NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-13-05
July 20, 2013

ConCORD: Tracking and Exploiting Cross-Node Memory Content Redundancy in Large-Scale Parallel Systems

Lei Xia

Abstract

In large-scale parallel systems scientific workloads consist of numerous processes running across many nodes. Their memory footprint is massive, however many studies from both our and others' works have shown that significant amount of memory content redundancy usually exists in these systems. This has consequences for services that enhance performance, reliability, or power. For example, leveraging content sharing could significantly reduce the size of a checkpoint of a group of nodes. As another example, it could speed VM migration by allowing the reconstruction of a VM's memory from multiple source VMs. Finally, a service that improves reliability by introducing memory redundancy could leverage existing content sharing to minimize the memory costs of any particular level of redundancy.

I argue that, a facility that continuously tracks and identifies the sharing of memory content, both within individual machines, and across machines, should be included in the infrastructure of a large-scale HPC system. To support my claim, I consider 1) the *opportunity* that this service could unveil, 2) the *feasibility* of creating the service, and 3) the *impact* that such a facility would have.

In my dissertation, to show the opportunity, I demonstrate that both intra-node and inter-node memory content sharing is common in parallel scientific workloads, through a detailed study of both kinds of sharing, at different scales, different granularities, and different times for a range of applications and application benchmarks. Next, I claim

that identifying and tracking inter-node memory content redundancy is possible with low overhead on a scalable system. To support my claim, my dissertation presents the design, implementation and evaluation of *ConCORD*, a distributed system that is able to scalably measure the amount of inter-node memory content sharing and identify the specific shared content regions in large-scale parallel systems with minimal performance impact. Finally, to enhance the impact of ConCORD, I propose a content-aware service command model, which allows HPC services to be enhanced and deployed to effectively exploit memory content redundancy existing in the system with minimal efforts. By leveraging the effective and low overhead content-aware interface from ConCORD, various of such HPC services would be greatly simplified and enhanced. I will also demonstrate this in my dissertation by building one of such services on top of ConCORD.

I have implemented ConCORD and linked it to Palacios, an OS independent embeddable virtual machine monitor to which I am a major contributor.

This effort was made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, and the Department of Energy (DOE) via grant DE-SC0005343.

Keywords: Memory Content Redundancy, Content Share Tracking, Parallel System, High Performance Computing

NORTHWESTERN UNIVERSITY

ConCORD: Tracking and Exploiting Cross-Node Memory Content Redundancy in Large-Scale Parallel Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Lei Xia

EVANSTON, ILLINOIS
June 2013

Thesis Committee

Peter A. Dinda, Northwestern University, Chair
Russell Joseph, Northwestern University
Aleksandar Kuzmanovic, Northwestern University
Patrick G. Bridges, University of New Mexico
Sanjay Kumar, Intel, Intel Labs

Abstract

ConCORD: Tracking and Exploiting Cross-Node Memory Content Redundancy in Large-Scale Parallel Systems

Lei Xia

In large-scale parallel systems scientific workloads consist of numerous processes running across many nodes. Their memory footprint is massive, however many studies from both our and others' works have shown that significant amount of memory content redundancy usually exists in these systems. This has consequences for services that enhance performance, reliability, or power. For example, leveraging content sharing could significantly reduce the size of a checkpoint of a group of nodes. As another example, it could speed VM migration by allowing the reconstruction of a VM's memory from multiple source VMs. Finally, a service that improves reliability by introducing memory redundancy could leverage existing content sharing to minimize the memory costs of any particular level of redundancy.

I argue that, a facility that continuously tracks and identifies the sharing of memory content, both within individual machines, *and across machines*, should be included in the infrastructure of a large-scale HPC system. To support my claim, I consider 1) the *opportunity* that this service could unveil, 2) the *feasibility* of creating the service, and 3) the *impact* that such a facility would have.

In my dissertation, to show the *opportunity*, I demonstrate that both intra-node and inter-node memory content sharing is common in parallel scientific workloads, through

a detailed study of both kinds of sharing, at different scales, different granularities, and different times for a range of applications and application benchmarks. Next, I claim that identifying and tracking inter-node memory content redundancy is possible with low overhead on a scalable system. To support my claim, my dissertation presents the design, implementation and evaluation of *ConCORD*, a distributed system that is able to scalably measure the amount of inter-node memory content sharing and identify the specific shared content regions in large-scale parallel systems with minimal performance impact. Finally, to enhance the impact of ConCORD, I propose a content-aware service command model, which allows HPC services to be enhanced and deployed to effectively exploit memory content redundancy existing in the system with minimal efforts. By leveraging the effective and low overhead content-aware interface from ConCORD, various of such HPC services would be greatly simplified and enhanced. I will also demonstrate this in my dissertation by building one of such services on top of ConCORD.

I have implemented ConCORD and linked it to Palacios, an OS independent embeddable virtual machine monitor to which I am a major contributor.

Dedication

To my wife Wenxuan Zhang and my daughter Emma Xia.

Acknowledgments

First of all, I would like to express my special appreciation and thanks to my advisor Professor Peter A. Dinda, for his enormous patience and great efforts to guide me through my Ph.D journey. His insights, inspiration and guidance have helped me find my own path and overcome many obstacles in this journey. He always suggests me to think broadly and independently, and encourages me to pursue what I am interested in. His constant encouragement and invaluable guidance throughout the past five years makes my Ph.D study so exciting and fruitful. This thesis would have been impossible without his support and mentoring.

Secondly, I would like to thank the rest of my committee members, Prof. Russell Joseph, Prof. Aleksandar Kuzmanovic, Prof. Patrick Bridges and Dr. Sanjay Kumar for their brilliant comments and suggestions and for letting my defense be an enjoyable moment. Specially, Sanjay provided me with wise counsel when I was an intern in Intel Labs. I want to thank him for his continuous support beyond my internship.

In addition, my thesis work was significantly benefited from various collaborations with other members from V3VEE project. In particular, I would like to thank Jack Lange, Chang S. Bae, Cui Zheng, Kyle Hale, Kevin Pedretti, Maciej Swiech, Yuan Tang, Feng Lu, Madhav Suresh and many others. Also I would like to thank my collaborators from Intel Labs, particularly, Xue Yang, Xingang Guo, Praveen Gopalakrishnan, York Liu and Sebastian Schoenberg.

The research described in this dissertation is part of the V3VEE project and was supported by the National Science Foundation (NSF) via grants CNS-0709168 and CNS-0707365, and the Department of Energy (DOE) via grant DE-SC000534.

I owe so much thanks to my father Xuanhu Xia, my mother Yongju Jin and my sister

Juan Xia who were continuously supporting me with love throughout my life and leaving me free in all my decisions. Finally I would like to dedicate this dissertation to my wife Wenxuan Zhang for her many years of patience and love, and my beautiful daughter Emma Xia for giving me unlimited happiness and pleasure.

Contents

List of Figures

Chapter 1

Introduction

Content-based memory sharing has been studied for many years, but it is primarily employed to deduplicate memory pages and reduce memory footprint size in virtualization platforms. Studies from both us and others have shown that significant amounts of intra-node and *inter-node* content sharing exist in scientific workloads that run in large-scale parallel systems [12, 140, 73]. However, very little work has been done on studying how we can leverage this content sharing, particularly this *cross-node* sharing, to improve or enable many services that enhance performance, reliability, or power in such large scale systems. In addition, a facility that can continuously track the sharing of memory content, both intra-node and inter-node sharing, is still lacking in the infrastructure of such large-scale parallel system.

1.1 Content-sharing tracking as a service

In modern large scale data centers, virtualization is usually applied to facilitate servicing multiple customers on single servers, in which multiple virtual machines (VMs) are collocated on a single physical machine. In such systems, memory efficiency is a key consideration. One technique for improving memory efficiency in virtualized systems is using content-based page sharing. In these systems, the hypervisor uses hashing and page com-

parison to identify duplicate pages and collapses all duplicate virtual pages into a single physical copy. In doing so, the primary aim is to capture sharing between VMs that are highly similar, significantly reducing the memory footprint of such VMs.

Similarly in large-scale parallel systems, scientific workloads usually generate massive memory footprints. Many studies from both our and others' works have shown that significant amounts of memory content sharing, both intra-node and inter-node, exist in these systems. Particularly, there is substantial additional sharing across nodes beyond that of sharing within individual nodes.

By leveraging memory content sharing, particularly *inter-node* sharing, many critical services that enhance performance, reliability, or power in high performance computing can be enabled or improved. For example, leveraging content sharing could significantly reduce the size of a checkpoint of a group of nodes. As another example, it could speed VM migration by allowing the reconstruction of a VM's memory from multiple source VMs. Finally, a service that improves reliability by introducing memory redundancy could leverage existing content sharing to minimize the memory costs of any particular level of redundancy. These content-aware services are illustrated as follows.

Content-aware checkpointing Harnessing peta- and exascale computational power presents a challenge as system reliability deteriorates with scale. The first driving service, checkpointing [4, 86] with rollback, is a well-known technique for fault-tolerance in which the application save it state in stable storage, usually in a parallel file system (PFS), and rolls back in the event of a node failure.

Checkpointing results in high overheads due to often simultaneous writes by all nodes to the PFS, which reduces the productivity of such systems. For example, when a large parallel application is checkpointed, tens of thousands of nodes may write their memory content to the PFS, producing many terabytes of data. Furthermore, the I/O bandwidth of

HPC systems generally does not increase at the same rate that computational capabilities and physical memory do. And the larger the system is, the more frequently checkpoints may be needed due to the lower mean time to failure. Even assuming storage is cheap, ever larger checkpoints at every higher rates will lead to an I/O bottleneck. This appears to be already occurring on petascale systems, and will certainly occur in exascale systems.

A facility that dynamically tracks the sharing of memory content could address this checkpointing problem. First, leveraging sharing would allow us to significantly reduce the size of a checkpoint by saving only a single copy of each distinct memory page. Second, it would reduce the total time and the I/O bandwidth needed to transfer and store the checkpoints.

VM migration VM migration [19, 112], our second driving service, is a useful feature provided by most virtualization systems. This capability is being increasingly employed in today's HPC systems to help provide fault tolerance [89]. Current VM migration services are mostly focused on migrating a single virtual machine across hosts. A facility that tracked content sharing could speed single VM migrations by allowing the reconstruction of the VM's memory from multiple source VMs. Furthermore, there are many cases in which migrating a *set* of VMs that runs a parallel application has been found useful [92]. By leveraging intra-node and inter-node memory content sharing, each distinct memory page in the group of migrating VMs could be copied only once during migration, and each destination VM could reconstruct its memory from multiple source VMs to make its migration process faster. The total amount of data to be transfered could also be reduced.

Redundant computation Redundant computation and process replication [90, 32] are employed to enhance the availability and reliability of high performance computing and mission critical systems. In these systems, a process's state is replicated and stored in its

partner nodes. If the process fails, these available replicas can recover and assume the original process's role quickly. Process replication offers a different set of tradeoffs compared to rollback recovery techniques. It completely masks a large percentage of system faults, preventing them from causing application failures without the need for rollback. However, process replication can be costly in terms of the large amount of extra memory that is needed, which is a large budgetary and power consumption item. By leveraging memory content sharing, there is a potential to reduce these costs by avoiding explicitly creating memory page replicas when memory pages with the same content already exist elsewhere. That is, we can potentially use the applications' own redundancy instead of making more.

All of these services can be enabled only if there is a facility to dynamically track the sharing of memory content in the system. In addition, I argue that such memory content sharing detection and tracking should be factored into a separate service, i.e, a facility that continuously tracks memory content sharing across the machines should be included as a part of infrastructure in a large parallel system. All these content-aware services, including memory deduplication should be built on top of this facility.

Factoring content sharing tracking into a separate service and facility helps to reduce the duplicated efforts of implementing such a sharing detection and tracking component in each of these services. In addition, a facility with efficient content sharing tracking enables developers to simplify or improve current services by exploiting memory content sharing without efforts to integrate content sharing detection into its service. Finally, factoring content-sharing tracking out as a service would allow us to focus on making the content sharing detection and tracking service itself more effective and efficient.

To show the feasibility of building such facility in large scale systems, I have designed and implemented ConCORD, a system that can detect and track inter-node memory content sharing in large-scale parallel systems.

1.2 ConCORD

To demonstrate that tracking, identifying and exploiting inter-node memory content redundancy in large scale parallel systems is possible with low performance impact, I have designed and implemented *ConCORD*, a distributed system that is able to scalably measure the amount of both intra-node and *inter-node* memory content sharing and identify the specific shared content regions in large-scale parallel systems with minimal performance overheads.

ConCORD is designed to fit into large-scale parallel systems, which targets to scale from small clusters with hundreds of nodes to large-scale parallel systems with hundreds of thousands of nodes. In addition, ConCORD works effectively and maintains low overheads as the system size grows. ConCORD is a distributed system, all of its running components can be distributed and launched on any available nodes in the system. This provides better scalability then centralized model, which would prevent the system from being scalable.

ConCORD serves as a base system that makes other content-aware services able to be built on top of it and effectively exploit content sharing to improve their performance. To better enable this, ConCORD exposes two simple, yet powerful interfaces to allow users and service applications to build their own content-aware services on top of ConCORD. These include the content-sharing query interface to examine the memory content sharing information, and the content-aware service command to enable easy building of content-aware services with minimal effort.

1.2.1 Content-sharing query interface

ConCORD provide a query interface for other service applications or users to check the amount of memory content sharing and examine the location of shared regions across a group of nodes. This memory content information exposed by ConCORD generally

includes:

- *The degree of memory content sharing*: The degree of memory content sharing reflect a rough information on amount of content sharing existing in the given set of nodes. It is defined as the ratio of number of unique memory blocks over all memory blocks for a group of nodes.

- *The number and locations of a certain block of content*: Given a memory block with specific content, ConCORD can also find all of the blocks existing in a set of node containing the same content. It gives the total number of such blocks, and locations of them.

- *Number and list of memory content blocks with a certain number of copies*: Finally, ConCORD can also identify memory content that is ”hot” among a set of nodes. It can identify all unique block of contents that are shared among a certain number of nodes.

Based on this content sharing information from ConCORD, many services can be built to effectively exploit the memory content sharing exising in their target nodes to improve themselves.

1.2.2 Content-aware service command

To further lower the barrier to build a content-aware service on top of ConCORD, I have proposed the content-aware service command model and incorporated it into ConCORD. Content-aware service command is a distributed concurrent service model and associated implementation that enables effectively building of content-aware services in large-scale parallel systems. It provides a powerful interface that enables many HPC services to effectively exploit and utilize memory content redundancy existing in the system with very

little effort. Content-aware services built on top of service commands are automatically parallelized and executed on the parallel systems. The service command execution system takes care of the details of partitioning of the task, scheduling subtasks to execute across a set of machines, and managing all of inter-node communication. This enables service developers to build and enhance their services to maximally exploit and utilize the memory content sharing without worrying about the complication of the implementation.

A content-aware service command enables the collective and parallel execution of a specified service, by disassembling the service into a series of certain operations and applying them on each of memory block with distinct content among all serviced nodes or virtual machines. For example, checkpointing of a machine can be completed by copying each memory page that contains distinct content in the machine's memory to some reliable storage. Migration of a virtual machine can be reduced to be transferring each memory block containing distinct content to the VM's new host.

1.2.3 ConCORD in virtualized environments

ConCORD can be deployed into either non-virtualized environments, in which all applications are running on native operating systems, or virtualized environments, in which all users applications are running inside virtual machines (VMs),

Most of ConCORD's components are designed to be agnostic to both environments, with the memory update monitor being the only exception. The memory update monitor is a component of ConCORD that collects and continuously tracks memory content in individual nodes being traced. It is deployed inside native operating system kernel in non-virtualized environments, and deployed inside VMM in virtualized environments. Dependant on where the memory update monitor is deployed, the actual mechanism to track memory updates is different. In my dissertation, I focus on building and applying ConCORD on virtualized environments, thus a memory update monitor is implemented inside

a VMM.

1.3 Virtualization in HPC

Virtualization has become exceedingly popular within enterprise and large scale data center environments. Virtual machines provide a layer of abstraction between a computer's physical hardware and an operating system (OS) running on top of it. This abstraction layer allows a Virtual Machine Monitor (VMM) to encapsulate and manage an entire operating system environment. Virtualization makes service isolation easier for a large scale data center, and allows more services to be consolidated on fewer physical servers while maintaining the service isolation. This allows data centers to reduce cost by reducing the number of physical machines substantially. In addition, virtualization enhances fault tolerance in data centers by separating the software services from the hardware. Virtualization allows any physical machine to run any software service without reconfiguring or reinstalling. This allows hot spare machines to be dramatically scaled back, as there is no longer a need to keep a 1:1 ratio between live machines and preconfigured replicas.

Many of the motivations for virtualization in data centers apply equally to HPC systems, for example, virtualization allows users to customize their OS environment (e.g. between full-featured OSes and lightweight OSes). and consolidate less-demanding users when appropriate. In addition, virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems by maximizing system flexibility and utility to a wide range of users [37, 52, 92].

Virtualization can improve the usability of large scale parallel systems. Since full system virtualization provides full compatibility at the hardware level, allowing existing unmodified applications and OSes to run. The machine is thus immediately available to be used by any legacy applications, increasing system utilization when ported application jobs

are not available. In addition, virtualization also provides new opportunities for fault tolerance, a critical area that is receiving more attention as the scale of supercomputer keeps increasing. Virtual machines can be used to implement full system checkpoint procedures, where an entire guest environment is automatically check-pointed at given intervals. This would allow the centralized implementation of a feature that is currently the responsibility of each individual application developer. Migration is another feature that can be leveraged to increase HPC system resiliency. If hardware failures can be detected and predicted, the software running on the failing node could be preemptively migrated to a more stable node. Finally, virtualization makes system management easier. Full system virtualization would allow a site to dynamically configure nodes to run different types of OSes without requiring rebooting the whole machine on a per-job basis.

However, HPC systems are naturally different from enterprise environments. First of all, HPC is generally characterized as an area of computing that runs highly tuned applications at extremely large scales, involving thousands of nodes. Additionally, the applications are typically tightly coupled and communication intensive, placing an overriding focus on performance. This is because even small performance losses can have dramatic multiplying effects on large scale tightly integrated systems. The adoption of virtualization in HPC systems can only occur if the performance overheads are truly minimal and do not compound as the system and its applications scale up.

Existing VMMs have been developed with a business centric and are primarily designed for enterprise environments. While this is a sensible approach, it is no longer suitable for use in HPC and other specialized environments. To address the lack of research into the area of HPC virtualization architectures, we have developed the Palacios Virtual Machine Monitor, which will be introduced in the following paragraphs.

1.3.1 Palacios

Palacios is an OS independent embeddable VMM specifically designed for HPC environments as part of the V3VEE project (`http://v3vee.org`), a collaborative community resource development project involving Northwestern University, the University of New Mexico, the University of Pittsburgh, Sandia National Laboratories and Oak Ridge National Laboratory to which I am a major contributor. Palacios is designed to provide a flexible VMM that can be used in many diverse environments, while providing specific support for HPC.

Currently, Palacios targets the x86 and x86_64 architectures (both hosts and guests) and is compatible with both the AMD SVM [6] and Intel VT [55] extensions. Palacios supports both 32 and 64 bit host OSes as well as 32 and 64bit guest OSes. Palacios implements full hardware virtualization while providing targeted paravirtualized extension. Palacios has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as Red Storm development cages consisting of Cray XT nodes. Palacios also suppports the virtualization of a diverse set of guest OS environments, including commodity Linux and other OS distributions, modern Linux kernels, and several lightweight HPC OSes.

Part of the motivation behind Palacioss design is that it be well suited for high performance computing environments, both on the small scale (e.g., multicores) and large scale parallel machines. Several aspects of Palacioss design that are suited for HPC includes:

- Minimalist interface: Palacios does not require extensive host OS features, which allows it to be easily embedded into even small kernels, such as Kitten and other light-weight HPC OSes.

- Full system virtualization: Palacios does not require guest OS changes. This allows it to run existing kernels without any porting, including Linux kernels and whole

distributions, and lightweight kernels like Kitten, Catamount, Cray CNL and IBMs CNK.

- Contiguous memory preallocation: Palacios preallocates guest memory as a physically contiguous region. This vastly simplifies the virtualized memory implementation, and provides deterministic performance for most memory operations.

- Pass-through resources and resource partitioning: Palacios allows host resources to be easily mapped directly into a guest environment. This allows a guest to use high performance devices, with existing device drivers, with no virtualization overhead.

- Low noise: Palacios minimizes the amount of OS noise injected by the VMM layer. Palacios makes no use of internal timers, nor does it accumulate deferred work.

- Extensive compile time configurability: Palacios can be configured with a minimum set of required features to produce a highly optimized VMM for specific environments. This allows lightweight kernels to include only the features that are necessary and remove any overhead that is not specifically needed.

Overall, Palacios provides a flexible, high performance virtualized system software platform for HPC systems. It has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as the Red Storm and Cray XK supercomputers.

In my dissertation, I mainly focus on implementing ConCORD within virtualized environments that target high performance computing systems. Although it is mostly agnostic to the type of VMM, for my implementation and evaluation presented in this dissertation, ConCORD has been built and linked with Palacios VMM.

1.4 Contributions

The contributions of my dissertation can be summarized as follows:

- I have articulated the benefits of including a facility that continuously tracks memory content sharing across the machine as a part of infrastructure for large-scale HPC systems.

- To demonstrate the opportunity of exploiting inter-node memory content sharing in HPC systems. I have presented a detailed experimental study to show that both intra-node and inter-node memory content sharing is common in parallel scientific workloads.

- By presenting my design, implementation and evaluation of ConCORD, I have demonstrated that scalably determining and tracking inter-node memory content sharing is possible with minimal performance impact.

- I have proposed the content-aware service command which enables building of various HPC services to effectively exploit memory content redundancy in large-scale systems with minimal effort. I have implemented the content-aware service command execution system and integrated it into ConCORD.

- I have proposed and built a content-aware checkpoint service based on the content-aware service command to demonstrate the power of my model.

1.5 Outline of dissertation

The remainder of my dissertation is organized as follows.

Chapter 2 presents my experimental study about memory content sharing in scientific workloads. I have conducted a detailed experimental study to examine the memory content redundancy on a range of parallel applications and application benchmarks. The experiment study shows that both intra- and inter-node memory content sharing are common

in parallel applications, and that there is substantial additional sharing across nodes beyond that of sharing within individual nodes. This provides a basis for potential efforts to effectively exploit memory content sharing to benefit many services in HPC systems.

Chapter 3 introduces ConCORD, a distributed system to dynamically track inter-node memory content sharing existing in the system. I will give a overview of ConCORD system, including its system architecture and design of its core components in this chapter.

Chapter 4 presents the design, implementation and evaluation of the memory update monitor, a driving component of ConCORD that is implemented inside VMM. To keep track of memory content over the system, ConCORD deploys a memory update monitor in each individual node to collect and continuously track memory content of local VMs.

Chapter 5 describes the design, implementation and evaluation of the distributed memory content tracer, a critical component of ConCORD. The content tracer enables ConCORD to track memory content locations and sharing across VMs dynamically and scalably in large-scale systems.

Chapter 6 introduces the content sharing query interface of ConCORD. The content sharing query interface is a simple interface exposed by ConCORD to allows service applications and users to dynamically examine the amount of memory content sharing and the shared content across a set of nodes.

In Chapter 7, I introduce the concept of the content-aware service command. The content-aware service command provides a powerful model for developers to quickly build content-aware service with minimal effort. I will elaborate on how we can build a service using it, and then present the design and implementation of a service command execution system in ConCORD.

In chapter 8, I propose the design of a content-aware checkpointing service for HPC systems, and demonstrate an approach to building such a service using the content-aware service command. The content-aware checkpointing service is straight-forward to build on

top of ConCORD's content-aware service command model. By evaluating the performance of the content-aware checkpointing, we also examine the effectiveness and efficiency of the content-aware service command framework. In addition, this demonstrating service also serves as a template implementation of HPC services using content-aware service command.

Chapter 9 elaborates on related work in the areas of content-based memory sharing, fault tolerance, virtualization in HPC, virtual machine migration and distributed primitives.

Finally, in Chapter 10, I conclude the dissertation with a summary of contributions and highlight opportunities for additional research.

Appendix A shows the C source code file for implementing content-aware checkpoint service, which comprise totally of only around 220 line of code.

Appendix B presents my works on VNET/P, a fast VMM-based overlay networking for bridging the cloud and high performance computing. VNET/P is a virtual networking system that has been directly embedded into Palacios and can achieve native performance on 1 Gbps Ethernet networks and very high performance on 10 Gbps Ethernet networks. Our work on VNET/P have demonstrated that it is feasible to extend a software-based overlay network into tightly-coupled environments.

Chapter 2

Memory Content Sharing Study

Memory content duplication has been effectively exploited for more than a decade to reduce memory consumption. By consolidating duplicate pages in the address space of an application, it can reduce the amount of memory it consumes without negatively affecting the application's perception of the memory resources available to it. There is considerable work that have been done previously on deduplicating memory content within individual nodes, but little work considers the opportunities of tracking and leveraging inter-node memory content sharing, particularly in a parallel computing context.

In this chapter, I will focus on the question of opportunity: does significant inter-node and intra-node memory content sharing actually exist in parallel workloads? I will describe a detailed experimental study to examine the memory content redundancy on a range of parallel applications and application benchmarks. The study considers both forms of sharing, at different scales, and across time.

As I will demonstrate in this chapter, both intra- and inter-node memory content sharing are common in parallel applications, especially, there is substantial additional sharing across nodes beyond that of sharing within individual nodes. This provides a basis for potential efforts to effectively exploit memory content sharing to benefit many services in HPC systems. And how to capture this memory content sharing drives my design of

ConCORD system.

In the rest of this chapter, I will describe my approach for collecting and analyzing memory content duplication in HPC applications, the benchmarks I have used and the results of my experimental study, followed by the observations and conclusion.

2.1 Methodology

We have investigated the memory content sharing in scientific applications by examining content sharing using a set of applications and benchmarks. In our experiments, each benchmark are started by MPI with a number of processes across physical nodes, with one or more process running on each physical node. And then all processes' memory contents were periodically examined and compare to each others to find all possible content sharing.

To be more specifically, during the execution of the benchmark, we periodically suspended all of processes synchronously, then dumped the binary memory content for each process to a memory dump file, then resumed all processes.

To identify duplicated content, each dumped memory file from the same suspend-resume cycle was sequentially scanned, an MD5 hash value is generated to describe the content of each memory block. A hash list is thus generated for each process' memory to hold all these different hash values. Hash collisions indicate intra-node sharing inside each process. In addition, the hash lists from each process are then compared to each other to find all inter-node memory content sharing.

The dumped memory of each process includes all memory regions that are mapped and writable in its address space, this includes all of its data, heap, stack and anonymous memory pages. We excludes all unmapped memory regions with possibly all zeros or other unmeaningful contents. Also, the read-only memory regions are excluded from the study, because most of these memory contains application and shared library code which

are obviously duplicated from each other. In our experiments, we are interested more on the type of sharing among the part of memory content that is dynamically changed during execution of the applications, instead of statically shared content.

In our study, we are interested in both intra-node memory content sharing and inter-node memory content sharing. We use intra-node and inter-node distinct memory block ratio to represent the level of memory content sharing. The metrics we use are defined here:

- *Total memory blocks* shows the total number of memory blocks of all processes.
- *Intra-node distinct blocks* gives the summation of the number of distinct memory blocks in each process.
- *Inter-node distinct blocks* gives the number of distinct memory blocks across all processes.
- *Intra-node distinct ratio* represents the ratio of *intra-node distinct blocks* over *total memory blocks*, The smaller of this ratio, the more intra-node memory content sharing. This ratio is labeled as *intra%* in the following graphs.
- *Inter-node distinct ratio* represents the level of both intra-node and inter-node content sharing, which is the ratio of *inter-node distinct blocks* over *total memory blocks*. The smaller of this ratio, the more memory content sharing exists across all nodes. This ratio is labeled as *inter%* in the graphs.

We report mainly the *intra-node* and *inter-node* distinct ratios of the tested parallel applications and benchmarks here. All the results are averaged from these numbers that are got in each suspend-resume cycles during the entire application execution. The memory block size we use is a single x86 page (4096 bytes), unless otherwise specified. The

suspend-resume cycle interval is set to 5 seconds, i.e, suspend-resume cycle happened every 5 seconds in our experiments.

Note that our measurement study is done using only writable memory regions of the application portion of the address space. If we considered read-only regions of the application memory and also the kernel memory contents, there will be more memory content sharing both inside and across nodes. Our measurement *underestimates* the amount of both kinds of memory content sharing that is available. That is, if we considered kernel memory, the opportunity would grow.

We have performed all of the tests on a cluster with 12 nodes. Each node is equipped with two Dual Core Intel(R) Xeon(TM) 2.00GHz CPU, 1.5GBytes RAM and 32GB disk, a Broadcom NetXtreme BCM5703X 1Gbps Ethernet card. The nodes are connected through a 1Gbps Ethernet switch.

2.2 Benchmarks

I have run a set of parallel applications and application benchmarks that are designed to run on large parallel systems to study their memory content sharing. These applications include:

- Moldy [105] is a general-purpose molecular dynamics simulation program. It is sufficiently flexible that it ought to be useful for a wide range of simulation calculations of atomic, ionic and molecular systems. We used MPI-based Moldy version in the experimental study.

- The NAS Parallel benchmarks (NPB) [128] is a set of benchmarks targeting performance evaluation of highly parallel supercomputers. In our test, NPB 2.4 version is used, which is MPI-based. 7 benchmarks from NPB are tested, which are BT, EP, LU, SP, CG, IS and MG.

Benchmark	Memory (MB)	Benchmark	Memory (MB)
bt.C.9	4364	cg.C.8	1292
ep.C.8	245	is.C.8	2377
lu.C.8	864	mg.C.8	3567
sp.C.9	2734	Moldy.8	5324
pHPCCG.8	3986	HPCCG.8	3987
miniFE.x.8	4350	miniMD.8	6313
Lammps.4	4265	HPCC.8	2778

Figure 2.1: Memory footprint size of tested applications. This shows that most of tested applications are memory-intensive.

- The HPC Challenge benchmarks (HPCC) [28] is a set of benchmarks targeting to test multiple attributes that can contribute substantially to the real-world performance of HPC systems. Version 1.4.1 is used in the test.

- The Sandia Miniapps [48], part of Sandia National Labs' Mantevo Project, is a set of small, self-contained programs that embody essential performance characteristics of key applications. We used miniFE, HPCCG, pHPCCG and miniMD in our test.

- Lammps [102] is a molecular dynamics simulator and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, which is also distributed by Sandia National Laboratories.

2.3 Results

In this section, I will present the results of my experimental study on the memory content sharing of these parallel workloads as described in last section. First of all, figure 2.1 presents the size of dumped memory files for each workload, which shows that most of the tested application benchmarks have significant and non-trivial memory footprints.

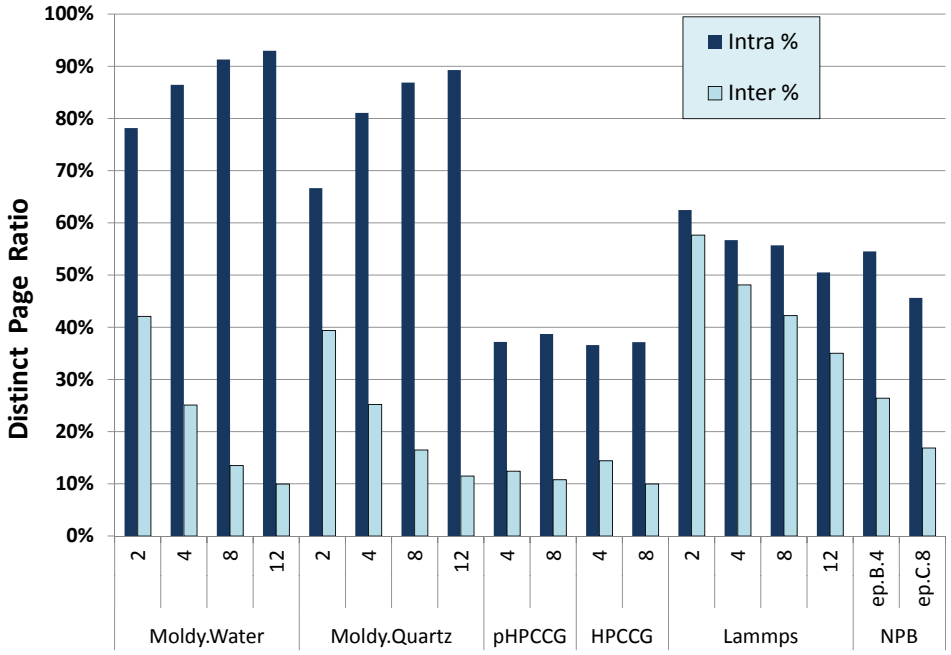


Figure 2.2: Parallel applications that have more inter-node sharing but less intra-node sharing.

The intra-node and inter-node distinct ratio for all of the applications and benchmarks are shown in Figures 2.2 (where inter-node sharing is dominant) and 2.3 (where intra-node sharing is dominant). Each application and benchmark is run with at least two different problem sizes, on different numbers of nodes, where both are encoded in the name. For example, *bt.C.9* means the bt benchmark with problem size C on 9 nodes.

The most important observation on Figures 2.2 and 2.3 is that memory content sharing of some form is common. This is the opportunity that I seek to take advantage of. Some VMM systems already employ memory-deduplication techniques to reduce memory pressure on a single node. In my thesis, I was hopeful that further deduplication is possible across nodes, and here I have found the evidence to support that.

Figure 2.2 shows the applications that have a significant amount of inter-node memory

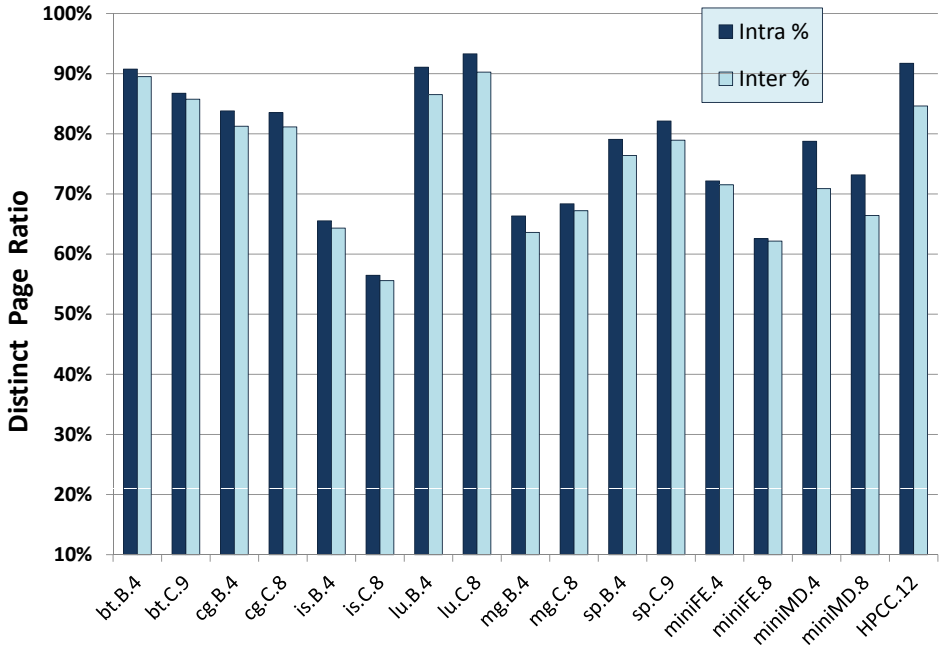


Figure 2.3: Parallel applications which have more intra-node sharing but less inter-node sharing.

content sharing. For example, Moldy can have up to a 10% inter-node distinct ratio, comparing to only a 78% intra-node distinct ratio. Also note that its inter-node distinct ratio keeps decreasing as the problem size and number of nodes increases.

If significant inter-node sharing exists, it could potentially be used to reduce the actual memory footprint by deduplicating pages based on this sharing. Figure 2.4 illustrates the potential memory footprint reductions for the Moldy application as a function of the problem size. The last column in the figure shows the size of memory the system could saved if all inter-node duplicated memory contents are removed. We can see for this application, (a) capturing inter-node sharing is critical, and (b) the size of saved space scales well with problem size. This behavior also occurs for the benchmarks of Figure 2.2.

