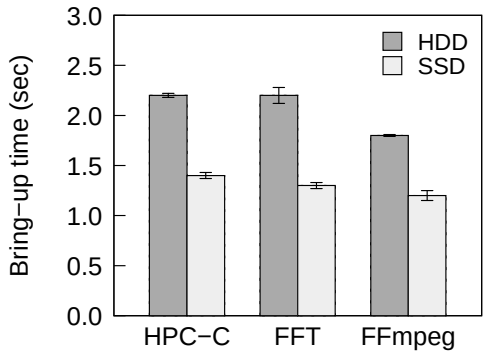


Figure 3.4: The time required to bring up a failed VM from a HDD- and a SSD-based fail-over image storage.

3.5.4 Restoration of a Failed VM

We now evaluate how quickly HydraVM brings a failed VM back up. We ran the three workloads in a VM in the protection mode of HydraVM; `hacp` was configured to take checkpoints of the VM every second. After executing about 50% of each workload and finishing the last checkpoint in this time period, we forced the VM to stop, to emulate the occurrence of a failure. HydraVM then switched to the recovery mode, and `hart` was invoked to restore the VM based on its fail-over image and a consistent disk state in the VM disk storage.

All workloads resumed execution correctly in the restored VM after a brief pause, during which HydraVM performs slim restore and brings the VM back up. As shown in Figure 3.4, HydraVM restores a VM from a fail-over image stored on a HDD in less than 2.2 seconds. If the fail-over image is on a SSD, the VM is restored even faster, using less than 1.5 seconds. In all experiments, we made sure that no data from the fail-over image is cached in memory prior to VM restoration. All data loaded during slim restore are loaded from the storage devices, so that we can include I/O time in our measurements.

HydraVM brings up a failed VM quickly by loading a minimal amount of information from the potentially gigantic fail-over image in the shared storage. In our experiments, less than 5MB of data out of the 1G fail-over image is loaded for slim restore, as shown in

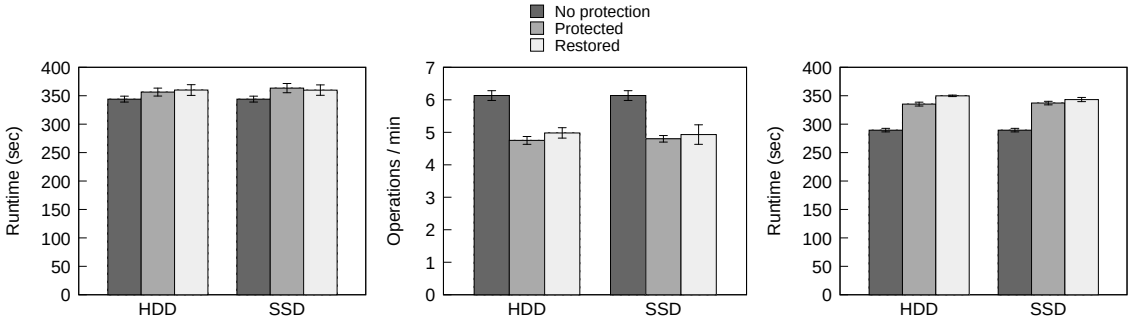
Workload	HPC-C		FFT		FFmpeg	
Fail-over image storage	HDD	SSD	HDD	SSD	HDD	SSD
Total data loaded (MB)	4.3	4.3	3.9	3.9	3.3	3.3
Number of page table pages loaded	1083	1083	975	975	823	825
Time to load all page tables (ms)	1342	635	1391	505	996	389

Table 3.2: Amount of data loaded and the loading time incurred during fail-over (slim VM restore).

Table 3.2. The majority of the data loaded are the VM page tables. The time to load and restore all page tables constitutes 30–60% of the bring-up times.

We further break down the bring-up time into three stages, in which (1) sufficient memory is reserved in the restoration host and a container for the VM is constructed, (2) the page tables, virtual CPU state, and other important data structures of the VM are loaded from the fail-over image and processed to initialize the VM, and (3) the virtual devices of the VM are re-connected and the VM is ready to begin execution. Our results show that on average, stage 1 and stage 3 take 0.4 and 0.2 second to finish, respectively; these times are independent of the types of device used in the fail-over image storage. The rest of the bring-up time is spent in stage 2.

This breakdown helps us reason about the fail-over behavior of HA approaches that use in-memory backup VMs. These approaches perform the tasks in stage 1 when setting up a backup VM in a dedicated host at the beginning of the protection of a primary VM. As the primary VM executes, the memory state of the backup, including the data structures mentioned in stage 2, are repeatedly updated to synchronize with that of the primary. When the primary VM fails, presumably, only the tasks in stage 3 are left to be completed during fail-over. Using in-memory backup VMs thus enables very fast fail-over, but at the cost of making unproductive memory reservation throughout normal VM operations.



(a) HPC-C (b) FFT (c) FFmpeg

Figure 3.5: The performance of the workloads under different conditions: *no protection* and not checkpointed, *protected* and configured to be checkpointed every second, and *restored* from a failure that occurs halfway through the workload executions.

3.5.5 Operation of a Restored VM

In the experiments described in Section 3.5.4, after the VM was restored from the emulated failure, the last 50% of the workloads executed while `odpf` supplying the memory contents of the VM on-demand. All three workloads completed execution correctly.

In HydraVM, the impact of a failure includes not only the extra time required to bring the failed VM back up, but also the slowdown of VM execution after restoration due to demand paging. To understand this impact, in Figure 3.5, we compare the performance of the workloads running under three conditions: (C1) *no protection* and not checkpointed, (C2) *protected* and configured to be checkpointed every second, and (C3) *restored* from a failure that occurs halfway through the workload executions. C1 and C2 are failure-free conditions. In C3, the VM is configured to be checkpointed every second (same protection as in C2) before failure.

For each workload, comparing the performances measured with and without protection (conditions C1 vs. C2) tells us the cost of protection, while comparing the performances measured with and without a failure (conditions C2 vs. C3) reveals the total cost of recovering from a failure in HydraVM. Overall, failure recovery does not incur an undue overhead. In some cases, the workload performance with a failure (and failure recovery) is better than

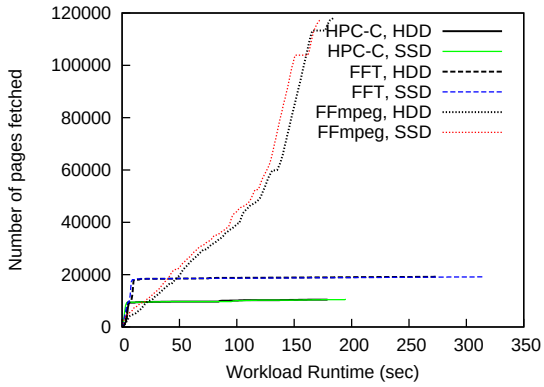


Figure 3.6: Number of VM pages fetched to execute the last 50% of the workloads in a restored VM.

that without a failure (simply under protection), since we did not re-engage protection after the VM is restored in these experiments. We chose to not re-engage protection, so that for HPC-C and FFmpeg, we can compare the execution time of the last 50% of the workloads running with and without demand paging (conditions C1 vs. C3), to understand the overhead incurred by the technique.

Recovering from a failure halfway through the execution (condition C3) using a fail-over image stored on a HDD, HPC-C executes for a total of 360 seconds, 4 seconds longer than the total runtime measured under protection without a failure (condition C2). Failure recovery is accomplished at a relatively low cost for HPC-C, mainly because the workload fetches a small amount of memory data on-demand to finish execution. As shown in Figure 3.6, during the second half of workload execution, only 44MB of data are fetched from the fail-over image storage. These fetches concentrate in the first few seconds of the restored execution: 88% of the data are fetched in 6 seconds from a HDD, and more quickly, in 4 seconds, from a SSD. Comparing the runtime of the second half of the workload under conditions C1 and C3 (with and without demand fetching) yields a consistent observation that demand paging incurs a relatively small overhead for HPC-C; fetching from a HDD and a SSD lengthens the runtime by an additional 7 and 6 seconds, respectively.

The costs of failure recovery are larger for FFmpeg. Without a failure, the workload

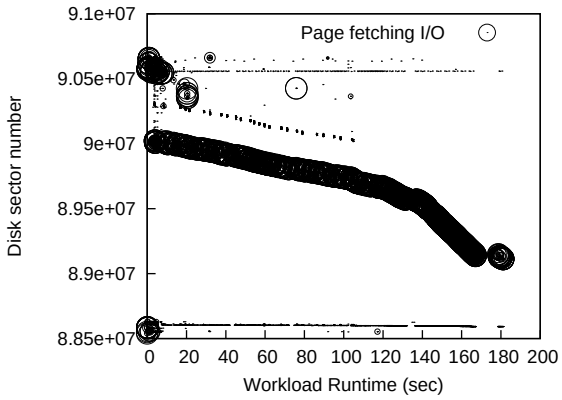


Figure 3.7: The storage I/Os incurred to demand fetch VM pages for FFmpeg after it resumes execution in a restored VM. Each dot represents an I/O, and the size of the dot represents the size of the I/O.

runs for 335 and 337 seconds under protection, using a HDD and a SSD in the fail-over image storage, respectively. When a failure occurs in the middle of the execution, the workload spends 15 more seconds to finish execution, if recovering from a fail-over image on a HDD. The cost of recovery is smaller when a SSD is used in the fail-over image storage; workload runtime is lengthened by 6 seconds.

These recovery costs result from the relatively large overheads of demand paging incurred by FFmpeg. After resuming in the restoration host, FFmpeg fetches a large amount of memory data, about 460MB, to finish execution. The workload also has a staged use of VM memory, as can be observed from Figure 3.6. It accesses the working set area-by-area, and fetches new memory pages at a slower pace than the other two workloads. Page fetching thus affects the execution of FFmpeg for a longer period of time, leading to increased overheads.

To gain a further understanding of the storage I/Os incurred for demand paging, we ran `blktrace` [33] in the fail-over image storage after the failed VM resumes execution. Using FFmpeg as an example, Figure 3.7 plots the I/O activities that actually occur on the storage device (a HDD in this case) to fetch page contents for the restored VM. Each dot represents an I/O request recorded by `blktrace`, and the size of the dot represents the size

of the request. We found that even though almost 120,000 4K pages are fetched during the execution of FFmpeg, only 9,000 I/O requests are sent to the device. The I/O pattern shows substantial spatial locality, and many of the I/Os fetched 512 disk sectors (256KB of data, 64 4K pages) at a time, as shown by the large dots in the figure. These observations suggest that the memory references made by the workload have good locality, and that NFS read-ahead buffering, exploited by `odpf`, is effective in pre-fetching near-by VM pages.

3.6 Discussions on Alternative Storage Architectures

We have demonstrated the feasibility of the HydraVM approach via a prototype implementation and its evaluation in a 4-node testbed. In this section, we discuss how HydraVM may take advantage of large, parallel storage systems to provide HA support for a large number of VMs.

Flat Datacenter Storage (FDS) [74] is an example of large, parallel storage system that may benefit HydraVM. FDS provides a shared, centralized permanent storage based on the disks or flash in hundreds or thousands of commodity storage servers in a cluster. In FDS, data are logically stored in blobs and accessed in tracts; FDS uses 8MB tracts to amortize the cost of disk seeks and achieve comparable performances for sequential and random accesses. FDS strives to distribute the tracts of a blob uniformly across a vast array of disks. When an FDS client accesses the storage, hundreds or thousands of disks may read or write data for the client in parallel, and the client achieves the combined throughput of all participating disks.

RAMCloud [75] is also a storage system characterized by scale and data parallelism. Different from FDS, RAMCloud provides DRAM storage by unifying the DRAM of hundreds or thousands of storage servers, and uses disks or flash to store replicas of the data in DRAM. It exploits scale and data parallelism not to serve client requests (those are satisfied quickly by DRAM accesses), but to speed up the recovery of a failed storage server. It strives to replicate the DRAM data of a server, in units of 8MB log segments, uniformly

across thousands of backup disks. When the server fails, the DRAM data can be recovered by thousands of disks reading their shares of the server's data replicas in parallel.

Since HydraVM utilizes permanent storage to eliminate backup memory reservation, below, we primarily use FDS as an example as we discuss how scale and data parallelism may benefit HydraVM; we will discuss RAMCloud's techniques where appropriate.

Consider HydraVM using FDS as the shared storage and storing each fail-over image as a blob. For a VM with 1GB memory, its fail-over image consists of 128 tracts, and ideally, each tract is stored on a different disk. During protection of the VM, HydraVM consolidates checkpoints into the fail-over image with a maximum of 128 disks working at the same time, each updating its share (a tract) of the fail-over image. In theory, even if all 1GB memory of the VM is dirtied and checkpointed, the checkpoint can be stored as fast as an 8MB tract can be written.

This example illustrates how FDS's scale and data parallelism may help HydraVM overcome the intrinsic slowness of permanent storage devices in the protection mode, but it is simplified in two aspects. First, in reality, deployment of hundreds of disks is expected to support HA for a large number of VMs for cost-effectiveness; checkpoints of multiple VMs must contend for disk bandwidth for their storage. Second, HydraVM must guarantee atomicity for the storage of each checkpoint across multiple disks and storage servers. We next discuss the two aspects in more detail.

The developers of FDS executed up to 180 concurrent, networked clients accessing an FDS cluster of 1,033 disks continuously, and observed a peak aggregate storage throughput of 67 GB/s, roughly 380 MB/s/client; each client has a 10 Gbps network bandwidth at its disposal. Consider each of these clients associated with the checkpoint stream of a VM protected by HydraVM. The measured storage throughput of FDS suggests that under this configuration, (1) each protected VM can have about 300MB of checkpoint data sent and stored in one second; (2) VMs with a writable working set smaller than 300MB may be checkpointed at least once every second. Using FDS enables HydraVM to store

checkpoints quickly and checkpoint VMs frequently, even when many VMs are protected at the same time. Frequent checkpointing reduces the loss of completed work upon failure. It also provides the possibility for HydraVM to support network buffering and transparent recovery of client-facing applications, which are currently unsupported, as discussed in Section 3.2.3; depending on running applications' writable working set size, an even larger FDS cluster or a stage buffer is needed.

To guarantee atomicity of checkpoint storage across multiple FDS disks, HydraVM may use a few intermediate servers, deployed between the protected hosts and the FDS cluster. Each protected VM runs its `harcv` in an intermediate server; one such server can run the `harcv` of multiple protected VMs. `harcv` receives a complete checkpoint from `hacp` in a memory buffer, and then writes to the tracts of the checkpoint on the appropriate FDS disks. It ensures all writes are successful; if any write fails, it retries the operation. The checkpoint buffer must not become unavailable before the checkpoint is completely stored. This may be achieved by replicating the intermediate server, or by using a backup battery in the server; when needed, the battery provides power of the server for checkpoints to be flushed.

Alternatively, tracts can be stored by a logging approach similar to that in the Log-structured File System (LFS) [86]; RAMCloud also uses a LFS-like approach to manage its DRAM and disk storage. This approach writes to a tract by appending the new content of the tract to storage, rather than overwriting the existing content of the tract. A partially stored checkpoint does not affect the utility of the fail-over image, so storing each checkpoint atomically will not be a concern. However, it is important to keep track of the tracts constituting the latest consistent fail-over image, which may not always be the latest version of the tracts. Also, this approach needs to run a garbage collector periodically to reclaim storage space from tracts that are no longer useful, *i.e.*, those that have newer contents and are not included in the fail-over image. Garbage-collecting tracts in their entirety can greatly reduce the collection overhead; if all the data in a tract becomes garbage together,

no extra I/Os are needed to discover live data and preserve them before reclaiming space from the tract. Otherwise, the extra I/Os caused by garbage collection can interfere with and slow down the storage of new checkpoints.

The recovery mode of HydraVM also benefits from the use of a large, parallel shared storage. When restoring a failed VM, `hart` can potentially load a large amount of data from the VM's fail-over image into the restoration host in a short amount of time. Thus, during slim restore, additional VM state (e.g., part of the VM's working set) can be loaded, to reduce the need of demand fetching and improve the performance of the restored VM. As the VM executes, `odpf` may also pre-fetch more memory pages in parallel. The VM's memory space can be populated faster, thus reducing the window during which the VM is susceptible to performance penalties. Note, however, that as the scale and data parallelism of the shared storage increase, the speed of storage I/Os may become less of a limiting factor in VM recovery, but the network bandwidth available to individual restoration hosts may become a new bottleneck.

3.7 Related Work

Upon detection of a host failure, failed VMs may be restarted automatically [21], but the runtime state of the VMs is lost upon restoration. To minimize such loss, two types of approaches have been proposed to keep track of and recover from a recent VM state in a failure. One records the low-level events, e.g., instructions and interrupts, executed by a protected (primary) VM, and replays them in a backup VM deterministically in lock-step [20, 32]. If the primary fails, the backup takes over execution from where the primary left off. Since the primary and backup must execute the exactly same sequence of instructions, these approaches require the VMs to have identical configurations. Therefore, they are not exempt from making unproductive memory reservation.

The other type of approaches replicates VM checkpoints continuously throughout normal VM operations. Most existing systems replicate checkpoints at fixed time intervals, and

buffer the VM network outputs during each interval until the checkpoint of the interval is fully replicated to and acknowledged by the backup [24, 39, 44, 47]. Alternatively, Kemari replicates a checkpoint each time before the VM is about to interact with external devices (e.g., disk and network) [93]. To reduce the application performance degradations caused by network buffering, RemusDB [69] buffers VM network outputs selectively. Targeting database applications running in a VM, RemusDB buffers only those network outputs that carry transaction control messages, *e.g.*, acknowledgements to `COMMIT` and `ABORT` requests. SecondSite [80] extends the use of checkpoint replication beyond a local area network to synchronize primary and backup VMs connected by Internet links for disaster recovery.

All the above mentioned systems make memory reservation for backup VMs. HydraVM adapts a periodic, incremental checkpointing technique similar to existing systems, but unlike these systems, trades off main memory with stable storage to provide a cost-effective HA alternative for long-running computation jobs and cluster applications. There are other systems that persist VM checkpoints in stable storage, but for different purposes than HydraVM. VNsnap [57] captures consistent snapshots of a distributed VM environment, to suspend and later resume the entire VM cluster. The system was evaluated with much longer intervals between snapshots (e.g., 10 minutes) compared to HydraVM. Since VNsnap is not designed for HA, the system does not support fast resumption from VM snapshots. Burtsev *et al.* implements transparent checkpoints in the Emulab network testbed, to provide controls over experiment execution without interfering the realism of experiments [35]. This system does not support fast resumption from VM checkpoints either, while HydraVM quickly brings up a VM with slim restore.

The proposed slim VM restore technique is built upon the core idea of fast instantiation of a VM with partially populated memory space. This idea has been applied to different types of problems. Post-copy VM migration [50] resumes the execution of a migrated VM in the destination host immediately after shipping the CPU state from the source VM, without having the entire memory state copied over. As the VM runs in the destination host,

memory pages are pulled/pushed from the source VM. Potemkin [95] quickly "forks" new VMs in single hosts from a reference image and creates private copies of VM pages upon modification of VM memory; its aim is to implement a large honeyfarm system based on lightweight VMs. SnowFlock [62] extends the idea of VM fork across machine boundaries. It creates multiple child VMs based on a condensed parent VM image in a group of hosts. As the child VMs execute, memory pages are fetched from the parent on-demand. The focus of SnowFlock is to spread application deployment rapidly, which is most useful in the area of parallel computations. HydraVM applies a similar technique, however, to address a completely different problem regarding the resource inefficiency of existing HA solutions.

3.8 Conclusions

In this chapter, we proposed HydraVM, a storage-based, memory-efficient way of achieving high availability in a virtualized environment. Unlike conventional approaches, which require twice the memory each protected VM uses, HydraVM requires minimal extra memory, relieving the tension between reliability and resource-efficiency, two critical operational goals of a virtualized environment. HydraVM maintains the VM state needed by fail-over in a shared storage, and recovers a failed VM promptly in any host that has access to the shared storage, allowing any host with available capacity to be used as the backup. HydraVM complements the HA toolbox currently available to administrators of a virtualized environment with a cost-effective alternative suitable for protecting long-running computation jobs and cluster applications.

CHAPTER IV

Application-Assisted Live Migration of Virtual Machines with Java Applications

4.1 Introduction

Live migration [38, 73] is to move a running virtual machine (VM) from a physical host to another with minimal disruption to the execution of the VM. It has been used for load-balancing [91, 98], fault-tolerance [71, 89], power savings [30, 40, 72], and performance enhancements [31].

To migrate VMs within a LAN, such as within a datacenter, the primary task is to migrate the contents of VMs' memory; VM disk contents can be stored in a shared storage. Most migration tools transfer VM memory by using a *pre-copy* approach. While a migrating VM continues to run on the source host, its memory pages are iteratively transferred to the destination host. All pages are sent in the first iteration, and at each following iteration, only those pages dirtied during the previous iteration are sent. Ideally, dirty pages should be transferred faster than new pages get dirtied, and the number of dirty pages pending transmission should decrease iteratively. When the VM is paused for the last iteration, a small number of dirty pages remain to be transferred. After this short *stop-and-copy*, the VM resumes execution in the destination, and the migration completes.

However, this ideal migration is not always achievable, since the underlying network

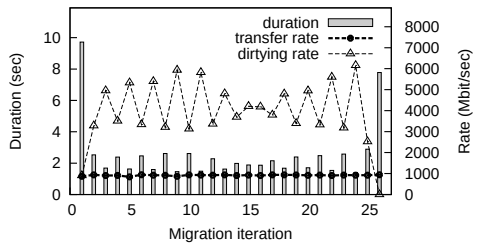


Figure 4.1: Live migration of a 2GB Xen VM running the Apache Derby database workload from SPECjvm2008.

can become a bottleneck. Figure 4.1 shows live migration of a 2GB database Xen VM over a gigabit Ethernet. Since the database application dirties memory pages much faster than the pages can be transferred, the number of dirty pages to be transferred does not decrease iteratively; hence the iterations do not keep becoming shorter. Migration cannot finish with a short stop-and-copy, but is forced to enter the last iteration after generating excessive network traffic (a total of 7GB). It incurs a long completion time (66 secs), causes a noticeable VM downtime (8 secs), and degrades application performance (by over 20%).