Figure 2.3 shows applications that have some degree of intra-node content sharing but

Problem Size	Number of Nodes	Total (MB)	Intra-Distinct (MB)	Inter-Distinct (MB)	Reduction (Intra) (MB)	Reduction (Inter) (MB)
$2^7 \times 2^7 \times 2^8$	2	29	19	11	10 (34%)	18 (62%)
$2^8 \times 2^8 \times 2^8$	4	161	131	41	30 (19%)	121 (75%)
$2^9 \times 2^9 \times 2^8$	6	489	417	91	72 (15%)	398 (81%)
$2^{10} \times 2^{10} \times 2^8$	8	1337	1161	220	176 (13%)	1116 (83%)
$2^{11} \times 2^{11} \times 2^8$	10	3057	2706	426	351 (11%)	2631 (86%)
$2^{12} \times 2^{12} \times 2^8$	12	5324	4753	612	571 (11%)	4713 (89%)

Figure 2.4: Memory that could potentially be reduced when memory contents are de-duplicated using intra- and inter-node content sharing in the Moldy application (with quartz as input).

only very little inter-node sharing. A natural question to ask is whether the latter is due to the block size in use. Will a finer granularity expose more sharing? Figure 2.5 shows results in which the block size was varied down to 128 bytes. At least down to this level, granularity has little effect. This is unfortunate, but there is also a bright side to these results (and the results for the benchmarks that do have significant inter-node sharing): they suggest that page granularity is sufficient. This is important because I expected to implement the ConCORD system in a virtualization context, where the page granularity is readily accessible, including with hardware assistance, while sub-page granularity is much more challenging to handle.

For some benchmarks, such as HPCC, miniFE and miniMD, the memory tracing results show that on average there are some but not much memory content sharing. However, if we look at sharing over time during execution, we find that there are phases in which the inter-node memory content sharing is considerable, as can be seen in Figure 2.6. This suggests that a facility for tracking memory content sharing must act dynamically.

Finally, we consider the modification behavior of the memory used by the tested applications. Figure 2.7 shows how frequently the memory content changes, i.e, the percentage

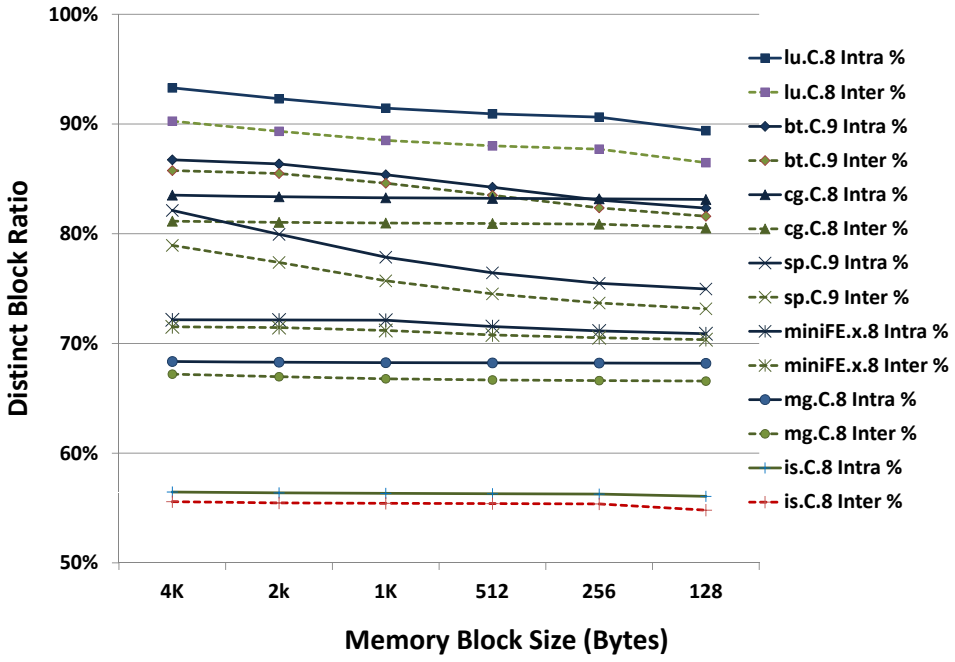


Figure 2.5: Memory content sharing using various memory block sizes.

of memory pages that have been modified in between each scan round, for various workloads with scan interval set to 2 seconds and 5 seconds. From the results, we can see that memory content update rates of different workloads varies a lot from each other, ranging from 65% to less than 5% for burst updates, and from 27% to less than 1% on average. This suggests that memory content in applications could change frequently, which requires that a facility must be able to track memory content updates to follow content sharing in the system.

2.4 Summary

In this chapter, I have presented the results of a detailed study of the memory content sharing of a range of parallel applications and application benchmarks. The study considers

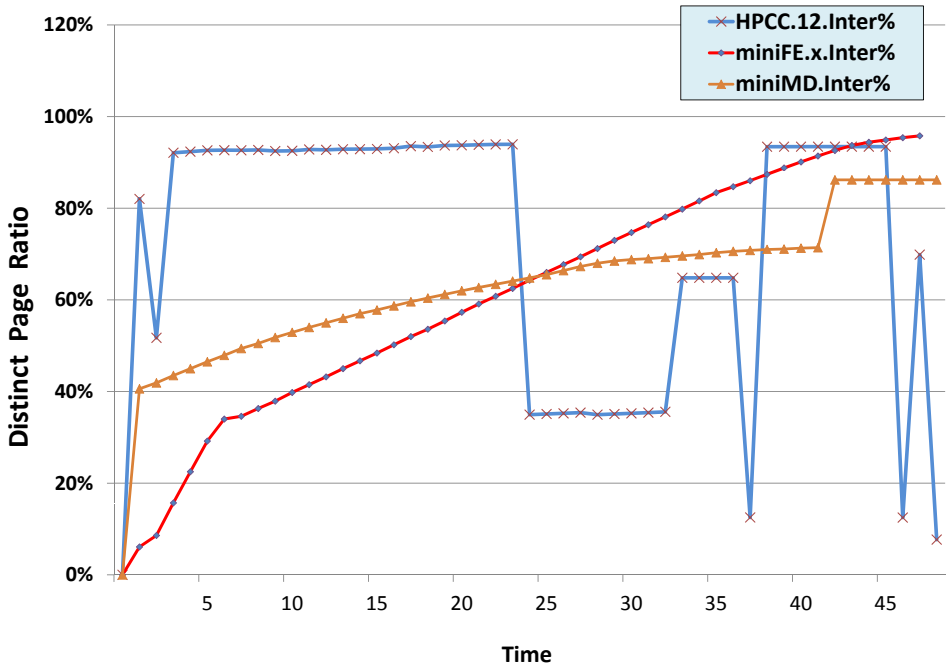


Figure 2.6: Applications which have different level of inter-node content sharing over the execution time.

both forms of inter-node and intra-node memory content sharing actually exist in parallel workloads, at different scales, and across time. Based on the results and analysis I have presented above, we can draw our key observations as following:

- The key observation from my experimental study is that intra- and inter-node memory content sharing is common in parallel applications. This suggests that there is opportunity for exploiting this memory content sharing to benefit many services in HPC systems.

- For some applications, there is substantial content sharing across nodes beyond that of sharing within individual nodes. This suggests that most of previous works on expoliting memory duplication on individual node is not sufficient. A system with

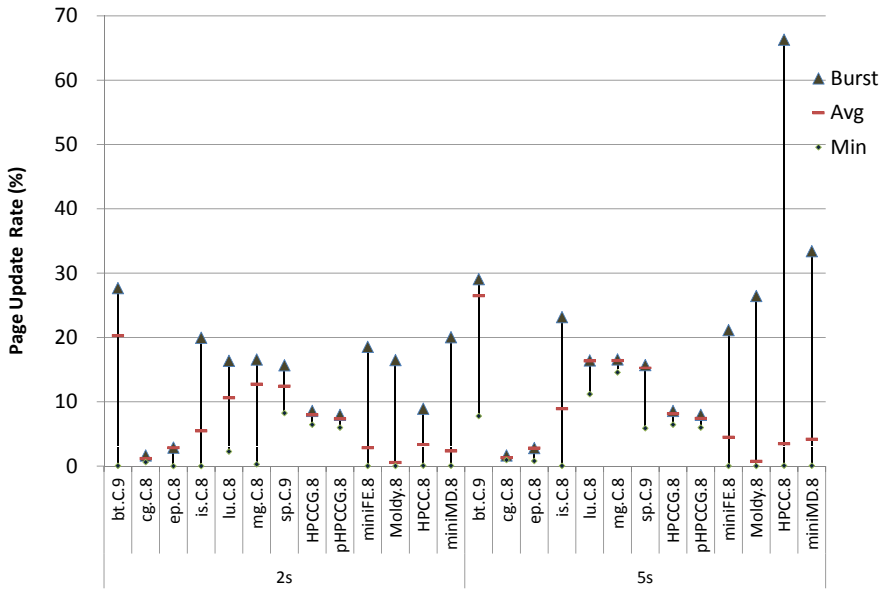


Figure 2.7: Memory update rate for each scan round for different workloads using 2s and 5s scan intervals.

ability to track both intra- and inter-node memory content sharing can bring more opportunities to benefit system services in HPC systems.

- Some applications update their memory content frequently, and show different levels of memory content sharing over different time of execution, this suggests dynamic tracking of memory content sharing is desirable.

- Also the results suggest that block granularity has little effect on exploiting more content sharing. This suggests that page granularity is sufficient.

Based on my study in this chapter, I have demonstrated that there is opportunity for exploiting inter-node memory content sharing to benefit many areas in high performance

computing and distributed systems. Hence, the further question would be how we can identify and track these memory content sharing across large number of nodes in an efficient and low overhead way.

Chapter 3

Overview of ConCORD

ConCORD is a distributed system for tracking and exloiting both intra-node and inter-node memory content sharing in large parallel systems. It is an online system that dynamically tracks the memory content sharing and the shared content with its locations existing in the system as it is running. It reports the current, or near-current memory content sharing in the system as the application changes its memory contents during its execution.

ConCORD exposes two simple yet powerful interfaces to allow users and service application programmers in the HPC community to build their own content-aware services on top of ConCORD. This includes a content-sharing query interface to examine the memory content sharing information, and a content-aware service command to enable easy building of content-aware services with minimal effort.

ConCORD is designed to fit into large-scale parallel systems. It has been desired to scale from small clusters with hundreds of nodes to large-scale parallel systems with hundreds of thousands of nodes. In addition, ConCORD works effectively and maintains low overheads as the system size grows. ConCORD is a distributed system, all of its running components can be distributed and launched on any available nodes in the system. This provides better scalability then a centralized model, which would prevent the system from being scalable. In addition, ConCORD is tolerant to communication and node failure.

ConCORD is a core contribution of my thesis. In this chapter, I will introduce ConCORD with its high level system overview, the major architectural components, and a brief discussion on its implementation. The details about the design and implementation of each of its core components are described in the following chapters.

3.1 System overview

In ConCORD, a node's memory is logically split into a sequence of blocks, called *memory blocks*. A fingerprint can be generated for each block based on its content, for example, by a hash function. This fingerprint, what we called a *memory content hash*, or simply *content hash*, is used in ConCORD to represent the content of a memory block. Any two memory blocks with the same content hash are considered identical to each other.

3.1.1 Hash-based content detection

Since ConCORD uses a hash value to represent the content of a memory block, there is always the possibility of collision, which means the different memory blocks are mapped to the same hash value by the system. However, the possibility of collision in most of widely used hash functions is considered very low. For example, MD5 generates 128bit hash value. If we consider a system with 1 Petabyte (2^{50} bytes) of 4KB memory pages hashed by ConCORD using the MD5 hash function, the collision probability is around 10^{-14}. This is considered to be less than the probability of physical memory corruption.

Generally, the size of the memory block in ConCORD is configurable, however, we use page granularity, i.e, 4KB page in x86 machine for easy implementation. Detecting memory content sharing with another block size is possible, but the smaller the memory block, the more extra CPU, memory and communication overhead is needed. In addition, from Figure 2.5, we can see that reducing the size of memory blocks does not increase

the detected amount of memory content sharing for most of applications. Given these concerns, ConCORD sticks with using 4KB page as the default memory block size.

Although currently ConCORD is designed to track identical memory blocks over VMs, it could be extended to identify those memory blocks that share mostly the same content. If that is a common case in some parallel workloads, then detecting and identifying these similarities could benefit many HPC services too, for example, the applications would only have to transfer the delta difference in content among memory blocks while they are doing checkpointing or migration.

Edit distance could be one way to define the similarity between two memory blocks. Two memory blocks with small edit distance could be defined as similar pages. Locality preserving hashing [17] could be applied as a fast approach to determine similarity between two memory blocks. Locality preserving hashing is the hash function where the relative distance between the input values is preserved in the relative distance between the output hash values. An alternative approach to detecting similar memory blocks is by randomly choosing a number of small parts of the content in the block and comparing them. If all of these parts are same, then these two pages are treated as similar to each other. This approach is proposed in Difference Engine [45].

3.1.2 Target systems

ConCORD is designed and targeted for large-scale parallel systems. There are several assumptions we have made on these systems. First, we assume that each node in the system is an independent failure boundary, which means that replicating computation and data can provide fault tolerance. Second, we assume that the network links between nodes have high throughput and low latency, which is very common for the network or interconnect between nodes in supercomputers, typically these local area networks that can achieve at least 1Gbps throughput with 10 to 100 μs latency. Also, the system supports efficient col-

lective communication, and the network communication bandwidth/latency is scaled as the system size increases. In addition, we assume that these systems are physically secure and well-administered. Overall, the system represents a carefully controlled environment with high throughput and low latency network connected among nodes and very few security concerns.

These assumptions help with the design of ConCORD. For example, the low latency and high bandwidth scalable interconnect means that the synchronization between all ConCORD nodes is not prohibitively expensive. Independent failure means we can an rely on data being available in more than one failure boundary (i.e., the physical memory of more than one node) while designing the recovery protocols.

3.1.3 Content-sharing query interface

ConCORD exposes two types of interface to allow users and service applications in HPC community to build their own content-aware services on top of ConCORD. These include: 1) a content-sharing query interface to examine the memory content sharing information, and 2) a content-aware service command to enable easy building of content-aware service with minimal effort.

ConCORD's content-sharing query interface allows services to check the amount of memory content sharing and examine the location of shared regions existing in the system that are perceivable to ConCORD. This content-sharing information includes:

- *The degree of memory content sharing (***DoS***)*: DoS is defined as the ratio of the number of unique memory blocks over the number of total memory blocks for a given group of nodes. It gives a rough information on amount of content sharing currently existing in the this group of nodes.

- *The number and locations of a memory block (***NoC/LoC***)*: Given a memory block

and a set of nodes, ConCORD can give the number and the list of nodes that have a copy of the data in the given memory block.

- *Number and list of memory blocks with a certain number of copies (***NoM/LoM***)*: NoM provides the number of memory blocks that appear at least in the given number of nodes among the defined set of nodes. NoM provides the number of hot memory blocks in the node set. Meanwhile, LoM returns a list of content hashes of such hot memory blocks.

Based on this set of queries, service applications can leverage the content sharing information, and actual shared memory regions to build their own services. In Chapter 6, I will present more details on the implementation of the query interface, optimizations for different types of queries, and a detailed evaluation of the query interface.

To further remove the barrier to building an HPC service that can efficiently exploit memory content sharing in the system, I have proposed the *content-aware service command* and integrated it in ConCORD.

3.1.4 Content-aware service command

The content-aware service command model is proposed to minimize the efforts spent on building a content-aware service on top of ConCORD. The content-aware service command is a service model and associated implementation that enables effectively building content-aware services in large-scale parallel systems. It provides a simple and powerful interface that enables many HPC services to effectively exploit and utilize memory content redundancy existing in the system. Content-aware services built on top of service commands are automatically parallelized and executed on the parallel systems. The service command execution system takes care of the details of the partition of the large task, scheduling subtasks to execute across a set of machines, and managing all of inter-node

communication. This enables service application programmers to build and enhance their services to maximally exploit and utilize the memory content sharing without worrying about the complication of the implementation.

A content-aware service command enables a collective and parallel execution of a specified service, by disassembling the service into a series of certain operations and applying them on each of memory block with distinct content among all serviced nodes. For example, checkpointing of a machine can be completed by copying each memory page that contains distinct content in the machine's memory into some reliable storage. Migration of a virtual machine can be reduced to transferring memory block containing distinct content to the VM's new host.

A content-aware service command includes a service-specified set of arguments, such as nodes involved or participant during the execution of this service, etc. In addition, a set of service operations (call-back functions) is specified to actual perform the service-specific task on each individual memory block with distinct content across the defined set of virtual nodes. This set of functions will be called by ConCORD, the service command run-time system, during distributed parallel execution of the command on some or all participating machines.

A content-aware service command can be executed in one of two operation modes: *interactive mode* and *batch mode*. In an interactive mode, all service-specific operations will be applied during the execution of the command, i.e, the service defined task is actually completed when the command is finished executing. On the other hand, in a batch mode, instead of applying the service-specified operations during the execution of the command, a plan or schedule will be generated when the command completes. This plan will be used later as a roadmap to direct each local node and VMs on how to apply these operations to complete the service defined job.

More details on the content-aware service command and its implementation associated

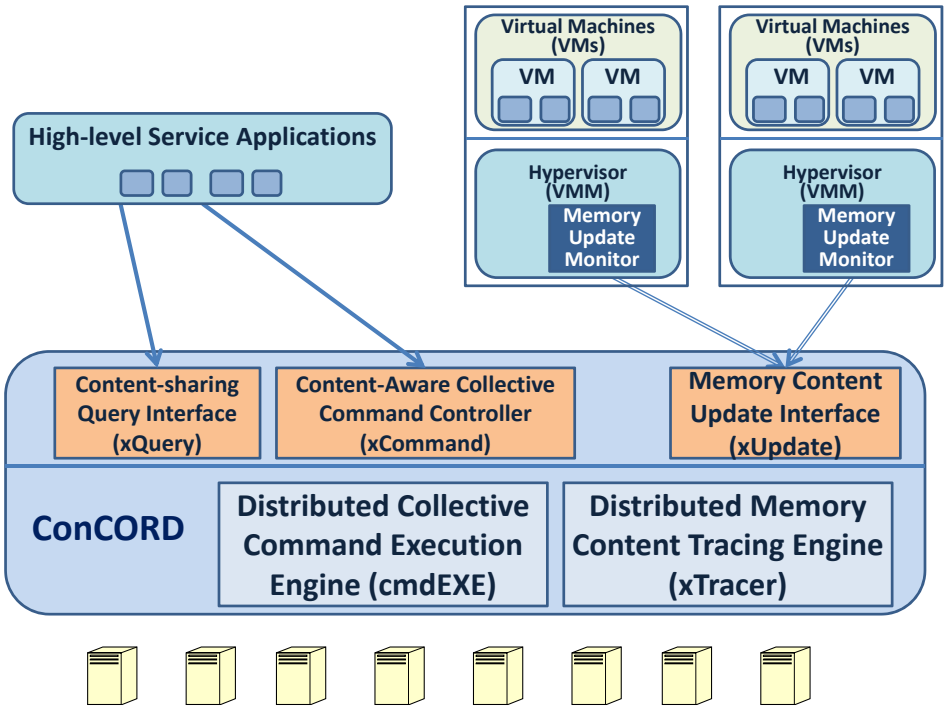


Figure 3.1: ConCORD overall architecture within virtualized environment

with ConCORD will be illustrated in Chapter 7.

3.1.5 System architecture

ConCORD can be deployed into either non-virtualized environments, in which all applications are running on native operating systems, or virtualized environments, in which all users' applications are running inside virtual machines (VMs),

In non-virtualized environments, the clients of the ConCORD could be a user-level service, such as checkpoint manager that performs a process-initiated checkpointing, or kernel checkpoint components that manage system checkpoints. The scientific applica-

tion developers can leverage the memory content sharing knowledge in system provided by ConCORD to reduce the time and storage space that are needed to checkpoint their executing processes, or enhance application availability through application-level memory content replication. Similarly, the kernel components could leverage the content-sharing to provide higher availability for the machine and its applications with less overhead, e.g., through system-initiated checkpointing and memory state replication. These components could gain the knowledge of intra-node/inter-node memory content sharing in the system through the *Content-sharing Query Interface* of the detection system. Similarly in virtualized environments, virtual machine monitors (VMMs) could leverage knowledge of inter-node memory content sharing among the virtual machines to enable more efficient virtual machines co-migration, group checkpointing, and etc.

Most of ConCORD's components are designed to be agnostic to both environments, i.e, most parts of ConCORD function exactly the same except the memory update monitor. The memory update monitor is deployed on each physical node inside OS kernel in the non-virtualized case, and inside the VMM in the virtualized case. Depending on where the memory update monitor is deployed, the approach to implement it may be different. In my dissertation, I focus on building and applying ConCORD on virtualized environments, thus a memory update monitor implemented inside a VMM is demonstrated in my thesis.

The high level architectures of ConCORD within virtualized environments is illustrated in Figure 3.1. The core functionality of ConCORD is to track memory contents sharing across virtual machines (VMs) in the site. To do so, ConCORD needs to discover which nodes in the network have a copy of a specific memory block. Since these VMs are running and their memory content changes over time, this information has to be discovered dynamically. To solve this problem, ConCORD mainly deploys two subsystems: 1) a site-wide distributed memory content tracer which allows ConCORD to locate VMs which have a copy of a given memory block using its content hash, and 2) a memory update monitor

deployed in each VMM instance on individual nodes to collect memory content and track memory updates from VMs and populate these updated content hashes into distributed content tracer.

Distributed memory content tracer The distributed memory content tracer is a site-wide distributed component that enables ConCORD to locate virtual machines having a copy of a given memory block using its content hash. It also allows ConCORD to find the amount of memory content sharing among machines in system wide. In Chapter 5, I will further elaborate the design, implementation and evaluation of such distributed components. In general, ConCORD employs a customized high performance light-weight distributed hash table (DHT) to store unique memory content hashes and map each content hash to a list of virtual nodes that have a copy of the corresponding memory block.

Memory update monitor A memory update monitor is run within the VMM to collect memory contents and track content updates in local VMs for each individual node. The memory update monitor is the heartbeat of ConCORD that drives ConCORD maintaining all recent memory content over the nodes. The basic mode of operation for the memory update monitor is to periodically step through the full memory of VMs being traced, identifying memory blocks that have been updated recently, and then sending memory content hashes for newly updated blocks to the ConCORD memory tracing engine. In addition, the memory update monitor maintains a local mapping table that allows ConCORD to efficiently locate a memory block's content from its hash. More about the design, implementation and evaluation of the memory update monitor in ConCORD will be discussed in chapter 4.

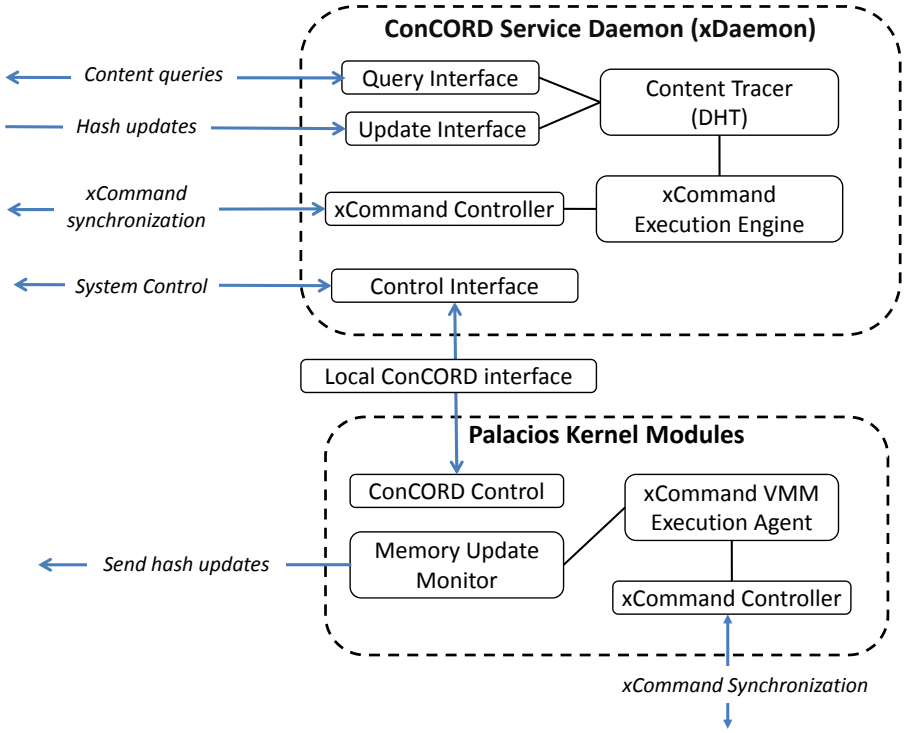


Figure 3.2: Overview of ConCORD's subsystems and components as linked with Palacios VMM

Content-aware service command execution engine ConCORD employs a distributed *content-aware service command execution engine* to scalably execute a collective command. This will be discussed in Chapter 7.

3.2 Implementation

3.2.1 Interface

Content-sharing query library The query interface is implemented as a user library which can be linked into any application program code. The library is responsible for

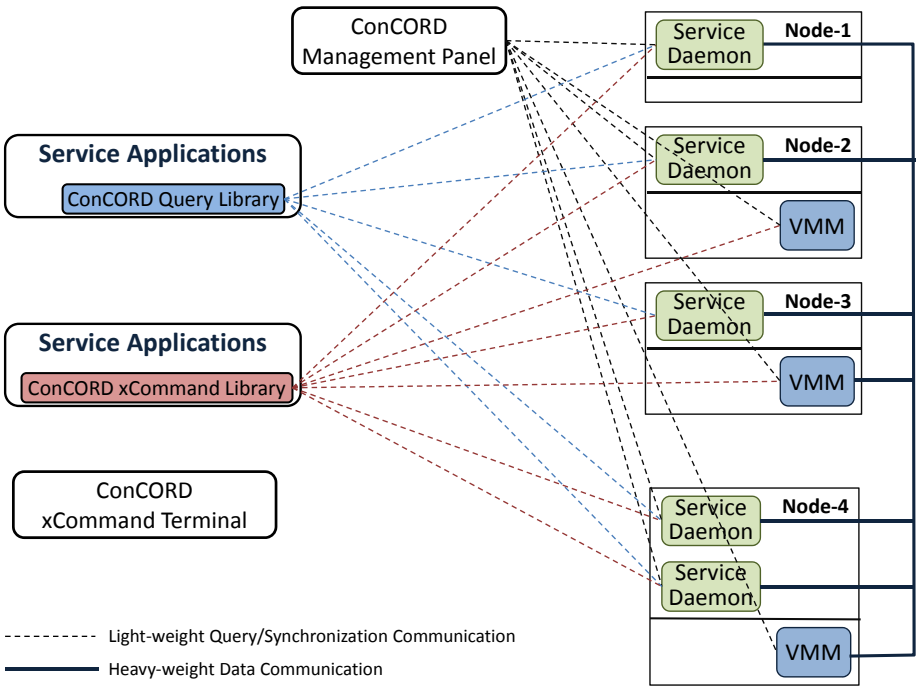


Figure 3.3: Overview of communication among ConCORD's subsystems and components.

communicating with ConCORD to launch a query and collect results. From the perspective of a service, the query interface is no more than standard library, with different types of queries exposed as standard function calls. Currently, the library is linkable to any Linux application, and the interface is provided in C language.

Content-aware service command library One kind of interface to operate service commands in ConCORD is implemented as a user level library. Similar to the content-sharing query library, the library can be linked to any applications. The service command library exposes a few functions to allow applications to create, configure and execute a service

command. The library can be initiated and used on multiple physical nodes inside different applications simultaneously. Each library maintains the status of its local service commands and controls the communication with ConCORD command execution engine independently. This allows multiple services to be started and executed in parallel in the system.

Content-aware service command control terminal In addition to the library which is linkable to a program, a user facility, the service command control terminal, is also provided to allow users to conduct the creation and execution of a service command directly by command line input. The control terminal is a simple command line interface. It allows the users to create, configure, execute and check the status of a command by command line input. The control terminal enables system administrators to directly control the execution of a service command using the command line instead of a piece of code written on top of service command library.

ConCORD management panel ConCORD management panel is a user utility to manage the running of ConCORD. It is able to run on any node. It can be used to control the start up, configuration change, management, and shutdown of ConCORD.

The other functionality of the management panel is maintaining the information of all of ConCORD's running components. At runtime, the ConCORD system is distributed and run as service instances in each individual physical node. The management panel maintains the mapping of these running instances to their physical node's information. In addition, the management panel also manages the mapping of VMs to their physical nodes.

When a client side library, such as the query library or the service command library initiates, it requests this mapping information from the management panel. The mapping of the ConCORD's running service instances to its physical node in ConCORD is relative

stable, because rearrangement of service instances happens very rarely in ConCORD. The node failure does not change mapping, only the permanently removal of a machine or moving of the ConCORD running instances across physical nodes changes this mapping. However, once the mapping is changed, all client libraries are notified.

3.2.2 Running components

Physical node A physical node is an independent physical machine. Each physical node may have several xDaemon instances and multiple VMs running.

Virtual Machine Monitor (VMM) A VMM instance can run in a physical node, with one or more VMs launched and run. Once there is VM running, a memory update monitor implemented inside VMM is started to collect the VM's memory contents and send them to ConCORD's distribute memory content tracer.

Memory update monitor Whenever a VM is first initiated and launched, the memory update monitor scans the entire VM's physical memory and generates a content hash for each memory page. Since all of these content hashes are considered to be new to ConCORD, they are sent to ConCORD's memory content tracer. After this, a memory page can be subsequently written by the VM. Writing changes the page contents and can invalidate an earlier identified instance of content sharing or create a new instance of content sharing. Therefore, the memory update monitor has to track updates to VM's memory, and keeps collecting the new memory contents and sending them to ConCORD.

To track memory updates, the memory update monitor has to periodically identify all memory pages that have been updated recently. One way to do this is by scanning the full memory periodically and comparing the content hash of each page with its old one to determine whether the page has been changed since last scan. This is a simple approach,

and it is easy to implement in Palacios. However, full memory scan could be very costly, since it has to rehash every page during each scan.

ConCORD service daemon (xDaemon) The core functionality of the ConCORD runtime system is split into a number of *ConCORD service daemons*. A service daemon is an instance running as a Linux user level process. It stores the memory content hashes detected and tracked in the memory content tracer (distributed hash table), handles a hash update request from memory update monitor and handles a query request from the client library. In addition, the service command execution engine is running in an xDaemon instance as the core component for handling the content-aware service command.

Each physical node may have one or several ConCORD service daemon instances running on it, which are differentiated by IP address and port. By adjusting the number of xDaemon instances running on each physical node, ConCORD can fit in heterogeneous systems with various computing power and network bandwidth limits

Service command execution engine The logic of the collective command execution engine is split into two components: one running as a part of xDaemon, while the other running inside VMM. As shown in Figure 3.2, the first implementation component is named the *xCommand execution engine*, while the part running in VMM is named the *xCommand VMM execution agent*. In addition, an *xCommand controller* is equipped in both service daemon and VMMs to be responsible for handling and communicating xCommand synchronization messages from and to the client-side service command execution library or control terminal.

3.2.3 Network communications

As shown in Figure 3.3, there are intensive network communications among different components of ConCORD. Generally, there are two types of network communications.

1-n broadcast query/synchronization communication . This is the type of communication between client libraries or the management node and xDaemon and VMM instances, such as communications between ConCORD management node and all service daemons and VMMs, or communications between the client side query library and all xDaemons instances,etc. This type of communication usually happens less frequently, only during query time or command execution time. Also the data size communicated is relatively small, thus requiring less bandwidth. Finally, it usually requires reliable communication, e.g, silence on lost messages is not acceptable.

Peer-to-peer data communication This is the type of communication between xDaemons and VMM instances. For example, content hash updates from a memory update monitor to an xDaemon instance, or content hash exchanges among xDaemon and VMM instances during the execution of a content-aware service command. This type of communication usually happens very frequently. The data size communicated usually is relative large, thus requiring high bandwidth. In addition, the latency on this communication is critical to system performance. Finally, it does not require reliable communication, e.g, the loss of small amounts of data over the network is tolerable.

Lightweight communication ConCORD uses UDP for network communications among all of its components, and between query clients. UDP is better scalable in large-scale systems than TCP, which needs a connection between every pair of communication peers. Based on UDP, I have implemented two types of communication: unreliable UDP and re-

liable UDP. Unreliable UDP messaging is just the standard UDP message, which means that each single UDP packet could get lost during transmission, and the order of UDP packets being received is not necessarily in the order they are sent. Reliable UDP in ConCORD means every single UDP message has been acknowledged by receiver, so sender can assure that the message has being correctly received. This acknowledge-based UDP enhancement assures the message has been received, but not necessarily that is intact, i.e. the received UDP message may be corrupted during transmission. However, since there are many mechanisms applied at hardware level and Ethernet/IP protocol to check the integrity of a network packet, we assume that the chance of receiving a corrupted UDP message in a parallel system connected by a modern network is extremely small. In other word, ConCORD assumes that every UDP message received is exactly the same as it being sent.

We anticipate that UDP's advantages will become more prevalent with even larger scales as connectionless communication protocols will be preferred to avoid having expensive connection establishments among a large number of nodes.

3.3 Palacios

As we have seen, ConCORD is able to work with both native case in which applications are running on native OS in physical node, or virtualized environments, in which scientific applications are running inside virtual node provided by virtual machine monitor (VMM). For my dissertation, I focus on implementation of ConCORD on virtualized environment. In this case, ConCORD has to cooperate with the VMM to read memory content from its virtual machines (VMs). Most of ConCORD components are independent of the type of VMM being used, the only exception is memory update monitor, which has to work with the VMM to collect and track memory content of the VMs. In my implementation,

I have linked ConCORD with the Palacios VMM, although it is pretty straightforward to link ConCORD to any other VMM.

Palacios is an open source VMM developed from scratch as part of the V3VEE project. At a high level Palacios is designed to be an OS independent, embeddable VMM that is widely compatible with existing OS architectures. In other words, Palacios is not an operating system, nor does it depend on any more specific OS. This OS agnostic approach allows Palacios to be embedded into a wide range of different OS architectures, each of which can target their own specific environment (for instance 32 or 64 bit operating modes). Palacios is intentionally designed to maintain the separation between the VMM and OS. In accordance with this, Palacios relies on the host OS for such things as scheduling and process/thread management, memory management and physical device drivers. This allows OS designers to control and use Palacios in whatever ways are most suitable to their architecture. Palacios is also designed to be as compact as possible, with a simple and clear code base that is easy to understand, modify and extend.

Palacios is an OS independent VMM, and as such is designed to be easily portable to diverse host operating systems. Palacios integrates with a host OS through a minimal and explicitly defined functional interface that the host OS is responsible for supporting. Furthermore, the interface is modularized so that a host environment can decide its own level of support and integration. Palacios is designed to be internally modular and extensible and provides common interfaces for registering event handler for common operations. Figure 3.4 illustrates the architecture of Palacios.

The Palacios implementation relies entirely on the virtualization extensions deployed in current generation x86 processors, specifically AMDs SVM [2] and Intels VT [38, 100]. A result of this is that Palacios only supports both host and guest environments that target the x86 hardware platform. However, while the low level implementation is constrained, the high level architecture is not, and can be easily adapted to other architectures with or

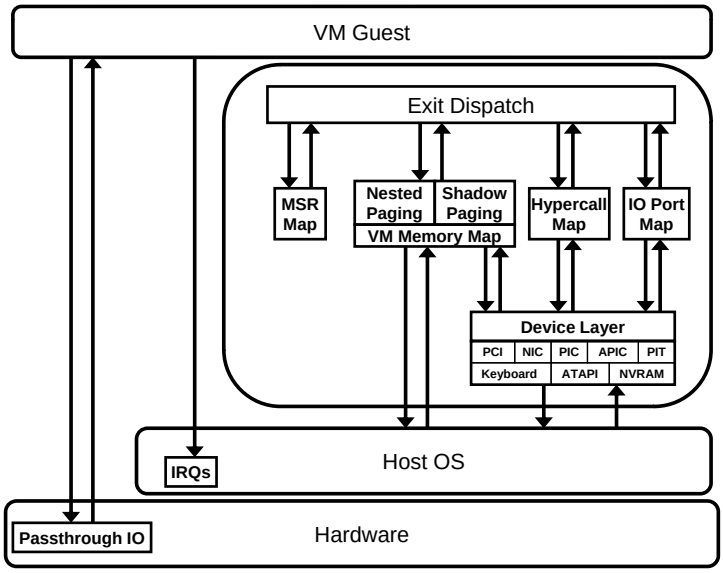


Figure 3.4: High level architecture of Palacios. Palacios consists of configurable components that can be selectively enabled. The architecture is designed to allow extensibility via event routing mechanisms [67].

with out hardware virtualization support. Specifically Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that comprise the PC platform. In addition to supporting full-system virtualized environments, Palacios provides support for the implementation of paravirtual interfaces.

Palacios fully supports concurrent operation, both in the form of multicore guests as well as multiplexing guests on the same core or across multiple cores. Concurrency is controlled with a combination of local interrupt masking and host OS provided synchronization primitives.

3.4 Conclusion

ConCORD is a distributed memory content sharing tracking system that is designed to fit into large-scale parallel systems. It works effectively with low overheads in large scale systems. ConCORD exposes powerful interfaces to allow users and service application programmers in HPC community to build their own content-aware services on top of ConCORD.

In this chapter, I introduced ConCORD's high level system design and its major architectural components. I will elaborate the design and implementation of each of its core components in the following chapters.

Chapter 4

Memory Content Update Monitor

The core functionality of ConCORD is to keep track of the memory content of all VMs running in the system. Since these VMs are running and their memory content changes over time, this information has to be discovered dynamically. To achieve this, ConCORD deploys a memory update monitor in each individual node to collect memory content and monitor their changes in VMs running on that node.

In this chapter I will describe the implementation of such memory update monitor in the Palacios VMM, followed by presenting my evaluation of its performance overheads. My evaluation results show that it is possible to dynamically track a VM's memory content updates within very minimal CPU overheads and network consumption. By presenting my implementation and evaluation, I have demonstrated a feasible approach to implementing a memory update monitor for ConCORD.

4.1 Introduction

In ConCORD, a memory update monitor runs on each individual node. It dynamically collects memory contents and monitors their updates to the VMs' memory running in the node. The core functionalities of a memory update monitor include:

- *Collecting and tracking memory contents.* Once a new VM is launched, the memory update monitor collects the memory content in its memory. Later, as the memory contents of the VM changes over time as it runs, the memory update monitor has to monitor these changes and collect new memory contents.

- *Hashing memory blocks.* ConCORD uses a content hash value to represent the content of a memory block, thus the memory update monitor collects a VM's memory contents by hashing its memory blocks. Once some of the blocks are updated, it needs to rehash them.

- *Populating memory contents in ConCORD.* ConCORD uses a system-wide distributed memory content tracer to keep track memory contents and content sharing over all VMs. To keep ConCORD updated on new memory contents from local VMs, the memory update monitor needs to send new content hashes from updated memory in local VMs to ConCORD periodically.

- *Mapping a content hash to memory pages containing corresponding content.* In addition, to allow ConCORD to efficiently locate a memory block's content from its content hash, the memory update monitor has to maintain a mapping from a given content hash to the memory blocks in VMs that have the corresponding content.

I will describe the approaches I have taken to implement each of these functionalities discussed above in the following section.

4.2 Memory virtualization in Palacios

In virtualized environments, a VMM cannot allow the guest to establish a virtual address mapping directly to any physical address, as this could allow the guest to conflict with VMM or other guests. At the same time, VMMs must maintain the illusion that the guest

is running on an actual machine in which mappings to any physical address are permitted. This requires a new memory address abstraction layer that is located between the physical hardware and what the guest perceives as hardware. Conceptually, this new abstraction is implemented using two levels of memory address mappings. Guest virtual addresses map to guest physical addresses using the guest's page tables, and guest physical addresses map to host physical addresses using a separate set of page tables created by the VMM.

There are usually two standard methods for VMMs to implement this virtualized paging. The first method is called shadow paging, which requires the VMM to fully emulate the address translation hardware in software. The second one, called nested paging, is introduced in recent versions of SVM and VT. It uses new hardware extensions to perform the additional address translation directly in hardware.

I will briefly describe how Palacios virtualizes a guest memory address space, as well as how Palacios implements both shadow and nested paging in this section before I describe how we can implement a memory update monitor inside Palacios.

4.2.1 Guest memory allocation

The most important aspect of Palacios' memory management system is that guest memory is preallocated in a single physically contiguous chunk. This means a guest's memory address space is a single block that is located at some offset in the host's physical memory space. Memory translations from guest physical addresses to host physical addresses are thus accomplished by simply adding a single offset value to every guest physical address.

4.2.2 Shadow paging

Shadow paging is implemented via a second set of page tables that exist in the shadow context. These are the page tables that are actually loaded into hardware when a guest begins executing. These shadow page tables map guest virtual addresses directly to host

physical addresses. These shadow page tables are generated via a translation process of the guest page tables (the page tables supplied by guest OS). The guest page tables translate guest virtual addresses to guest physical addresses, which are not valid memory locations since the guest memory space is located at some offset in physical memory. Therefore when a guest attempts to activate its set of guest page tables, the operation is trapped into Palacios where a set of shadow page tables is loaded into the hardware instead. Furthermore, any changes made to the guest's page tables are trapped by Palacios, which makes corresponding changes to the shadow page tables.

4.2.3 Nested paging

More recent CPUs include hardware support for virtualized paging. This support takes the form of a second layer of page tables that translates from guest physical addresses to host physical addresses. In essence nested page tables are a direct hardware representation of the VM's memory map that is readable by the CPU. This allows a guest OS to directly load its own page tables and handle most page faults without requiring VM exits to first determine if shadow page table updates are necessary. Unfortunately this requires an additional hardware cost for each page table lookup, since it must traverse two page tables instead of one. In general the software complexity needed for nested paging is considerably less than for shadow paging.

4.3 Implementation

As part of ConCORD, I have implemented a prototype memory update monitor within the Palacios VMM. The implementation consists of around 2000 lines of C code in Palacios and its Linux kernel module. It is completely transparent to the use of VMs and does not require any changes to host Linux kernel. Additionally, my implementation requires only

a small number of lines of changes to the existing Palacios code (less than 50 lines). In this section I will describe the my implementation, followed by evaluation of it in next section.

4.3.1 Hash function

Since ConCORD uses a content hash to represent the content of a memory block, a hash function is used by memory update monitor to generate the content hash for each block in the VM's memory. The memory update monitor is hash function-agnostic, any hash function can be plugged in. In the current implementation, the memory update monitor is configured to use MD5 as default hash function.

It is important to point out that while ConCORD needs a good hash function, it is not necessarily a cryptographic hash. We do not assume that the VMs are adversarial, which is a common assumption in HPC environments. For this reason, we can potentially reduce computational overhead by using non-cryptographic hash functions, for example, I have extended SuperHash [130] to generate a 128 bit hash as the same length as MD5 with less computing effort. I have compared the impacts of using different hash functions on performance overhead in Section 4.4, and we will see that, by applying less CPU-intensive non-cryptographic hash functions, the memory update monitor can reduce its CPU cost to 1/3 compared to using a standard cryptographic hash function.

4.3.2 Memory block size

In ConCORD, a virtual machine's memory is logically split into a sequence of blocks. A content hash is generated for each block to represent its content. Generally, the size of a memory block in ConCORD is configurable, however, in my current implementation, I use page granularity, i.e, 4 KB page in x86 machines, for easy implementation. Detecting memory content sharing with another block size is possible, but may cause extra CPU, memory and communication overheads. In addition, from Figure 2.5, we can see that

reducing the size of memory blocks does not increase the detected amount of memory content sharing for most of applications. For these concerns, ConCORD sticks with using 4KB page as default memory block size.

4.3.3 Tracking memory updates

Whenever a VM is first initiated and launched, the memory update monitor scans the entire VM's physical memory, generates a content hash for each memory page. Since all of these content hashes are considered to be new to ConCORD, they are sent to ConCORD's memory content tracer. After this, a memory page can be subsequently written by the VM. Writing changes the page contents and can invalidate earlier identified content sharing or create new content sharing. Therefore, the memory update monitor has to track updates to the VM's memory, and keeps collecting these new memory contents and sending them to ConCORD.

To track memory updates, the memory update monitor has to periodically identify all memory pages that have been updated recently. One way to do this is by scanning the full memory periodically and comparing the content hash of each page with its old one to determine whether the page has been changed since last scan. This is a simple approach, and it is easy to implement in Palacios. However, a full memory scan could be very costly, since it has to rehash every page during each scan.

Fortunately, most of architectures have mechanisms to allow us to track memory changes more efficiently, for example, the x86 architecture generally allows us to track memory page changes through its paging system. It incorporates a detailed model of paging that includes dirty bits on the page table entries(PTEs). The hardware ensures that the dirty bit is set on the first write.

As we have discussed in last section, VMM usually needs to set up an address translation mechanism which maps guest physical address to physical address. This can be done

by either shadow paging or nested paging. When shadow paging is used, the VMM installs a shadow page table whenever the guest operating system tries to install a process's OS-level page table. The shadow page table contains address mapping from the virtual addresses of each process in VMs to the VMMs desired host physical memory pages. The shadow page table is loaded to hardware, instead of page table set in guest OS. Thus, from the processor's perspective, the shadow page table is the actual page table it sees and uses to perform address translation for each instructions it executes. In turn, the processor sets the dirty bit of a shadow page table entry whenever that memory page is being write.

The memory update monitor can use this dirty bit in the shadow page table entry to detect modification to the memory pages. Because Palacios controls these page tables and the latter mapping, it can easily determine the guest physical pages that have been modified. In addition, it can manipulate and reset the dirty bit after each check round, in order to get updated memory pages for the next period of time. This allows the memory update monitor to restrict its periodic re-hashing only to those memory pages that have been modified since last rehashing round. Specifically, in each scan round, memory update monitor iterates through entries in all shadow page tables belonging to one VM, identifies all guest physical memory pages that have been updated since last scan (e.g, those entries that are either dirty or newly added). The guest physical address is used to identify a physical page in VM.

The dirty bit is not enabled by current nested paging hardware, which means my current implementation is limited to shadow paging. When nested paging is the only option, an alternative approach to keep track of page modifications is to set all VM pages to be read-only. Whenever the VM tries to write to a page, the operation is trapped by VMM, and memory update monitor marks the page as modified. The page then stays in a writable state until memory update monitor resets it to read-only state for next round of re-hashing. This means that writes to a page can only be trapped once between two re-hashing round.

This approach is also suitable for the case when VM exhibits a relative low update rate. Instead of periodic re-hashing at a fixed time interval, re-hashing only happens whenever the number of modified pages is more than a threshold number. This can eliminate the overhead of scanning the entire shadow page table periodically. However, when the VM is writing to a read-only page at the first time, it causes an exit into VMM, which brings more latency to the VM's performance. Thus, if the workload within the VM is write-intensive, tracking modified pages in this way could be expensive.

4.3.4 Collecting memory contents

As discussed above, during each scan round, the first step is to identify all memory pages that have been modified. Then the next step is sending these new content updates to ConCORD. To do this, all memory pages identified as modified since the last round are re-hashed by the memory update monitor, and new content hashes are generated for these pages.

Depending on the number of pages being updated each round, rehashing all the modified pages could be a time and CPU consuming operation. This frequent rescanning and rehashing could significantly slow down VM and its applications, if the workload within the VM is write-intensive. One approach we are using is to perform the periodic re-hashing in a separate kernel thread which can be run in a separate idle CPU core while the VM is running. This avoids long pauses of the VM.

4.3.5 Populating memory contents

In order to keeping ConCORD updated of new memory contents, during each scan round, all new content hashes generated from the re-hashing are sent to ConCORD's memory content tracer.

Consider a memory page that has been updated since last scan round. The memory

update monitor first identifies the page as updated, then rehashes the page to generate new content hash for it. After this, both the old content hash and new content hash of that page are sent to the ConCORD memory content tracer. This enables ConCORD to remove the old content hash while inserting new content hash.

The memory update monitor uses a hash update request to send new content hash to the memory content tracer. A hash update request contains a single hash content, with operation field to indicate either adding or removing this content hash from ConCORD. In addition, a VM id is included in each request to help ConCORD to track the source of this memory content. The format of the hash update request is defined as follows.

```
/* hash update operation types */
#define XCORD_HASH_UPDATE_ADD           0x11
#define XCORD_HASH_UPDATE_REMOVE        0x12

struct xcord_update_msg {
    uint8_t update_op;

    uchar_t hash[HASH_LEN];
    uint16_t vm_id; /* VM id*/
};
```

Hash update requests are sent to the ConCORD memory content tracer through UDP messages. Since the hash request is small in size (around 20 bytes), multiple hash update requests can be aggregated to single UDP message to increase the update throughput.

For each hash update, the memory update monitor is able to determine which ConCORD instance (xDaemon instance) this request should be sent to. This will be discussed more in the next chapter, but generally speaking, ConCORD employs a zero hop distributed hash table, which makes computing a content hash's responsible instance straightforward and can be done locally without any other node's help.

4.3.6 Scan interval

The interval during each scan round (*scan interval*) is another factor to consider. On the one hand, a short scan interval enables the memory update monitor to track more recent memory updates, and sends these new contents to ConCORD. Therefore, more frequent scanning improves ConCORD's accuracy because new memory contents can be quickly found and recorded. On the other hand, more frequent scanning brings more performance overhead, which includes CPU costs for scanning the page table entries and rehashing updated pages, and network overhead to send hash updates.

In my current implementation, the default scan interval of memory update monitor is 2 seconds. Our evaluation in Section 4.4 shows the memory update monitor uses only around 5% extra CPU for most of applications. For a few applications with much higher memory update rate, this could cause up to 25% extra CPU overhead, however, by increasing interval to 5 second, it reduces the cost significantly.

The scan interval could be adaptively adjusted by the memory update monitor to control the system overheads. In this approach, memory update monitor would dynamically determines the scan interval instead of using a fixed interval. If the memory update rate becomes too high, it increases the scan interval to reduce the performance impact to the system, and slows down the updates to ConCORD to prevent a flooding to the network. The adaptive scan interval approach will be added to memory update monitor in my future work.

4.3.7 Consistency

Since a hash update request is sent through a UDP message, it could be delayed or lost during transmission. The memory update monitor does not check whether ConCORD has received the request correctly. This results in the content hashes in the memory update

monitor potentially being different from these in ConCORD. Furthermore, As we have discussed, there is also inconsistency between the real content in current VM's memory and the content hashes maintained by the memory update monitor due to the lag of periodic re-hashing behind memory updates by the VMs.

For these reasons, the memory contents detected by ConCORD are not always consistent with the actual memory content existing in the VMs, i.e, a hash content in ConCORD may not have corresponding memory pages existing anymore in the VMs, or a memory page content may not be known by ConCORD. This together results in the memory content sharing reflected by ConCORD not always being accurate. However, this is tolerable since ConCORD works in the best-effort strategy. This inaccuracy is in its nature by design. Each node hereby has ground truth, which services are based on as well.

In our current implementation, ConCORD does not check whether the real content is still valid and exists in the VM. An alternative design choice would be using a copy-on-write mechanism to keep a copy of page with old content until its content hash has been removed from ConCORD. This would enable ConCORD to guarantee that all content hashes in ConCORD always have corresponding memory pages existing somewhere in VMs. However, this may consume significant amount of extra memory to keep the outdated memory pages.

4.3.8 Identifying content location

A content hash in ConCORD corresponds to a memory page with unique content in some VMs. To efficiently locate a VM's memory page given a content hash, the memory update monitor also maintains a local mapping table that maps a content hash to the local memory pages with corresponding contents.

This mapping table is created for each VM. The key of each entry in the table corresponds to a content hash, while the content of the entry is a list of addresses of memory

pages in this VM that have corresponding content. We will see in Chapter 7 that this mapping table is also very important for ConCORD's service command execution system because it allows it to locate a memory page for a given content hash.

During each scan round, a modified page is removed from the entry indexed by its old content hash, and inserted into the entry indexed by its new content hash. Note that this mapping table created by the memory update monitor is different from the distributed hash table that I will elaborate on next chapter. The distributed hash table is used by ConCORD for tracing memory content over all VMs system-wide, it maps a content hash to all VMs that have a copy of memory pages with corresponding content.

4.4 Evaluation

In this section, I will present my evaluation of the implementation of memory update monitor inside Palacios. Mainly I am focusing on investigating the CPU and network overheads caused by the memory update monitor in tracking and updating memory contents when different types of workloads are running inside VMs.

4.4.1 Methodology

In my evaluation, a number of VMs were launched on an eight-node cluster. One VM was running in each physical node, with various parallel and scientific workloads spanning and running in these VMs. A memory update monitor is running in each physical node to collect and monitor memory content updates for the VM running on the same node.

I measured the CPU times that are used by the memory update monitor during one scan round. This includes the CPU time used to scan the VM's shadow page table, rehash updated memory pages and send hash updates to ConCORD. Since the memory update rate varies during each round, I report the maximal (burst), average and minimal CPU times

that are consumed from each round during the entire execution of the workloads.

In addition, I also measured the size of data sent by the memory update monitor during one round. Similar to the CPU time, the burst, average and minimal numbers are reported.

4.4.2 Testbed

All the evaluation in this chapter were performed on a cluster with 24 nodes. Each node is equipped with two Dual Core Intel(R) Xeon(TM) 2.00GHz CPU, 1.5GBytes RAM and 32GB disk, a Broadcom NetXtreme BCM5703X 1Gbps Ethernet card. The nodes are connected through a 1Gbps Ethernet switch. The guest OS implementation was based on Linux 2.6.30.4, with MPICH2 for launching tested parallel workloads.

4.4.3 CPU utilization

Figure 4.1 shows the CPU time that is spent by the memory update monitor in each physical node for one scan round, which includes time to scan VM's shadow page table to identify all pages that have been updated from last scan, rehash all updated memory pages and send content hash updates over network.

To investigate the impact of hash function on CPU consumption, two different hash functions, MD5 and Superhash, are used to generate content hash for memory pages, and also two scan intervals, 2s and 5s, are applied.

For most of the workloads, using the MD5 hash function, it takes on average less than 128ms to scan and rehash memory updates when the scan interval is set to 2s. This is less than 6.4% overhead. With a scan interval of 5s, the CPU time required is even less, resulting in a 2.6% overhead. However, several applications (such as BT and MG) which have much higher memory update rates, need on average 512ms CPU time (25% overhead) during each scan round when the interval is 2s. For these applications, when the system extends its scan interval to 5s, the CPU times needed for each scan round do not

increase too much, which makes the CPU overhead decrease to less than 10%. Even for less frequently occurring burst updates during application execution, the memory tracer consumes less than 512ms, which is 25% overhead for most of the workloads. To reduce this overhead, the system can set the scan interval to 5s, which only increases the CPU time a little bit, resulting in an overhead of less than 10% even during the burst updates.

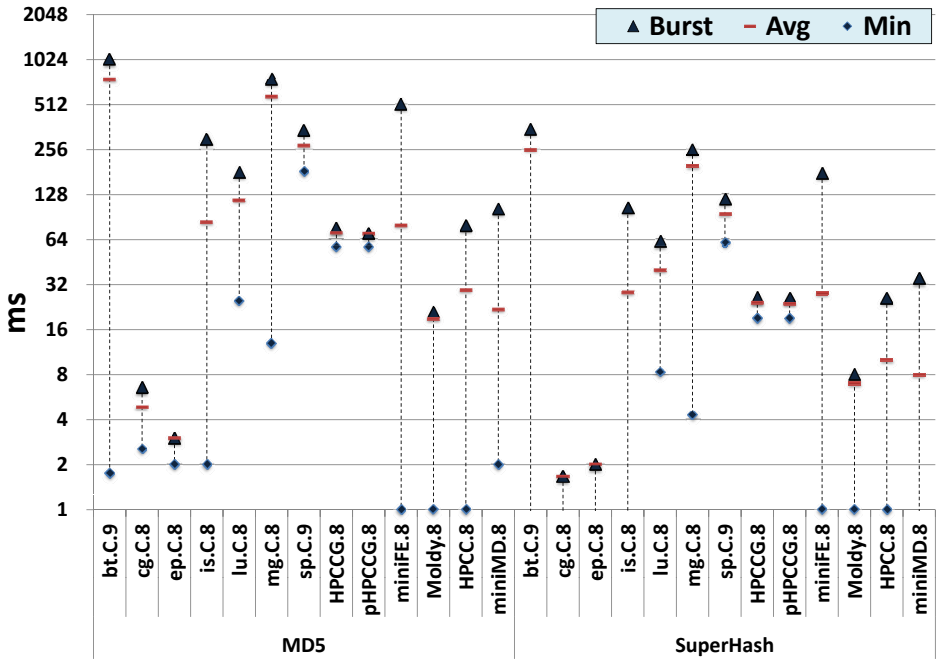
Compared with MD5, when the SuperHash hash function is used, the CPU overheads are about 1/3 as high. That means for most of the workloads, the overheads of the memory tracer are less than 2.2% with a 2s interval and less than 1% when interval is 5s, and less than 9% in 2s and 4% in 5s interval during burst updates.

As we discussed, the memory update monitor could dynamically adjust the scan interval for memory hash updates. If it detects that the current memory update period would cause CPU overhead to surpass a preset threshold, the memory update monitor could increase the scan interval. If it dips below a higher threshold, it could decrease the interval.

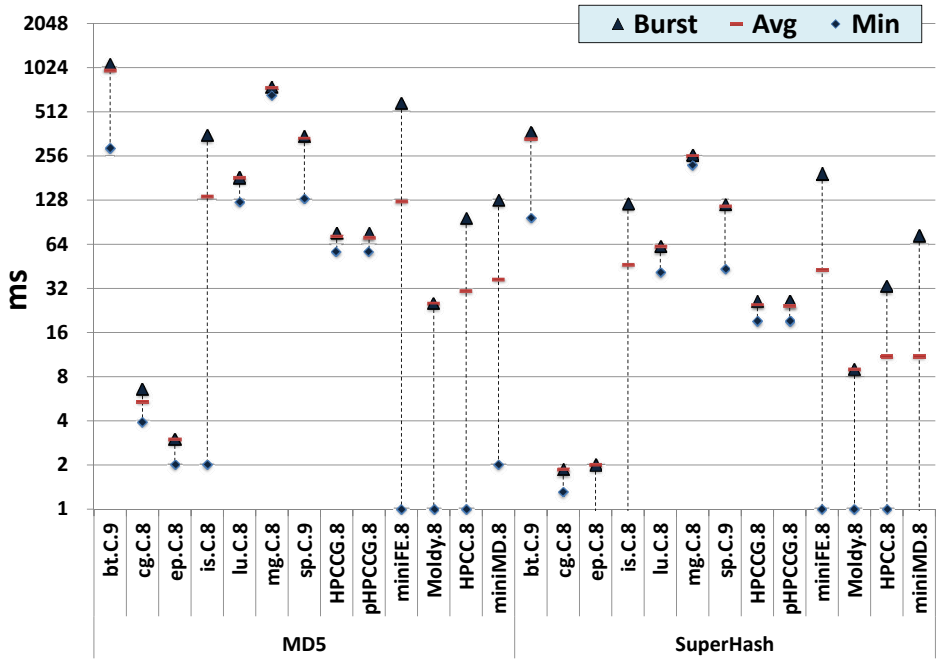
4.4.4 Network utilization

Besides the CPU overhead, I also evaluated the data traffic that is needed for each node to send its memory content hash updates to ConCORD distributed memory content tracing engine during each scan round. Figure 4.2 shows the amount of data that is sent over the network by each node during a 2s and 5s scan interval. Since MD5 and the extended version of SuperHash generate hash values of the same length, the data traffic needed are the same for both hash functions. From the figure, we can see that for most of the workloads, the memory update monitor sends less than 512Kbytes data during each round on average and less than 1024Kbytes during burst updates. This suggests that the network overhead of the memory tracer is generally less than 1% on 1Gbps network, which is very acceptable in our target cluster and HPC systems.

Our results show that it is possible to dynamically track a VM's memory content up-



(a) 2s Interval



(b) 5s Interval

Figure 4.1: CPU Time spent by the memory update monitor on each node during one scan round. One VM is running on each node. We show results from using 2s and 5s scan intervals with MD5 and SuperHash functions. For most of the workloads when using MD5 hash, it needs less than 128ms to finish memory updates in 2s interval, which is less than 6.4% overhead. While in 5s interval, it need less than 128ms, which is less than 2.6% overhead. When switching to SuperHash, the overheads are around 1/3 of the ones using MD5.

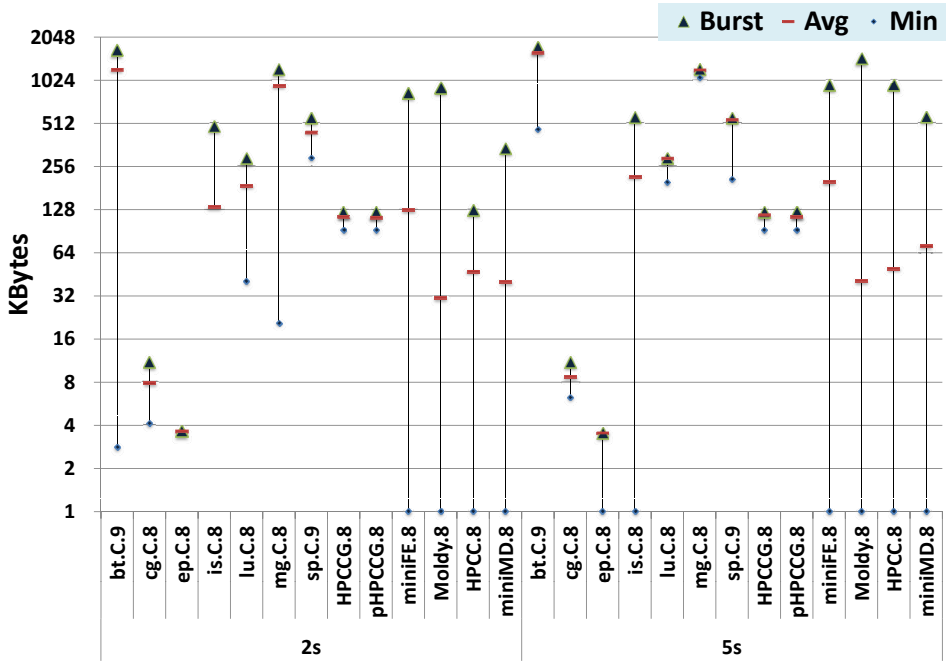


Figure 4.2: Amount of data sent over the network by each memory update monitor during one scan round to send content hashes to ConCORD memory content tracing engine. We show results from using 2s and 5s scan intervals with MD5. For most of the workloads, the memory update monitor sends less than 512Kbytes data during each round averagely and less than 1024Kbytes during burst updates.

dates running in individual nodes within less than 5% extra CPU overhead and less than 1% network consumption for most of workloads we have been running. Even for a few workloads that have occasionally high memory update rates, the memory update monitor could adaptively extend the scan interval to avoid burst CPU and network overheads if it is desirable.

4.5 Discussion

As we have seen, the hash computation imposes the most of the CPU overhead for the memory update monitor, especially for workloads with frequent updates, which can consume non-negligible CPU time that may limit performance of the system. I have already shown two ways to reduce the CPU overhead, by 1) using less CPU intensive non-cryptographic hash functions, or 2) extending the scan interval during burst update periods.

There may be other ways to further reduce the computational burden from CPU, particularly by exploiting the computation power of newly emerging hardwares or other potential hardware assistant, particularly, by a GPU.

GPU accelerated hashing Massively multi-core processors, like Graphics Processing Units (GPUs) provide one order of magnitude higher peak performance than traditional CPUs. They are getting more popular and wider deployed in current HPC systems. Much ongoing research has been exploring the feasibility of harnessing the GPU's computational power to improve system performance. For example, [38, 33] has demonstrated the use of GPU offloading to accelerate hash-based computationally intensive primitives and improve performance for massive storage systems. Similarly, we could apply this strategy here in the memory update monitor to further relieve CPU by offloading memory content rehashing onto the GPU.

4.6 Conclusion

In this chapter I presented the design and implementation of the memory update monitor for collecting and tracking memory contents on VMs running in local physical nodes. This memory update monitor was implemented in the Palacios VMM. My evaluation results show that it is possible to dynamically track a VM's memory content updates running

in individual nodes with less than 5% extra CPU overhead and less than 1% network consumption for most of workloads. By presenting the design with evaluation of my implementation, I have demonstrated a feasible approach to implementing the memory update monitor.

Chapter 5

Site-wide Memory Content Tracing

To be able to track memory content sharing across VMs in the system, ConCORD needs to discover which VMs in the network have a copy of a specific page. Since these VMs are running and their memory content changes over time, this information has to be discovered dynamically. To solve this problem in ConCORD, we use two subsystems: a site-wide distributed memory content tracer which allows ConCORD to locate VMs having a copy of a given memory block using its content hash, and a memory update monitor running in individual nodes to periodically track memory updates from VMs and send the updated contents to distributed memory content tracer.

I have discussed the implementation of memory update monitor in the last chapter. In this chapter, I will introduce the design and implementation of ConCORD's distributed memory content tracer (*content tracer* for short). Generally, the content tracer uses a light-weight implementation of a distributed hash table (DHT) to store all unique content hashes and map each content hash to a list of VMs that have a copy of the memory block with corresponding content. This chapter mainly describes the design, implementation and evaluation of this light weight DHT used by the content tracer. ConCORD's content tracer also exposes an interface to allow other applications to examine memory content sharing and shared content locations. I will describe the details of the interface and its implementation

in Chapter 6.

5.1 Introduction

The distributed hash table was originally targeted on wide-area distributed systems. DHT systems offer a simple API equivalent to the well-know hash table data structure: *push*(**key**, **value**) to store an item and *get*(**key**) → **value** to retrieve a value. DHTs offer a reliable distributed storage system for small data objects and enable fast lookups even in large scale systems.

There have been many distributed hash table (DHT) algorithms and implementations proposed over the years. Most of these DHTs are targeted to wide-area distributed systems, and scale logarithmically with system scales. Some works have appeared recently in using traditional DHTs in high performance computing systems, such as C-MPI [137] and ZHT [75]. But they are designed and used as general key-value stores for storing various sized objects, which is still too general for ConCORD.

Instead of using a traditional DHT as a key-value store, ConCORD employs a DHT to track memory contents and content sharing over a large set of VMs. These differences make ConCORD able to greatly simplify its DHT's design and focus on reducing the overhead of maintaining such a DHT. First, I have illustrated the major design differences by comparing the DHT needed by ConCORD with traditional DHTs.

- *Targeted system environments*: Most of traditional DHTs are targeted for wide-area distributed systems. Because distributed systems are naturally loosely-coupled, most of their design focuses on considering discovering peers, fault tolerance given node failure, network disconnection, frequently having peers join and leaving, etc, but performance is not major concern here. In contrast, ConCORD targets high performance computing systems, where each node in such systems is aware of every other

and the network topology. In addition, in an HPC system, the number of nodes are relative stable, so node joins would be very infrequent while node leaves could happen due to machine failure. Thirdly, the network capacity between nodes is usually 2 to 3 orders higher than in distributed systems. Finally, in a physically secured and well-administered HPC system, an attack through software is not a major concern. In contrast to traditional DHT, performance of the DHT in ConCORD is the major focus.

- *Type of key*: The key used in hash entries of ConCORD DHT is fixed-size, i.e, the size of a content hash value. For example, this is a 16 bytes string if ConCORD uses MD5 as hash function. However, in traditional DHT, the key for an object is usually a string with various possible lengths.

- *Type of object*: Traditional DHTs are used as a distributed key-value store, an object with various size and formats could be stored and indexed by its key. The size of an object could range from bytes to megabytes, or even gigabytes. However, in ConCORD, the DHT is not used to store particular objects or data, instead on to track memory content across VMs. The key itself reflects a specific memory block content, while the object stored for this key is just a list of the VM IDs identifying which VMs have a copy of memory page with the specified content. Although it is dependent on the implementation, in our implementation, this list is usually very short in size.

- *Query type* In a traditional DHT, the query only goes to one node with a single hash entry involved, such as reading an object with given key. However, in ConCORD a single query can involve single hash entry in one node, or most or even all of the hash entries in all DHT nodes, such as query for content sharing degree across a

group of VMs. This requires the DHT be designed to efficiently handle these type of collective queries.

- *Fault Tolerance* To provide a reliable and high available service, traditional DHTs usually have to replicate key/value pairs in multiple nodes in case of node failure. However, this adds more overhead and enlarges query latency. In ConCORD, fault tolerance for the DHT itself is not a critical requirement, since the failure of node means loss of some part memory content information. This brings only more inaccuracy of the memory content sharing from the perspective of ConCORD compared to the real memory content sharing existing in the system.

- *Persistence*: Traditional DHTs have strong persistence requirements on their objects/data stored on the hash tables. The objects still have to be retrievable even if a physical node crashes or fails. This requires DHT to store the objects/data on stable storages instead of keeping everything only in memory. For many HPC systems, compute nodes usually have no local persistent storage, which would require a DHT to write the data to a remote IO node, thus degrading system performance. However, in ConCORD, all DHT entries can be stored in memory only, as there is no persistence requirement for these hash entries.

- *Consistency*: For a traditional DHT, which is used as persistence storage system, the entries in the DHT must be always reflecting updated user objects, i.e, when an application sends an update request on an entry, it assumes that the entry in DHT is correctly updated. This requires the communication peer to rely on reliable communication, while the DHT has to add additional error checking to make sure everything goes correctly. However, in ConCORD, such strong consistency is not necessary, for example, a content hash update could be lost, which results in that the memory content information perceived by ConCORD is not always exactly the same as it exists

in real system. This is tolerable and indeed is the case for which ConCORD is designed. Such weak consistency can simply the design, while potentially reducing the cost of both communication and CPU.

In addition, for high reliability and availability, an object in the DHT is usually replicated and stored in more than one physical node in case of node failure. Traditional DHTs usually require strong consistency, which requires the DHT to make more efforts to keep all duplicated copies of the same object consistent to each other. However, since ConCORD is a best-effort system, it does not require such strong consistency, which can release the DHT's from having to maintain the consistency.

Due to all of these difference on system environments and requirements, ConCORD avoid many of design concerns arising from traditional DHTs to make it less heavy-weighted and focus mainly on improving its performance and reducing the complexity of implementation.

5.2 Zero-hop DHT in ConCORD

ConCORD uses a hash-based approach to detect identical memory content. Each memory block in all VMs is hashed, the content hash of the block is used to compare with each other to find possible content sharing. To allow ConCORD to locate VMs having a copy of a given memory block using its content hash, the content tracer deploys a system-wide distributed hash table to hold all unique content hashes for memory blocks from all VMs.

The basic entry of this DHT is a <content hash, list(VM id)> mapping pair. The key of each entry is the fixed-size memory content hash representing all memory blocks existing in the VMs with a specific content. The value of the entry is a list of VM IDs, of the VMs having a copy of such memory block. The DHT provides two operations to allow memory

update monitors to update entries in DHT: *insert(content hash, VM id)* and *remove(content hash, VM id)*.

In addition to the update operations, the content tracer exposes a query interface that allows users to discover various content sharing information across a group of VMs. I will describe this interface in detail in Chapter 6. In order to be able to respond to these kinds of queries, the DHT also calculates and maintains some content sharing metrics, such as the degree of content-sharing, the number of content-shared blocks, etc. This metrics are calculated based only on information stored locally, and are updated along with every update operation.

The DHT is partitioned and spread among all of ConCORD's running xDaemon instances. Similar to classic DHTs, all of these xDaemon instances form an ring-style overlay logically. Each node in the overlay is identified by a unique ID, which defines its position in the ring. Each xDaemon instance is responsible for storing and maintaining a certain number of partitions of the table. However, contrarily to other DHTs, ConCORD's DHT uses zero-hop routing design. This means given a content hash, a party can compute its content hash owner nodes immediately and locally without the need of any assistance or information from other peer nodes.

I have chosen this design because the primary requirement for ConCORD is performance. For example, a VM with 1GB memory size is identified by upto 262,144 memory hashes, each requiring a lookup to determine if and where the duplicated pages is available in other nodes. This high number of lookups imposes stringent requirements on the DHT latency and throughput. In addition, the target system is high performance computing systems, instead of distributed systems like what most of DHTs are designed for. Our targeting systems are relatively stable environments, resistance to high churn is not the main requirements for DHT here.

5.3 Implementation

The basic entry of the DHT in memory content tracer is a <content hash, list(VM id)> mapping pair. The key is the memory content hash representing memory blocks existing in VMs with a specific content. The value of the entry is a list of the VM IDs of VMs that have a copy of such a memory block. The VM ID is used here to locate the VM that has a copy of page with corresponding content. In implementation, the DHT uses a dynamic bitmap in each of its hash entries to represent such a VM list. A dynamic bitmap is generally a bit array, with each bit representing a unique VM ID in the system.