To alleviate the network bottlenecking problem during migration and its undesirable consequences, approaches have been proposed to speed up memory transfer using high-speed networks [54], slow down memory dirtying by throttling application execution [38], or reduce the amount of memory contents to be transferred, e.g., by using compression [92]. However, these approaches incur high resource costs or application performance penalties. The OS's knowledge can also be utilized to reduce the amount of memory transfer by not sending clean page cache pages and free pages [60], but the benefit is limited. Page cache misses may degrade application performance at the destination, and skipping free pages may only benefit the migration of lightly-loaded VMs.

In this chapter, we propose to reduce the amount of memory transfer by exploiting application semantics. Specifically, we design *application-assisted live migration*, which skips transfer of VM memory pages that need not be migrated for the execution of running applications at the destination. We build a framework for the proposed approach based on Xen. Our framework places a paravirtualized stub in the guest VM to enable collaboration

between the migration tool and applications in the guest. We ask the applications, which know best their semantics, to identify areas in their memory that need not be migrated. Based on the applications' inputs, we maintain a *transfer bitmap* that guides the migration tool to transfer or skip over VM memory pages.

Using the proposed framework, we build *JAVMM*, which migrates *Java VMs*—VMs running various types of Java applications—without transferring garbage[1] in Java memory. Targeting Java applications does not restrict JAVMM's applicability, since these applications are nearly ubiquitous; with over 9 million developers worldwide, Java has become the global standard for web-based content and enterprise software, and runs in 89% of computers in the U.S.[14]. Java applications are increasingly being deployed and run in VMs for flexible resource sharing and easy deployment. Various types of Java cloud services are being widely used [4], many of which are provisioned based on VMs for elasticity. To migrate Java VMs fast and with little performance impact is therefore an important task.

In JAVMM, Java Virtual Machine (JVM)[2] assists in migration on behalf of Java applications. Right before a migrating Java VM is paused for the last iteration, the running JVM performs a garbage collection. After the last iteration is completed, the VM resumes execution at the destination in a post-collection state: in the *Young generation* of the Java heap, only one survivor space may contain live objects, which survived the collection. JAVMM migrates the surviving objects in the last iteration, and skips transfer of the entire Young generation throughout migration.

We prototyped JAVMM using HotSpot JVM [7] in the proposed framework, and evaluated it in terms of three metrics commonly considered for live migration: the time and resources used by migration, and the impact of migration on running applications' performance. Our experimental results show that compared to Xen live migration, which is agnostic of application semantics, JAVMM can reduce the completion time, network traffic

[1] *i.e.*, Java objects that are no longer used

[2] Note the difference between a JVM, the application-level VM that executes Java bytecode, and a Java VM, a general-purpose VM in which Java applications and their JVMs run.

and application downtime caused by Java VM migration, all by more than 90%, when the running Java application has a high object allocation rate and needs a large Young generation space, without incurring noticeable performance degradation to the application.

The primary contributions of this chapter are as follows.

- We propose application-assisted live migration, and establish a generic framework to skip migration of VM memory selectively based on application semantics.
- Using the proposed framework, we build JAVMM to migrate Java VMs skipping over garbage with JVM's assistance, without customizing each Java application.
- Via an in-depth evaluation of JAVMM, we demonstrate the utility of application-assisted live migration.

The remainder of this chapter is organized as follows. Section 4.2 reviews existing approaches to alleviating network bottleneck during live migration. Section 4.3 presents our approach, a generic framework for application-assisted live migration. Section 4.4 describes JAVMM, built based on the proposed framework for efficient migration of Java VMs. Section 4.5 evaluates JAVMM experimentally. We discuss the applications and possible extensions of this research in Section 4.6, and conclude the chapter with Section 4.7.

4.2 Related Work

To alleviate network bottlenecking during live migration, approaches have been proposed to send dirty memory pages faster, generate dirty memory pages slower, or send less data for the dirty memory pages generated.

Huang *et al.* [54] proposed to transfer memory pages faster using high-speed networks capable of Remote Direct Memory Access (RDMA) like InfiniBand. The network remains a potential bottleneck even with high-speed links, though, considering the increasing computation power of individual VMs and the fact that multiple VMs may be migrated at the same time.

Clark *et al.* [38] proposed to slow down the memory-dirtying rate by moving processes to a wait queue after they generate more than a certain number of dirty pages. This may degrade application performance, and as the authors noted, one must be careful not to throttle interactive services.

Our approach falls in the third category, which transfers less data for the dirty memory pages generated. Compression [55, 92] and deduplication [41, 42, 99] are popular in this category, trading CPU for network bandwidth. Our approach skips transfer of selective memory pages, performing no computations on the pages skipped and incurring a minimal CPU overhead.

There are also other approaches that skip transfer of selective memory pages, according to different criteria than ours. Some skip over frequently dirtied pages during live iterations, but those pages must be transferred in the last iteration [26, 53, 63, 65], risking a long VM downtime. Page cache pages can be skipped over in all iterations if the storage has an identical copy of the pages; the contents skipped need to be reproduced [56], or VM performance may degrade after migration [60]. Post-copy migration skips over all memory pages and removes the pre-copy stage. To run the VM in the destination, pages are fetched from the source [50, 51], incurring performance penalties. Free pages can be skipped over and not fetched upon access [60], by exploiting knowledge of the migrating OS, but only in lightly-loaded VMs we may find a considerable number of free pages to be skipped.

Our approach exploits knowledge of the migrating applications to skip transfer of selective memory pages. JAVMM skips transfer of garbage in the frequently-dirtied Young generation of the Java heap. It need not reproduce the contents skipped, and does not degrade application performance. Our work is closest to the memory deprotection technique discussed in RemusDB [69], a VM-based high-availability system for databases. To reduce system overhead, the authors explored omission of selective memory contents from VM checkpoints based on application inputs, although data structures to be suitably omitted by this technique are yet to be identified.

4.3 Application-Assisted Live Migration

We take a *white-box* approach to reducing the amount of memory transfer for efficient VM live migration: we propose to skip transfer of selective VM memory based on application semantics, by exploiting applications' assistance.

4.3.1 What Memory to Skip Migrating?

Generally, memory contents that are reproducible or not required for correct application execution need not be transferred during migration; these contents also need no replication in high-availability systems [69].

Examples of reproducible contents include those recoverable from application logs and intermediate results that can be recomputed. It may be beneficial to skip migrating these contents if regenerating them in the destination is faster than transferring them from the source.

Memory contents not required for correct application execution include caches and garbage. Caches of various kinds, e.g., web cache and database buffer pool, need not be migrated if the performance drop caused by empty caches at the destination can be mitigated or is acceptable. Garbage is memory content that is no longer being used. It is a good candidate to skip, since in its absence, applications execute correctly and without performance degradation. Garbage exists in any applications written in languages that do not deallocate memory explicitly; Java, C# and most scripting languages fall in this category.

4.3.2 Challenges and Design Principles

To skip migration of selective application memory, the key challenge is to let the migration tool and running applications collaborate. The migration tool needs to know which memory pages to skip transferring. For the memory contents not transferred, the applications need to recover or not access them in the destination host.

Traditionally, the migration tool and an application in the guest VM are unaware of the execution of each other. They do not, and have no existing channel to, communicate. They also address memory differently: the migration tool transfers VM memory pages based on Page Frame Numbers (PFNs), *i.e.*, the page numbers in the VM's contiguous memory space, while the application executes based on Virtual Addresses (VAs). For the migration tool and the application to collaborate, the *communication gap* and *semantic gap* between them must be bridged.

We design a framework to enable their collaboration and be able to skip migration of selective application memory, following three principles; each principle describes the responsibility of one software component in our framework.

- The guest kernel provides system-level support for bridging the communication gap and semantic gap between the migration tool and running applications. It coordinates between the migration tool and the applications as they perform migration collaboratively, so that the migration tool need not interact with each application individually.

- A running application decides which areas of its memory need not be migrated, and informs the migration tool of it. The application should make this decision, since it knows best the semantics of its memory, e.g., what each memory area is used for and when the content is needed.

- The migration tool needs to know which memory pages to skip transfer, without incorporating application semantics. This way the tool becomes generic (i.e., application-independent), and can thus be used for different applications without modification. This also minimizes changes to existing live migration mechanisms.

4.3.3 A Generic Framework

Figure 4.2 provides an overview of our framework, prototyped based on Xen 4.1; our guest VM runs Linux 3.1. We added a Xen management command to invoke application-

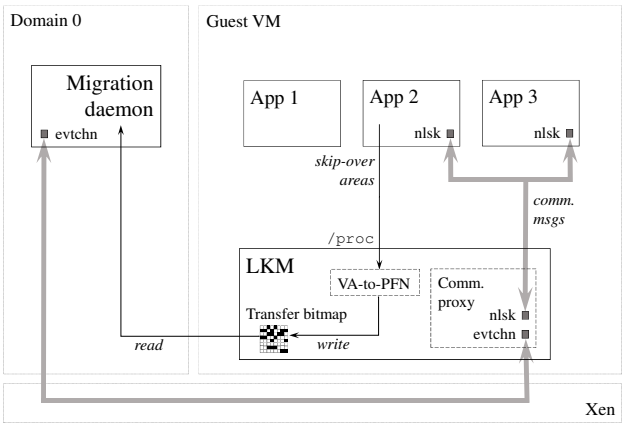


Figure 4.2: A generic framework for application-assisted live migration.

assisted live migration. Once invoked, our migration daemon executes. Our migration daemon is a modified version of Xen's. It communicates with applications in the guest through the guest kernel, and skips transfer of memory pages guided by a *transfer bitmap*. We provide guest kernel support in a Loadable Kernel Module (LKM).

4.3.3.1 Bridging the Communication Gap

Our LKM serves as a communication proxy between the migration daemon and the applications in the guest. It interacts with the migration daemon using event channel, the event notification primitive provided by Xen. A special event channel port is created when the guest VM is created, through which the migration daemon can communicate with the LKM throughout the migration process.

The LKM interacts with the applications using netlink sockets, a special socket family for communication between kernel- and user-space processes. We use netlink because it is bi-directional, asynchronous and capable of multicasting. Upon loading, the LKM creates a netlink socket, and associates it with a multicast group, which the applications subscribe to. The migration daemon communicates with the applications simply by contacting the LKM, and the LKM multicasts a netlink message to notify all subscriber applications. The LKM also relays messages from the applications to the migration daemon.