The DHT provides two operations to allow memory update monitors to update entries in DHT, which include *insert(content hash, VM id)* and *remove(content hash, VM id)*. When a content hash update is received by an xDaemon instance for inserting a content hash, it looks for the hash entry indexed by this content hash. If there is no entry exist, creates a new hash entry. The bit corresponding to source VM in the hash entry's VM bitmap is set. If the request is removing a content hash, the bit is cleared in the hash entry. If the hash entry ends up with all bits being unset, it is removed from DHT.

5.3.1 VM ID management

In ConCORD, each VM is assigned a unique ID. ConCORD maintains the mapping from a VM-ID to the VM's location information, such as physical node it is running on. VM-ID is an integer number assigned from a range of integers, also called a *VM-ID space*. For example, a VM-ID space could be represented by [0, 1024], which means all VMs in the system can be assigned to a id ranged from 0 to 1024. The size of a VM-ID space defines the maximum possible number of VMs in the system. If the actual number of VMs surpass this size, ConCORD then has to expand its VM-ID space dynamically.

A VM ID is reusable, when a VM terminates, its VM id is recycled. However, before

the ID can be assigned to other VMs, its corresponding bit in each entry of DHT has to be cleared. This clearing process happens only when xDaemon instance is relative idle. The ID for a terminated VM is not recycled until its corresponding bit in all hash entries has been cleared.

VM-ID space expansion If the number of VMs in the system surpasses the size of the VM-ID space, or if there is no ID available for a new VM, ConCORD then expands its VM-ID space. VM-ID space expansion usually resizes the length of the VM-ID space to its next power of 2. For example, if current VM-ID space size is 128, and the VM-ID space expansion is needed, then the VM-ID space is resize to 256, and so on.

Dynamic bitmap for VM list The DHT uses a dynamic bitmap in each of its hash entry to represent a list of VMs that have a memory page with such corresponding content. Dynamic bitmap is generally a bit array, with each bit representing a unique VM ID in the system. The length of this bitmap is dynamically changeable. Generally, the number of bits in this bitmap is the same as the size of current VM-ID space. Once ConCORD expands its VM-ID space size, each hash entry needs to resize its VM bitmap. However, this VM bitmap resizing does not happen at once during VM-ID space expansion, instead, the bitmap of a hash entry is resized only when there are updates to that hash entry. For example, when there is a hash update request to insert a VM-ID to an entry, but the corresponding bit in hash entry's VM bitmap is not assigned yet, DHT checks the current size of VM-ID space and expands its VM bitmap accordingly.

5.3.2 Content hash ownership management

In ConCORD, an xDaemon instance is the basic component that holds the content hash partitions. In ConCORD, each ConCORD xDaemon instance is assigned a universal unique

id (UUID) in the ring-shaped ID space.

Each content hash value can be easily mapped to a 64-bit integer. This integer can be defined as the *name* of the content hash. The name of a content hash is used to determine the xDaemon instance that is responsible for storing it. The entire name space N (a 64-bit integer) is evenly distributed into m partitions where m is a fixed big number indicating the maximum number of xDaemon instances that can be used in ConCORD, and this is configurable at compile time. For example, when ConCORD running with k daemons, each daemon holds m/k partitions, with each partition storing N/m different content hashes. Each partition can then be assigned to one or more (for redundancy) xDaemon instances. For simplicity and performance consideration, each partition is stored by only one instance in my current implementation.

The partition ownership is determined in a very simple way, by modding the partition number with the total number of xDaemon instances running in ConCORD. For this simple partition ownership management scheme, given a content hash, it is straightforward and simple to determine its responsible xDaemon instance, without any need to query other nodes, or some forms of mapping table.

xDaemon instance management

In static membership, every xDaemon instance at ConCORD bootstrap time has all the information about how to contact other xDaemon instances in ConCORD. In the static case, when ConCORD starts, it reads the xDaemon instance to node mapping information from a configuration file. Once the membership is established, no new xDaemon instances would be allowed to join the system, although the xDaemon instances can move between physical nodes. An xDaemon instance could leave due to machine failures, and ConCORD assumes a failed node could be recovered soon and restart the xDaemon instances on it, or that these instances can be restored on other backup nodes. So over the long term, the

number of xDaemon instances in the system is fixed.

In contrast to static membership, in dynamic membership, xDaemon instance may join and leave at any time. This makes the management of DHT and other peer information much more complicated. Due to this implementation complication, in addition to the consideration that the number of nodes in our targeted HPC systems is relative stable, ConCORD does not support dynamic membership.

5.3.3 Fault tolerance

Communication failure

A communication failure between the memory update monitor and the DHT is mostly causing the loss of content hash updates from the VMMs. As we discussed in Chapter 4, this results in that the memory content hashes stored in DHT being inaccurate. This is by nature in ConCORD's design because it works in a best-effort strategy.

xDaemon instance failure

Communication failure or physical node failure can also cause some of the xDaemon instances to not be available. This means all content hash entries maintained by these instances are not available. If the xDaemon instance fails due to a node crash, then all content hash entries in this instance are permanently lost.

One or a small number of xDaemon instance failures is tolerable, since the ConCORD memory content tracer is by nature not exactly accurate about the memory contents existing in the VMs. Therefore, lost of one or small number of xDaemon instances just brings only more inaccuracy temporarily.

To provide more reliability on this, the same content hashes could be distributed and stored in multiple xDaemon instances. However, this would cause more memory overheads to store the duplicated copies, and more communications for each update, in addition to

communications to keep them consistent. In my implementation, I am trying to make it as simple and efficient as possible, so I did not use a replication strategy in ConCORD's DHT design.

5.3.4 Persistence

The content tracer's DHT is a distributed in-memory data structure. By default, it does not write any of its hash entries into persistent storage. All key-value pairs are kept in memory so that it can achieve low latency in both updates and lookups when compared to other persistent hash maps that need to hit disk for either lookup or update operations.

5.4 Evaluation

5.4.1 Testbed and metrics

The metrics measured and reported are:

- *Latency*: The time taken for a DHT update request from the memory update monitor running on a remote node mainly reflects the latency of the network, instead of the operation latency of the hash table itself. However, in this chapter, we are mainly interested in measuring the operation overhead of hash table itself, instead of network latency. Therefore, I have measured the time taken for updating a hash entry through a local call inside an xDaemon instance.

- *Memory Utilization*: Besides the DHT update latency, I also measured the memory overhead of the DHT. I have measured the memory consumed by each xDaemon instance, as a function as the number of hash entries existing in its local hash table. In order to show the actual memory used by DHT, I have implemented a simple memory allocator inside ConCORD, which manages a large chunk of memory blocks

allocated from the standard malloc and assigns the necessary data chunk for hash entries.

- *Message Loss Rate*: Since a memory content update message is sent through UDP, it could get lost. The lost update message is one cause of ConCORD's inaccuracy in its detected memory content sharing knowledge. Therefore, in my evaluation, I have also measured the loss rate of update messages during intensive memory content update periods.

All the evaluation in this chapter were performed on a cluster with 6 PowerEdge R415 machines. Each node is equipped with two Quadcore 2.4 GHz X3430 Intel Xeon CPU, 8GBytes RAM, a Broadcom NetXtreme II 1Gbps Ethernet card. The nodes are connected through a 1Gbps Ethernet switch.

5.4.2 Results

DHT operation latency

Figure 5.1 shows the time taken to update a hash entry in the DHT. Four update cases are shown on the figure: creating a new hash entry(*insert-hash*), inserting a block into an existing hash entry (*insert-block*), deleting a hash entry(*delete-hash*) and deleting a block from a hash entry without deleting the actual entry (*delete-block*). The result from the figure shows that time spent on four different update operations is relatively independent of the current hash table size, which is what we expect by using a hash table data structure.

Memory Utilization

Figure 5.2 shows the memory usage (MBytes) by each xDaemon instance as a function of memory size per VM in each host. There is one VM and one xDaemon instance running in each physical node. The overhead percentage is calculated as the percentage of bytes used

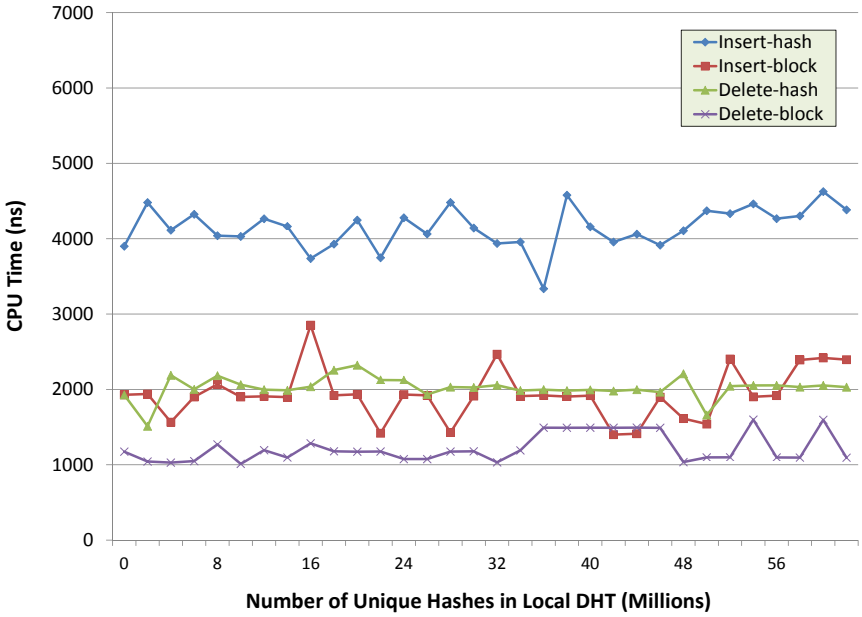


Figure 5.1: CPU time consumed for four types of DHT update operations as function of number of hash entries currently already existing in the DHT. *Insert-hash* is insert operation that causes a new memory hash entry being created, while *Insert-block* inserts a memory block into existing memory hash entry. Similarly, *Delete-hash* deletes a memory hash entry while *Delete-block* only removes a memory block from a memory hash entry. The results suggest that these operations are constant regardless of the number of hash entries currently existing in the DHT.

by xDaemon on per bytes of VM's memory. We can see from the figure that, when the standard memory allocator is used, xDaemon's memory overhead is around 5% for large number of hash entries, while comparing with our customized memory allocator, which reduces the memory overhead to half (around 2.5% for large number of entries).

Update Message Loss Rate

Since the DHT update message is based on UDP, it gets lost some times, although this is rare in our targeted parallel system. I have measured the lost rate of update messages

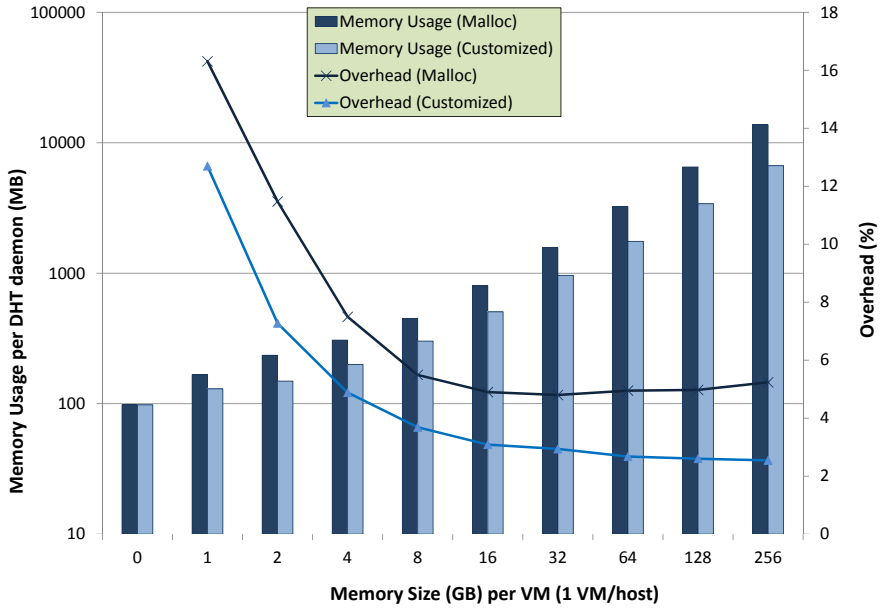


Figure 5.2: Memory usage by xDaemon instances running content tracer on each physical node as a function of the number of hash entries currently existing in each of xDaemon's local hash table. Those marked as *Malloc* measure memory usage by using the default memory allocator in the standard C library, while the items marked as *Customized* correspond memory usage using our own customized memory allocator that avoids certain overheads and limitations of the default memory allocator.

during an intensive VM update periods. In my test case, 2 VMs are running on each of the nodes, with each assigned 4GB memory. During the test, ConCORD is started on a different number of physical nodes, with 2 VMs launched on each physical node. Once the test starts, each VM scans its entire memory space, hashes each memory page and sends the content hash to ConCORD content tracer simultaneous. After all VM's memory content has been updated to ConCORD. We counted the totally number of blocks in all VMs, and compared with the total number of blocks perceived by ConCORD. The difference between these two numbers are due to the loss of update messages. The result is shown in

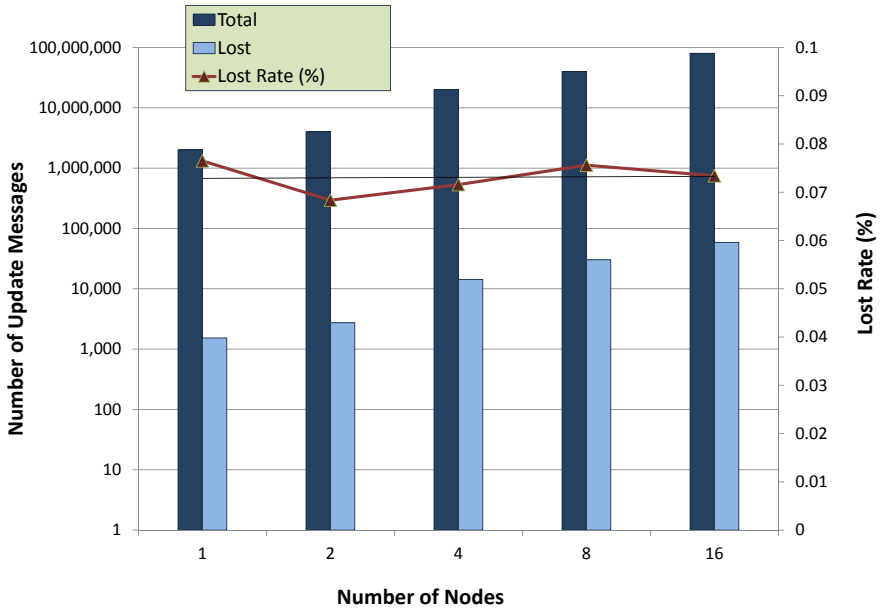


Figure 5.3: Update message that are sent and lost during regular periodic updates from memory update monitor. The lost rate is relatively small, which is less than 0.1%.

Figure 5.3, which suggests that even in an intensive update period, of update message get lost less than 0.1%.

5.5 Conclusion

I have discussed the design and implementation of ConCORD's distributed memory content tracer, which is mainly an implementation of a light-weight implementation of distributed hash table. The DHT stores all unique content hashes in the system and maps each content hash to a list of VMs that have a copy of the memory block with corresponding content. Our target system environments and requirements enable us to simplify our design on the DHT used by ConCORD. My evaluation shows that the DHT is likely to scale

well in large scale systems.

Chapter 6

Content-sharing Query

In this chapter, I will describe the content-sharing query interface exposed by ConCORD. The chapter starts with a definition of content-sharing queries, followed by a discussion of the design and implementation of the query interface in ConCORD. Finally my evaluation of the query interface is presented.

6.1 Content-sharing query

A *content-sharing query* is a query to ConCORD to check the amount of memory content sharing and to examine the location of shared regions on a set of virtual machines. Depending on the type of queried information, content sharing queries are categorized into two types: node-wise queries and collective queries.

Node-wise query A node-wise query is a type of query that checks only local content sharing information, which only requires information stored in one ConCORD instance. For example, checking how many memory blocks with a given content hash exist in a set of VMs.

Collective query A collective query is a type of query that examines system-wide content sharing information. This kind of information has to be gathered from all or most of the ConCORD instances.

Before presenting the implementation of the query interface in ConCORD, the formal definitions for query types are defined first.

6.1.1 Node-wise query

Node-wise queries in ConCORD mainly comprise of two kinds of queries. The first one is checking how many memory blocks containing given contents exist in a set of virtual machines. The other one is finding the actual locations of those memory blocks that have a given content, i.e, which VMs have a copy of memory blocks containing a given content hash. These two queries are defined as follows.

- **Number of Copies of a Memory Block (NoC)** Given a memory content hash and a set of virtual machines, we define the *NoC* as the the number of virtual machines in this set that have a memory block containing the given content.

  Input: $NS \subseteq N$: Given set of virtual machines; mc: the given memory content.
  Output: $A(NS, mc)$: number of virtual machines that have a memory block with content mc.

  $$NoC(NS, mb) = \sum_{i \in NS} |\{mb\} \cap M_i|$$

  The NoC gives the number of virtual machines that have the given block of memory content in their memory.

- **Locations of Copies of a Memory Block (LoC)** Given a memory content hash and a set of virtual machines, the LoC gives a subset of those virtual machines that have a memory block containing the given content.

Input: $NS \subseteq N$: Given set of virtual machines; mc: the given memory content.

Output: $LoC(NS, mc)$, set of virtual machines that have a memory block with content mc.

$$LoC(NS, mb) = \{i | i \in NS \text{ AND } mb \in M_i\}$$

6.1.2 Collective query

The collective query interface in ConCORD currently supports three kinds of queries. The first one returns the degree of memory content sharing (*DoS*) existing in a set of VMs. DoS reflects the rough amount of content sharing in the system. The second kind of query returns the number of hot memory blocks among a given set of VMs. Hot memory blocks is defined as these memory blocks with their contents shared at least in *k* different VMs. ConCORD provides a query interface to acquire the number of hot memory blocks and their actual contents across a set of VM. This enables users to gain more detailed information on the amount of content sharing and actual shared contents rather than DoS. And the last one returns the actual memory content hashes of these hot memory blocks. The three kind of queries are defined formally as below.

First of all, let us define some variables that are used for query type definitions.

Definitions

$$N = \{0, 1, 2..n\} : \text{set of virtual machine in the system}$$

$$M_i : \text{set of memory blocks in virtual machine } i$$

$$MD_i : \text{set of memory blocks with distinct contents in virtual machine } i$$

$$|M_i| : \text{number of memory blocks in virtual machine } i$$

$$|MD_i| : \text{number of memory blocks with distinct contents in virtual machine } i$$

- **The Degree of Memory Content Sharing (DoS)** *DoS* is defined to represent the degree of memory content sharing in the given set of virtual machines. The content sharing in DoS includes both intra-node and inter-node sharing.

  Input: $NS \subseteq N$: Given set of virtual machines.

  Output: $DoS(NS)$: Degree of memory content sharing across all virtual machines in NS.

  $$DoS(NS) = \frac{|\bigcup_{i \in NS} M_i|}{\sum_{i \in NS} |MD_i|}$$

  DoS is a ratio with a value between 0 and 1. The smaller the value, the more memory content sharing exists in the given set of virtual machines.

- **The Intra-node Content Sharing Degree (DoSi)** To differentiate the intra-node and inter-node content sharing, we further define *Intra-node Content Sharing Degree (DoSi)* and *Inter-node Content Sharing Degree (DoSx)*.

  Input: $NS \subseteq N$: Given set of virtual machines.

  Output: $DoSi(NS)$: Degree of intra-node memory content sharing across all virtual machines in NS.

  $$DoSi(NS) = \frac{\sum_{i \in NS} |M_i|}{\sum_{i \in NS} |MD_i|}$$

- **The Inter-node Content Sharing Degree (DoSx)** The *DoSx* is defined as following:

  Input: $NS \subseteq N$: Given set of virtual machines.

  Output: $DoSx(NS)$: Degree of intra-node memory content sharing among all virtual machines in NS.

  $$DoSx(NS) = \frac{|\bigcup_{i \in NS} M_i|}{\sum_{i \in NS} |M_i|}$$

The same as $DoSi$, $DoSx$ is a value between 0 and 1. The smaller the value, the more inter-node memory content sharing exists in the given set of virtual machines. Note that $DoS = DoSi \times DoSx$.

- **Number of Memory Blocks with a Certain Number of Copies (NoM)** *NoM* gives the number of memory contents that appear at least in a given number of nodes.

  Input: $NS \subseteq N$: Given set of virtual machines k: number of copies.
  Output: $NoM(NS, k)$: number of memory blocks that share same content at least in k virtual machines in NS.

$$NoM(NS, k) = |LoM(NS, k)|$$

- **List of Memory Blocks with a Certain Number of Copies (LoM)** Beyond the number of memory blocks shared among nodes, the *LoM* defines the exact memory contents that are shared. It is defined as :

  Input: $NS \subseteq N$: Given set of virtual machines k: number of copies.
  Output: $LoM(NS, k)$: set of memory blocks that share same content in at least of in k virtual machines in NS.

$$LoM(NS, k) = \{m | (m \in \bigcup_{i \in NS} M_i) \text{ AND } (NoC(NS, mb) \geq k)\}$$

6.2 Query interface

The query interface is implemented as a user library which can be linked into any application program code. The library is responsible for communicating with ConCORD to launch a query and collect results. From the perspective of a service application program, the query interface is no more than a standard library, with the different types of queries exposed as standard C function calls. These calls include:

- *double degree_of_sharing(NodeSet nodes, ShareType type)*: This returns degree of content sharing in the required types across the given set of nodes. The ShareType could be “intra-node”, “inter-node” or “both”.

- *NodeSet get_location(hash_value)*: Returns a list of node each of which has at lease one memory block that generates the given content hash.

- *int get_copies(hash_value)*: Returns the number of copies of the memory blocks that exist in system which generate the given content content.

- *HashSet hashList_of_shared(NodeSet nodes, int k)*: Get a list of content hashes which represent the memory blocks that have at least k copies across a set of nodes.

Currently, the library is linkable to any Linux application, and the interface is provided in the C language. The actual C interface of the library is shown as follows.

```
/* query interface initiation,
 * must be called before use of any other query functions */
int xcord_query_client_init(int num_daemon, char * daemon_map_file);

/* node-wise interface */

/* Returns the number of memory blocks
 * existing in the given VM set that generates
 * the given content hash. */
int xcord_block_copies(uchar_t hash[HASH_LEN],
                       struct vm_set * vms,
                       int block,
                       int (*call_back)(int qtype,
                            int ret,
                            struct vm_set * vm_lst,
                            void * data)
                       void * data);

/* Returns a list of VMs which has at least one memory
 * block that generates the given content hash.
 */
struct vm_set *
xcord_block_locations(uchar_t hash[HASH_LEN],
```

```
struct vm_set * vms,
int block,
int (*call_back)(int qtype,
      int ret,
      struct vm_set * vm_lst,
      void * data)
void * data);;
```

The node-wise query interface contains two core functions, one for getting the number of memory blocks with given a content hash, and the other for examining the actual list of VMs that have a memory block in their memory with the given content hash.

Both of the functions support blocking and nonblocking mode calls. In blocking mode, the function call returns when the query completes or fails. While in nonblocking mode, it returns immediately. For calling code to get the query results, a callback function is supplied in the call. Once the query completes or fails, this callback function is invoked with returned query results. The nonblocking mode function call is useful when multiple queries on different content hashes need to run as it enables the queries to be sent and handled in parallel to improve the overall query throughput.

```
/* Collective queries interface */
double xcord_degree_of_sharing(struct vm_set * vms, int timeout);

/* number of memory contents that exist at least k VMs */
int xcord_num_hot_blks(struct vm_set * vms, int level);

/* a list of memory blocks that exist at least in k VMs */
struct xcord_hash_list *
xcord_lst_hot_blks(struct vm_set * vms,
                   int level);
```

The collective query interface includes three core functions, all of which examine some level of content sharing information on a set of given VMs. The first one returns the *DoS* of the set of VMs, while the second one gives the *NoM* and the last one returns the *LoM* on a set of given VMs.

In contrast to functions for node-wise queries, the collective query function calls are blocking in current implementation, which returns only when the query completes or fails. In my future work, I will add the nonblocking option to these functions.

6.3 Query Handling

Query handling involves two parties: client library and xDaemon instance. In a node-wise query, only one xDaemon instance is involved, while all xDaemon instances are involved during handling of a collective query. I will describe the communication strategy between these two parties, and the process logic to handle a query in each party in this section.

6.3.1 Client library

For node-wise queries, the query library directly sends the query request to the xDaemon instance which is responsible for the content hash in the query. As we discussed in Chapter 5, ConCORD uses a zero-hop DHT to store the content hashes. Given a content hash, any node can compute the ID of the xDaemon instance that is responsible for storing it. When the query library initiates, it gets the mapping of xDaemon instances with their physical nodes. The mapping is relative stable, it changes rarely. Once the mapping is updated by the ConCORD management node, it notifies the local library to update it.

Therefore, the library code computes the ID of the target xDaemon instance, maps it to a <ip:port> where this instance is actually running on. Then a query request is sent to the instance. Once the xDaemon instance receives the request, it looks it up in its local hash table to determine the result, and returns it back with the query reply.

The query message communicated between the library and xDaemon instance is defined as follows. Both the query request and query reply use the same message format. The message is designed to accommodate various types of queries, including both node-

wise query and collective query. The query message is packed into a UDP message and sent over the network.

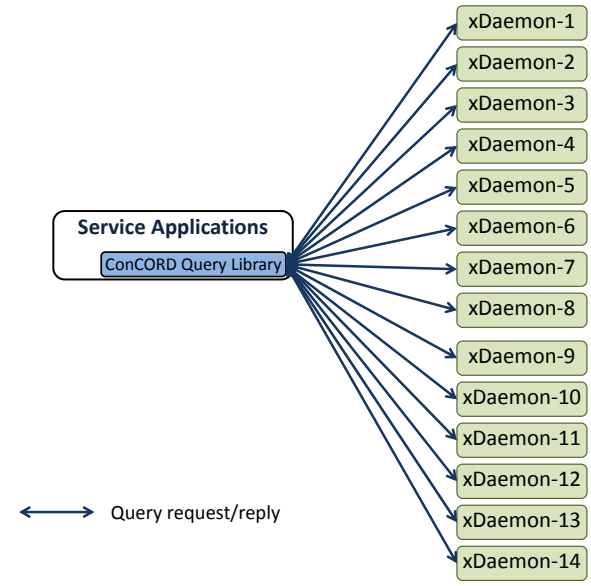
```
struct xcord_query_msg {
    uint8_t query_op;
    uint32_t request_id;  /* unique id to identify request */
    uint16_t xdaemon_id;  /* xDaemon serving this request */

    char client_ip[IP_LEN]; /* calling client's ip and port */
    uint16_t client_port;

    uint32_t num_blks;
    uint32_t num_hashes;

    struct xcord_hash_list hash_lst;
    struct vm_set vm_lst;
}__attribute__((packed));
```

6.3.2 Collective query

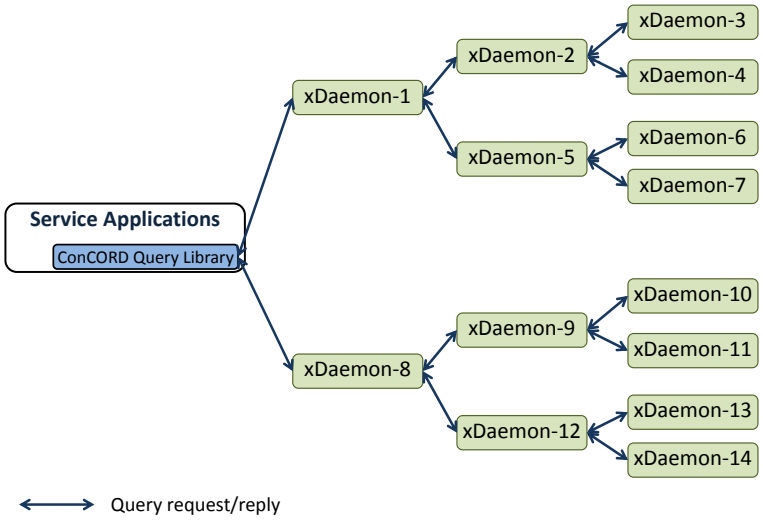
As we have discussed before, the ConCORD memory content tracer is a fully distributed system, with the system-wide information split across, stored and maintained by a large number of running xDaemon instances. To handle a collective query, ConCORD has to acquire system-wide content sharing information from all of these running instances. Therefore, during processing of a collective query, the client's library is required to communicate with all available xDaemon instances. It sends out query requests to them, collects all information replied and computes (or merges) results together to the final results.

Depending on which instances the library directly communicates with, two strategies can be used for implementing this communication process: broadcast based or a tree based communication. These two communication topologies for implementing collective query is illustrated in Figure 6.1.

- *Broadcast Reduction Topology*: In broadcast reduction topology, the library communicates with all available xDaemon instances directly. Once there is a collective



(a) Broadcast Reduction



(b) Tree Reduction

Figure 6.1: Topology of query request/reply in two different implementation of collective query.

query call from local program code, it sends a query request to all available xDaemon instances and gathers replies from all of them, merges (or summarizes) these results together to form the final results, and returns them to the caller.

- *Tree Reduction Topology*: In contrast with broadcast topology, when a tree reduction is applied, xDaemons instances are logically organized as a hierarchy. Each xDaemon instance is assigned within different layers in the hierarchy. During a query, a xDaemon instance talks to these instances that are within one layer distance. Using tree reduction, the client library communicates directly to only a few of top layer xDaemon instances, and these instances are responsible for relaying the query request to the next layers, and then collecting and gathering the replies from them. For example, as shown in Figure 6.1, to handle a collective query, client library sends the query request to the xDaemon instances in top layer, instance-1 and instance-8, which then pass the request down to next layer of instances, and so on. The replies come back in the reverse direction, each instance being responsible for collecting replies from its next layer's instances (child instances), summarizing them and replying to its up layer instance (parent instance).

The tree topology reduces the computational and communicational pressure on the client library node, while it puts more burdens on xDaemon instances, particularly those instances sitting in the top layers. Which strategy is better depends on various factors. The tree topology performs better when the client library is running on a relative less powerful machine while the xDaemon instances are running on more powerful servers, or when the client library is connected to xDaemon instances through a slow network while xDaemon instances are connected with each other over a fast network.

In the following section, I present my evaluation of the implementation of collective query handling using both strategies, with a comparison of their performance.

6.3.3 xDaemon instance

In xDaemon, a thread runs to handle incoming query requests. Once a node-wise query request is received, the content hash is used to search in the local part of the DHT for corresponding entry. If the entry is found, the VM list in the entry is compared with the given VM list in the query request, and the disjoint set of two VM lists is generated. Either the length, or the actual VM IDs in this disjoint set is encoded in the query reply, and sent back to the client library.

When receives a collective query request, each xDaemon instance handles the request independently based on the content hashes stored in the local part of the DHT. For example, to handle a *DoS* query, each xDaemon instance returns the number of blocks and number of unique content hashes stored in its local hash table to client library. The library then summarizes the total number of blocks and the total number of unique content hashes from all xDaemon, and computes final DoS value by dividing these two numbers.

The query to get the number or list of hot memory blocks works in a similar way. Each xDaemon instance returns the number or list of hot memory blocks detected by its local hash tables. Then the client library adds these numbers or merges all lists together to be the final results.

6.3.4 Target VM Set

In a content-sharing query, a VM set is always specified as the target set of VMs. ConCORD handles the query by checking the required content sharing information among this set of VMs. This VM set can be any subsets of all VMs monitored by ConCORD. However, specifying an arbitrary VM set in a query, particularly a collective query, requires ConCORD to parse its entire distributed hash table to compute the results during query time. For example, to handle a *DoS* query, each xDaemon instance has to check the total

number of blocks and the total unique content hashes existing in the given set of VM. To compute these two numbers, xDaemon instance has to parse all hash entries in its hash table to determine whether each content hash exists in any of the VMs in the given VM set. This usually takes a long time, and large CPU overhead if the hash table size is large.

The primary reason is because the query is targeted to an arbitrary VM set. No information about this set of VMs exists in ConCORD prior to the query. However, in practice, queries on any arbitrary VM set are usually not common. Instead, many VMs are usually bound to each other naturally to form a set, for example, a set of VMs that run one parallel application or collaborative services. Most of the collective queries are targeted to such kind of relative stable VM sets, instead to arbitrary sets of VMs.

For this reason, ConCORD allows the registration of a set of relevant VMs, a *VM group* in advance. After the VM group is registered, ConCORD pre-computes and maintains content sharing information for this VM group. The content sharing information about this VM group is keeping updated when there are content hash updates to any VMs in the group. Therefore, when there is a collective query targeting on this VM group, the xDaemon instance does not need to parse the hash table and compute the content sharing at query time. Instead, all information is already pre-computed and ready to return to the ‘client library. This can significantly reduce the response time for collective queries. As we will see in the evaluation results presented in Section 6.4, the response time for collective queries on registered VM group is 3 to 4 order of magnitude lower than those targeting an arbitrary VM set.

When a set of relevant VMs is created and launched in the system, system administrator is allowed to register this set of VMs with ConCORD. ConCORD maintains a list of registered VM groups on each xDaemon instance, and pre-computes the information on both DoS and number of hot memory for each registered VM group locally. Once a content update comes, the source VM is checked to find whether it belongs to any regis-

tered VM groups. If any VM group is found, their content sharing information is updated accordingly.

6.3.5 Fault Tolerance

Two types of failures could happen during query handling: message loss and node failure, here we briefly discuss how these failures are handled in the ConCORD query interface.

Message Loss Since the query request and reply are sent through reliable UDP, as I have described in Section 3.2.3, these messages are automatically re-transmitted if lost.

xDaemon Instance Failure For a node-wise query, if the target xDaemon instance fails before or during query handling, then this query is aborted and the client library returns a failure to calling code.

During handling of a collective query, one or small number of xDaemon instance failure is tolerable. If one xDaemon instance fails, the client library senses this by resending the query request several times. With no response, the client library removes this xDaemon instance from its xDaemon list, and does not communicate with it any more. The ConCORD memory content tracer is by nature not exactly accurate about the memory contents existing in VMs. Therefore, missing query replies from one or small number of xDaemon instances only possibly adds more inaccuracy to the final query results.

6.4 Evaluation

In this section, I will present my evaluation on ConCORD's query handling. I mainly focus on measuring the response times for different types of queries. Response time measures the average latency that is needed for a service application program to get a response from

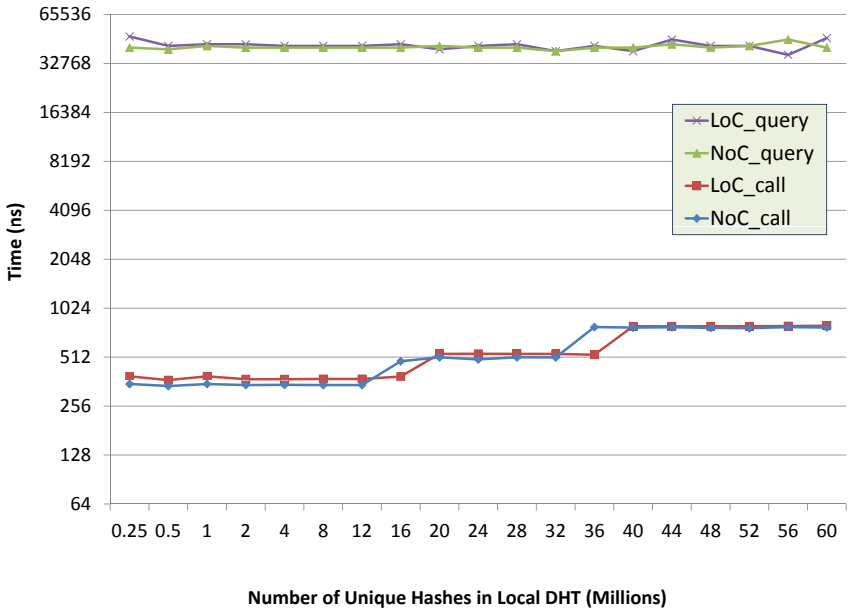


Figure 6.2: Response time of node-wise queries, as local function call from the same process, or a local query from a process residing on the same host machine.

a query function call. I have measured and reported the average response time by running the same queries 1000 of times, and averaging their response time for each test.

I performed all of the tests in this section on a cluster with 20 nodes. Each node is equipped with two Dual Core Intel(R) Xeon(TM) 2.00GHz CPU, 1.5GBytes RAM and 32GB disk, a Broadcom NetXtreme BCM5703X 1Gbps Ethernet card. The nodes are connected through a 100 Mbps Ethernet switch.

6.4.1 Node-wise query

First, we consider the node-wise query. I have measured both kinds of node-wise query: *NoC* and *LoC*. A node-wise query comprises only a two way network communication between the client and an xDaemon instance. Therefore, its response time is mainly de-

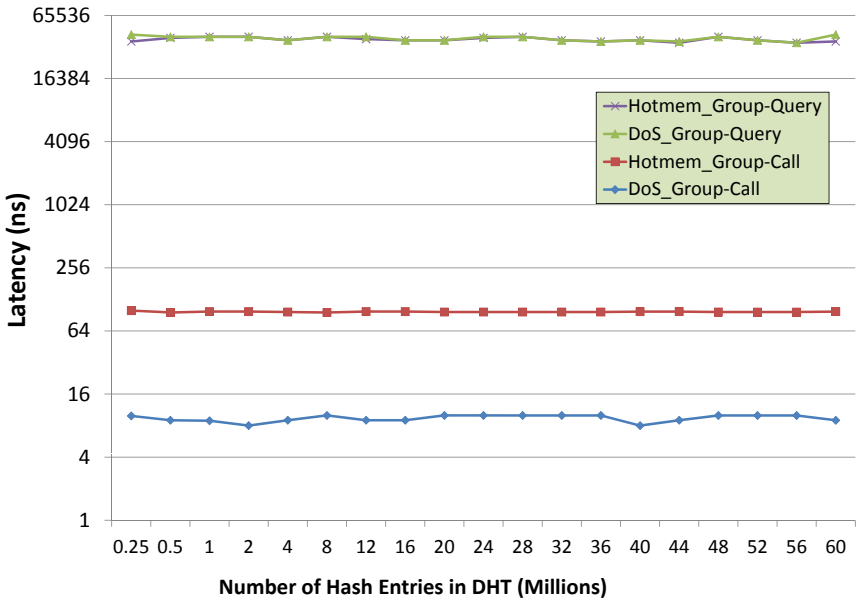


Figure 6.3: Response time of collective queries on a pre-registered VM group, as a local call inside xDaemon instance, and a local query from user library residing on the same host machine.

termined by the latency of small packets on the network. To determine the response time without the impact of network latency, I have measured the latency of these two types of queries in three scenarios:

- *function call in xDaemon*: In this scenario, I measured the response time of the actual node-wise query handling function inside the xDaemon instance. This measures the actual time needed to handle a node-wise query request inside xDaemon. For node-wise query, this is the time spent by the xDaemon instance to query a content hash in the local hash table and generate the disjoint VM set. The results are shown in Figure 6.2, labeled *LoC-call* and *NoC-call*.

- *query in local node*: In this scenario, I run the query library within an process on the same physical node as its target xDaemon instance. The response time of a query in this scenario is mainly determined by the network latency through the loop network device. The results are shown in Figure 6.2, labeled as *LoC-query* and *NoC-query*.

- *query in remote node*: In this scenario, the query library is running on separate physical node with its target xDaemon instance. The query response time in this case is largely determined by the network latency. The results are shown in Figure 6.3.

Figure 6.2 shows the response times for node-wise queries measured as either a local function call or a local query, with different numbers of content hash entries stored in its target xDaemon instance. We can see that response times for both queries are relatively stable, independent of the number of entries in the hash table.

Meanwhile, figure 6.3 shows the response time of node-wise queries on a system with increasing number of hash entries split into various number of xDaemon instances running, while the number of hash entries in each instance keeps constant. The results are as expected, the response time for both queries are stable as the number of‘ xDaemon instances increases. This is because ConCORD employs zero-hop DHT. This enables always only two parties are involved to hand a node-wise query, client library and target xDaemon instance, no matter how many xDaemon instances running in the system.

6.4.2 Collective query

I have measured two types of collective queries: 1) a query for the degree of sharing (*DoS*), and 2) a query for the number of hot memory blocks (*Hotmem*).

I am interested to see how fast the collective query is handled on a single xDaemon instance. I measured this by starting ConCORD on a single physical node with only one xDaemon instance running, and sending a collective query to this single xDaemon in-

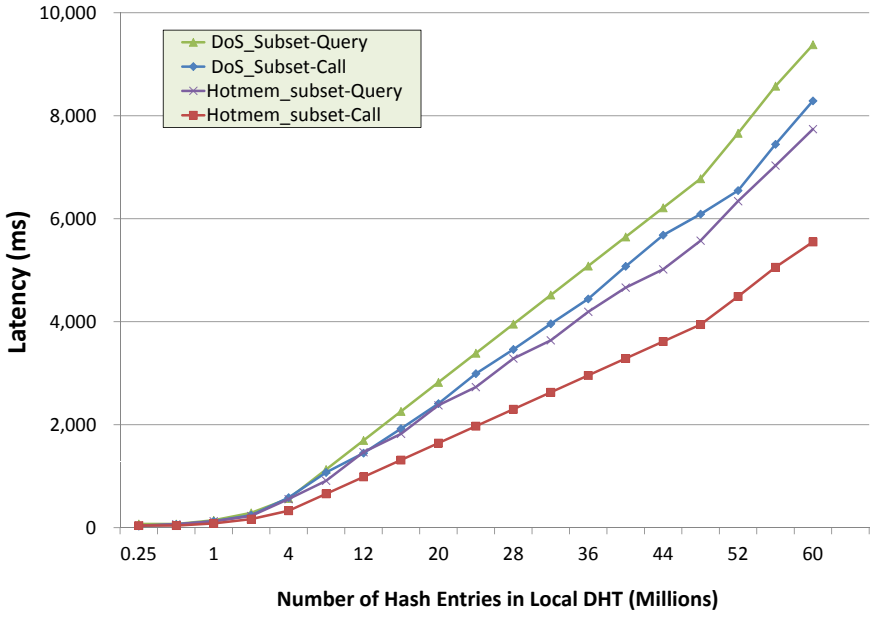


Figure 6.4: Response time of collective queries target an arbitrary set of VMs, as a local function call from the same process and a local query from a process residing on the same host machine.

stance. The hash table in the xDaemon instance is filled with different number of unique content hashes in each test. Figure 6.4 shows the response times for these collective queries targeting an arbitrary set of VMs, which means the targeting VM set has not been registered with ConCORD. The response times in the figure are measured as the local internal function call (same as in node-wise query measurements) and as the local query from running client library in the same physical node.

The figure shows: 1) the response time of a collective query on single xDaemon instance is almost linear with the size of its hash table, 2) the response time of a collective query is of 3 orders of magnitude higher than the network latency. Most of time is spent on xDaemon to iterate through its entire hash table. This is because the query targets an arbi-

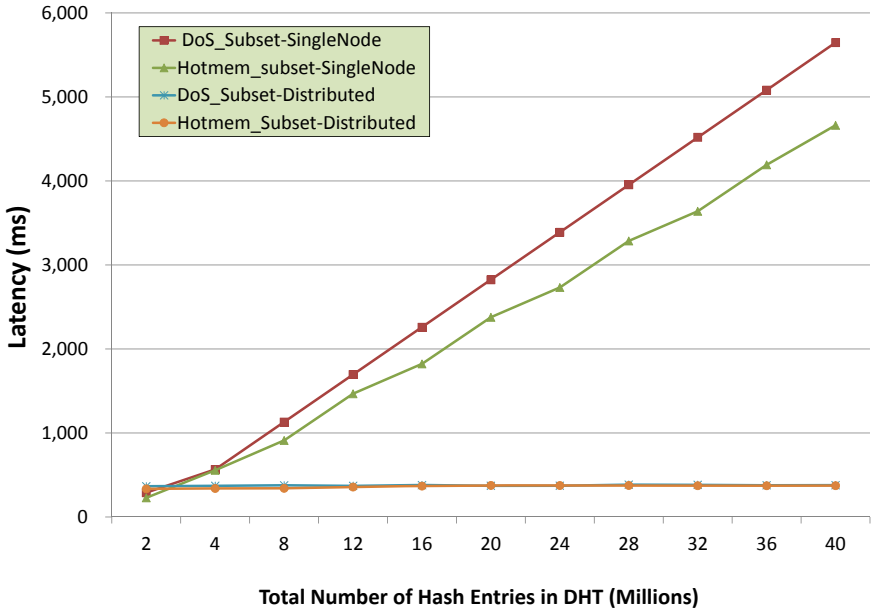


Figure 6.5: Response time of collective queries on arbitrary VM set, as ConCORD is running on single node or running in a number of distributed nodes. In distributed approach, the number of hash entries in each xDaemon instance is keeping at 2 million.

trary set of VMs, and so xDaemon does not have any pre-computed results. All information has to be parsed and collected during the query execution, which is very time-consuming.

One way to reduce this response time is to distributed these hash entries over more xDaemon instances, and let each xDaemon instance perform part of query in parallel. Figure 6.5 compares the response times of the same query on ConCORD using two strategies: 1) ConCORD uses one single xDaemon instance to store all content hashes and handle query, 2) ConCORD distributes the same number of content hashes into multiple xDaemon instances to keep the hash table in each instance at a fixed size. The result shows that this distribution strategy can provide good scalability since response time stays constant as the number of total hash entries increases.

With the distribution strategy, the collective query can achieve a good scalability with constant response times as the number of content hashes in the system increases. However, response time is still large (around 300 ms). As we have discussed, the primary reason is because this query is targeted to an arbitrary and unregistered VM set, so no information about this set of VMs exists prior to the query. Instead, ConCORD allows a set of relevant VMs be registered as a VM group in advance. ConCORD pre-computes and maintains the content sharing information for each of these registered VM groups view. In this case, when a collective query arrives, the xDaemon instance does not have to parse its hash table at query time, all information is ready instead. This can significantly reduce the response time for collective queries. As we can see in Figure 6.3 response times for collective queries on registered VM groups is 3 to 4 order of smaller than these targeting to arbitrary VM sets. The time for xDaemon instance to handle the query in this case is very small (*DoS-Group-call* and *Hotmem-Group-call*), only around 10-70 nanoseconds.

Finally, we compare response times of collective queries using different communication strategies: broadcast reduction and tree reduction, with ConCORD running on a different number of nodes in each test. The result is shown in Figure 6.6. From the results, we can see tree reduction is not better than broadcast reduction up to 20 nodes, although the trend seems to suggest that if the number of nodes increases, tree reduction may perform better than linear reduction. But we have no idea of this beyond 20 nodes yet.

6.5 Conclusion

To allow service developers to easily ask questions about content sharing existing in the system, ConCORD exposes a content-sharing query interface. They can check both the amount of memory content sharing that exists, and examine the actual shared content. I have described this query interface in this chapter. I started with the definitions of different

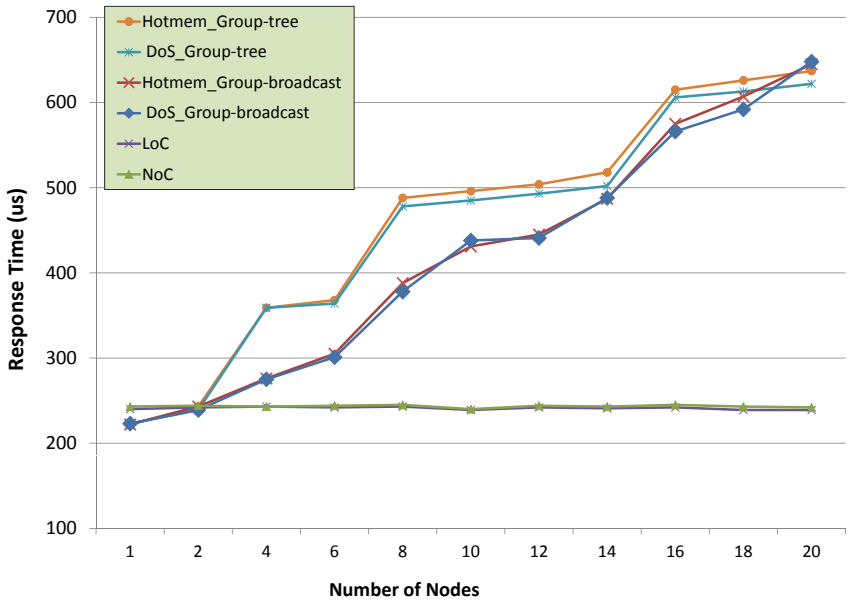


Figure 6.6: Response time of node-wise and collective queries as the function of number of ConCORD nodes. Collective queries are targeted to a pre-registered VM group, by using both broadcast reduction and tree reduction communication strategies. Each physical node runs 2 VMs, and each VM is allocated with 4GB memory. The query client is running on a separate physical node.

types content-sharing queries supported in ConCORD, followed by a discussion on the design and implementation of them in ConCORD. Finally, I presented my evaluation of ConCORD's query interface.

Chapter 7

Content-aware Service Command

Content-aware service command (*service command* for short) is a service model and associated implementation that enables effective construction of content-aware services in large-scale parallel systems. It provides a simple and powerful interface that enables many HPC services to effectively exploit memory content redundancy existing in the system. Content-aware services built on top of service commands are automatically parallelized and executed on the parallel systems. The service command execution system takes care of the details of partitioning of the task, scheduling subtasks to execute across a set of machines, and managing all of inter-node communication. This enables service application programmers to build and enhance their services to maximally exploit and utilize the memory content sharing without worrying about the complication of the implementation.

In the rest of this chapter, I will first introduce the content-aware service command with a detailed description of its configuration options. Then I will present a brief tutorial with an example to explain how easily we can build a content-aware service using the service command. Furthermore, I will describe my implementation of the service command execution system in ConCORD, followed by an example to show the execution of a service command. Finally, I will describe my evaluation of the service command execution system and conclude the chapter.

7.1 Model and Introduction

A content-aware service command enables a collective and parallel execution of a specified service by disassembling the service into a series of specific operations and applying them on each memory block with distinct content among all serviced nodes or virtual machines. For example, checkpointing of a machine can be completed by copying each memory page that contains distinct content in the machine's memory into some reliable storage. Migration of a virtual machine can be reduced to transferring each memory block containing distinct content to the VM's new host.

A content-aware service command comprises a set of service-specific parameters and a set of service operations (call-back functions). The parameters are used to configure the subjects of a service, such as virtual nodes that this service is applied on, and virtual nodes that can contribute to speed the service, the operation mode of the service, etc. The set of service operations are the actual places that define service-specific operations that will be applied to each individual memory block with distinct content existing in any serviced machines' memory. This set of operations will be performed by ConCORD's service command run-time system during the distributed execution of the service command.

A command execution comprises of two phases: a collective phase followed by a local phase. During collective phase execution, each memory block with distinct content existing in participating nodes' memory is applied on service-specific operation. The operations performed on different memory blocks are parallelized in all nodes. When the collective phase ends, all results that have been done so far are collected and sent to each SVM. Therefore, during local phase, each node go through all blocks in its memory and identify all unfinished memory blocks, and perform the operation on these blocks locally.

7.1.1 Service command parameters

A set of parameters can be configured for a collective command. These parameters are described as follows.

- *Service Virtual Machines (SVMs)*: A set of virtual machines that this command is applied on, i.e, this is the virtual machines on which this service is going to be performed during this time of execution. For example, the SVMs of a service command performing checkpointing are those virtual machines that are going to be checkpointed.

- *Participating Virtual Machines (PVMs)*: A set of virtual machines that are going to participate and contribute to speed the execution of the service. For example, The PVMs of a checkpoint service are those virtual machines that are not checkpointed, but that share some memory contents with the checkpointing VMs. They can help to save some memory blocks during the service which may speed the execution of the service.

- *Service Mode*: The operation mode for a service command, which could be either interactive mode or batch mode. In general, a service command executed in interactive mode completes the whole service during its execution, while a command executed in batch mode only computes a roadmap directing VMs to perform the service, without actually performing it.

- *pause-SVM*: This parameter determines, during execution of the service command, whether SVMs should be paused. This option is critical to some services. For instance, during checkpointing, a VM usually has to be paused to guarantee the saved checkpoints are consistent with each other.

- *pause-PVM*: This parameter determines, during execution of the command, whether the PVMs should be paused. This option is similar to *pause-SVM*.

- *Timeout*: Maximum time that this command is allowed to spend on the collective phase. As I will explain in the following, an execution of a service command comprises two phases: a collective phase and a local phase. Collective phase is the phase in which the service-specific operations are being applied on memory blocks in parallel on all PVMs. This parameter is used to control the maximum time that could spend during this phase before local phase execution begins.

- *Service-data*: This parameter is used for service-specific data. This data is considered private data by the service command, and is stored by the service command execution system as a data blob. Later, the data is supplied to all service operation function calls. This enables service-defined data to be shared by operation functions.

7.1.2 Service operation functions

In addition to a set of parameters, a set of operation functions can be specified in a service command. These functions will be called at different phases during the execution time of the command by the command execution system. These operation functions are the core part of the service command. They are the actual places to define service-specific operations to be applied on the individual blocks of distinct memory content existing in SVMs' memory.

The set of operation functions comprises several defined interfaces by the service execution system. Service developers implement these functions to use them in service command system. They are called in the context of user space, which is inside the service command execution engine running as part of an xDaemon instance, or in kernel space, which is running inside the command execution agent as part of the VMM. The functions

are as follows.

- *parse-input*: This function is called once on the service command system to parse the service specific configuration file. The function is supposed to read the service specific configuration file, parse all configurations and convert them to a string and attach it to the service command's private data field. Then xCoord sends this private data along with other parameters for a service command to ConCORD's service command execution system when this service starts.

- *collective-start*: This function is executed exactly once on each xDaemon instance by the service command execution engine and the VMM execution agent for each SVM and PVM prior to collective phase in the execution of the command. This is usually the place to allocate and initialize resources that are needed by the following operations during the collective and local phases. For example, in checkpointing service, the checkpoint file is created and opened in this function for subsequent writes of memory blocks.

- *collective-cmd*: This is the core function defined for the collective phase in the execution of the command. This function is called once on each memory block with distinct memory content existing in SVMs during collective phase. The function call is executed inside the VMM's execution agent. This is the place to apply a service-specific operation on each memory block with distinct content in SVMs. Note that the operation may be applied more than once on a specific block of same content. Therefore, the operation inside this function must be idempotent, i.e, applying the operation on the same object more than one time generates the same effects as applying the function on the object exactly once.

- *select-vm*: For each distinct memory content, there may exist many VMs that have

memory blocks with this content. The service command execution system has to choose one or more VMs to perform the actual operation defined in the *collective-cmd* function during the collective phase. The *select-vm* function gives service developers the option to select one or more specific VMs to perform the collective operation on each block of distinct content. If this function is not supplied, the service execution engine randomly chooses one from all VMs which have the blocks.

- *collective-finalize*: This function is called and executed exact once on by the command execution engine in each xDaemon instance and for each SVM and PVM by VMM's command execution agent, at the end of the collective phase and before the local phase of the command's execution. This is usually the place to reduce and gather results from work that has been done during the collective phase, and clean up and free resources that are unneeded anymore in the local phase. This is also a barrier, all calls to *collective-cmd* on every VM will be guaranteed to have returned before this function finishes.

- *local-start*: Similar with *collective-start*, this function is executed exactly once on each xDaemon instance and VMM execution agent for each SVM (PVMs are not involved in local phase execution of a service command). This function is called after *collective-finalize*, and indicates the start of the local phase execution. This is usually the place to allocate and initialize resources to be used during local phase execution on each VMM instance.

- *local-cmd*: This function is the place to define operations that are to be performed on each memory block (not just those with distinct content) in SVMs. The memory block given to the function is marked with whether at least one memory block with the same content as this block has had *collective-cmd* successfully applied to it during the collective phase. If it has been successfully applied, the return argument

from the collective call for this memory block is also supplied. This allows the local phase to skip or augment work done by the collective phase.

- *local-finalize*: Similar to *collective-finalize*, this function is executed exactly once by each xDaemon instance and VMM execution agent for each SVM at the end of local phase execution. This is usually the place to clean up service specific resources allocated and used during the local phase or the entire command execution, e.g, closing opened checkpoint files or closing network connections for transferring data.

- *parse-ret-arg*: This function is called once on xCoord and supplied with the service specific data returned from the execution of a service command. The function is supposed to parse the service-specific return data and generate any output from it, for example, to print errors.

A service command is not required to implement all of these functions. Each service command may specify a subset of them. Furthermore, as we will see in following, if a service command is set to execute in batch-mode, none of these functions are executed during the execution of the service command.

The actual signature for each of the operation functions are defined as follows using the C language.

Service operations prototypes

```
struct xcord_service_ops {
  int (*parse_input)(struct xcord_service * xservice,
                     char * input_file);

  int (*collective_start)(uint8_t component_type,
                          uint16_t c_id,
                          struct xcord_hash_list * hashes,
                          void ** arg,
                          uint16_t * arg_size);
```

```
  int (*collective_cmd)(uint16_t vm_id,
                        struct xcord_hash_item * hash,
                        char * mem_block,
                        int block_size,
                        void ** arg,
                        uint16_t * arg_size);

  int (*select_vm)(struct xcord_hash_item * hash,
                   uint16_t daemon_id,
                   struct vm_set * vm_lst,
                   void ** arg,
                   uint16_t * arg_size);

  int (*collective_finalize)(uint8_t component_type,
                             uint16_t c_id,
                             struct xcord_hash_list * hashes,
                             void ** arg,
                             uint16_t * arg_size);

  int (*local_start)(uint8_t component_type,
                     uint16_t c_id,
                     struct xcord_hash_list * hashes,
                     void ** arg,
                     uint16_t * arg_size);

  int (*local_cmd)(uint16_t vm_id,
                   struct xcord_hash_item * hash,
                   uint32_t page_no,
                   char * mem_block,
                   int block_size,
                   void ** arg,
                   uint16_t * arg_size);

  int (*local_finalize)(uint8_t component_type,
                        uint16_t c_id,
                        struct xcord_hash_list * hashes,
                        void ** arg,
                        uint16_t * arg_size);

  int (*parse_ret_arg)(void * ret_arg, uint16_t arg_size);
};
```

7.1.3 Operation modes

A content-aware service command can be configured to execute in one of two operation modes: *interactive mode* and *batch mode*. In general, in an execution of a service command in interactive mode, all service-specific operations are applied on specified memory blocks during the execution of the command. The callback functions are invoked and then the service defined task is actually complete when the service command finishes. On the other hand, in batch mode, instead of applying the service-specific operations during the execution of the command, an execution plan or roadmap is generated. This roadmap gives a step by step outline to direct each node or VM to apply different operations on their local memory blocks. By completing all these steps on each node, the defined service finishes. By separating out the plan creating, it is possible to reuse the plan or further optimize it.

Interactive-mode

A service command executed in interactive-mode performs the actual service-specific operations on memory blocks during its execution. The command execution system takes care of scheduling each individual operation, arranging some nodes to perform it and guaranteeing all required operations are correctly performed. This simplifies the service developers' work. To create a content-aware service, service developer simply create and configure a service command, then start its execution by ConCORD. Once the execution finishes, the defined service is completed. To build a service using interactive-mode, service developers do not have to consider any implementation issues regarding how to scalably parallelize the execution of operations on a large number of nodes, how to schedule these operations, manage data communication, or how to keep data consistent. All of these are taken care of by ConCORD's command execution system.

However, there are a few limitations for a service command executed in interactive

mode. First, to enable the command's execution in interactive mode, the service specific operation to be applied on each individual memory block are required to be idempotent. This is because in execution of a service command, the service-specific operation is guaranteed to be applied on each memory block with distinct content at least once, but not exactly once. The same operation could be applied to the same memory content more than once. The idempotency of the operation allows it to be applied on the same object multiple times without any effect than applying it exactly once. For example, during checkpointing service, each memory block with distinct content is saved to reliable storage at least once, but some memory blocks could be saved more than one time, which means the same memory content could be appeared in multiple places in the final checkpoint file. This does not impact the correctness and consistency of a checkpoint, however.

In addition, during the execution of a service command, the service-specific operations are performed on all required memory contents in parallel, which requires that the operation be able to be performed independently. Furthermore, the order in which the operation is performed on different memory contents is not guaranteed. If a service requires some operations to be applied on memory contents in a specific order, execution of the command in an batch-mode may be a better option.

Finally, executing a command in interactive mode may not be always the most efficient way. Since ConCORD does not have the best knowledge of underlying system infrastructure or architecture, it is difficult for it to choose the best system-specific schedule for individual operations. System-specific global scheduling strategies may better utilize the networking and system resources or achieve a better load balancing, etc. If a service developer thinks he knows more about the system and can do better, then batch-mode execution also him the option to do so.

Batch-mode

As we have discussed, the execution of a service command in batch mode does not actually apply the service-specific operations. Instead the execution system just collects the memory content information and returns a roadmap showing the list of operations that need to be performed. A example of such roadmap may look like: "... step 21: write memory page in address xxx on VM-2 to file checkpoint.2, step 22: write memory page in address yyy in VM-3 to checkpoint.3, etc". The service can then schedule the execution of these steps, maintaining their dependencies and making sure all required operations are completed.

The advantages that the batch-mode execution of a service command enables include:

- It allows service to organize and schedule the operations globally. This leads to be able to more efficiently utilize system and network resources, or to generate better load balance across different machines.
- It allows service to perform operations in a specific and customized order, or to follow some customized scheduling strategy.
- It enables more options to allow service to pursue better flexibility and efficiency if they have better knowledge of the system topology and other dynamic load information than ConCORD has available.
- It enables service applications to make sure the operations are applied on each memory content exactly once, which releases the requirement that operations have to be idempotent as required in interactive-mode.

However, batch mode requires more implementation efforts on service. Service developers have to handle more complex situations, such as data inconsistency since the data may not be available in some destination anymore when service is performing the operations.

7.2 Tutorial for building content-aware service

In this section, I will present a simple tutorial for how we can build a content-aware service using the service command. The general steps of using the service command model are discussed briefly, followed by an example with step-by-step descriptions for building a collective VM reconstruction service, in which multiple VMs contribute to speed the migration of a single VM by sending the shared memory content to the destination host in parallel.

7.2.1 General Steps

To build a content-aware service, we need to define a service command, with appropriate parameters, and implement the set of operation functions.

First, to be able to apply a content-aware service command, your service should be mainly performing some operations on memory content, and this operation can be performed on each of memory block independently. To configure a service command, you will first define the parameters for the command. At minimum, you need to determine the SVMs and PVMs. PVMs may share some memory content with SVMs. You can examine which VMs share more memory content with SVMs through natural knowledge, for example, that this set of VMs is running the same applications or closely related applications. Or you can determine it through ConCORD's content-sharing query interface.

The core part of configuring a service command is to implement all or a subset of the operation functions, particularly the *collective-cmd* and *local-cmd* functions.

The following list summarizes the general steps to build a content-aware service using service command.

- step 1: Determine your set of service VMs (SVMs).
- step 2: Determine your set of contributing VMs (PVMs).

- step 3: If any service-specific resources are needed during the collective phase, allocate and initialize them in the `collective-start` and deinitialize and release them in the `collective-finalize` function.

- step 4: Determine the operation that needs to be performed on each individual block with distinct memory content in parallel by all VMs during the collective phase. Next, determine whether any service-specific data has to be passed to *local-cmd* after the collective phase is done. Then implement these in the `collective_cmd` function.

- step 5: Similarly, if any service-specific resources are needed during the local phase, allocate and initialize them in the `local-start` function and deinitialize and release them in the `local-finalize` function respectively.

- step 6: Determine the operation that needs to be performed on each memory block in each SVM, and implement it in the `local-cmd` function.

7.2.2 Example

In this section, I will show a step-by-step example of building a collective VM reconstruction service on top of the content-aware service command.

Considering the migration of a virtual machine across hosts. The conventional way is to copy all memory pages of the migrating VM from source host to destination host, in addition to other smaller data representing the status of the VM (*non-memory state*), such as the values of registers, device status, etc). When the destination host receives all of this data, it is able to reconstruct the VM and resume it locally. Such a migration strategy can be described as the pseudo-code in Algorithm 1.

During this migration process, the most time-consuming part is copying the VM's entire memory content across physical hosts. Suppose there are some other VMs running

Algorithm 1 VM Migration

1: connect to *remote host*
2: $N \leftarrow$ Number of memory pages in the VM
3: **for** $pageno := 0$ to $N - 1$ **do**
4: send $(pageno : memory[pageno])$ to *remote host*
5: **end for**
6: send *non-memory state* to *remote host*
7: close network connection

on the hosts on the same network as the destination host, and many of these VMs share some memory content with the migrating VM. Then these VMs can contribute and possibly speed this migration process by sending their shared contents to the destination host in parallel. These VMs would be PVMs, while the migrating VM being SVM. The idea here is that by receiving memory contents from multiple PVMs, the destination host is able to gather and reconstruct SVM's memory faster than from a single source. Perhaps this would let you "dead-migrate" a VM with multiple sources as fast as live-migrating it from a single source. Or perhaps the source and destination hosts are connected over a very slow network, but the other participating VMs are on the same network as the destination, and thus you would be able to transfer the VM much faster using the participants than copying it over the slow network from the source.

It would be hard to build such a VM migration strategy without knowledge of the memory content sharing across all participating VMs in different physical hosts. Even with this information on hand, it would be still complicated to implement such a migration protocol as it has to synchronize data transition across all sending hosts properly to minimize the duplicated memory contents transferred through network.

With the content-aware service command model, these implementation efforts could be separated from service developers. Instead, they would be handled by the service command execution system in ConCORD. The service developers' can then focus on implementing

the service-specific part of these operations needed to achieve VM migration.

Let us take a step by step to build such a collective VM reconstruction service using the content-aware service command model. This will help to better understand the service command model. First, the set of SVMs has to be determined. In our example, there is only one VM, which is the migrating VM. Then, we have to choose a set of PVMs. This can include any VMs that share some amount of memory contents with the migrating VM. You can examine this sharing information using ConCORD query interface as I have described in Chapter 6. Another factors to consider in adding a VM to the set of PVMs include the loads of its host and the network connection conditions between its host and the destination host.

Next, we have to choose the operation mode for the service command. In this example, we will use interactive mode, which means after the service command completes, the VM's complete state will have been migrated to the new host. In addition, in this example, we have to specify the destination host information as the service's private data. This enables all VMs to be able to make a connection to the destination host and send data to it. Finally, what is left is to implement the set of service operation functions.

Since the destination host information is required for all VMs, we implement the `parse_input` function, which takes the input of the host information and converts it to a blob of string data and attaches it as the private data for this service command. The pseudo-code is shown in Algorithm 2.

Algorithm 2 parse-input

1: Read *host info* from text file or other input
2: Convert it to a string of <*ip:port*>
3: Attach the string as the private data of service command

Next, we implement `collective_start`, the function that will be called in all xDaemons instances and VMM execution agents for each PVM and SVM. Before VMs can

send memory content through the network, each VM instance has to initiate a network connection to the destination host. This initiation work is performed in `collective_start`. In addition, the destination host also needs some information to reconstruct the migrating VM's memory using the received memory blocks that will come from multiple sources. To do so, the migrating VM sends a mapping table, which maps each content hash to one or more memory pages' offsets in the source VM. Given this mapping table, the destination host can map each received page using its content hash to find its locations in the VM's memory. The final pseudo code for `collective_start` is shown in Algorithm 3.

Algorithm 3 collective-start

```
1: connect to remote host
2: if this is a SVM then
3:     send MapTale(hash → memory offset) to remote host
4: end if
```

`select_vm` is used to specify how the execution system chooses a VM from all of those which may have the memory blocks with the required content in collective operation. If this function is not implemented, the execution system randomly picks one. Here, random picking should be fine for us, and thus we do not implement `select_vm`.

`collective_cmd` is the core part of any service command for performing operations on each memory block with distinct content. In our VM migration example, this operation would send a memory block to the destination host. During the collective phase execution, each VM (either SVM or PVM) is selected by command execution engine to transfer memory block with specific content. If a block with the given content (identified by the content hash) does not exist in the selected VM's memory, then the `collective_cmd` returns NOTCOMPLETED to indicate it is incapable of completing the operation on this memory content, which suggests the command execution engine to choose other VMs. The pseudo code is shown in Algorithm 4.

Algorithm 4 collective-cmd

```
1: if a memory block with given memory hash exists in local VM then
2:     send the memory block to remote host
3:     return COMPLETED
4: else
5:     return NOTCOMPLETED
6: end if
```

In `collective_finalize`, we close the network connections created in all the PVMs since PVMs will not participate in the local phase execution. However, network connection in the migrating SVM will still be used during the local phase execution, thus we keep it open. The pseudo code is shown in Algorithm 5.

Algorithm 5 collective-finalize

```
1: if this is a PVM then
2:     close network connection to remote host
3: end if
```

There is nothing to do in `local_start` for our migration service.

Finally, the migrating VM transfers those memory pages that have not been copied to the destination during the collective phase. For example, some of memory pages may just be updated in the migrating VM and have not yet been perceived by ConCORD. This is critical step for guaranteeing that all memory pages with distinct content are copied to the destination, even these pages with new contents that ConCORD is unaware of.

During local phase execution, the VMM's command execution agent goes through each memory page of the SVM and calls `local_cmd` on each. Each page is marked as `COMPLETED` if a memory page with the same content has been successfully executed in `collective_cmd` i.e, it has been already sent to destination during collective phase. Otherwise, the memory page is marked as `NONCOMPLETED`. Thus, in `local_cmd`, we send all memory pages that are marked as `NOTCOMPLETED`. The pseudo code is shown in

Algorithm 6.

Algorithm 6 local-cmd

1: **if** *block* is *NOTCOMPLETED* **then**
2: send <*hash:memory-content*> to *remote host*
3: **end if**

The last thing to do, in `local_finalize`, the migrating VM has to send the non-memory state to destination host. Then the network connection to the destination host is closed. The pseudo code is shown in Algorithm 7.

Algorithm 7 local-finalize

1: send *non-memory state* to *remote host*
2: close network connection to *remote host*

After the operation functions are implemented, we need to compile these functions and register them to ConCORD. Current implementation of ConCORD only allows static linking of the operation functions. Once there are new service operation functions registered, ConCORD has to be stopped and rebuilt with these functions. To support dynamic run-time addition of service implementations will be one of the possible future works.

Registering a service operation function set is pretty simple. Sample code is shown as follows. The `xcord_register_service_ops` macro takes a service operation structure and a function set name, and registers it with ConCORD. Later when a service command is configured, a function set name can be specified, and ConCORD searches for the actual operation function structure and uses it in the execution of service command.

```
struct xcord_service_ops migration_ops = {
  .parse_input = f_parse_input,
  .collective_start = f_collective_start,
  .collective_cmd = f_collective_cmd,
  .collective_finalize = f_collective_finalize,
  .local_start = f_local_start,
  .local_cmd = f_local_cmd,
```

```
  .local_finalize = f_local_finalize,
  .parse_ret_arg = f_parse_ret_arg,
};

xcord_register_service_ops("migration",
                           &migration_ops)
```

In this section, I have described the general steps to build a content-aware service using the service command model, with a step-by-step description of how we can build a collective VM reconstruction service as an example. This helps us to gain a more intuitive understanding of how service command model works. In Chapter 8, I will present a working example of another content-aware service, a content-aware group checkpointing, with a detailed implementation and evaluation, to further demonstrate the power of the content-aware service command model.

7.3 Implementation

In this section, I will describe how the service command framework is implemented in ConCORD. The interfaces for using the content-aware service command framework in ConCORD include a user library and a service command terminal.

7.3.1 Service command library

One interface to operate service commands in ConCORD is implemented as a user level library. Similar to the content-sharing query library, the library can be linked to any application. The service command library exposes functions that allow services to create, configure and execute a service command.

The core functions in the service command library include:

```
int xcord_service_execution_env_init(char * xcord_serv_ip,
                                     uint16_t port);
```

```
struct xcord_service_cmd *
xcord_create_service(char * service_name, char * ops_name,
                     struct vm_set * svm,
                     struct vm_set * pvm,
                     uint16_t timeout,
                     void * arg,
                     uint16_t arg_size);


struct xcord_service_cmd *
xcord_create_service_from_file(char * conf_file);

int xcord_execute_service_cmd(struct xcord_service_cmd * command,
                              int block,
                              int (*call_back)
                                (struct xcord_service_cmd * command)
                             );

int xcord_service_status(struct xcord_service_cmd * command);

void xcord_free_service_cmd(struct xcord_service_cmd * command);
```

A service command structure can be created either by given parameters, or through a service configuration file. Then a call to `xcord_execute_service_cmd` starts the execution of the command. This call could be either in a blocking or unblocking mode. In blocking mode, the call to `xcord_execution_service_cmd` blocks until the service command completes, fails or aborts. Nonblocking mode allows the call to return immediately. To allow the program to track the execution status of the service command, a callback function can be specified. The function is called whenever the service command completes, fails or aborts. In addition, `xcord_service_status` is provided for program to check the execution status of an executing service command.