4.3.3.2 Bridging the Semantic Gap

The applications identify areas in their memory that the migration daemon can skip transfer. They specify each *skip-over area* by a VA range, and pass the VA range to the LKM via a `/proc` entry. The LKM finds the PFNs of the skip-over area by page table walks, while the application continues its normal execution. The LKM may consider a smaller VA range than that specified by the application. It aligns the start and end VAs of the specified range to the immediate next and previous page boundaries, respectively, to ensure pages found in the skip-over area can be skipped by the migration daemon in their entirety.

4.3.3.3 Skipping Transfer with a Transfer Bitmap

The LKM records the PFNs of the skip-over areas in a *transfer bitmap*. When transferring VM memory, the migration daemon examines the transfer bitmap, in addition to the dirty bitmap maintained by the hypervisor.

The transfer bitmap is created in the guest when the LKM is loaded, and is shared with the migration daemon when migration begins. It uses one bit per VM memory page (PFN), based on the same page size used by the dirty bitmap; assuming 4KB pages, the transfer bitmap uses 32KB per GB of VM memory, incurring a negligible memory overhead. Each transfer bit is either set (1) or cleared (0). A set transfer bit indicates the page needs to be migrated; the migration daemon transfers the page if its marked dirty in the dirty bitmap. A cleared transfer bit indicates migration of the page can be skipped; the migration daemon does not transfer the page, even if it is marked dirty.

4.3.3.4 Updating the Transfer Bitmap

The transfer bitmap is initialized with all bits set; by default, memory pages are transferred if they are marked dirty. Figure 4.3 illustrates how the transfer bitmap is updated. When migration begins, the LKM makes the first bitmap update. It queries the applications

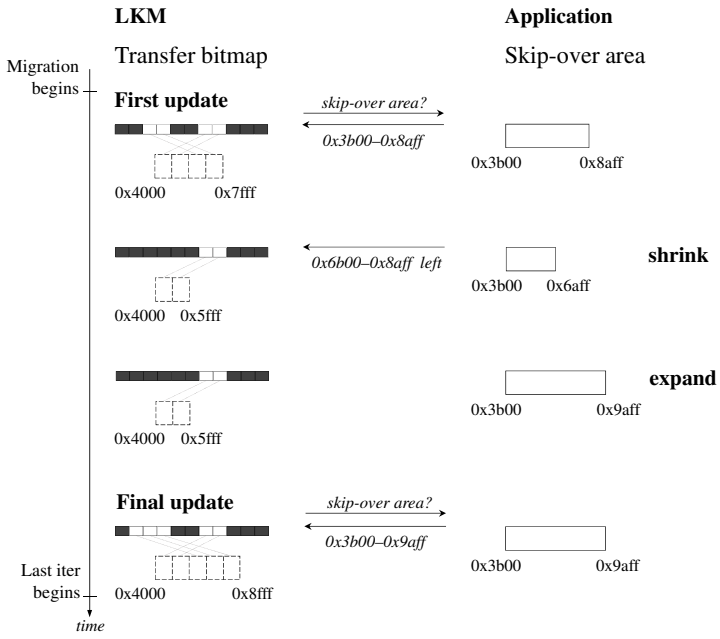


Figure 4.3: An example of transfer bitmap updates.

for skip-over areas. For each area in the applications' response, it remembers the VA range, finds the associated PFNs, and clears the corresponding transfer bits. Therefore, pages in the skip-over areas are not transferred even if they are dirtied.

In parallel with, and after, the first bitmap update, the VM continues to run, and each skip-over area may expand or shrink, *i.e.*, VA ranges and the associated PFNs may join or leave the area. Subsequent updates to the transfer bitmap may be needed.

A skip-over area is expected to shrink infrequently and by a small amount during migration, or the benefit of skipping its migration decreases. When the area shrinks, the application should notify the LKM of the VA ranges leaving the area. The LKM updates its memory of the area's VA range accordingly, and immediately, sets the transfer bits of the PFNs leaving the area. Since the pages may later get dirtied in a memory area requiring migration, setting their transfer bits immediately ensures transfer of their dirty contents in the iteration following the dirtying; this guarantees the correctness of migration.

Given the VA ranges leaving a skip-over area, the LKM does not find the PFNs leaving the area via page table walks, because the VA ranges may have been freed, in which case the

associated PFNs are reclaimed and no longer found in the page tables. The LKM maintains a cache of PFNs with cleared transfer bits. It queries the cache by the VA ranges leaving the area to quickly find the PFNs that must have their transfer bits set. The cache uses little memory: 1MB per GB of skip-over area with 4-byte entries (a 0.1% overhead).

When a skip-over area expands, transfer bitmap updates are not required. Not clearing the transfer bits of the PFNs joining the area does not affect the correctness of migration, although the pages may be unnecessarily migrated. To reduce the runtime overhead, the application does not notify when a skip-over area expands. The LKM does not clear the transfer bits of the PFNs joining the area until in the final bitmap update, which is performed right before the last iteration begins. Dirty pages in the expanded space of a skip-over area will be skipped in the last iteration to reduce VM downtime.

In the final bitmap update, the LKM queries the applications again for skip-over areas. It compares the VA ranges in the response of the applications with those in its memory. For any expanded space, it finds the PFNs joining the areas via page table walks, and clears their transfer bits. For any shrunk space, it sets the appropriate transfer bits based on the cached PFNs. Immediately after the final bitmap update is completed, the VM is paused and the last iteration begins. In the short window of the final bitmap update, the skip-over areas should be prevented from shrinking; this ensures the transfer bits of all the pages leaving the areas are set.

In our current implementation, if a PFN joins or leaves a skip-over area with no changes in the area's VA range, the transfer bitmap is not updated. This happens when a virtual page in the area has its PFN mapping changed in three possible ways: (1) from *null* to p, when a page frame is allocated; (2) from p_{old} to p, when the page is remapped due to page sharing, compaction and migration (within the VM); and (3) from p_{old} to *null*, when the page is swapped out. For (1), migration finishes correctly without clearing the transfer bit of the allocated page, which joins the skip-over area. For (2) and (3), we assume the absence of these events in skip-over areas during migration, but the LKM can be extended to update

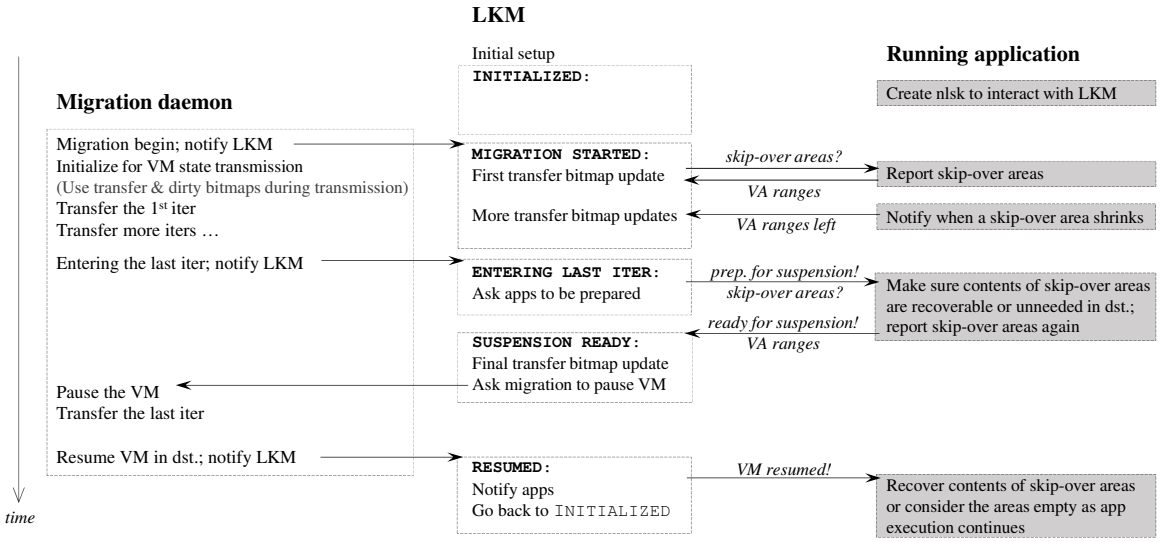


Figure 4.4: The workflow of application-assisted live migration.

the transfer bitmap for these events with further assistance from the guest kernel.

4.3.3.5 Migration Workflow

Figure 4.4 shows the workflow of application-assisted live migration. Our LKM coordinates between the migration daemon and applications in the guest as they collaborate through different stages of migration. To ease its job of coordination, the LKM transitions between states of operation based on the messages exchanged with the migration daemon and the applications, and takes different actions in each state as described next.

Before migration. Once the guest VM is created, the LKM may be loaded in preparation for possible migration. Upon loading, the LKM sets up the communication proxy and the transfer bitmap, and then enters the `initialized` state, ready for migration. If an application has memory areas that need not be transferred during migration, it creates a netlink socket as it runs in the VM, to communicate with the LKM and assist in migration.

Migration begins. The migration daemon connects with the LKM once it is started. The LKM enters the `migration started` state, and multicasts a netlink message to query running applications for skip-over areas. Based on the applications' responses, it performs

the first transfer bitmap update. As the VM continues execution, the migration daemon transfers memory pages based on both the transfer bitmap and the dirty bitmap. The LKM will be notified by the applications if a skip-over area shrinks, and it updates the transfer bitmap immediately for the pages leaving the area.

Entering the last iteration. The migration daemon contacts the LKM again before pausing the VM and `entering the last iteration`. The LKM multicasts a netlink message, asking the applications to prepare for VM suspension. This message also queries the applications for the current VA ranges of the skip-over areas, which are needed by the final transfer bitmap update.

To prepare for VM suspension, the applications ensure that when the VM resumes running in the destination, the contents of their skip-over areas, which are not transferred to the destination, are recoverable or unneeded. For example, they may need to execute to a known recoverable state, flush caches or collect garbage. Once completing the actions required, they notify the LKM, passing along the current VA ranges of the skip-over areas.

Knowing that the applications are `suspension-ready`, the LKM performs the final transfer bitmap update, and then notifies the migration daemon to suspend the VM and proceed with the last iteration. The contents of the skip-over areas should remain recoverable or unneeded until VM suspension is completed.

Migration finishes and VM resumed. After the last iteration finishes, the migration daemon activates the VM at the destination, and notifies the LKM that VM execution has `resumed`. The LKM asks the applications to execute recovery logic for their skip-over areas, or to consider those areas empty, as they continue to run. It then returns to the `initialized` state in preparation for the next migration.

4.4 JAVMM: Java-Aware VM Migration

Using our framework for application-assisted live migration, we have designed and implemented JAVMM, which migrates Java VMs assisted by JVM.

In designing JAVMM, we considered skipping transfer of both the JVM code cache and garbage in the Java heap. The code cache stores native code compiled for performance enhancements. If it is not migrated, applications can resume running interpreted in the destination, but we have observed a non-trivial performance drop in such a case. Since the code cache is small relative to the Java heap, we decided to migrate it as usual, and focus on skipping the transfer of garbage in the Java heap.