The library can be initialized and used on multiple physical nodes inside different applications simultaneously. Each library maintains the status of its local service commands and controls the communication with the ConCORD command execution engine independently. This allows multiple services to be started and executed in parallel in the system.

7.3.2 Service command control terminal

In addition to the library, a user facility, the service command control terminal, is also provided to allow administrators to conduct the creation and execution of a service command directly by command line input. The control terminal is a simple command line interface. It allows administrators to create, configure, execute and check the status of executing service commands. The following statements are supported in the terminal.

- *service create* service-name service-conf-file: create a service and configure it using local configuration file.
- *service profile* service-name: check the configuration of a local registered service command.
- *service status* service-name: check the execution status of a service command.
- *service start* service-name: start an execution of a service command.
- *service abort* service-name: abort an execution of a service command.
- *service remove* service-name: remove the service command.

Similar to the user library, multiple control terminals can be started on different physical nodes simultaneously. Each terminal maintains its local service commands and communicates with ConCORD independently. This allows system administrators to start a control terminal in any machine they want, as long as the machine is able to connect with all ConCORD instances.

7.3.3 Execution overview

The major ConCORD components that are involved in the execution of a service command are described below.

- The Service Command Coordinator (*xCoord*) is the initiator of a service command, e.g, the service command library running inside a service application, or the service

command control terminal. xCoord synchronizes with all other execution components during different execution phases.

- The service Daemon (*xDaemon Instance*): The service command execution engine is running as part of each xDaemon instance. During the execution of a service command, the xDaemon instance uses the content sharing information from its local part of DHT to direct the distribution of collective operations to SVMs and PVMs.

- Service Virtual Machine (*SVM instance*): SVMs are VMs that the service command is scoped to. A service command execution agent is running inside each VMM instance. It performs the actual operations on memory blocks from the VM during collective and local phase execution. When we refer to *SVM instance*, we mean the execution agent processing the memory of that SVM. If there are multiple SVMs in one physical node, the VMM actually starts multiple execution agents (SVM Instances), each for one SVM.

- Participating Virtual Machine (*PVM instance*): Similarly to an SVM instance, when we refer to a *PVM instance*, we mean the execution agent in the VMM running for that PVM.

- *VM Instance*: This refers to either an SVM or PVM instance when there is no need to differentiate them in the discussion.

An execution of a service command comprises two phases: the collective phase and the local phase. The following describes the flow of a service command execution.

- When a service command is required to start from a client's library (xCoord), xCoord sends the parameters along with its operation function name to all xDaemon instances, SVM instances and PVM instances. Each of these instances saves and

registers this service locally. Then each xDaemon instance generates a list of unique content hashes existing on any SVMs according to its local part of DHT.

- Once initialization of the service command in the xDaemon instances is complete, xCoord notifies all execution components to start the collective phase execution of the command. During collective phase, each xDaemon instance sends a list of content hashes to each PVM and SVM, and requires the VM to perform *collective-cmd* on their local memory blocks with the given content hashes. For each content hash, the VM can either find a local memory block with the same content and perform the collective operation on it, or fail to find such a memory block and notify the xDaemon instance. If this content hash cannot be completed by a VM, the xDaemon tries to get other VMs which it thinks also have a copy of the block to perform the operation. The *collective-cmd* is applied on memory blocks with distinct content in parallel in all SVMs and PVMs, where the xDaemon instance tries to distributed tasks evenly to each VM instance.

- At the end of collective phase, each xDaemon instance generates a list of content hashes for which the collective operations have been successfully performed so far. This list of content hashes is sent to each SVM. Each SVM receives such a list from all xDaemon instances, and merges them together to form a final completed hash list.

- During the local phase of execution, each SVM parses its local memory, compares each page's content with the received completed hash list. It marks a page as completed if at least one block with same content has been successfully executed in the *collective-cmd* during the collective phase. Otherwise it marks the page as not-completed. Finally, a service-defined local operation is applied to each local page,

with the completeness of the page also provided. This *local-cmd* performs different operations on a page depending on its completeness.

To summarize the execution of a service command, during the collective phase, xDaemon instances essentially send out individual memory block jobs to different VMs (both SVMs and PVMs), and collect the return results. When the collective phase ends, xDaemon instances send all results that have been done so far to each SVM. Therefore, during local phase, each SVM can identify all unfinished works and perform them locally.

7.3.4 Execution details

A detailed execution flow with the functions performed by each component during each phase of execution is described below.

Initialization

A service command coordinator (xCoord) is the component that synchronizes the execution steps between the collective phase and the local phase. Usually, the node running the service command library or the service control terminal functions as the xCoord. When xCoord starts, it gains host information about all xDaemon and VM instances running in the system from ConCORD's management node.

xCoord Once a service command is created locally in xCoord, it is stored in xCoord's local commands' database. Once a service command starts to execute, either by calling `xcord_execute_service_cmd` from the library function, or typing `service start` command in service control terminal, xCoord sends the "service start request" (`SERV-START`) to each PVM/SVM instance and xDaemon instance. All parameters of the service command are associated with the request. `Service-start-request` registers the service command with all Daemon instances, SVM instances, and PVM instances.

xDaemon Instance When the xDaemon instance receives a "service start request" with a list of SVMs and PVMs for the service command, it scans the local part of the DHT, generates a list of hash items to be performed for this service (`SH-List`). The `SH-List` contains all content hash items that xDaemon thinks exist in the SVMs' memory according to the local DHT.

The format of `SH-List` is defined as follows.

```
SH-List:{{Content-Hash:{VM-List}:arg:sTag}, ...}
sTag: COMPLETED|NOTCOMPLETED
```

During initialization, the `sTag` of each item in `SH-List` is initialized as `NOTCOMPLETED`. As we have discussed before, the DHT in ConCORD may inaccurately perceive content hashes as existing currently in VMs due to slow or lost VM hash update messages. This means `SH-List` may not reflect exactly all of memory contents in SVMs. Instead the list only reflects what memory contents ConCORD believes to exist in the SVMs.

After the `SH-List` is generated and initiated, the xDaemon instance calls `collective_start` specified with the service command. Once the call returns, it sends a "request to start collective phase" (`SERV-COLLECTIVE-START-REQ`) message to xCoord to notify it that this xDaemon is ready to start collective phase.

SVM/PVM instance (VM instance) The execution flow in a SVM instance and PVM instance during initiation is exactly the same, therefore I do not differentiate them in my discussion in this section, *VM instance* is used to refer to either of them.

When a VM instance receives a request to start service from xCoord, it registers the service command locally. If `pause-SVM` or `pause-PVM` is set, the corresponding SVMs or PVMs are paused, and their memory updates from last scan are sent to ConCORD.

Each VM instance maintains a local content hash list (`Local-Hash-LST`) that reflects all unique memory contents in this VM. This list is used to determine whether local VM has the required memory contents when xDaemon instances request it to perform operations on specific memory content during collective phase. Also, as we have described in Chapter 4, the memory update monitor keeps maintaining such a hash list, in addition to a hash table to map each content hash to one or more memory blocks in VM. The list and mapping table can be directly used by VM instance for this purpose.

Finally, `collective_start` is called in each VM instance, whereever each VM instance sends a `SERV-COLLECTIVE-START-REQ` message to xCoord, indicating it is ready to start the collective phase.

Once xCoord has received `SERV-COLLECTIVE-START-REQ` from all xDaemon instances and VM instances. it sends a "collective start request" (`SERV-COLLECTIVE-START`) to all xDaemon instances, which starts the collective phase execution of the service command.

Collective Phase Execution

All xDaemon, SVM and PVM instances are involved in the collective phase execution of a service command. The execution flow in an SVM instance and a PVM instance is identical, thus I do not differentiate them from each other in my description here.

xCoord xCoord sends (`SERV-COLLECTIVE-START`) to all xDaemon instances to start the collective phase execution of a service command. It then blocks waiting "collective phase complete" (`SERV-COL-DONE-XDAEMON`) messages from all xDaemon instances.

xDaemon Instance When an xDaemon instance receives a `SERV-COLLECTIVE-START`, it starts the collective phase execution of the command. First, it iterates through the

SH-List. For each hash item that is marked NOTCOMPLETED, it calls select_vm to choose one (or K) VM(s) that are responsible for performing service operations on this content hash. If select_vm is not specified, then it randomly chooses one VM from the hash's VM list. Once the VM(s) are chosen, it sends a "hash item operation request" (HCMD-REQ) to all selected VM(s).

HCMD-REQ is a message to request a VM instance to perform the service-defined operation on the memory block with given content hash. The message is defined as:

```
HCMD-REQ:
{VM-ID, xDaemon-ID, num_hashes, {hash:arg, hash:arg, }}
```

Note, in practice, instead of sending each request separately, xDaemon aggregates multiple HCMD-REQ requests into one UDP message to best utilize network bandwidth.

After all HCMD-REQs are sent to the corresponding VMs, the xDaemon instance waits for "hash item operation complete" (HCMD-COMPLETE) from VM instances until the "iteration timeout" (ITER-TIMEOUT) time has passed. It then repeats the above operation for another iteration, generating HCMD-REQ requests for all content hashes that are not completed yet, and sends them to the VM instances.

HCMD-COMPLETE is a message from a VM instance that notifies its work status on the requested operations. It can indicate either completeness of the operation, or the inability of completing it.

```
HCMD-COMPLETE message:
  {VM-ID, xDaemon-ID, num_hashes,
      {hash:ret_arg:sTag, hash:ret_arg:sTag, }}
```

For each HCMD-COMPLETE received, if the opertion is COMPLETED, the xDamon instance marks the hash item in SH-List as COMPLETED, and saves ret_arg as the

item private data. Otherwise, if it is NONCOMPLETED, the sending VM is removed from the hash item's VM list. If that empties the item's VM list, the hash item is removed from SH-List.

The xDaemon instance keeps sending these requests in each iteration, until either 1) all items in SH-List are completed, or 2) the collective timeout specified in the service command expires. After the collective phase execution is terminated, the xDaemon instance calls collective_finalize with SH-List as an input. Then it sends "collective phase complete" (SERV-COL-DONE-XDAEMON to xCoord to indicate the collective phase is done.

The collective phase execution happens in an xDaemon instance can be summarized in the following pseudo code.

```
SH-List: {{Hash:{VM-List}:sTag:arg},...}
  sTag: COMPLETED|NOTCOMPLETED

HCMD-REQ: {VM-ID, xDaemon-ID, num_hashes, {hash:arg, hash:arg,}}
HCMD-COMPLETE:
  {VM-ID, xDaemon-ID, num_hashes,
    {hash:ret_arg:sTag, hash:ret_arg:sTag,}}

collective-phase:

  collective_start(SH-List, global_arg)
  Foreach VM-ID in SVMs and PVMs:
    Create HCMD-REQ-VM-ID: {VM-ID, xDaemon-ID, 0, {}};
  endFor
```

```
Foreach {Hash:{VM-List}:sTag:arg} in SH-List:
  if(sTag is NOTCOMPLETED):
    {Chosen-VM} = select_vm(Hash, {VM-List})
    Foreach (VM-ID in {Chosen-VM})
      add({Hash:arg}, HCMD-REQ-VM-ID);
    endFor
  endif
endFor

Foreach VM-ID in SVMs and PVMs:
  sendMsg(host(VM-ID), HCMD-REQ-VM-ID);
endFor

start MSG-ACK Receiving Thread;
sleep (GCL-ITER-TIMEOUT);
if (item in SH-List is NOTCOMPLETED
AND round_time < GCL-TIMEOUT):
  round_time += time_passed;
  goto collective-phase;
else:
  stop MSG-ACK Receiving Thread;
endif

call collective_finalize;
sendMsg(host(xCoord), SERV-COL-DONE-XDAEMON);
```

```
DONE

MSG-ACK Receiving Thread:
  LOOP:
    Receive a HCMD-COMPLETE message: {VM-ID, {hash:sTag:ret_arg,}}
Foreach H:{hash:sTag:ret_arg} in HCMD-COMPLETE:
      Find I:{HashV:{VM-List}:sTag:arg} in SH-List
       where I.HashV equals H.hash
      if(H.sTag == COMPLETED):
        I.sTag = COMPLETED
        I.arg = H.ret_arg
      elseif (H.sTag == NOTCOMPLETED)
        remove I.VM-ID from I.VM-List
      endif
    endFor

DONE
```

SVM/PVM Instance (VM instance) The core part of the collective phase is execution in a VM instance is handling requests to perform service-defined operations on the local memory blocks with given content hashes. When a VM instance receives a HCMD-REQ request, it looks for the content hash in local hash list. If there is a match, it calls `collective_cmd` once with memory block that corresponds to the given content hash. When the calls return successfully, it sets the hash item as `COMPLETED`, attaches the returned data and sends `HCMD-COMPLETE` back to the xDaemon instance. Otherwise, it

sets the hash item to be NONCOMPLETED and sends it back.

This operation is repeated with every HCMP-REQ request received, until the SERV-COL-DONE message is received from xCoord. Then each VM instance calls collective_finalize and completes the collective phase execution of the command. The collective phase execution that happens on a VM instance can be illustrated in pseudo code as follows.

```
Local-Hash-LST: {VM-ID:{Page-No:Hash:State:Memory-Content}, ...}
HCMD-REQ: {VM-ID, xDaemon-ID, num_hashes, {hash:arg, hash:arg, }}
HCMD-COMPLETE:
  {VM-ID, xDaemon-ID, num_hashes,
      {hash:ret_arg:sTag, hash:ret_arg:sTag, }}
  sTag: COMPLETED|NONCOMPLETED


collective_phase:

  For each HCMD-REQ:{xDaemon-ID, {hash:arg, hash:arg, }} received:
    HCMD-COMPLETE = {VM-ID, xDaemon-ID, {}};
    for each {hash:arg} in HCMD-REQ:
      if(hash in Local-Hash-LST):
        collective_cmd(hash:arg, memory, &ret_arg)
        If(success):
          put <hash:ret_arg: COMPLETED> in HCMD-COMPLETE;
        else
          put <hash:NONCOMPLETED> in HCMD-COMPLETE;
        endif
      else:
```

```
        put <hash:NONCOMPLETED> in HCMD-COMPLETE;
      endif
    endfor

    Send HCMD-COMPLETE to its xDaemon-ID;
  endFor

  if(SERV-COL-DONE is received)
    call collective_finalize;
    goto local_phase;
  else
    repeat collective_phase;

DONE
```

When xCoord has received all the "collective phase complete" (`SERV-COL-DONE-XDAEMON`) messages from all xDaemon instances, xCoord sends a "collective complete" (`SERV-COL-DONE`) to each VM instance to end the collective phase execution in its local VM. Now, the collective phase execution of this service command completes.

Local Phase Execution

Only xDaemon and SVM instances are involved in local phase execution. Execution flow in these components is described as follows.

xCoord xCoord waits for a "local start request" (`SERV-LOCAL-START-REQ`) from all xDaemon instances. Then it sends a "local start request" (`SERV-LOCAL-START`) to all

SVM instances, which starts the local phase execution on them. After this, xCoord blocks waiting for "local phase complete" (`SERV-LOCAL-DONE-VM`) message from all SVMs, and then finishes the execution of the service command once it has received them.

xDaemon Instance Before local phase execution starts, each xDaemon instance has to compile a list of hash items that have been completed during the collective phase. The list is sent to all SVM instances. Specifically speaking, for each SVM instance, each xDaemon instance generates a "collective completed list" (`CL-FIN-List`), which contains all hash items in `SH-List` that are in the destination SVM and also have been marked as `COMPLETED`.

```
CL-FIN-List: {VM-ID, xDaemon-ID, num-hashes, {hash:arg, ...}}
```

When the lists are ready, the xDaemon instance sends each of them to its target SVM. The list has to be acknowledged by the SVM instance(`CL-FIN-List-ACK`). If no acknowledgment is received from an SVM, the xDaemon retransmits the list to it. After acknowledgments from all SVM instances are received, the xDaemon instance sends a `SERV-LOCAL-START-REQ` to xCoord to indicate it is done sending `CL-FIN-Lists` to the SVMs. Once xCoord has gathered `SERV-LOCAL-START-REQ`s from all xDaemon instances, it notifies the SVM instances to start local phase execution.

The pseudo code to describe how the xDaemon generates `CL-FIN-List` is as follows.

```
CL-FIN-List: {VM-ID, xDaemon-ID, num-hashes,
                  {hash:arg, hash:arg, ...}}
CL-FIN-List-ACK: {VM-ID, num-hashes}

CL-FIN-List Generation:
```

```
Foreach VM-ID in S-VMs:
  create CL-FIN-List-VM-ID: {VM-ID, xDaemon-ID, 0, {}};
endFor

Foreach {hash:{VM-List}:sTag:arg} in SH-List:
  if(sTag is COMPLETED):
    Foreach VM-ID in {VM-List}:
      if (VM-ID in {SVMs}):
        put {hash:arg} into CL-FIN-List-VM-ID;
      endif
    endFor
  endif
endFor

Foreach VM-ID in S-VMs-List:
  sendMsg(host(VM-ID),CL-FIN-List-VM-ID);
endFor

Foreach VM# in S-VMs:
  Wait for VMs CL-FIN-List-ACK-VM-ID;
  if(Timeout)
    resends CL-FIN-List-VM-ID;
endFor

sendMsg(host(xCoord), SERV-LOCAL-START-REQ);
```

```
DONE
```

SVM Instance Before the local phase starts, each SVM instance receives a `CL-FIN-List` from each xDaemon instance. It merges all of them together to form the `Merged-CL-FIN-LST`. Once the `SERV-LOCAL-START` message is received from xCoord, the SVM instance starts the local phase execution of the service command. The following shows the data types of these items.

```
CL-FIN-List: {VM-ID, xDaemon-ID, num-hashes, {hash:arg, hash:arg, ...}}
CL-FIN-List-ACK: {VM-ID, num-hashes}
Merged-CL-FIN-LST: {VM-ID:{Hash:arg}, ...}
Local-Mem-LST: {VM-ID:{pageno:hash:state:memory}, ...}
```

The merging process is shown in following pseudo code.

```
Merge_Fin_Lst:

  Create Local-Serv-LST: {VM-ID:{}}
  While(SERV-LOCAL-START not received) {
    For each CK-FIN-List: {VM-ID, xDeamon-ID, {hash:arg, hash:arg, ...}}
      Check integrity of CK-FIN-List.
      CK-FIN-List-ACK = {VM-ID, num-hashes}.
      Send CK-FIN-List-ACK to its source xDaemon.
```

```
      For each {hash:arg} in CK-FIN-List:
        If(hash in Local-Mem-LST):
          Add {hash:arg} into Merged-CL-FIN-LST.
        endIf
      endFor
    endFor
  endWhile
```

In the local phase, each SVM instance iterates through all the memory blocks in the SVM. For each memory block, if the content hash of that block is in the `Merged-CL-FIN-LST`, it marks the block as completed, otherwise, as non-completed. Then it calls `local_cmd` once for each of these memory blocks. The pseudo code is as follows.

```
local_phase:

  call local_start;
  For each {pageno:hash:state:memory} in Local-Mem-LST:
    if(hash is in Merged-CL-FIN-LST (hash:arg))
      local_cmd(hash, arg, pageno, memory);
    endIf
  endFor

  call local_finalize;
  sendMsg(host(xCoord), SERV-LOCAL-DONE-VM);
```

Finally, when this completes, the SVM instance sends a SERV-LOCAL-DONE-VM message to xCoord to notify it that this SVM has finished local phase execution,

7.3.5 Fault tolerance

In the above illustrative process, we did not consider any failures that could happen in the system, which are mainly message loss or physical node failures. I will now discuss how service command execution can tolerate different kinds of failures.

Message Loss

Since all communications between xCoord and xDaemon/VM instances are based on reliable UDP messaging as I have described in Section 3.2.3), lost synchronization messages from/to xCoord are automatically re-transmitted until acknowledged.

The hash request/reply messages (HCMD-REQ/HCMD-COMPLETE) exchanged by a xDaemon and VM instances are loss tolerant. If a hash request gets lost, then the operation is indeed not sensed and preformed on the VM instance for the request. The xDaemon instance will send the request again in the next collective iteration. If a hash reply gets lost. This means a VM instance has successfully applied the operation on a memory block with given content hash, but xDaemon does not know it. In this case, the xDaemon sends the same request again in next iteration, and thus the same operation may be applied again by some other VMs on their memory blocks with given content hash. This is ok because it is required that the service-defined operation on each block of distinct memory content is idempotent.

The final hash complete list (CL_FIN_LST) communicated between an xDaemon and SVMs are required to be acknowledged by the SVMs to make sure the SVMs know the list of all completed content hashes. However, this is just for performance consideration, not correctness. Lost of a small portion of this list does not affect the correctness of the service

command, since the SVM treats the lost content hashes as non-completed, and thus applies the operation on them during the local phase execution. No work is missed just completed suboptimally.

xCoord Failure

If xCoord fails during the execution of a service command, the service is aborted. The communication between xCoord and xDaemon instances or VMs is based on reliable UDP messages, i.e, each message received is acknowledged. If xCoord fails, the synchronization messages from xDaemon and VMs are not acknowledged, which causes re-transmission of these messages. After a certain number of re-transmissions, xDaemons and VMs sense the failure, and abort execution of the service command locally.

xDaemon Failure

One or a small number of xDaemon instance failures is tolerable during the execution of service command. When xCoord detects the failure of a xDaemon instance, it removes it from the command's xDaemon list for this execution and continues to execute the command without it. Even if the xDaemon comes back later, xCoord does not use it during the current execution of the service command.

The ConCORD memory content tracer is by nature not exactly accurate about the memory contents existing in VMs, and the content-aware service command is designed to be tolerant to this inaccuracy. Therefore, loss of one or small number of xDaemon instances just increases the amount of this inaccuracy, it does not impact the correctness of service command execution, but merely generates more performance overhead.

VM Instance Failure

The execution of a service command cannot complete if an SVM instance fails. An SVM instance failure causes immediate abortion of the execution. However, PVM instance failure is tolerable. After several tries to receive reply from a PVM instance, xDaemon just removes this PVM from its VM list, and all xDaemon instances do not send hash request to it anymore.

7.4 Execution example

I will now use the VM reconstruction example as we described in Section 7.2.2 to illustrate the execution of a service command. We will take a close look at the main data stored and communicated in between xDaemon and VM instance. This gives a more intuitive description of the execution of the service command.

In this example, there are 4 VMs involved. VM-1 is the SVM, i.e. the VM to be migrated. The rest of three VMs are PVMs. Each VM has 8 memory pages, some of which are shared with others. The VMs' memory content is shown in Table 7.1, with the content of each page represented by its content hash. The memory pages that share the same content with the migrating VM are marked as bold and underlined in the table. We can see from the table that each PVM shares some memory contents with the migrating VM, thus they can contribute to its migration.

Next, Table 7.2 shows the content hashes and their VM lists as stored in the memory content tracer's DHT in ConCORD. All unique content hashes are split among and stored in three xDaemon instances. As we can notice, the content hashes stored in ConCORD is a little outdated compared to the VMs' memory contents. The entries marked bold and underlined are entries with obsolete information. For example, the memory page with content *AB* does not exist in VM-2 anymore, and memory page having *AH* is not in VM-

Table 7.1: Memory Pages in each VM

SVM		PVMs					
VM-1		VM-2		VM-3		VM-4	
page	hash	page	hash	page	hash	page	hash
1	AA	1	BA	1	CA	1	BA
2	AB	2	BB	2	**AB**	2	**AB**
3	AC	3	**AC**	3	DE	3	**AC**
4	AD	4	**AD**	4	CD	4	**AD**
5	AE	5	CG	5	**AE**	5	DE
6	AF	6	BF	6	BF	6	AF
7	AG	7	BG	7	CG	7	**AG**
8	AJ	8	BH	8	CH	8	DH

1 either. Instead, VM-1 has a page with new content *AJ*, which has not been reflected in ConCORD yet. We will see how this inaccuracy of ConCORD impacts the execution process of a the service command, and how the service command system tolerates it and makes sure the content-aware service completes correctly.

Table 7.2: Status in ConCORD DHT

xDaemon-1		xDaemon-2		xDaemon-3	
Hash	VM list	Hash	VM list	Hash	VM list
AA	{1}	**AB**	{**1,2,3**}	AC	{1,2,4}
AD	{1,2,4}	AE	{1,3}	**AF**	{**1,3,4**}
AG	{1,4}	**AH**	{**1**}	BA	2,4
BF	{2,3}	BG	{2}	BH	{2}
CA	{3}	CD	{4}	CG	{2,3}
CH	{3}	DH	{4}	DB	{4}
DE	{3,4}				

One of the core data structures generated and used by an xDaemon instance throughout the entire collective phase execution is `SH-List`. This list contains all content hash item appearing in all SVMs' memory that are perceived by ConCORD. Each xDaemon instance generates its part of such a list. The entries of `SH-List` are kept updated during collective phase execution. The `SH-List` generated by each xDaemon, in addition to the updates to

the lists during each iteration in collective phase execution are shown in Table 7.3.

Table 7.3: SH-Lists in each xDaemon in each round

xDaemon		Before 1st Iteration		After 1st Iteration		After 2nd Iteration	
	Hash	VM-List	Status	VM-List	Status	VM-List	Status
1	AA	{1}	NON	{1}	**CMPLED**	{1}	CMPLED
	AD	{1,2,4}	NON	{1,2,4}	**CMPLED**	{1,2,4}	CMPLED
	AG	{1,4}	NON	{1,4}	**CMPLED**	{1,4}	CMPLED
2	AB	{1,2,3}	NON	{**1,3**}	NON	{1,3}	**CMPLED**
	AE	{1,3}	NON	{1,3}	**CMPLED**	{1,3}	CMPLED
	AH	{1}	NON				
3	AC	{1,2,4}	NON	{1,2,4}	**CMPLED**	{1,2,4}	CMPLED
	AF	{1,3,4}	NON	{**1,4**}	NON	1,4	**CMPLED**

At the beginning of the 1st iteration, entries in `SH-List` are initiated as `NOTCOMPLETED`. Next the hash content requests are sent to the VMs from each xDaemon. The hash request lists being sent by each xDaemon during each iterations are illustrated in Table 7.4. In our example, two iterations happen before the collective phase execution finishes. In the first iteration, all hash requests, i.e, sending a memory block with a given content hash to the destination host, are completed by the targeted VMs, except for hash content *AB*, *AH* and *AF* which are not due to the obsolete content information perceived by ConCORD. The source VM is removed from the content hash's VM list when a failure on a content hash request is received. In particular, *AH*'s VM list is empty after VM-1 is removed from its list. There the entry is removed from `SH-List`. To complete the operations on the remaining two hash contents, xDaemon instance-1 and instance-2 have to send hash requests for content hash *AB* and *AF* to other VMs. This time they get completed and replied to by these VMs.

After two iterations, all hash entries have been completed, which ends the collective phase execution. Before the local phase execution starts, each xDaemon compiles a hash list that contains all hash contents that have been completed during the collective phase

(CL-FIN-LIST). The CL-FIN-LIST sent to each VM by each xDaemon are shown in the last column of Table 7.4.

Table 7.4: Hash list sent to VMs by each xDaemon during each round

	HCMD-REQ (1st Iteration)		HCMD-REQ (2nd Iteration)		CL-FIN-LIST	
xDaemon	VM	Hash List	VM	Hash List	VM	Hash-List
1	1 2 4	{AA} {AD} {AG}	1 2 4		1	{AA,AD,AG}
2	2 3 1	{AB} {AE} {AH}	1 2 3	{AB}	1	{AB,AE}
3	4 3	{AC} {AF}	3 4	 {AF}	1	{AC,AF}

Finally, when the SVM (VM-1 in our example) has received a CL-FIN-LIST from all xDaemons, it merges them together to form the Merged-CL-FIN-LIST, which is shown in Table 7.5. In the local phase execution, the SVM instance iterates through all memory blocks in the SVM's memory. For each block, if its content hash is on the Merged-CL-FIN-List, it means a memory block with the same content has already been sent to the destination host, and nothing needs to be done during local phase. If it is not on the list, this memory block is sent to the destination host. This step is necessary to make sure that all of memory blocks in the migrating VM have been sent to destination.

7.5 Evaluation

I have performed a set of evaluations on my implementation of the content-aware service command framework in ConCORD. Two metrics are measured in my evaluation: 1) the execution time of a content-aware service command, and 2) the network overhead during the execution of a content-aware service command.

Table 7.5: VM's movement during local phase in service VM (VM-1)

VM Local Memory List	Hash Merged-CL-FIN-List	Status	Local Phase Command
AA	AA	CMPLED	Nothing
AB	AB	CMPLED	Nothing
AC	AC	CMPLED	Nothing
AD	AD	CMPLED	Nothing
AE	AE	CMPLED	Nothing
AF	AF	CMPLED	Nothing
AG	AG	CMPLED	Nothing
AJ		**Missed**	**Send Block**

All the evaluation in this chapter were performed on a cluster with 6 PowerEdge R415 machines. Each node is equipped with two Quadcore 2.4 GHz X3430 Intel Xeon CPU, 8GBytes RAM, a Broadcom NetXtreme II 1Gbps Ethernet card. The nodes are connected through a 1Gbps Ethernet switch.

7.5.1 Command Execution Time

To measure the execution time of a content-aware service command independent of the specific service implementation, I execute a service command configured with empty operation functions, i.e, all of its operation functions are implemented but do nothing inside the function. This enables us to measure the time spent purely on ConCORD components during the command's execution, excluding time spent on service-specific operations for a real service command.

I have measured the execution times by running a set of service commands in each one of two scenarios. The first set of tests was executing a service command on a fixed number of SVMs (8 SVMs in our case) and xDaemon instances repeatedly. In each run, every VM is allocated with a different memory size. This set of tests is intended to measure the execution time as a function of average VM's memory size. The second set of runs

applies a service command on different numbers of VMs and xDaemon instances, with the memory size of every VM in each test being fixed to 1 GBytes. In the second set of tests, the number of xDaemon instances is kept the same as the number of VMs, e.g, when testing 8 VMs, there are 8 xDaemon instances running, with each two of them on a physical nodes.

In addition, for each test, I executed every service command in both interactive-mode and batch mode, and the execution times were measured for both modes. Execution time of a service command in batch mode reflects mostly the time spent by the collective phase execution, while execution time of a service command in interactive mode reflects time spent by both the collective phase and the local phase execution.

Figure 7.1 shows the execution times for the tests on a fixed number of VMs, and Figure 7.2 presents results for tests on a variable number of VMs. For test with fixed number of VMs, the results show that the time to execute a service command is linear with the total memory size of all service VMs. On the other hand, for tests with a variable number of VMs, the execution time of a service command is almost constant as the number of SVMs increases. This suggests that the content-aware service command achieves a good scalability when number of serviced VMs increases.

7.5.2 Network Overhead

Besides the execution times for a service command in various scenarios, I have also measured the network communications during the execution of a service command. Specifically, I have added a piece of code to the xDaemon implementation to count the number of bytes it has sent and received during a period of time. Figure 7.3 presents the number of KBytes that have been sent and received by each xDaemon instance during the execution of a service command on various numbers of VMs. Each VM is assigned with a fixed memory size in different tests, and the number of xDaemon instances is the same as the

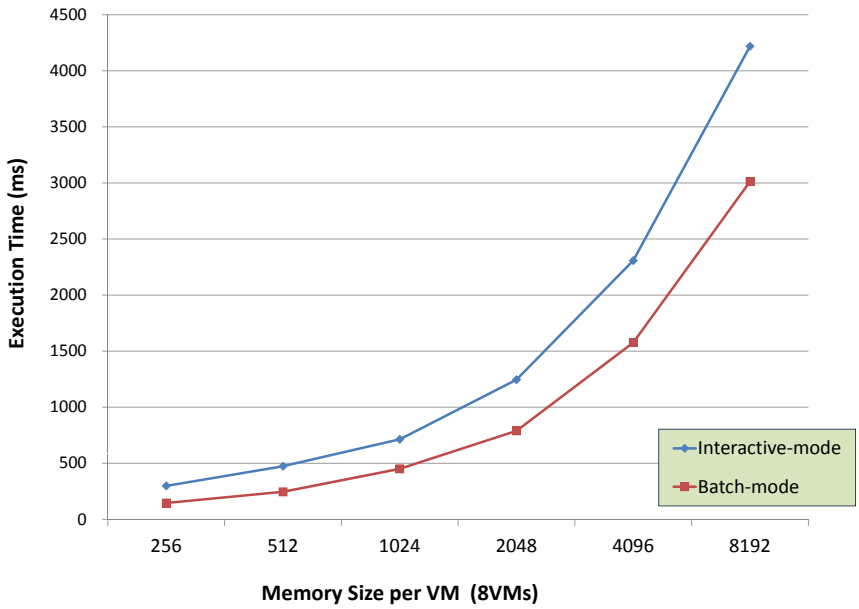


Figure 7.1: The times to execute a service command on a fixed number of SVMs with different memory sizes during each test. The execution times of both interactive mode and batch mode are linear to the total memory size of SVMs.

number of VMs. The results show that the data communicated between each xDaemon to the VMs is nearly constant as the number of SVMs increases.

7.6 Conclusion

The content-aware service command model is a powerful model that enables effectively construction of content-aware services in large-scale parallel systems. Content-aware services built on top of service commands are automatically parallelized and executed on the parallel system. This enables service developers to build and enhance their services to maximally exploit and utilize the memory content sharing with minimal implementation effort.

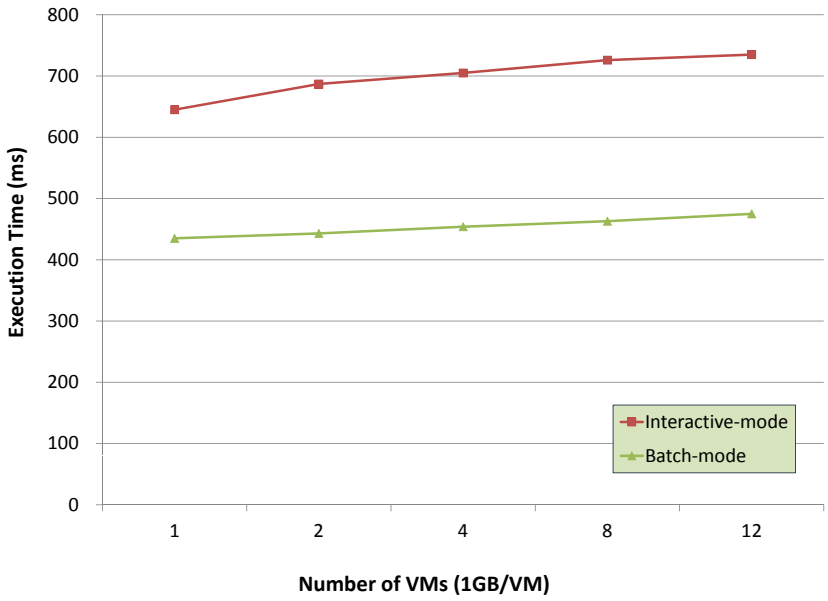


Figure 7.2: The times to execute a service commands on different number of SVMs with fixed memory size in each of them. The number of xDaemon instance is the same as the number of SVMs. The execution times seen in both interactive mode and batch mode are almost constant as the number of service VMs increases.

In this chapter, I first introduced the the content-aware service command model and the configuration options of a service command. Then I showed the general steps to build a content-aware service using the model with a step-by-step demonstration of an example service. I explained the execution flow happening in each ConCORD component during the execution of a service command, with an illustrative example. Finally, I presented my evaluation of the service command execution system in ConCORD, including times taken to execute a service command in various scenarios, and the network overhead during the execution of a service command.

In the following chapter, I will present another content-aware service, a group check-

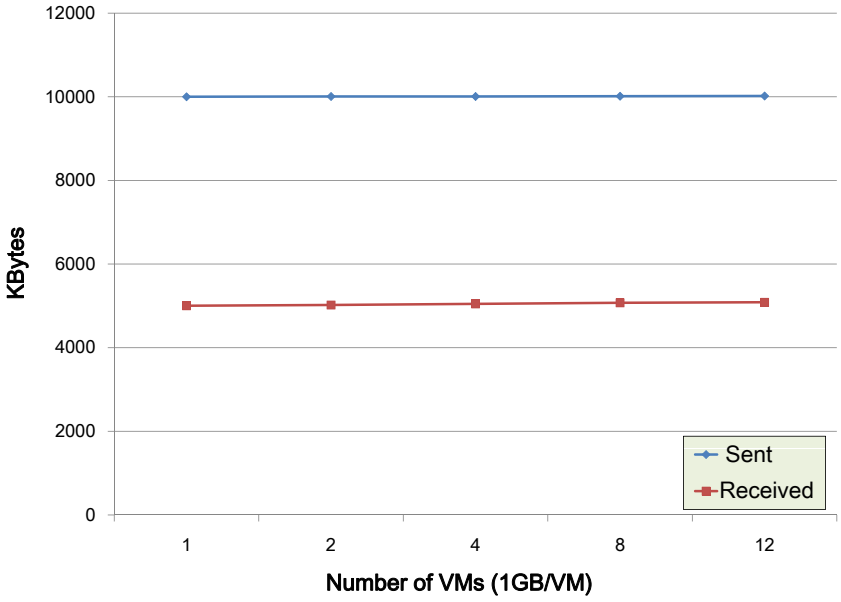


Figure 7.3: The data sent and received by each xDaemon instance during the execution of a service commands on different number of VMs with a fixed memory size. The number of xDaemon instances is the same as the number of VMs. The data communicated from/to each xDaemon instance is kept constant as the number of total service VMs increases.

pointing service that has been built on top of service command, including detailed implementation and evaluation.

Chapter 8

Content-aware Checkpointing Example

Disk-based coordinated checkpointing with rollback-recovery has been the well known technique for fault tolerance in high performance computing systems for several decades. This approach generally works as follows: periodically all nodes are stopped, the system state of each node is saved to stable storage, and then the nodes continue with the computation. Upon a failure, the stored checkpoints are read from stable storage to restore each node from the last saved state.

Such checkpointing results in high overhead due to often simultaneous writes of all nodes to the parallel file system (PFS), which reduces the productivity of such systems in terms of throughput computing. For example, when a large parallel application is checkpointed, tens of thousands of processes will each write its several GB of data containing its memory contents into the backup file system, with total checkpoint size in the order of TBs. Furthermore, the I/O bandwidth of supercomputers and parallel systems does not increase at the same rate as computational capabilities and available RAM size and so large checkpoints increasingly lead to an I/O bottleneck. There is already high overhead in a peta-scale system. The exascale systems in the near future will have a significant larger number of nodes and more memory. In addition, the larger the system, the more frequently it may require checkpoints.

Checkpoint performance impacts the scalability of large-scale applications to such a degree that many HPC applications employ their custom application-specific checkpoint mechanism to minimize the saved checkpoint state and therefore the time to checkpoint. However, this approach requires intimate knowledge of the application's internal computational logic and data structures, and thus is typically very difficult to generalize to other applications.

I have proposed content-aware group checkpointing as an approach to solve this problem. Content-aware checkpointing leverages the memory content sharing across the checkpointing nodes to reduce the checkpoint size. The general idea is that by utilizing content sharing information, content-aware checkpointing needs to store only a single copy of memory block with unique memory content over all nodes. This can potentially reduce the size of checkpoints significantly, and thus reduce the total time and the I/O bandwidth needed to transfer it.

My checkpointing system efficiently checkpoints the state of a collection of VMs, for example, a group of VMs running a parallel application. In this chapter, I will first demonstrate my implementation of such a content-aware checkpointing service on top of ConCORD's content-aware service command, and then present my evaluation of this content-aware service.

8.1 Implementation

The content-aware checkpointing works as the following way. First, a global shared block file is created for each checkpoint. This file is used to save memory blocks with distinct contents across all virtual machines to be checkpointed. During the checkpointing process, each virtual machine tries to writes some of its memory blocks to this shared file simultaneously. By carefully arrangement each VM writes memory blocks that contain different

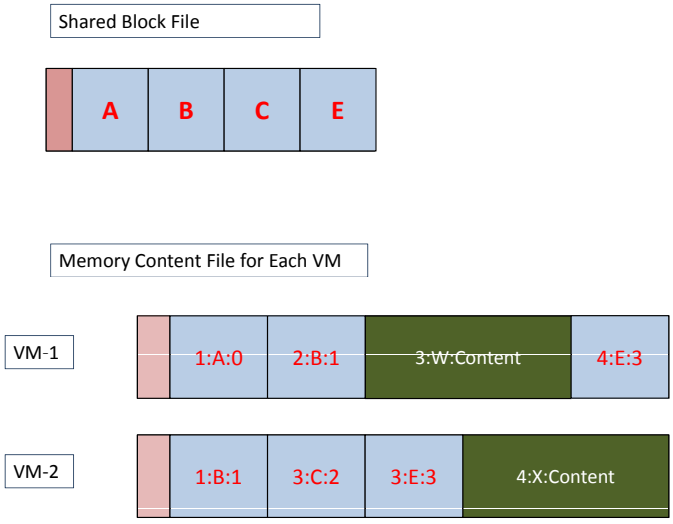


Figure 8.1: Checkpoint files format.

memory contents from other VM's writes. Therefore, the memory block from different VMs that share the same content is saved only once in the file. This will significantly reduce the final checkpoint file size if there is large amount of memory content sharing among this set of VMs.

In addition, to allow each VM to restore its original memory from this shared file, a memory content file is created for each VM. This file comprises a mapping from each of its memory block number to either an offset to a memory block saved in shared block file, or the actual content of the memory page. The structures for both type of files are shown in Figure 8.1.

Take the file contents given in Figure 8.1 as an example. This checkpoint contains memory contents from two VMs. Each VM has 4 memory pages. Two pages in one

VM share the same content with the other VM, the pages with content B and E. The final checkpoint comprises three files. The shared block file saves four memory pages with distinct contents. In the memory content file for each VM, one entry is saved for each of its memory page. The entry contains the memory page number, its content hash and either an offset in the shared file, or the actual memory content. For example, the first entry, 1:A:0, in VM-1's memory content file means the first memory page of the VM is stored in the block starting from offset 0 in the shared block file. The fourth entry, 4:X:Content, in VM-2's memory content file, means the fourth memory page of the VM is saved in local content file.

To restore a VM's memory from such a checkpoint, each VM loads its memory content file and parses each entry in the file. For every page mapped to an offset, it reads the actual memory content from the block starting from this offset in the shared block file. If a memory page is followed by its original content, just simply copies the content from its own file to VM's memory page.

This content-aware checkpointing service is straightforward to build on top of ConCORD's content-aware service command model. The general steps are:

- *Collective Phase* During the collective phase, all participating VMs write some of memory blocks with distinct content from their local memory to the shared block file in parallel.

- *Local Phase* In the local phase, each VM checks each of its memory pages. If a memory page with the same content has already been saved to the shared block file, adds an entry in VM's memory content file to map the page number to the offset of the saved block in shared file. Otherwise, it writes the actual content of the page into local memory content file and adds an entry for it.

I have built this content-aware group checkpointing service using the content-aware

service command model. My implementation comprises only of around 230 lines of C code, which has been attached in Appendix A of this thesis. Next, I will describe the core implementation code for the service.

8.1.1 Implementation Code

First, a global data structure is defined and attached to the service command as service-defined private data. This data structure is used to hold the local state data shared throughout command execution time on each single execution instance across different calls to service specific functions. In the checkpointing service, this global data contains the filename for the shared block file, and filename prefix for each individual VM's memory content file, and an integer to store the file handler after creating and opening the shared block file during the collective phase and local VM's content file during the local phase.

The global data structure is defined as below.

```
struct ckpt_arg {
  char shared_ck_file[NAME_LEN];
  char local_ck_file_prefix[NAME_LEN];
  sint32_t file_id;
};
```

When creating a service command, these filenames are specified in a configuration file. The *f-parse-input* function is implemented to parse this configuration file to generate the global data. The data is attached to service command's private data field, later the command execution system passes this data as an argument to all service operation functions. Note, after the execution starts, each instance (xDaemon instance and VM instance) saves a copy of data locally so any changes to this global data by one operation function call in one instance is only visible to the following operation function calls running in the same instance. The service developers can use this to save local state data that needs to be passed from call to call in on the same ConCORD component.

In `parse_input`, a checkpoint configuration file is parsed, and the filenames for the global shared checkpoint and for local checkpoints are read and converted to a private data as a service command.

```
static int
f_parse_input(struct xcord_service_cmd * xcommand,
              char * input_file)
{
  FILE * ck_conf;
  struct ckpt_arg * ckarg;

  ck_conf = fopen(input_file, "r");
  if(ck_conf == NULL) {
    PrintError("Fail to open %s\n", input_file);
    return -1;
  }

  ckarg = xcord_malloc(sizeof(struct ckpt_arg));
  ckarg->file_id = -1;

  fscanf(ck_conf, "%s, %s",
         ckarg->shared_ck_file,
         ckarg->local_ck_file_prefix);

  xcommand->arg_size = sizeof(struct ckpt_arg);
  xcommand->arg = ckarg;

  return 0;
}
```

Next, the global shared block file is opened locally from the parallel file system (PFS). We assume here that a PFS provides appending operations that allows multiple machines to append blocks of data to the same file. We assume PFS can properly serialize the data and write it to the file, and that it also guarantees that a block of data from one call will be written into a continuous chunk in the file. The order of blocks written into the file from different senders need not guaranteed. In addition, we assume appending a block of data to a file in the PFS returns the actual offset of this block in the file after the block has been successfully written to file. We use this offset as a pointer to direct where to find a memory

block when a VM is restoring from the shared block file.

Here, the shared block file is opened locally in each SVM and PVM instance. The file descriptor is stored in the command's private data for later use by other service functions.

```
static int
f_collective_start(uint8_t component_type,
                   uint16_t c_id,
                   struct xcord_hash_list * hashes,
                   void ** arg, uint16_t * arg_size)
{
  if(component_type == XCORD_COMPONENT_SVM
      || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg = (struct ckpt_arg *)*arg;
    ck_arg->file_id = (sint32_t)
        pfs_open_file((ck_arg->shared_ck_file);
    if(ck_arg->file_id < 0) {
      PrintError("Fail to open global shared checkpoint file\n");
      return -1;
    }

    *arg = (void *)ck_arg;
    *arg_size = sizeof(*ck_arg);
  }

  return 0;
}
```

Inside `collective_cmd`, if a request content hash has a corresponding memory page in the local VM, the memory page is written to the shared block file, and the offset of the block in the shared block file is returned. This offset is then stored as the content hash's private data, which will be used in `local_cmd` to determine whether the block has been saved.

```
static int
f_collective_cmd(uint16_t vm_id,
                 struct xcord_hash_item * hash,
                 char * mem_block, int block_size,
                 void ** arg, uint16_t * arg_size) {
  struct ckpt_arg * ck_arg = (struct ckpt_arg *)(*arg);;
  sint64_t offset = 0;
```

```
  offset = pfs_append_file((struct fs_client *)ck_arg->data,
                           ck_arg->file_id,
                           mem_block, block_size);
  if(offset < 0) {
    return -1;
  }

  hash->arg = xcord_malloc(sizeof(sint64_t));
  hash->arg_size = sizeof(sint64_t);
  *((sint64_t *)hash->arg) = offset;

  return 0;
}
```

The shared block file descriptor is closed in `collective_finalize`, and in `local_start`, each SVM creates a memory content file in its local storage. In `local_cmd` we iterate over all memory pages of the VM. An entry is created for each local memory page in this content file. The entry contains either a pointer (offset) to a memory block saved in the shared block file, or the actual memory content of the block. Finally, the local memory content file is closed in `local_finalize`.

```
static int
f_collective_finalize(uint8_t component_type,
                      uint16_t c_id,
                      struct xcord_hash_list * hashes,
                      void ** arg, uint16_t * arg_size) {
  if(component_type == XCORD_COMPONENT_SVM
       || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);

    pfs_close_file((struct fs_client *)ck_arg->data,
                   ck_arg->file_id);
        ck_arg->file_id = -1;

    return 0;
  }
}


static int
f_local_start(uint8_t component_type,
              uint16_t c_id,
```

```
                struct xcord_hash_list * hashes,
                void ** arg, uint16_t * arg_size) {

  if(component_type == XCORD_COMPONENT_SVM) {
    char local_ck_file[NAME_LEN];

    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);
    snprintf(local_ck_file, NAME_LEN, "checkpoints/%s-vm%d.ck",
             ck_arg->local_ck_file_prefix, c_id);

    ck_arg->file_id = open(local_ck_file, O_WRONLY | O_CREAT);
    if(ck_arg->file_id < 0) {
      PrintError("Fails to open local chekpt file %s\n",
                 local_ck_file);
      return -1;
    }
  }

  return 0;
}


static int
f_local_cmd(uint16_t vm_id,
            struct xcord_hash_item * hash,
            uint32_t page_no,
            char * mem_block, int block_size,
            void ** arg, uint16_t * arg_size) {
  struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);
  char header[32];
  int size = 0;
  sint64_t offset = -1;

  /* Local ckpt block layout:
   *   page_no(4B):Hash(16B):Offset(8B):[MEMORY_DATA]
   */
  memcpy(header, (char *)&page_no, sizeof(page_no));
  size += sizeof(page_no);
  memcpy(header+size, hash->hash_val, HASH_LEN);
  size += HASH_LEN;

  if(hash->s_tag == SERVICE_HASH_ITEM_COMPLETED
       && hash->arg_size == sizeof(sint64_t)) {
    offset = *((sint64_t *)hash->arg);
  }

  memcpy(header+size, (char *)&offset, sizeof(offset));
```

```
  size += sizeof(offset);

  if(write(ck_arg->file_id, header, size) < 0) {
    PrintError("Fails to write to local checkpt file");
    return -1;
  }

  /* the block has not been writen to shared checkpoint
   * file by any VMs, write to local VM checkpoint file
   */
  if(offset == -1) {
    if(write(ck_arg->file_id, mem_block, block_size) < 0) {
      PrintError("Fails to write to local checkpt file\n");
      return -1;
    }
  }

  return 0;
}


static int
f_local_finalize(uint8_t component_type, uint16_t c_id,
                 struct xcord_hash_list * hashes,
                 void ** arg, uint16_t * arg_size) {
  if(component_type == XCORD_COMPONENT_SVM
      || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);

    close(ck_arg->file_id);
  }

  return 0;
}
```

An example configuration of the service command is shown as follows. A checkpoint argument file is also given. The service configuration file indicates there are 5 SVMs that are to be checkpointed, with the aid of 3 additional PVMs. The checkpoint argument file specifies checkpoint file names.

```
/*
 * Below is the service Configuration File
 * checkpoint.conf
 */
```

```
##Service configuration file
name=Checkpoint
ops_name=checkpoint
svm=5:{"guest-node1-1","guest-node1-2","guest-node3-2","lei-guest-4"
    ,"tony-guest-1"}
pvm=3:{"tony-guest-2","someone-g-3","lei-guest-2"}
timeout=10
arg_file=checkpoint_arg.conf

## checkpoint argument file
/* checkpoint_arg.conf */
checkpoint_shared_file.ck,
local_checkpoint_file
```

8.2 Evaluation

I have run a set of performance tests to evaluate my implementation of the content-aware checkpoint service. I was mainly interested in two aspect of the performance: 1) the size of checkpoint generated, and 2) the total time taken to complete the service.

Checkpoint Sizes

Reducing checkpoint file size is the primary goal for our content-aware checkpointing, given the anticipated scale of storage requirements on massively parallel systems. To compare the size of checkpoint files generated from content-aware checkpointing with other conventional approaches, we consider four strategies for generating a checkpoint.

- *Raw*: The raw checkpoint strategy is saving the entire memory content from each VM into a separate file. The total size of all those files is reported. In the raw checkpoint, each VM writes its memory to its local file system independently.
- *Raw-gzip*: All files from raw checkpointing are concatenated, and the concatenated checkpoint file is compressed by gzip with its best compression option (–best). The size of the compressed file is reported.

- *ConCORD*: Using the content-aware checkpoint strategy I have described in this chapter, the checkpoint comprises of a shared block file and a memory content file for each VM. All of these files are concatenated together to one large checkpoint file. The size of this file is reported.

- *ConCORD-gzip*: Similar to *Raw-zip*, the merged checkpoint file from the content-aware checkpointing is compressed by gizp with the best compression option, and the size of the compressed file is reported.

Four types of benchmarks have been carefully chosen to evaluate the performance of content-aware checkpointing service in cases with different amount and types of memory content sharing. The benchmarks I used during evaluation include:

- *Moldy*: As we have seen from chapter 2, the Moldy application presents a significant cross-node memory content sharing among its running processes but very little intra-node sharing. I use Moldy here to show how content-aware checkpointing performs on the type of applications with lots of inter-node but little intra-node sharing.

- *HPCCG*: In contrast to Moldy, HPCCG shows a lot of intra-node sharing.

- *NPB*: The NAS parallel benchmarks show moderate to little sharing for both the intra-node or inter-node cases. I use this set of benchmarks to evaluate content-aware checkpointing at the situation with moderate to little sharing.

- *No-share*: In this scenario, I evaluate content-aware checkpointing in the case where there is no sharing at all, neither intra-node or inter-node. This case is intentionally constructed, because some content sharing always exists in a real world application. To construct this case, I start a number of VMs, pause these VMs and manually fill the VM's memory with totally different content. This makes sure there are no blocks sharing the same content over all the checkpointed VMs.

Figure 8.2 compares the sizes of checkpoints generated by four checkpoint strategies, while Figure 8.3 normalizes the results by showing the compression ratio achieved by the three strategies, compared to the raw checkpoint (the raw checkpoint shows as 100%). The degree of sharing (*DoS*) among the checkpointed VMs is also given in Figure 8.3.

We can see that the compression ratio achieved by content-aware checkpointing is almost consistent with the *DoS* reported by ConCORD. In addition, gzip does not help much to reduce size of checkpoint on either raw checkpoints or content-aware checkpoints. One possible reason could be that the most of content sharing in this case is inter-node, which means the blocks with same content are usually located far away from each other in the final checkpoint file. This makes it hard for gzip to utilize the redundancy to compress the file. However, content-aware checkpointing can utilize the content sharing that exists globally, no matter how far apart two blocks with the same content are located from each other.

Similarly, Figure 8.4 and 8.5 compare checkpoint sizes and the compression ratio for the HPCCG application. HPCCG presents a significant amount of intra-node sharing. Similar to Moldy, the compression ratio achieved by content-aware checkpointing is consistent with the *DoS* reported by ConCORD. However, in contrast with Moldy, gzip can further compress both the raw checkpoint file and the content-aware checkpoint file to less than 15% of raw size. This may suggest that intra-node memory content sharing can be leveraged very well by using gzip.

Figure 8.6 and 8.7 compare the checkpoint sizes and the compression ratios for various NAS benchmarks. The NAS benchmarks contain several testing suites, each of which has either moderate or little content sharing. The results show that content-aware checkpointing can always exploit this content sharing and achieve a compression ratio on the final checkpoint size that is consistent with the *DoS*.

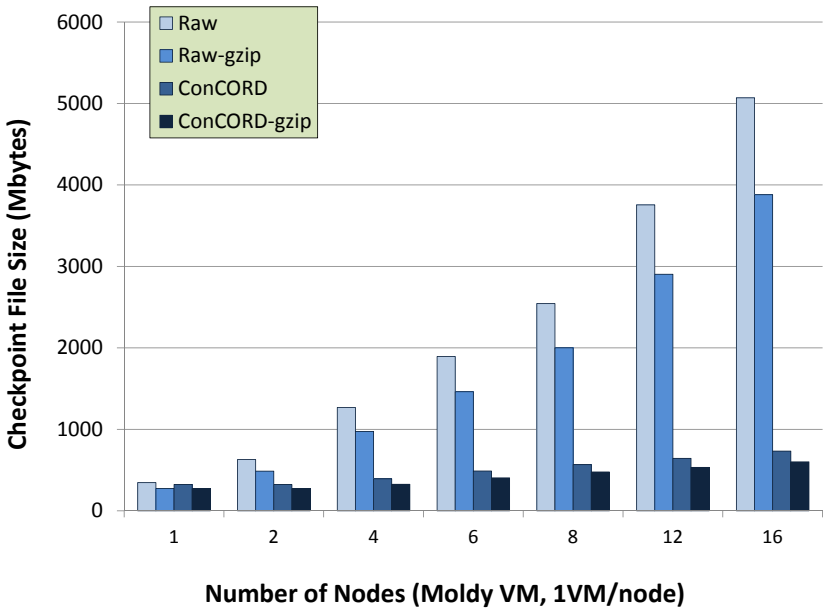


Figure 8.2: The checkpoint file size achieved by different checkpoint approaches, as a function of number of nodes, for Moldy application, which has a significant amount of inter-node content sharing.

Finally, I have compared the compression ratio achieved by the four checkpoint strategies in the case where no content sharing exists at all. The results are shown in Figure 8.8. We can see that in this case, the checkpoint from content-aware checkpointing is slightly larger than the raw checkpoint. This is because content-aware checkpointing has to save additional metadata for each memory page to allow a VM to restore its memory from checkpoint (e.g, a pointer for a duplicated saved block, hash for a block, etc). However, this overhead is very limited (less than 3% of total checkpoint size and it keeps almost constant as number of VMs increases). Also as I mentioned, this case is intentionally constructed as the worst case scenario. In real world applications, more or less content sharing always exists, which makes content-aware checkpoint always generates smaller

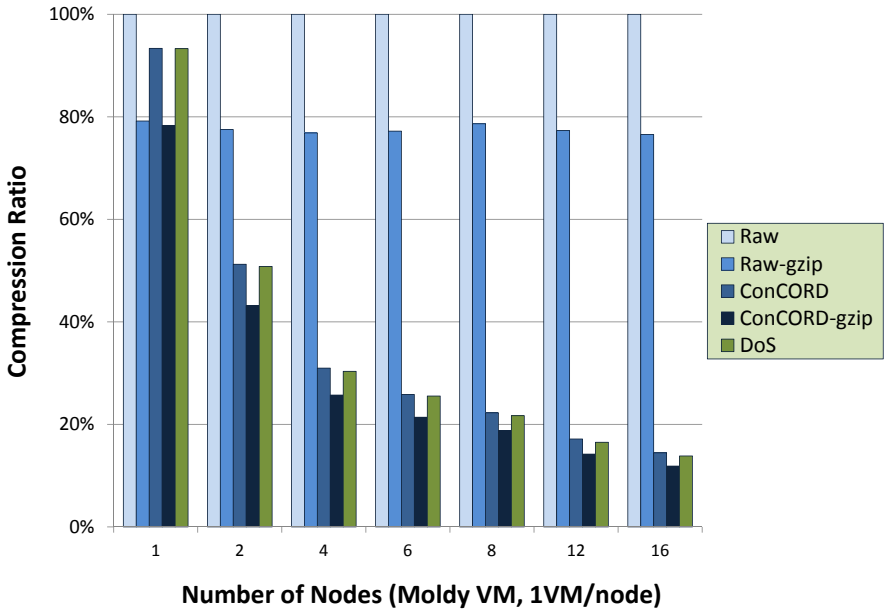


Figure 8.3: The compression ratio of the checkpoint as a function of number of nodes, for Moldy application, which has significant amount of inter-node content sharing.

checkpoint.

Checkpoint Time

I also measured the time needed to checkpoint a set of VMs by using three checkpointing approaches: raw checkpointing, raw checkpointing with gzip and the content-aware checkpoint. To avoid the large latency to write memory contents to a network file server, the memory blocks are checkpointed to a local RAM disk. This help us to factor out the impact of a parallel file system implementation in our final results. In the tests using raw checkpointing, each VM writes its memory to local RAM disk independently, while in the tests using raw checkpointing with gzip, each individual checkpoint file is compressed locally in parallel. I measured the checkpoint time in two scenarios. The first scenario is

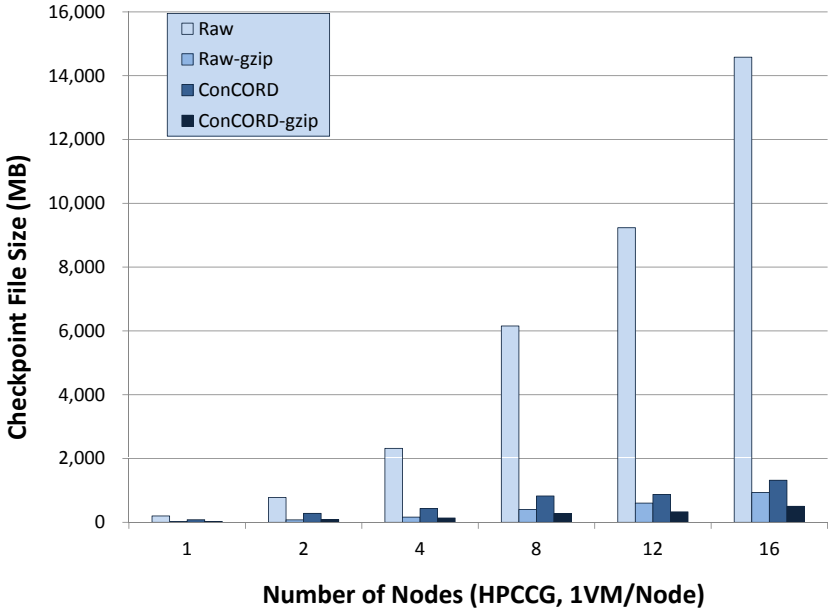


Figure 8.4: The checkpoint file size achieved by different checkpoint approaches, as a function of number of nodes, for the HPCCG application which presents a significant amount of intra-node content sharing.

checkpointing a fixed number of VMs (8 VMs in these cases), with each VM allocated different memory sizes in each test. The second scenario is checkpointing a different number of VMs at a time, with each VM having a fixed memory size (1 GBytes). In the second scenario, the number of physical nodes that ConCORD is running on is kept the same as the number of VMs, e.g, when checkpointing 4 VMs, 4 xDaemon instances are started on separate physical nodes. In addition, in each test, I executed the service command in both interactive-mode and batch-mode, and measured the times spent on both modes. Batch mode reflects the time purely spent on the execution of the service command, without actually performing any disk writes at all.

Figure 8.9 shows the results for the fixed number of VMs, and Figure 8.10 shows re-

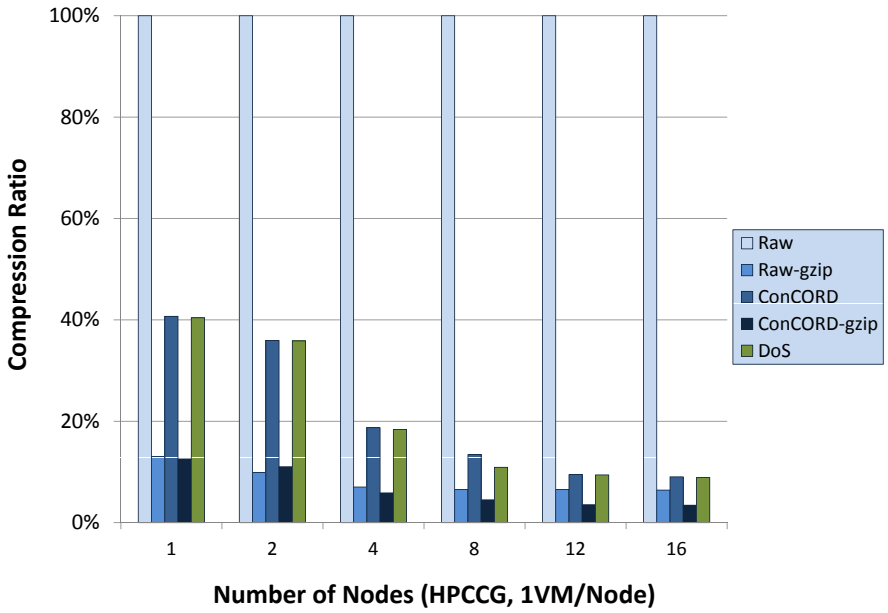


Figure 8.5: The compression ratio of different checkpoint approaches, as a function of number of nodes for the HPCCG application, which presents a significant amount of intra-node content sharing.

sults from the variable number of VMs. The results from checkpointing a fixed number of VMs show that the checkpoint time is linear with the total memory size of the VMs. For the variable number of VMs, the checkpoint time is almost constant as the number of VMs increases. This suggests that content-aware checkpointing can achieve good scalability as the number of checkpointed VMs increases. In addition, the raw checkpointing approach with using gzip to compress files uses significantly more time than the content-aware checkpointing. This suggests that content-aware checkpointing can achieve almost the same compression ratio as raw checkpointing with compression but uses significant less amount of time.

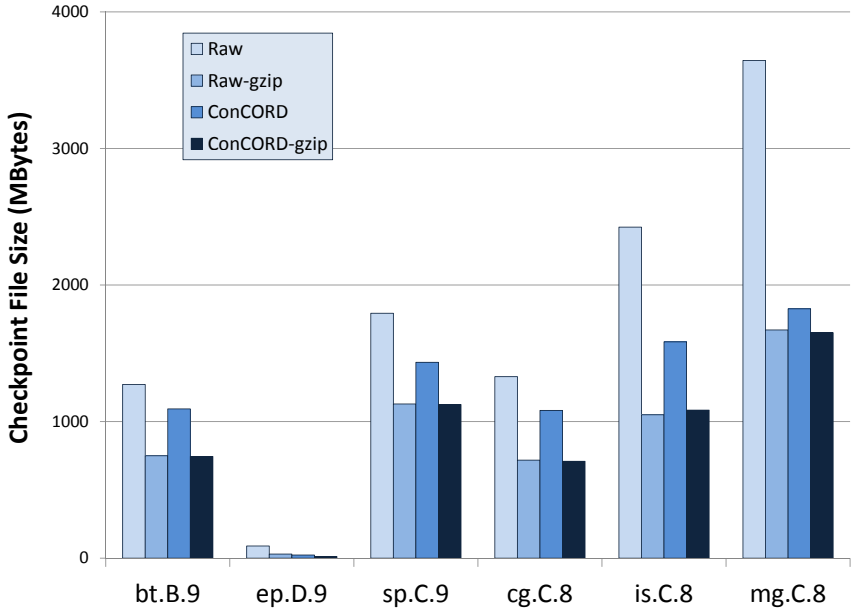


Figure 8.6: The checkpoint file sizes achieved by different checkpoint approaches as a function of number of nodes for various NAS benchmarks, which have moderate to little content sharing. The names of tested applications are shown in x-axis, for example, *bt.B.9* stands for running *bt* test application in 9 VMs, with each VM in one node.

8.3 Conclusion

In this chapter, I proposed the content-aware group checkpoint service, which leverages the content sharing across the checkpointed VMs and other VMs to significantly reduce the size of checkpoints. I have built the content-aware checkpoint using ConCORD content-aware service command. The implementation is very simple, comprising only about 230 line of code. This shows that the content-aware service command model enables service programmers to create effective content-aware services with minimal effort. These services can efficiently exploit and utilize memory content sharing without worrying about the complicated implementation of synchronization and parallelization of the jobs collec-

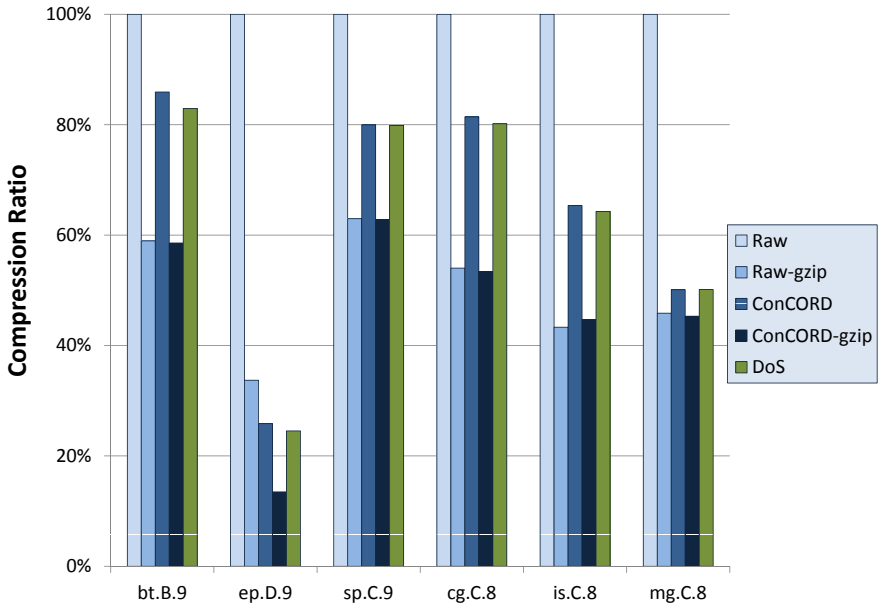


Figure 8.7: The compression ratio of various checkpoint strategies as a function of number of nodes various NAS benchmarks, which have moderate to little content sharing. The names of tested applications are shown in x-axis, for example, *bt.B.9* stands for running *bt* test application in 9 VMs, with each VM in one node.

tively.

My evaluation shows that the content-aware checkpoint service can significantly reduce the checkpoint size if there is a significant amount of content sharing among checkpointed VMs. Even in the worst case with no sharing at all, it has only a tiny and constant overhead. The time to checkpoint a set of VMs show the service scales well as the number of VMs increases, which is significantly less that using raw checkpointing with compression.

Finally, the implementation of the content-aware group checkpointing also serves as a template for building similar services on top of the content-aware service command model.

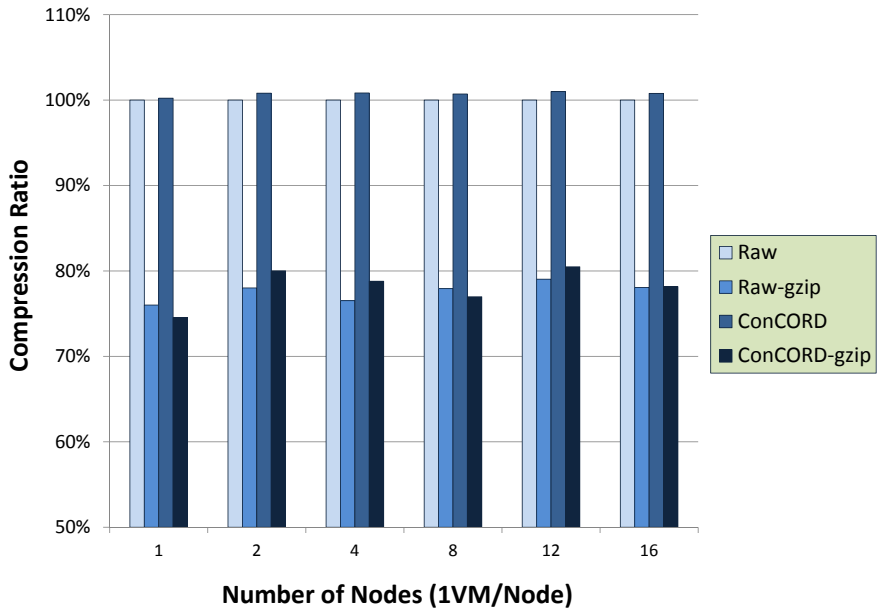


Figure 8.8: The compression ratio of various checkpoint strategies as a function of number of nodes with no memory content sharing inside or across VMs at all. Checkpoint generated by content-aware checkpointing are less than 3% larger than raw checkpointing. This overheads keeps constant as the number of VMs increases.

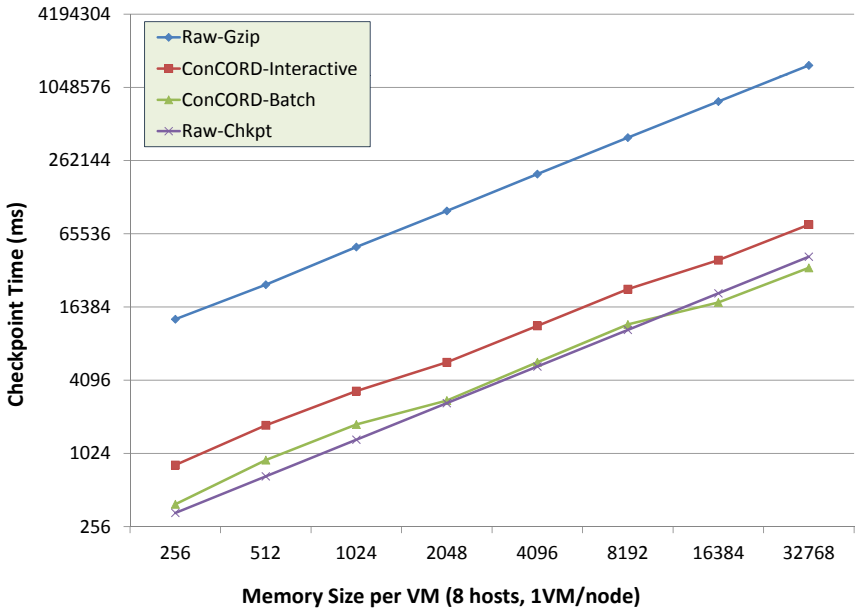


Figure 8.9: The service time for checkpointing a fixed number of VMs with different memory sizes. The service times seen in three checkpointing approaches are all linear with the total memory size of VM.