4.4.1 Background on Java Heap Management

As a Java program runs, objects are created in the heap of its JVM. Most implementations of JVM (e.g., Oracle's HotSpot and JRockit and IBM's JVM) use a *generational* heap. The remainder of this chapter is presented in the context of HotSpot, based on which JAVMM is prototyped. The general principles and our design of JAVMM are also applicable to other JVM implementations.

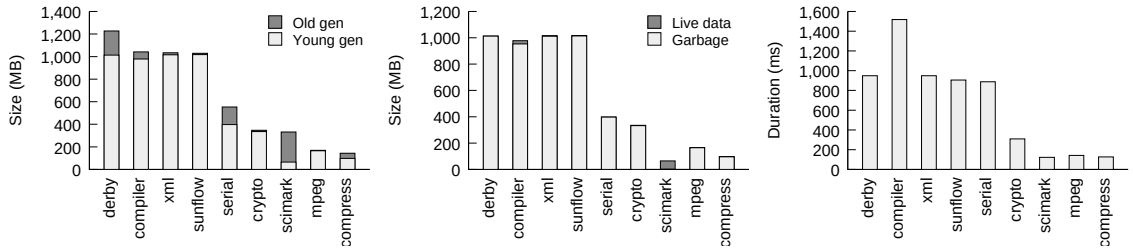
In HotSpot, the heap is divided into *Young* and *Old* generations. The Young generation is further divided into three spaces: *Eden* and two survivor spaces, *From* and *To*. Most objects are allocated in the Eden. When the Eden gets filled up, JVM performs a *minor* garbage collection (GC) to reclaim memory from garbage in the Young generation. Java (application) threads execute to a *Safepoint* [6] and pause for a minor GC, so that GC threads can move objects in the heap in a consistent manner. A minor GC copies live data in the Eden to the To space. Live data in the From space are either copied to the To space, or promoted to the Old generation if they have survived a number of minor GCs. At the end of a minor GC, the Eden is completely empty. The From and To spaces swap roles: From becomes the one that holds live data, and To becomes empty.

4.4.2 Garbage in Java Heap

To understand Java heap usage, we experiment with the SPECjvm2008 suite [16], a benchmark suite for measuring the performance of Java runtime environments. We run one

Workload	Description
derby	Apache Derby [1] database with business logic
compiler	OpenJDK 7 front-end compiler [13]
xml	Apply style sheets to XML documents
sunflow	An open-source image rendering system [17]
serial	Serialize and deserialize primitives and objects
crypto	Sign and verify with cryptographic hashes
scimark	Compute the LU factorization of matrices
mpeg	MP3 decoding
compress	Compression by a modified Lempel-Ziv method

Table 4.1: Description of the SPECjvm2008 workloads used in our experiments.



(a) Java heap memory usage (b) Garbage in a minor GC (c) Duration of a minor GC

Figure 4.5: Java heap usage and GC behavior of sample workloads from SPECjvm2008 running in a 2GB VM; see Table 4.1 for workload descriptions. The Young generation of the Java heap is allowed to use at most 1GB memory.

workload from each benchmark category for 10 minutes in a 2GB VM, using HotSpot and its parallel garbage collector; Table 4.1 describes the workloads used. HotSpot is allowed to grow the Young generation to the maximum size of 1GB and the Old generation to use the rest of the VM memory.

Figure 4.5(a) shows the average memory consumption of the Java heap. For 8 of the 9 workloads evaluated, the Young generation grows faster and uses more memory than the Old generation; up to 98% of the heap memory is consumed by the Young generation. Only scimark uses more memory in the Old generation, since that workload has more long-lived than short-lived objects. We observed that for derby, compiler, xml and sunflow, the Young generation quickly grows to the maximum size of 1GB to accommodate the large number of objects created by the workloads; these workloads have high object allocation rates.

A large portion of the Young generation memory may contain garbage when the lifetime of the objects is short. For all workloads except scimark, over 97% of the Young generation memory is garbage collected in a minor GC, as shown in Figure 4.5(b). The amount of garbage is significant for the four workloads using a 1GB Young generation. We observed that these workloads fill the Young generation and trigger a minor GC frequently, every 3 seconds or so; each minor GC reclaims almost all of the Young generation memory. Throughout workload execution, this pattern repeats, and the entire Young generation is continuously dirtied.

Figure 4.5(c) shows the average time required to collect Young generation garbage by a minor GC. Our results suggest that it may be faster to collect the garbage than to transfer them over a bottleneck network link. This applies to all workloads except scimark, which has exceptionally small amounts of garbage. Even for compiler, which has the longest GC duration of the workloads, its 950MB of garbage takes 1.5 seconds to be collected, but would take more than 7 seconds to be transferred over the gigabit Ethernet link in our testbed. Note, however, that for Old generation garbage, collection may not be faster than transmission. In our experiments, a full GC can take as long as 4 seconds to collect only 93MB of garbage in the Old generation.

In summary, for a wide range of Java workloads we have made the following observations.

Observation 1. The Young generation can be large and continuously dirtied, due to the high object allocation rate of the workload.

Observation 2. A significant portion of the Young generation memory may contain garbage, due to the workload's use of short-lived objects.

Observation 3. Collecting Young generation garbage may be faster than sending them over a bottleneck network link.

4.4.3 JAVMM

The Young generation can generate a large number of dirty pages during the migration of a Java VM, yet many of the dirty pages may contain garbage (Observations 1 and 2). JAVMM thus skips transfer of the garbage with assistance of JVM, which knows where garbage objects are located in memory.

Garbage objects are scattered among live data, and their locations keep changing as objects become unreferenced. It is impractical to keep track of the locations of garbage objects in order to skip their migration. Instead, JAVMM enforces a minor GC to collect garbage for efficient migration, since collection may be faster than network transmission (Observation 3).

Built on the framework described in Section 4.3.3, JAVMM enforces a minor GC only once during migration, when running applications are notified by the LKM to prepare for VM suspension. After the enforced GC completes, the VM is suspended. In the Young generation, the Eden and To spaces are empty, and only the From space may contain live data, *i.e.*, the data surviving the enforced GC. These live data are the only Young generation data that will be used when the VM resumes running in the destination.

JAVMM makes sure to transfer these live data in the last iteration, and throughout migration, it skips transfer of the memory pages in the Young generation, even if they are dirtied. JAVMM is thus beneficial for migrating Java VMs with a large and frequently-dirtied Young generation; this typically happens when the running Java applications are characterized by high object allocation rates.

4.4.3.1 System Overview

In JAVMM, JVM provides all the assistance needed for VM migration on behalf of Java applications; no modifications to Java applications are required. Figure 4.6 shows how JAVMM is built based on our framework for application-assisted live migration; our prototype uses HotSpot JVM (OpenJDK 7) and its parallel garbage collector.

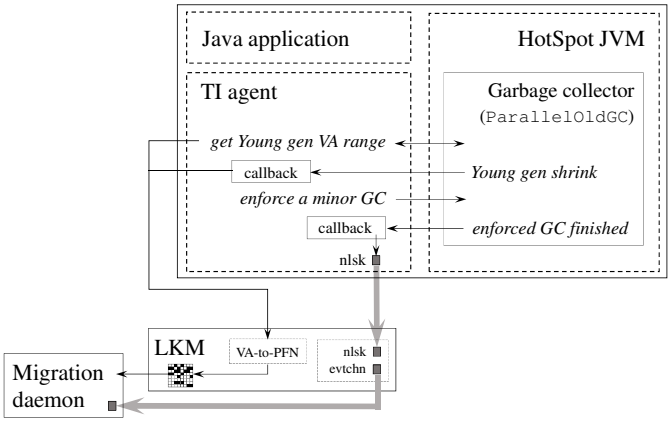


Figure 4.6: An overview of JAVMM, which is built on our framework for application-assisted live migration. This is a zoom-in view of Figure 4.2 with JVM/Java application being the running application.

We enable JVM to communicate with our LKM and collaborate with the migration daemon through the LKM. In prototyping JAVMM, we wanted to provide most of the functionalities required of JVM as pluggable modules, and minimize modifications to the core HotSpot code. We thus implemented an agent using JVM Tool Interface (TI) [10], a native programming interface for inspecting and controlling JVM. The TI agent compiles to a dynamic library to be loaded as Java applications run; it runs in the same OS process as the JVM/Java applications. JVM interacts with the LKM through the TI agent. When the functionality required is beyond the current scope of TI, we extend TI with the necessary modifications to HotSpot.

4.4.3.2 Workflow of JAVMM

Figure 4.7 shows the workflow of JAVMM; it details how JVM accomplishes the actions required of an application assisting in migration, sketched in the gray boxes of Figure 4.4.

As a Java application runs, our TI agent is loaded. It creates a netlink socket to communicate with the LKM.

The agent is notified by the LKM when migration begins, and is queried for skip-over areas. It obtains the VA range of the Young generation from JVM, and tells the LKM. Based on the agent's response, the LKM performs the first transfer bitmap update. It clears

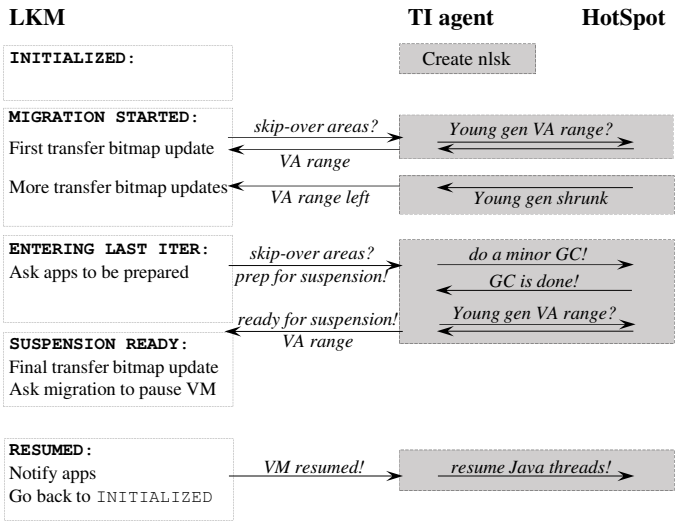


Figure 4.7: The workflow of JAVMM, with details of JVM's and our TI agent's actions to fulfill the requirements of an application assisting in migration shown in Figure 4.4.

the transfer bits of the Young generation pages, so the pages will not be transferred even if they are dirtied.

During migration, the agent notifies the LKM when memory pages leave the Young generation, so that the transfer bitmap can be updated. In HotSpot, memory pages may be freed from the Young generation at the end of a GC. We slightly modify HotSpot to notify when this happens, based on TI's notification interface of GC events. A callback in our agent is invoked to pass to the LKM the VA range with memory pages freed, and the LKM immediately sets the transfer bits of the pages leaving the Young generation.

The agent is notified by the LKM again when migration is about to enter the last iteration, and is asked to prepare for VM suspension. It enforces a minor GC to collect Young generation garbage; we modify HotSpot to ensure that this GC is not silently ignored.[3]

As usual, Java threads execute to a Safepoint and pause, and JVM performs a collection. Once the collection is finished, a callback in our agent is executed; at this time, the Eden and To spaces are empty, and the Java threads are still paused. Without giving JVM control to

[3]HotSpot may ignore GC requests when several requests are enqueued at about the same time due to simultaneous allocation failures in multiple threads—only one of these requests needs to be executed.

release the Java threads from the Safepoint and resume their execution, the agent notifies the LKM that the application is ready for VM suspension. The Java threads are thus prevented from using the heap, and this ensures the Eden and To spaces remain empty until VM suspension is completed.

Along with the notification of the application being suspension-ready, the agent passes to the LKM the current VA range of the Young generation and also that of the occupied From space, which contains the live data surviving the enforced GC. Based on these information, the LKM performs the final transfer bitmap update. It considers the occupied From pages "leaving" the Young generation, and sets their transfer bits, in order to ensure transfer of live Young generation data in the last iteration.

Once the final transfer bitmap update is completed, the migration daemon suspends the VM, and finishes migration with the last iteration. When the VM resumes in the destination, our agent is notified by the LKM. It returns control to JVM, which in turn releases the Java threads from the Safepoint. The Java application then resumes execution with all live data available in the destination.

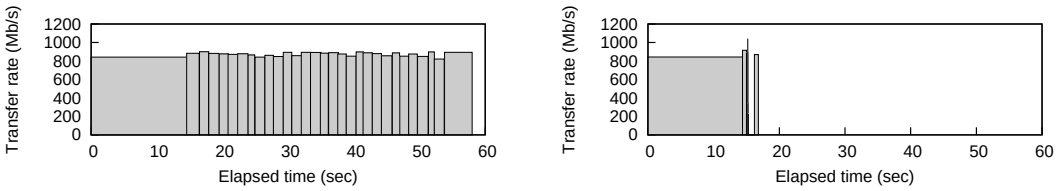
4.5 Evaluation

We now evaluate JAVMM in comparison with Xen VM live migration, which is a traditional pre-copy approach that is agnostic of the applications running in the migrating VM.

4.5.1 Experimental Setup

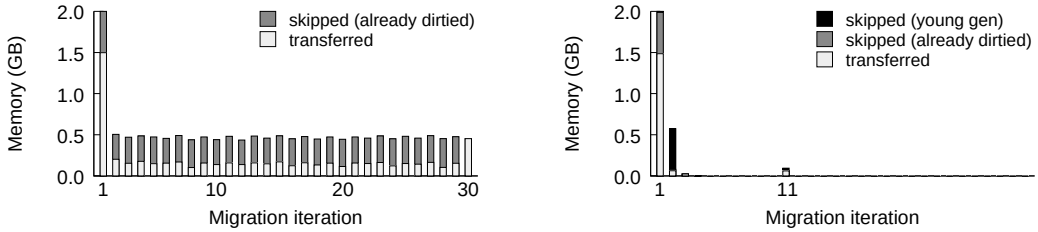
Our evaluation uses both real-life applications and benchmarks from SPECjvm2008 [16], the same benchmark suite used to profile Java heap usage in Section 4.4.2.

We run each workload for 10 minutes in a VM configured with 2GB memory and 4 vCPUs. Halfway through the workload execution, we migrate the VM, between two HP Proliant BL465c blades in the same gigabit Ethernet LAN; each blade is equipped with two



(a) Xen (b) JAVMM

Figure 4.8: Progress of migrating a VM running the compiler workload from SPECjvm2008. Each box represents a migration iteration; the width shows the duration and the area shows the amount of traffic sent. In (b), the second last iteration of JAVMM generates little network traffic while waiting for the workload to execute to a Safepoint (0.7 sec) and a minor GC to be done (0.1 sec).



(a) Xen (b) JAVMM

Figure 4.9: Amount of memory processed when migrating a VM running the compiler workload from SPECjvm2008. In (b), the 4–10th iterations of JAVMM each process less than 2MB of dirty memory.

dual-core AMD Opteron 2.2 GHz CPUs and 12GB RAM.

Alongside each workload, we run a custom analyzer that sends out the number of operations completed by the workload once every second. We observe workload throughput from outside of the VM using a time source that is not affected by temporary suspension of the VM, which happens before completing migration.

Each experiment is repeated at least three times. Unless otherwise mentioned, we report the average of the measurements, and show 90% confidence intervals in bar graphs.

4.5.2 Progress of Migration

We begin by analyzing how a Java VM is migrated iteratively by Xen and JAVMM, respectively. We use a VM running the compiler workload from SPECjvm2008 as an exam-

ple; see Table 4.1 for the workload description. Figure 4.8 plots the progress of migrating the VM in an experimental run. We plot each iteration by a box, and show the duration and the amount of traffic sent by the width and area of the box, respectively.

In the first iteration, Xen and JAVMM perform equally well. They both skip sending about 500MB of memory, as shown in Figure 4.9. Xen skips over pages that are dirtied before transmission, since such pages may be sent in the next iteration. Prototyped on Xen, JAVMM also skips over pages that are already dirtied, and in addition, all Young generation pages. The workload is using a 512MB Young generation when migrated, and most of the space is skipped over by both Xen and JAVMM in the first iteration.

Xen and JAVMM start to progress differently from the second iteration. Although they both have more than 500MB of dirty memory pending transmission in the second iteration, they transfer different amounts. JAVMM sends only 64MB of the dirty memory, skipping over both repeatedly dirtied pages and Young generation pages. Xen has to send more than 200MB of the dirty memory, since it can only skip over repeatedly dirtied pages.

Since JAVMM sends less dirty data, it finishes the second iteration faster, during which less memory gets dirtied. As a result, it has even less dirty data to send in the third iteration. JAVMM reduces the amount of memory transfer effectively as iterations progress. After 10 iterations, little dirty memory remains to be sent. JAVMM then finishes migration with a short stop-and-copy at the 11th iteration, using 17 seconds and sending 1.6GB of network traffic.

However, for Xen, the amount of memory transfer does not decrease over the iterations. Migration is forced to enter stop-and-copy when it reaches the maximum 30 iterations allowed by Xen's default; the stop-and-copy takes long, since over 400MB of dirty memory remains to be sent. Xen finishes migration taking 58 seconds and sending 6.1GB of network traffic, i.e., over 3x longer time and more traffic than JAVMM.

4.5.3 Performance of Migration

Next, we evaluate JAVMM for workloads with different characteristics of Java heap usage.

Workload characterization. When profiling sample workloads from SPECjvm2008 in Section 4.4.2, we found the workloads fall in the following three categories according to Java heap usage; see Table 4.1 for description of the workloads.

- **Category 1.** The Young generation quickly grows to the maximum size, since the workload has a high object allocation rate. The derby, compiler, xml and sunflow workloads are in this category.

- **Category 2.** The Young generation grows faster than the Old generation, albeit not maximally utilized. The workload has a medium object allocation rate. The serial, crypto, mpeg and compress workloads are in this category.

- **Category 3.** The workload has a small Young generation and a large Old generation, since the object allocation rate is low, and most of the workload data are long-lived. Scimark is the only workload in this category.

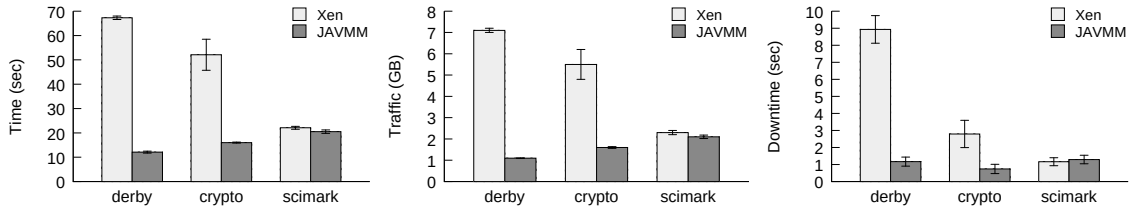
Our observations on object allocation rates are consistent with the measurements by other researchers [90].

We evaluate JAVMM using one workload from each category. For Category 1, which is the most favorable workload scenario for JAVMM, we evaluate derby; in the workloads of this category, derby uses the largest Old generation, which JAVMM has to transfer. For category 2, we evaluate crypto. For category 3, which is the least favorable workload scenario for JAVMM, we evaluate scimark.

Derby, crypto and scimark are all CPU-intensive workloads. They use up 90% of CPU, and perform no network I/Os. Table 4.2 shows their experimental settings. While each workload can use a maximum 1GB Young generation, when migrated, the Young genera-

Workload	Max allowed Young gen (MB)	Observed when migrated Young gen (MB)	Old gen (MB)
derby	1024	1024	259
crypto	1024	456	18
scimark	1024	128	486

Table 4.2: Workloads with different characteristics of Java heap usage and their experimental settings.



(a) Total migration time (b) Total migration traffic (c) Workload downtime

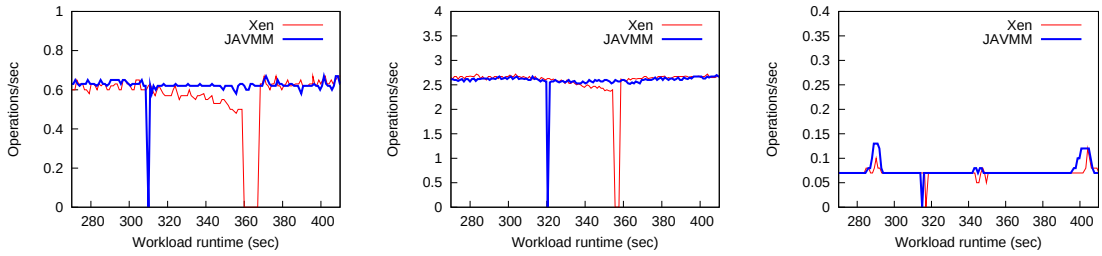
Figure 4.10: Performance of JAVMM and Xen live migration for workloads with different characteristics of Java heap usage.

tions of derby, crypto and scimark are using 1GB, 0.4GB and 0.1GB of memory, respectively.

How fast does JAVMM migrate a Java VM? Figure 4.10(a) shows the time required to migrate the VMs running the three workloads. JAVMM migrates the derby VM fastest, taking only 12 seconds. Compared to Xen, which takes over a minute to migrate the VM, JAVMM reduces the migration time by 82%. JAVMM also achieves a 69% reduction of migration time for the crypto VM. For scimark, JAVMM can skip over little Young generation memory. It migrates the VM using a comparable amount of time as Xen.

How much resource does JAVMM use for migration? Figure 4.10(b) shows the amount of network traffic transferred to migrate the VMs. For derby and crypto, JAVMM migrates the VM sending even less traffic than the VM size, while Xen sends up to 3.5x the VM size of migration traffic. Compared to Xen, JAVMM reduces migration traffic for derby and crypto by 84% and 72%, respectively. For scimark, JAVMM achieves a 10% reduction of migration traffic.

Thanks to the reduced data transfer, JAVMM also uses up to 84% less CPU time than



(a) Derby (b) Crypto (c) Scimark

Figure 4.11: Effect of VM migration on the throughput of running application, *i.e.*, the number of operations completed per second. Migration begins after the application runs for 300 seconds.

Xen in migrating the VMs. In these experiments, JAVMM uses at most 1MB of memory for the transfer bitmap and PFN cache.

How much does JAVMM affect application performance? Figure 4.11 shows the throughputs of the workloads. For each workload, the VM is migrated after the workload runs for 300 seconds. Using JAVMM, the workload experiences no noticeable throughput degradation during migration, except the short pause before migration finishes. When migrated by Xen, the workload can experience an extended downtime.

Figure 4.10(c) shows the workload downtime. The downtime includes the time spent in the last iteration and the resumption time. The resumption time is required to reconnect VM devices and activate VM execution in the destination; this time is short, only about 170 ms in our measurements. For JAVMM, the downtime also includes the time required to finish the enforced GC while the workloads are paused at a Safepoint, as well as the time required by the final transfer bitmap update; the latter is completed quickly, within 300 μs in all our experiments.

Derby experiences 1.2 seconds of downtime when the VM is migrated by JAVMM, 83% shorter than the 9-second downtime when the VM is migrated by Xen. Derby dirties the 1GB Young generation rapidly, but JAVMM can still reduce the amount of memory transfer iteratively, by skipping transfer of Young generation pages. In the last iteration, JAVMM sends only 11MB of dirty data, skipping over Young generation garbage, while

Workload	Max allowed Young gen (MB)	Observed when migrated Young gen (MB)	Old gen (MB)
xml	1536	1536	28
derby	1024	1024	259
compiler	512	512	86

Table 4.3: Workloads with high object allocation rates and their experimental settings.

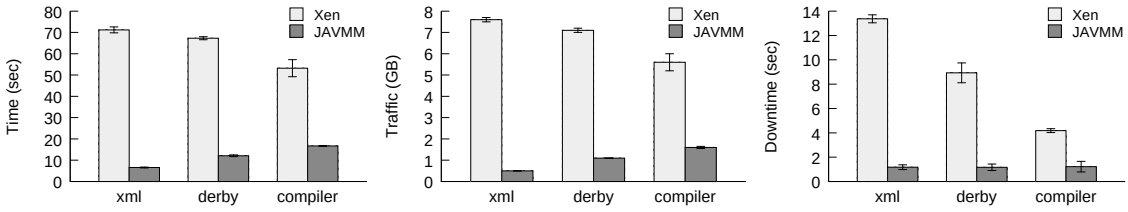
Xen has over 900MB of dirty data to be sent. JAVMM thus reduces the downtime of derby significantly compared to Xen, even though it uses 0.9 second to finish the enforced GC; the GC duration can be further shortened with Java heap fine-tuning or increased parallelism. For crypto, JAVMM also achieves a 73% shorter downtime than Xen.

However, for scimark, JAVMM imposes a 10% longer downtime than Xen; scimark is paused for 1.2 and 1.3 seconds when the VM is migrated by Xen and JAVMM, respectively. For this workload, JAVMM takes time to perform the enforced GC, but the amount of data to be transferred in the last iteration is not reduced. Most of scimark's objects are long-lived. They survive the GC enforced, and must be sent in the last iteration.

Summary. JAVMM is advantageous in migrating the VMs running derby and crypto, representatives of workloads with non-trivial Young generation sizes and object allocation rates. Compared to Xen, JAVMM migrates these VMs achieving shorter completion time, smaller network traffic and shorter downtime. Scimark represents workloads using a small Young generation and many long-lived objects. Compared to Xen, JAVMM migrates this VM with a slightly longer downtime, although it achieves comparable, or slightly better migration time and traffic.

4.5.4 Impact of Young Generation Size

We conducted a second set of experiments for the workloads most favorable for JAVMM, Category 1 workloads with high object allocation rates. We evaluate the benefit of using JAVMM for these workloads with varying sizes of Young generation, focusing on the same three evaluation questions discussed in Section 4.5.3.



(a) Total migration time (b) Total migration traffic (c) Workload downtime

Figure 4.12: Performance of JAVMM and Xen live migration for Category 1 workloads with different size Young generations.

We experimented with derby and two additional workloads, xml and compiler, from Category 1. All three workloads are CPU-intensive and without network I/Os. We specify different maximum sizes for the Young generations of the workloads, as shown in Table 4.3. When migration begins, the Young generations of xml, derby and compiler all reach the maximum sizes. They are using 1.5GB, 1GB and 0.5GB of memory, namely, 75%, 50% and 25% of the VM memory, respectively.

Figure 4.12(a) shows the time required to migrate the VMs running the three workloads. With high object allocation rates, the workloads dirty the entire Young generation space rapidly. For Xen, the larger the Young generation, the more dirty memory are repeatedly transferred, and the longer it takes to migrate the VM. On the contrary, JAVMM migrates the VMs with larger Young generations faster, since more dirty memory are skipped over. JAVMM thus achieves greater reductions of migration time for the VMs with larger Young generations, than Xen. For the xml, derby and compiler workloads, JAVMM migrates the VM using 91%, 82% and 69% less time than Xen, respectively.

A similar trend is observed for the amount of network traffic sent for migrating the VMs, as shown in Figure 4.12(b). For JAVMM, the larger the Young generation, the less migration traffic is sent, and it achieves a greater traffic reduction than Xen. JAVMM sends 93% less traffic than Xen to migrate the VM running xml, which has the largest Young generation of the workloads.

Figure 4.12(c) shows the downtime incurred by the workloads before migration is com-

pleted. When the VM is migrated by Xen, the workloads with larger Young generations incur longer downtimes. A large portion of the Young generation keeps getting dirtied until the VM is paused for the last iteration, due to the workloads' high object allocation rates. Xen has up to 1.5GB of data to send in the last iteration, resulting in up to 13 seconds of downtime.

For JAVMM, there is not a direct relationship between downtime and the Young generation size, since downtime also affected by other factors, *i.e.*, the duration of the enforced GC and the amount of surviving data to be sent in the last iteration. The three workloads experience about 1.2 seconds of downtime when the VM is migrated by JAVMM, up to 91% shorter than their respective downtimes incurred when migrated by Xen.

4.6 Discussions on Applications and Extensions

When to use JAVMM? JAVMM is most beneficial for the cases which are most problematic to traditional pre-copy approaches—when the VM to be migrated runs Java applications with large Young generations and high object allocation rates.

In some cases, JAVMM should be used with consideration of the resulting application downtime. The first is when the application requires long minor GCs, since the duration of the enforced GC increases downtime. The second is when the application has a high object survival rate. Many objects may survive the enforced GC and must be transferred during stop-and-copy. Scimark is such an example. The third is when the application is read-intensive for which traditional pre-copy approaches can reduce downtime effectively; the GC enforced by JAVMM is likely to increase downtime.

Use JAVMM for large VMs with fast networks. Our evaluation has shown benefits of JAVMM by migrating a 2GB VM over a gigabit Ethernet. These benefits remain as VMs configured with tens or hundreds of GBs of memory are migrated over 10 Gbps or faster networks, since in such scenarios, the VM processing power, application memory footprints and memory-dirtying rates likely increase proportionally. As we continue to

deploy JAVMM in upgraded environments, the underlying network may remain as much a bottleneck as in our current testbed.

Use JAVMM with other garbage collectors. While the design of JAVMM is orthogonal to the choice of garbage collector, we are particularly interested in porting JAVMM to run with collectors that use a non-contiguous VA range for the Young generation for performance evaluation and optimization. HotSpot's garbage-first garbage collector [43] is one such example.

Support large and multiple applications. The LKM updates the transfer bitmap on applications' behalf. It can coordinate concurrent bitmap updates from multiple applications, and prevent the applications from manipulating others' memory. While the LKM can notify a set of applications with multicast, care is needed to collect responses from all of them and handle any straggler. We are also investigating parallelization of transfer bitmap updates to handle large skip-over areas efficiently.

4.7 Conclusions

In this chapter, we have proposed application-assisted live migration, skipping transfer of selective VM memory pages based on application semantics. We have built a generic framework for the proposed approach, which is then used to build JAVMM, a system that migrates VMs running Java applications skipping transfer of garbage in Java memory. Our experimental results have shown that JAVMM can migrate a Java VM with up to more than 90% less completion time, less network traffic and shorter application downtime than Xen live VM migration, which is agnostic of application semantics. JAVMM also incurs a lower CPU cost than Xen live VM migration and a negligible memory overhead. In JAVMM, JVM is enabled to provide all the assistance needed for migration on behalf of Java applications; no modifications to Java applications are required by JAVMM for efficient migration of a VM.

CHAPTER V

Conclusions

In this thesis, we have explored ways to replicate VMs for HA using resources efficiently, and to migrate VMs fast, with minimal execution disruption and using resources efficiently. We now summarize the contributions of this thesis and the directions in which the research of this thesis can be extended.

5.1 Thesis Contributions

To reduce the network traffic of checkpoint replication in a HA system, we have shown that checkpoint compression can be applied, adapting to the workload types and resource constraints in the system, by evaluating and comparing the strengths and weaknesses of different compression methods. To the best of our knowledge, this is the first detailed evaluation and characterization of checkpoint compression methods in the context of supporting HA, considering gzip, delta and similarity compressions. Based on the evaluation results, we provide guidelines for their selection and usage.

To reduce the memory requirement of maintaining backup VMs for HA, we have shown that a memory-efficient HA alternative is feasible, by building HydraVM, a storage-based HA approach for VMs. HydraVM uses a new combination of well-known system techniques, including incremental VM checkpointing, demand paging and pre-fetching, to solve the real-world problem of providing resource-efficient HA support for VMs. Our

prototype implementation and evaluation of HydraVM demonstrate the applicability of this solution.

Finally, we have shown the utility of running applications' assistance in VM live migration. We have established a generic framework for application-assisted live migration, which selectively skips transfer of VM memory based on application semantics. Using this framework, we have built JAVMM, which migrates Java VMs skipping transfer of garbage in Java memory by leveraging JVM's assistance. Our evaluation of JAVMM in comparison with Xen live migration, which is agnostic of applications running in migrating VMs, has validated the effectiveness of our approach.

5.2 Future Directions

The research in this thesis can be extended in the following directions:

- **Hybrid and automatically selected checkpoint compression methods in HA systems.**

  Our characterization of compression methods shows that it is useful to combine the strengths of different compression methods in a hybrid approach. Cully *et al.* [39] briefly discussed using gzip and delta compression together to achieve greater reductions of checkpoint traffic, but our evaluation suggests that combining them could potentially incur high CPU and memory costs at the same time. We propose to combine a lightweight technique, like similarity compression, with heavyweight ones, such as gzip. Coarse-grained similarity compression (e.g., based on 1K chunks) can be used to achieve a meaningful, though not significant, reduction of checkpoint sizes, at a low computing overhead. The remaining checkpoint data can be compressed greater and faster with gzip. If for some workloads, similarity compression reduces checkpoint traffic effectively already, gzip need not be performed.

  We would also like to develop heuristics that utilize the insights from our evaluation

to automate the selection of compression methods according to workload types and resource constraints, and even to dynamically adjust the selection decisions at runtime. Such heuristics are especially useful for building an intelligent HA system to provide VM protection at a large scale using resources efficiently.

- **Transparent failure recovery of client-facing applications by a storage-based HA approach.**

  It is challenging for a storage-based HA approach like HydraVM to checkpoint a protected VM frequently like approaches using in-memory backups do and support network buffering, which in turn enables transparent failure recovery for client-facing applications running in the VM, since each checkpoint takes much longer to store on permanent storage devices than in memory. Large, parallel storage systems help overcome the intrinsic slowness of permanent storage devices, as discussed in Section 3.6, but it can be difficult to rely solely on the storage system's scale and data parallelism to achieve the kind of checkpointing frequencies required by network buffering to not incur undue delays on network packets, especially when running applications have large writable working sets. It would be useful to use a small amount of memory in a stage buffer, to hold and coalesce part of the VM state (e.g., the most frequently checkpointed pages) before writing them to storage. This limited use of memory speeds up checkpoint storage during VM protection, and can also help in fast VM recovery.

- **Exploiting greater intelligence of running applications in VM live migration.**

  In application-assisted live migration, we have presented a framework for the migration tool to be informed by running applications and know, for each memory page of the migrating VM, whether to send or skip the page. We plan to extend this framework so that the migration tool can receive more guidance from running applications and exploit the greater intelligence therein. The tool can then perform a richer set

of operations to reduce the amount of memory transfer for efficient live migration. For example, the tool can apply compression on the memory pages that are not being skipped over. This not only reduces memory transfer further, but also uses compression, a CPU-expensive operation, at a lower cost. The transfer bitmap in our framework can be augmented to have more than one bit for each VM memory page, and indicate the suitable compression methods to apply before sending the page over the network.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apache Derby database in Java. http://db.apache.org/derby.

[2] Basic Compression Library. http://bcl.comli.eu.

[3] The FFmpeg multimedia tool. http://www.ffmpeg.org.

[4] Four Java cloud platforms reviewed. http://www.javaworld.com/article/2078443/mobile-java/four-java-cloud-platforms-reviewed.html.

[5] GNU zip utility. http://www.gzip.org.

[6] HotSpot glossary of terms. http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html.

[7] HotSpot virtual machine. http://openjdk.java.net/groups/hotspot/.

[8] The HPC Challenge benchmark. http://icl.cs.utk.edu/hpcc.

[9] Hyper-V server virtualization technical overview. http://download.microsoft.com/download/A/2/7/A27F60C3-5113-494A-9215-D02A8ABCFD6B/Windows_Server_2012_R2_Server_Virtualization_White_Paper.pdf.

[10] JVM Tool Interface (TI). http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html.

[11] Live migration on KVM. http://www.linux-kvm.org/page/Migration.

[12] LVM2 resource page. http://sourceware.org/lvm2.

[13] OpenJDK 7. http://openjdk.java.net/projects/jdk7/.

[14] Popularity of Java applications. http://www.java.com/en/about/.

[15] The RUBiS benchmark. http://rubis.ow2.org.

[16] The SPECjvm2008 benchmark suite. http://www.spec.org/jvm2008.

[17] Sunflow open source rendering system. http://sunflow.sourceforge.net.

[18] A TPC-C-like benchmark of VoltDB. http://community.voltdb.com/node/134.

[19] VMware distributed resource scheduler (DRS). http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf.

[20] VMware Fault-Tolerance (FT). http://www.vmware.com/products/fault-tolerance.

[21] VMware High-Availability (HA). http://www.vmware.com/products/vi/vc/ha.html.

[22] VoltDB in-memory database. http://community.voltdb.com.

[23] zlib compression library. http://zlib.net.

[24] Anurag Agarwal, Dharmesh Shah, Nagaraj Kalmala, Neelakandan Panchaksharam, Rajeev Bharadhwaj, Sameer Lokray, Srikanth Sm, and Thomas Bean. Method and apparatus for transactional fault tolerance in a client-server system, Oct. 2009. Patent, US 7610510.

[25] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. VMFlock: Virtual machine co-migration for the cloud. In *Proceedings of the 20th Symposium on High Performance Distributed Computing*, 2011.

[26] Javanshir Farzin Alamdari and Kamran Zamanifar. A reuse distance based precopy approach to improve live migration of virtual machines. In *Proceedings of the 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, pages 551–556, 2012.

[27] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet caches on routers: The implications of universal redundant traffic elimination. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 219–230, 2008.

[28] Ashok Anand, Vyas Sekar, and Aditya Akella. SmartRE: An architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, pages 87–98, 2009.

[29] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2nd edition, 2013.

[30] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. Jettison: Efficient idle desktop consolidation with partial VM migration. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 211–224, 2012.

[31] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.

[32] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, Feb. 1996.

[33] Alan D. Brunelle. blktrace user guide. `http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html`.

[34] Anton Burtsev, Mike Hibler, and Jay Lepreau. Aggressive server consolidation through pageable virtual machines. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (Poster Session)*, 2008.

[35] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. Transparent checkpoints of closed distributed systems in Emulab. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 173–186, 2009.

[36] B. Callaghan, B. Pavlowski, and P. Staubach. NFS version 3 protocol specification. Technical report, IETF, 1995. RFC 1813.

[37] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 133–138, 2001.

[38] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Cristian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machine. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[39] 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*, pages 161–174, 2008.

[40] Tathagata Das, Pradeep Padala, Venkata N. Padmanabhan, Ramachandran Ramjee, and Kang G. Shin. LiteGreen: Saving energy in networked desktops using virtualization. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[41] Umesh Deshpande, Brandon Schlinker, Eitan Adler, and Kartik Gopalan. Gang migration of virtual machines using cluster-wide deduplication. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 394–401, 2013.

[42] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th International Symposium on High Performance and Distributed Computing*, pages 135–146, 2011.

[43] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, pages 37–48, 2004.

[44] Yuyang Du and Hongliang Yu. Paratus: Instantaneous failover via virtual machine replication. In *Proceedings of the 8th International Conference on Grid and Cooperative Computing*, pages 307–312, 2009.

[45] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–130, 2008.

[46] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Survey*, 34(3), Sep. 2002.

[47] Balazs Gerofi and Yutaka Ishikawa. RDMA based replication of multiprocessor virtual machines over high-performance interconnects. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 35–44, 2011.

[48] Balazs Gerofi, Zoltan Vass, and Yutaka Ishikawa. Utilizing memory content similarity for improving the performance of replicated virtual machines. In *Proceedings of the 4th Conference on Utility and Cloud Computing*, 2011.

[49] 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. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 309–322, 2008.

[50] Michael 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*, pages 51–60, 2009.

[51] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Enabling instantaneous relocation of virtual machines with a lightweight VMM extension. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 73–83, 2010.

[52] Kai-Yuan Hou, Mustafa Uysal, Arif Merchant, Kang G. Shin, and Sharad Singhal. HydraVM: Low-cost, transparent high availability for virtual machines. Technical report, HP Labs, 2011.

[53] Bolin Hu, Zhou Lei, Yu Lei, Dong Xu, and Jiandun Li. A time-series based precopy approach for live migration of virtual machines. In *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, pages 947–952, 2011.

[54] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 11–20, 2007.

[55] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive memory compression. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–10, 2009.

[56] Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. Efficient live migration of virtual machines using shared storage. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 41–50, 2013.

[57] Ardalan Kangarlou, Patrick Eugster, and Dongyan Xu. VNsnap: Taking snapshots of virtual networked infrastructures in the cloud. *IEEE Transactions on Services Computing*, 5(4), 2012.

[58] Jacob Kloster, Jesper Kristensen, and Arne Mejlholm. On the feasibility of memory sharing: Content-based page sharing in the Xen virtual machine monitor. Master's thesis, Department of Computer Science, Aalborg University, 2006.

[59] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th Conference on File and Storage Technologies*, 2010.

[60] Akane Koto, Hiroshi Yamada, Kei Ohmura, and Kenji Kono. Towards unobtrusive VM live migration for cloud computing platforms. In *Proceedings of the Asia-Pacific Workshop on Systems*, 2012.

[61] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 40–46, 2002.

[62] 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*, pages 1–12, 2009.

[63] Zhaobin Liu, Wenyu Qu, Tao Yan, Haitao Li, and Keqiu Li. Hierarchical copy algorithm for Xen live migration. In *Proceedings of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 361–364, 2010.

[64] Maohua Lu and Tzi-Cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Proceedings of the 39th Conference on Dependable Systems and Networks*, pages 534–543, 2009.

[65] Fei Ma, Feng Liu, and Zhen Liu. Live virtual machine migration based on improved pre-copy approach. In *IEEE International Conference on Software Engineering and Service Sciences*, pages 230–233, 2010.

[66] Richard McDougall and Jennifer Anderson. Virtualization performance: Perspectives and challenges ahead. *SIGOPS Operating System Review*, 44(4), Dec. 2010.

[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 European Conference on Computer Systems*, 2008.

[68] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[69] Umar Farooq Minhas, Shriram Rajagopalan Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *PVLDB*, 4(11), 2011.

[70] Derek G. Murray, Steven H, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[71] 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*, pages 23–32, 2007.

[72] Ripal Nathuji and Karsten Schwan. VirtualPower: Coordinated power management in virtualized enterprise systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 265–278, 2007.

[73] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[74] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 1–15, 2012.

[75] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles*, pages 29–41, 2011.

[76] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2), May 2005.

[77] Dick Pountain. Run-length encoding. *Byte*, 12(6), 1987.

[78] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 1st Conference on File and Storage Technologies*, 2002.

[79] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981. TR-15-81.

[80] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. SecondSite: Disaster tolerance as a service. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2012.

[81] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the USENIX Annual Technical Conference*, pages 143–156, 2008.

[82] 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 *Proceedings of the European Conference on Parallel Processing*, 2011.

[83] Alan Robertson. Linux-HA heartbeat system design. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, Oct. 2000.

[84] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3), Aug. 2013.

[85] Ohad Rodeh and Avi Teperman. zFS - a scalable distributed file system using object disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[86] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer System*, 10(1):26–52, Feb. 1992.

[87] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 377–390, 2002.

[88] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Operating System Review*, 44(4), Dec. 2010.

[89] D. P. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D. M. L. Brown, and J. Nieplocha. Transparent system-level migration of PGAS applications using Xen on InfiniBand. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 74–83, 2007.

[90] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009.

[91] 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*, pages 1–12, 2008.

[92] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th Conference on Virtual Execution Environments*, pages 111–120, 2011.

[93] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshi Moriai. Kemari: Virtual machine synchronization for fault tolerance. In *USENIX Annual Technical Conference (Poster Session)*, 2008.

[94] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 193–204, 2010.

[95] 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 20th ACM Symposium on Operating Systems Principles*, pages 148–162, 2005.

[96] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, 2002.

[97] 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*, pages 121–132, 2011.

[98] 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*, pages 229–242, 2007.

[99] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 88–96, 2010.

[100] Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 1000 Islands: Integrated capacity and workload management for the next generation data center. In *Proceedings of the 2008 International Conference on Autonomic Computing*, 2008.