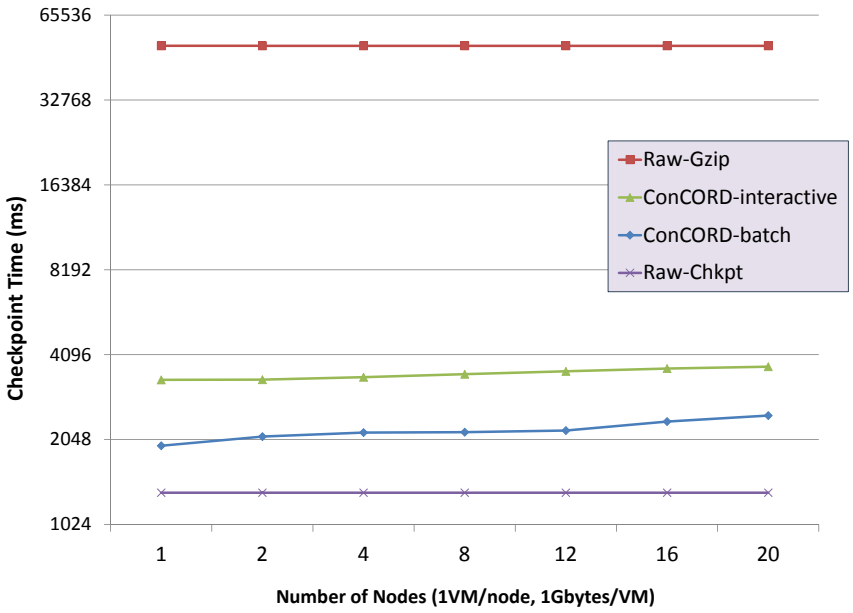


Figure 8.10: The service time for checkpointing different numberz of VMs with fixed memory size in each VM. The number of physical nodes ConCORD is running is the same as the number of VMs checkpointed. The service times seens in the content-aware checkpointing are almost constant as the number of VMs increases.

Chapter 9

Related Work

In this chapter, I will elaborate various previous work that are related to my discussion in this thesis.

9.1 Virtualization in HPC

Virtualization is about separating the physical from the logical. Traditionally, the software is bound to the hardware, allowing better utilization of the resources at hand, but on the other hand creating compatibility issues and limitations. The virtualization approach dictates the separation of the two, to allow higher compatibility and even independence of the software from the hardware running it.

Virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems by maximizing system flexibility and utility to a wide range of users [37, 52, 92]. In addition, there are many examples of the benefits available from a virtualization layer [89] for HPC.

Many of the motivations for virtualization in data centers apply also equally to HPC systems, for example, allowing users to customize their OS environment (e.g. between full-featured OSes and lightweight OSes). Additionally, virtualization allows multiplexing of less-demanding users when appropriate. Finally, virtualization is relevant to a number

of research areas for current petascale and future exascale systems, including reliability, fault-tolerance, and hardware-software co-design. The adoption of virtualization in HPC systems can only occur if it has minimal performance impact in the most demanding uses of the machines, specifically running the capability applications that motivate the acquisition of petascale and exascale systems. Virtualization can not succeed in HPC systems unless the performance overheads are truly minimal and that those overheads that do exist do not compound as the system and its applications scale up.

There has been considerable interest, both recently and historically, in applying existing virtualization tools to HPC environments [106, 30, 37, 52, 122, 123, 145]. while most of this work has been exclusively in the context of adapting or evaluating Xen and KVM on cluster platforms. Palacios and Kitten are a new OS/VMM solution developed specifically for HPC systems and applications [72, 71].

9.2 Content-based memory sharing

Many studies have shown the benefits of using memory sharing between VMs collocated on the same host. Transparent page sharing in a virtual machine hypervisor was first implemented in the Disco system [14]. It required guest operating system modification and detected identical pages based on factors such as origin from the same location on disk. In Disco, reading from a special copy-on-write disk involves checking to see if the same block is already present in main memory and, if so, creating a shared mapping to the existing page.

Content-based page sharing was introduced in VMware ESX [133] and Xen [63]. These implementations use background hashing and page comparison in the hypervisor to transparently identify identical pages. Potemkin [132] uses flash cloning and delta virtualization to enable a large number of mostly-identical VMs on the same host. Flash cloning

creates a new VM by copying an existing reference VM image, while delta virtualization provides copy-on-write sharing of memory between the original image and the new VM. Difference Engine [45] has demonstrated that even higher degrees of page sharing can be obtained by sharing portions of similar, but not identical pages.

Satori [85] implements memory sharing mechanism in the Xen environment by detecting opportunities for page sharing while reading data from a block device.

Kernel Samepage Merging (KSM) [8] lets the system share identical memory pages amongst different processes. It can also be used in conjunction with QEMU/KVM to share identical regions of memory between multiple co-located VMs. KSM finds duplicate pages by scanning through the memory, and then merges the duplicate pair into a single page, which is marked as "copy-on-write".

The above works have the goal of reducing memory pressure on individual nodes by deduplicating instances of intra-node memory content sharing. In contrast, the ConCORD system detects and tracks both intra-node *and inter-node* sharing of memory content across all the nodes. Additionally, we have argued that such detection and tracking should be factored into a separate service, on top of which other services, including deduplication, can be built.

The work cloest to ours is Memory Buddies [136], which uses memory fingerprinting to discover VMs with high sharing potential and then co-locates them on the same host. It uses a centralized controller to gather fingerprints from each node, which is likely to limit scalability. In contrast, ConCORD uses a scalable distributed hashing approach.

LBFS [87] exploits similarities between files or versions of the same file to save bandwidth. It avoids sending data over the network when the same data can already be found in the server's file system or the client's cache. LBFS uses this technique to reduce bandwidth compared to traditional network file systems on common workloads.

A number of storage systems use data deduplication to reduce their storage footprint

or to reduce the volume of data transferred across the network. These storage systems often adopt a "content addressable storage (CAS)" approach which names the data-blocks based on their content. In this context, hashing is used as a naming technique: data-block names are simply the hash value of the block's data. To store a new file, the system divides the file into blocks, hashes them, and compares the blocks' hashes with the hashes of the already stored blocks. This way the system identifies new data blocks and uses references to already stored blocks thus reducing the storage costs. CAS systems are divided into two categories, fixed-length and variable-length chunk. Venti [103], CASPER [124] and LBCAS [64] use fixed-length chunk. DeepStore [144] and NEC-hydra [29] use variable-length chunk.

Finally, ConCORD is not a distributed memory sharing/caching system such as Memcached [2] or RAMCloud [95]. Such general-purpose distributed memory caching systems are used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read.

9.3 Virtual machine migration

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centers and clusters. It allows a clean separation between hardware and software, and facilitates system maintenance, load balancing, fault tolerance and power-saving.

Virtual machine migration for workstations was first discussed in [112]. It presents mechanisms for offline migration of multiprogrammed workloads running on workstations. In offline migration, VMs are suspended, state is stored in temporary storage and then transferred to the destination system, where execution is resumed. This technique involves a large downtime and server based applications avoid it to maintain customer

satisfaction.

[19] discuss live migration of virtual machines using pre-copy techniques for commercial workloads. They introduce the concept of a writable working set (WSS) and report love downtimes. [51] presents an implementation of pre-copy live migration of on Xen using InfiniBand and evaluate it for NAS parallel benchmarks (MPI) running on clusters using one core per node. [89] discusses proactive fault tolerance mechanisms for scientific applications using live migration. SnowFlock [66] propose a mechanism for quickly cloning live VMs on new host machines in a local network, using demand-paging and multicast distribution of data.

Post-copying techniques for live migration are discussed by [49]. In post-copy live migration, the remote machine is started and processes are migrated without copying the memory pages, which are copied on demand. The downside of this approach is the significant slowdown that the migrated machine/application might suffer. First touches of a page on the destination system usually result in a network transfer, which has high startup costs. As opposed to pre-copy which is bandwidth bound, post-copy is a latency bound technique. In post-copy approach, all memory pages are transferred only once during the whole migration process and the baseline total migration time is achieved.

Virtualization can provide significant benefits in data centers by enabling virtual machine migration to eliminate hotspots. [135, 96] propose some virtual machine migration strategies to balance VM loads among physical nodes. Based on periodically collected resource usage status, some virtual machines are migrated from overloaded machines to light-loaded ones. In addition, by monitoring health status of nodes, such as temperatures or disk error logs, node failures can be anticipated.

A proactive scheme for fault tolerance [89] migrates VMs from unhealthy nodes to healthy ones using live migration of virtual machines. Remus [22] provides instantaneous failover by keeping an up-to-date replica of a VM in a separate host.

Live migration of virtual machines also provides a flexible way to implement power budget, power reservation and power-saving [91, 47, 50]. By consolidating VMs on several light-loaded physical machines, some idle nodes can be powered off.

A checkpointing/recovery and trace/replay approach (CR/TR-Motion) is proposed [78] to provide fast VM migration. The approach transfers execution trace file in iterations rather than dirty pages, which is logged by a trace daemon. Apparently, the total size of all log files is much less than that of dirty pages. So, total migration time and downtime of migration are drastically reduced.

Live Gang Migration [26] optimizes the live migration of a group of co-located VMs on the same host, by deduplicate the identical memory pages in these co-located VMs before migrate them.

VMFlock [5] presents a migration service optimized for cross-datacenter transfer and instantiation of groups of VMs images that comprise an application-level solution. VMFlock employs two main techniques: First, data deduplication within the VMFlock to be migrated, and among the VMs in the VMFlock and the data already present at the destination datacenter. Second, accelerated instantiation of the application at the target datacenter after transferring only a partial set of data blocks and prioritization of the remaining data based on previously observed access patterns originating from the running VMs.

Shrinker [107] is a live virtual machine migration system leveraging the memory content sharing between VMs in geo-distributed data centers to improve migration efficiency between data centers interconnected by wide-area networks. It uses distributed content-based addressing to detect memory pages of the migrated VM that are already available on the destination site, removing the need to transfer these pages over a WAN.

9.4 Parallel and distributed primitives

MapReduce [25] is a software framework introduced by Google to support distributed computing on large data sets for processing highly distributable problems across huge datasets using a large number of computers. In the MapReduce programming model, user specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines.

A distributed hash table (DHT) [116, 23] is a class of a decentralized distributed system that provides a lookup service similar to a hash table. <key, value> pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Distributing information within networks can be complicated if hosts have only limited knowledge of the properties of the network. This leads to problems when it is highly important that a specific information has to reach one particular host or all hosts within the entire network. Epidemic algorithms follow the paradigm of nature by applying simple rules to spread information by just having a local view of the environment. According to this fact, these algorithms are easy to implement and guarantee message propagation in heterogeneous and not all the time coherent environments

Traditionally multicast communication has been used to transmit data to a number of receivers. The inherent location-transparency of multicast also makes its use attractive for peer to multi-peer communication and resource location and retrieval. With multicast

communication it is possible to implement distributed systems without any explicit need to know the precise location of data. Instead, peers find each other by communicating over agreed upon communication channels. To find a particular data item, it is sufficient to make a request for the data on the agreed upon multicast channel and any node that holds a replica of the data item may respond to the request. This property makes multicast communication an excellent choice for building a system that replicates data.

9.5 Fault tolerance in large-scale systems

Some applications can benefit from algorithm-based checkpoint free techniques [16]. Such techniques are very efficient and should be applied whenever possible. However, many applications need other resiliency schemes. Checkpoint/Restart is one of the most commonly used HPC fault-tolerance mechanism. Checkpoint/Recovery protocols periodically record process or system states to storage devices that survive tolerated failures.

However, as computing systems increase in scale, the likelihood of a failure impacting the system increases significantly requiring more frequent checkpoints. Checkpoint/Restart overhead must be reduced in order to make it to be employed for future extreme scale systems.

Many optimizations have been proposed in checkpointing strategies, for example, diskless checkpointing [101, 13, 15, 99, 80] was proposed as a solution to avoid the I/O bottleneck, concurrent checkpointing [58, 74, 82], checkpointing filesystems [11, 1] remaps applications preferred data layout to one which is optimized for the underlying file systems thus achieving reduction of checkpoint time, Remote checkpointing [147] leverages network resources to save checkpoints to remote checkpoint servers providing performance gains in environments where I/O bandwidth to the network is more abundant than that to local storage devices. These strategies generally reduce checkpointing latency without

actually reducing the size of checkpoints. The other strategies include memory exclusion [100], incremental checkpointing or speculative checkpointing [111, 4, 33], which actually reduce the latency of checkpointing by reducing the amount of data to commit.

Finally, there is multilevel checkpointing [86, 9], which analyzes the severity of failures and uses that to determine and control checkpointing to multiple storage targets, including memory-based checkpoints, local checkpoint storage and remote checkpoints. It shares some of the advantages and disadvantages of memory-based checkpointing and local storage techniques. Unlike these techniques, however, multi-level checkpointing has the flexibility to choose between multiple levels of storage based on system design parameters, making it a promising technique for exascale systems.

Besides checkpointing, object replication is also commonly employed to enhance the availability of data-intensive services [62, 20, 24, 90, 129, 115, 146] in distributed systems. In particular, many systems use a fixed number of replicas for every object. As an alternative to full object replication for data redundancy, erasure coding [97, 98] represents an object by n fragments, out of which any m fragments can be used to reconstruct the object.

Process Replication, also known as redundant computation or state machine replication, is another technique for tolerating faults and enhance availability in distributed and mission critical systems [113, 126, 32]. In this approach, a process's state is replicated such that if the process fails, its replica is available or can be generated to assume the original process's role without disturbing the other application processes. State machine replication offers a different set of trade-offs compared to rollback recovery techniques. In particular, it completely masks a large percentage of system faults, preventing them from causing application failures without the need for rollback. Process replication can be costly in terms of space if replicas have dedicated resources or time if replicas are co-located with other primary processes.

Chapter 10

Conclusion

This dissertation has described ConCORD, a distributed system which can dynamically track inter-node memory content sharing in a parallel system, as well as the content-aware service command model, a model that enables construction of various HPC services to effectively exploit memory content sharing in large-scale systems with minimal effort.

Scientific workloads running in a large-scale parallel system usually generate massive memory footprints. The study I have presented in this dissertation, along with some related study from other research groups, have shown that significant amounts of memory content sharing, both intra-node and inter-node, exist in these systems. In my dissertation, I argued and elaborated that many critical services that enhance performance, reliability, or power in high performance computing can be enabled, simplified or improved by leveraging those memory content sharing, particularly *inter-node* sharing.

However, all of these services can be enabled or enhanced only if there is a facility to dynamically track the sharing of memory content in the system. We further argued that such memory content sharing detection and tracking should be factored into a separate service, i.e, a facility that continuously tracks memory content sharing across the machines should be included as a part of infrastructure in a large parallel system. All these content-aware services should be built on top of this facility.

My dissertation has focused on the feasibility of building such a dynamic content tracking facility, and the impact that such system would have to high performance computing systems. To demonstrate the feasibility, I have designed, implemented and evaluated ConCORD, a distributed system that can dynamic detect and track inter-node memory content sharing with low overheads in large scale parallel systems. ConCORD exposes powerful interfaces to allow users and developers in HPC community to build their own content-aware services on top of ConCORD.

ConCORD serves as a base system that makes other content-aware services able to be built on top of it and effectively exploit content sharing to improve their performance. To better enable this, ConCORD exposes two simple, yet powerful interfaces to allow users and service applications to build their own content-aware services on top of it. These include the content-sharing query interface to examine the memory content sharing information, and the content-aware service command to enable easy building of content-aware services with minimal effort.

The query interface enables other service applications or users to check the amount of memory content sharing and examine the location of shared regions across a group of nodes. Based on this content sharing information from ConCORD, many services can be built to effectively exploit the memory content sharing existing in their target nodes to improve themselves.

To further lower the barrier to build a content-aware service on top of ConCORD, the content-aware service command model is proposed and incorporated into ConCORD to minimize the efforts spent on building a content-aware service. Content-aware service command is a distributed concurrent service model and associated implementation that enables effectively building of content-aware services in large-scale parallel systems. It provides a powerful interface that enables many HPC services to effectively exploit and utilize memory content redundancy existing in the system with very little effort. Content-aware

services built on top of service commands are automatically parallelized and executed on the parallel systems. This enables service developers to build and enhance their services to maximally exploit and utilize the memory content sharing without worrying about the complication of the implementation.

Finally, to demonstrate the power of the content-aware service command model, I have proposed a content-aware group checkpointing service and discussed its implementation using the content-aware service command. I have demonstrated that very few effort is needed to build such content-services.

10.1 Summary of contributions

The contributions of my dissertation can be summarized as follows:

- **Memory content sharing study.** I have presented a detailed experimental study on the memory content sharing of a range of parallel applications and application benchmarks. The study considered both forms of inter-node and intra-node memory content sharing actually exist in parallel workloads, at different scales, and across time. The results of the study have shown both intra-node and inter-node memory content sharing are common in parallel applications, especially, there is substantial additional sharing across nodes beyond that of sharing within individual nodes. This study has demonstrated that there is opportunity for exploiting inter-node memory content sharing to benefit many areas in high performance computing and distributed systems.

- **The concept of content-aware service.** I introduced the concept of content-aware services, of these services that can be enabled, simplified or improved by leveraging memory content sharing existing in such systems. I discussed the possible approaches that can be taken to build three of such content-aware services in HPC

systems to exploit the memory content sharing, including the virtual machine co-migration, content-aware group checkpointing and the memory replication service.

- **Articulate benefit of factoring out content sharing detection service.** I have articulated the benefits of factoring memory content sharing detection and tracking into a separate service, i.e, a facility that continuously tracks memory content sharing across the machines should be included as a part of infrastructure in a large parallel system. All these content-aware services should be built on top of this facility.

- **ConCORD.** I have designed and implemented ConCORD system, with thorough evaluation of its performance of various major components and interfaces. Targeted to fit into large-scale parallel systems, ConCORD exposes powerful interfaces to allow users and service application programmers in HPC community to build their own content-aware services on top of it. I have presented the thorough evaluation to show that ConCORD works effectively and maintains low overheads as the system size grows. Through building and evaluating of ConCORD, I demonstrated that factoring inter-node memory content sharing tracking into separate service is possible with minimal performance impact.

- **Content-aware service command model.** I have proposed the concept of content-aware service command model. The content-aware service command model is a powerful model that enables effectively construction of content-aware services in large-scale parallel systems. I have implemented the content-aware service command execution system, integrated it into ConCORD and evaluated its performance. My evaluation on the service command models showed that both the times taken to execute a service command and the overheads during the execution of a service command scale well as the system size increases.

- **Content-aware group checkpointing.** I proposed content-aware group checkpointing service which leverages the memory content sharing across the checkpointing nodes to reduce the checkpoint size. I implemented the content-aware checkpoint using ConCORD content-aware service command. The implementation is simple and straightforward, which demonstrates the power of the content-aware service command model, and shows that very few effort is needed to build such content-services. I have evaluated the performance of my implementation of group checkpointing service. The results shows that the content-aware checkpoint service can significantly reduce the checkpoint size if there is a significant amount of content sharing among checkpointed VMs. Finally, the implementation of the content-aware group checkpointing also serves as a template for building similar services on top of the content-aware service command model.

Beyond this thesis, I have done many works on enhancing the functionality and improving the performance of virtualization systems for high performance computing in my Ph.D research, which include:

- **VNET/P.** I developed the latest version of VNET/P [138, 21, 139] and embedded it into current Palacios VMM for our proposed use virtual overlay network in HPC environment to bridge HPC to cloud computing. It is able to deliver extremely high performance over 10 Gbit Ethernet. It can also provide a standard Ethernet interface to guests and applications based on Infiniband and other special purposed interconnection which appear commonly in HPC and supercomputers. A more elaboration on VNET/P is described in appendix B of this dissertation. In addition, I have demonstrated an approach to further improve throughput and reduce latency of the virtual overlay network by injecting overlay's functional components into guest operating system through VMM-based code injection without user or guest's coop-

eration [46].

- **Palacios.** V3VEE project is designing and implementing Palacios, an OS independent VMM. I have been contributing to Palacios for five years. I have implemented most of networking components of Palacios, including the original implementation of virtual PCI, porting of VNET into Palacios, implementing of socket interface in Palacios, implementing of virtIO based virtual network devices and Dom0 guest support. I have also been a major contributor to the effort of embedding Palacios into Linux host.

- **I/O virtualization approaches.** I have implemented and evaluated several mechanisms for improving the functionality and performance of I/O virtualization in various computing areas.

  - **Virtual passthrough I/O (VPIO).** I introduced the terminology of VPIO [142, 143], and I have designed and implemented the VPIO prototype system in Palacios with two example device models for NE2000 and RTL8139 network devices.

  - **Virtual WiFi.** Virtual WiFi [141] is a wireless LAN virtualization approach that enables each virtual machine to establish its own connection with self-supplied credentials through one physical wireless LAN network interface. I have implemented and evaluated the virtual WiFi prototype system.

10.2 Future work

My future research plans include both extending ConCORD and investigating more content-aware services that can be enabled, simplified or improved based on it.

- **Extending ConCORD for wider target systems.** In my dissertation, I mainly considered deploying ConCORD in large-scale parallel systems, however, the idea behind ConCORD is not merely intended for parallel systems. It can apply to other large-scale systems, for example, cloud computing systems. Optimization of ConCORD to be running on commercialized cloud computing systems will be one direction of my future work.

- **Investigating more content-aware services.** In my dissertation, I have demonstrated two of content-aware services that can be enabled by ConCORD, content-aware group checkpointing and collective virtual machine co-migration. Besides, there are many other services in HPC systems that could be enabled or improved by leveraging ConCORD, several of these are briefly discussed below. The other direction of future work would be to investigate all of these possible services that can be enabled, improved or simplified by ConCORD.

  – **Memory replication system for high availability.** Redundant computation, object replication and process replication are commonly employed to enhance the availability and fault tolerance of data-intensive services, such as on wide-area networks and enterprise-area networks with decentralized management, and on centrally-managed local-area clusters, high performance computing and mission critical systems. In these replication systems, all memory objects are assigned a certain number of replicas and are stored in the memory of a local or remote machine. The system maintains certain level of redundancies automatically as application executing. This usually consumes much extra memory space. Since memory is regarded as a key budget and power constraint in exascale systems, it is unclear if the benefits of replicating memory justify the qualitative gains.

However, with the knowledge of memory content sharing across nodes provided by ConCORD, there is a potential to reduce large number of explicitly created replicas for those memory blocks which are already naturally replicated in other nodes in the system. That is, we can potentially use the applications' own redundancy instead of making more.

– **Determine the best points to perform system services.** Application workloads could have diverse memory content share patterns during different phases of its execution. In some phases, the degree of content sharing across parallel nodes could be higher than the other phases. There is potential gain in many content-aware services, such as system checkpointing or migration, during such phases. Since ConCORD can monitor and potentially predict the degree of memory content sharing across nodes for each phase of the application execution in the system. With this monitoring and predicting capabilities, ConCORD could serve as a consultant to tell at which point it is good time for the system to perform certain content-aware services to maximally utilize the content-sharing across nodes.

– **Power efficient system enhancement.** To reduce the power usage of a cluster or a system, one potential approach is to lower the process' voltage of its nodes. However, this would reduce the reliability of the system components. With ConCORD and an optimized memory replication system based on it, it might be possible to increase the availability of the entire parallel system and transparently provide higher availability to system and application levels by hiding reduced reliability of the machines due to intended voltage lowering for power reducing.

Appendices

Appendix A

Content-aware Checkpoint Implementation Code

This appendix gives the complete code for content-aware checkpointing service. This is to illustrate how complex and straightforwardly this kind of service can be implemented using service command model.

Listing A.1: checkpoint-ops.c

```
/* checkpoint_ops.c
 * Lei Xia
 */
#include <core/xcord.h>
#include <core/xcord_service.h>
#include <core/xcord_service_ops.h>
#include <pfs_client.h>

struct ckpt_arg {
  char shared_ck_file[NAME_LEN];
  char local_ck_file_prefix[NAME_LEN];
  sint32_t file_id;
};

static int
f_parse_input(struct xcord_service_cmd * xcommand,
              char * input_file)
{
  FILE * ck_conf;
  struct ckpt_arg * ckarg;
```

```
  ck_conf = fopen(input_file, "r");
  if(ck_conf == NULL) {
    PrintError("Fail to open %s\n", input_file);
    return -1;
  }

  ckarg = xcord_malloc(sizeof(struct ckpt_arg));
  ckarg->data = 0;
  ckarg->file_id = -1;

  fscanf(ck_conf, "%s, %s",
         ckarg->shared_ck_file,
         ckarg->local_ck_file_prefix);

  xcommand->arg_size = sizeof(struct ckpt_arg);
  xcommand->arg = ckarg;

  return 0;
}

static int
f_collective_start(uint8_t component_type,
                   uint16_t c_id,
                   struct xcord_hash_list * hashes,
                   void ** arg, uint16_t * arg_size)
{
  if(component_type == XCORD_COMPONENT_SVM
      || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg = (struct ckpt_arg *)(*arg);
    ck_arg->file_id = (sint32_t)
        pfs_open_file(ck_arg->shared_ck_file);
    if(ck_arg->file_id < 0) {
      PrintError("Fail to open global shared checkpoint file\n");
      return -1;
    }

    *arg = (void *)ck_arg;
    *arg_size = sizeof(*ck_arg);
  }

  return 0;
}

static int
f_collective_cmd(uint16_t vm_id,
                 struct xcord_hash_item * hash,
```

```
                char * mem_block, int block_size,
                void ** arg, uint16_t * arg_size) {
  struct ckpt_arg * ck_arg = (struct ckpt_arg *)(*arg);;
  sint64_t offset = 0;

  offset = pfs_append_file(ck_arg->file_id,
                           mem_block, block_size);
  if(offset < 0) {
    return -1;
  }

  hash->arg = xcord_malloc(sizeof(sint64_t));
  hash->arg_size = sizeof(sint64_t);
  *((sint64_t *)hash->arg) = offset;

  return 0;
}

static int
f_collective_finalize(uint8_t component_type,
                      uint16_t c_id,
                      struct xcord_hash_list * hashes,
                      void ** arg, uint16_t * arg_size) {
  if(component_type == XCORD_COMPONENT_SVM
       || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);

    pfs_close_file(ck_arg->file_id);
        ck_arg->file_id = -1;

    return 0;
  }
}

static int
f_local_start(uint8_t component_type,
              uint16_t c_id,
              struct xcord_hash_list * hashes,
              void ** arg, uint16_t * arg_size) {

  if(component_type == XCORD_COMPONENT_SVM) {
    char local_ck_file[NAME_LEN];

    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);
    snprintf(local_ck_file, NAME_LEN, "checkpoints/%s-vm%d.ck",
             ck_arg->local_ck_file_prefix, c_id);
```

```
    ck_arg->file_id = open(local_ck_file, O_WRONLY | O_CREAT);
    if(ck_arg->file_id < 0) {
      PrintError("Fails to open local chekpt file %s\n",
                 local_ck_file);
      return -1;
    }
  }

  return 0;
}

static int
f_local_cmd(uint16_t vm_id,
            struct xcord_hash_item * hash,
            uint32_t page_no,
            char * mem_block, int block_size,
            void ** arg, uint16_t * arg_size) {
  struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);
  char header[32];
  int size = 0;
  sint64_t offset = -1;

  /* Local ckpt block layout:
   *   page_no(4B):Hash(16B):Offset(8B):[MEMORY_DATA]
   */
  memcpy(header, (char *)&page_no, sizeof(page_no));
  size += sizeof(page_no);
  memcpy(header+size, hash->hash_val, HASH_LEN);
  size += HASH_LEN;

  if(hash->s_tag == SERVICE_HASH_ITEM_COMPLETED
       && hash->arg_size == sizeof(sint64_t)) {
    offset = *((sint64_t *)hash->arg);
  }

  memcpy(header+size, (char *)&offset, sizeof(offset));
  size += sizeof(offset);

  if(write(ck_arg->file_id, header, size) < 0) {
    PrintError("Fails to write to local checkpt file");
    return -1;
  }

  /* the block has not been writen to shared checkpoint
   * file by any VMs, write to local VM checkpoint file
   */
  if(offset == -1) {
```

```
    if(write(ck_arg->file_id, mem_block, block_size) < 0) {
      PrintError("Fails to write to local checkpt file\n");
      return -1;
    }
  }

  return 0;
}

static int
f_local_finalize(uint8_t component_type, uint16_t c_id,
                 struct xcord_hash_list * hashes,
                 void ** arg, uint16_t * arg_size) {
  if(component_type == XCORD_COMPONENT_SVM
      || component_type == XCORD_COMPONENT_PVM) {
    struct ckpt_arg * ck_arg=(struct ckpt_arg *)(*arg);

    close(ck_arg->file_id);
  }

  return 0;
}

static int
f_parse_ret_arg(void * ret_arg, uint16_t arg_size){

  return 0;
}

struct xcord_service_ops checkpoint_ops = {
  .parse_input = f_parse_input,
  .collective_start = f_collective_start,
  .collective_cmd = f_collective_cmd,
  .collective_finalize = f_collective_finalize,
  .local_start = f_local_start,
  .local_cmd = f_local_cmd,
  .local_finalize = f_local_finalize,
  .parse_ret_arg = f_parse_ret_arg,
};

xcord_register_service_ops("checkpoint",
                           &checkpoint_ops)
```

Listing A.2: checkpoint.conf

```
/*
 * Below is the service Configuration File
 * checkpoint.conf
 */
##Service configuration file
name=Checkpoint
ops_name=checkpoint
svm=5:{"guest-node1-1","guest-node1-2","guest-node3-2","lei-guest-4"
   ,"tony-guest-1"}
pvm=3:{"tony-guest-2","someone-g-3","lei-guest-2"}
timeout=10
arg_file=checkpoint_arg.conf
```

Listing A.3: checkpoint-arg.conf

```
/* checkpoint_arg.conf */
checkpoint_shared_file.ck,
local_checkpoint_file
```

Appendix B

VNET/P: Bridging the Cloud and High Performance Computing Through Fast Overlay Networking

Cloud computing in the "infrastructure as a service" (IaaS) model has the potential to provide economical and effective on-demand resources for high performance computing. In this model, an application is mapped into a collection of virtual machines (VMs) that are instantiated as needed, and at the scale needed. Indeed, for loosely-coupled applications, this concept has readily moved from research [34, 110] to practice [93]. As we describe in Section B.2, such systems can also be adaptive, autonomically selecting appropriate mappings of virtual components to physical components to maximize application performance or other objectives. However, *tightly-coupled* scalable high performance computing (HPC) applications currently remain the purview of resources such as clusters and supercomputers. We seek to extend the adaptive IaaS cloud computing model into these regimes, allowing an application to dynamically span both kinds of environments.

The current limitation of cloud computing systems to *loosely-coupled* applications is not due to machine virtualization limitations. Current virtual machine monitors (VMMs) and other virtualization mechanisms present negligible overhead for CPU and memory

intensive workloads [52, 84]. With VMM-bypass [79] or self-virtualizing devices [104] the overhead for direct access to network devices can also be made negligible.

Considerable effort has also gone into achieving low-overhead network virtualization and traffic segregation within an individual data center through extensions or changes to the network hardware layer [88, 40, 61]. While these tools strive to provide uniform performance across a cloud data center (a critical feature for many HPC applications), they do not provide the same features once an application has migrated outside the local data center, or spans multiple data centers, or involves HPC resources. Furthermore, they lack compatibility with the more specialized interconnects present on most HPC systems.

Beyond the need to support our envisioned computing model across today's and tomorrow's tightly-coupled HPC environments, we note that data center network design and cluster/supercomputer network design seem to be converging [3, 41]. This suggests that future data centers deployed for general purpose cloud computing will become an increasingly better fit for tightly-coupled parallel applications, and therefore such environments could potentially also benefit.

The current limiting factor in the adaptive cloud- and HPC-spanning model described above for tightly-coupled applications is the performance of the virtual networking system. Current adaptive cloud computing systems use software-based overlay networks to carry inter-VM traffic. For example, our VNET/U system, which is described in more detail later, combines a simple networking abstraction within the VMs with location-independence, hardware-independence, and traffic control. Specifically, it exposes a layer 2 abstraction that lets the user treat his VMs as being on a simple LAN, while allowing the VMs to be migrated seamlessly across resources by routing their traffic through the overlay. By controlling the overlay, the cloud provider or adaptation agent can control the bandwidth and the paths between VMs over which traffic flows. Such systems [119, 109] and others that expose different abstractions to the VMs [134] have been under continuous

research and development for several years. Current virtual networking systems have sufficiently low overhead to effectively host loosely-coupled scalable applications [31], but their performance is insufficient for tightly-coupled applications [94].

In response to this limitation, we have designed, implemented, and evaluated VNET/P, which shares its model and vision with VNET/U, but is designed to achieve near-native performance in the 1 Gbps and 10 Gbps switched networks common in clusters today, as well as to operate effectively on top of even faster networks, such as Infiniband and Cray Gemini. VNET/U and our model is presented in more detail in Section B.2.

VNET/P is implemented in the context of our publicly available, open source Palacios VMM [72], which is in part designed to support virtualized supercomputing. A detailed description of VNET/P's design and implementation is given in Section B.3. As a part of Palacios, VNET/P is publicly available. VNET/P could be implemented in other VMMs, and as such provides a proof-of-concept that overlay-based virtual networking for VMs, with performance overheads low enough to be inconsequential even in a tightly-coupled computing environment, is clearly possible.

The performance evaluation of VNET/P (Section B.4) shows that it is able to achieve native bandwidth on 1 Gbps Ethernet with a small increase in latency, and very high bandwidth on 10 Gbps Ethernet with a similar, small latency increase. On 10 Gbps hardware, the kernel-level VNET/P system provides on average 10 times more bandwidth and 7 times less latency than the user-level VNET/U system can.

In a related paper from our group [21], we describe additional techniques, specifically optimistic interrupts and cut-through forwarding, that bring bandwidth to near-native levels for 10 Gbps Ethernet. Latency increases are predominantly due to the lack of selective interrupt exiting in the current AMD and Intel hardware virtualization extensions. We expect that latency overheads will be largely ameliorated once such functionality become available, or, alternatively, when software approaches such as ELI [39] are used.

Although our core performance evaluation of VNET/P is on 10 Gbps Ethernet, VNET/P can run on top of any device that provides an IP or Ethernet abstraction within the Linux kernel. The portability of VNET/P is also important to consider, as the model we describe above would require it to run on many different hosts. In Section B.5 we report on preliminary tests of VNET/P running over Infiniband via the IPoIB functionality, and on the Cray Gemini via the IPoG virtual Ethernet interface. Running on these platforms requires few changes to VNET/P, but creates considerable flexibility. In particular, using VNET/P, existing, unmodified VMs running guest OSes with commonplace network stacks can seamlessly run on top of such diverse hardware. To the guest, a complex network of commodity and high-end networks looks like a simple Ethernet network. Also in Section B.5, we describe a version of VNET/P that has been designed for use with the Kitten lightweight kernel as its host OS. Kitten is quite different from Linux—indeed the combination of Palacios and Kitten is akin to a "type-I" (unhosted) VMM—resulting in a different VNET/P architecture. This system and its performance provide evidence that the VNET/P model can be successfully brought to different host/VMM environments.

Our contributions on VNET/P are as follows:

- We articulate the benefits of extending virtual networking for VMs down to clusters and supercomputers with high performance networks. These benefits are also applicable to data centers that support IaaS cloud computing.
- We describe the design and implementation of a virtual networking system, VNET/P, that does so. The design could be applied to other VMMs and virtual network systems.
- We perform an extensive evaluation of VNET/P on 1 and 10 Gbps Ethernet networks, finding that it provides performance with negligible overheads on the former,

and manageable overheads on the latter. VNET/P generally has little impact on performance for the NAS benchmarks.

- We describe our experiences with running the VNET/P implementation on Infiniband and Cray Gemini networks. VNET/P allows guests with commodity software stacks to leverage these networks.

- We describe the design, implementation, and evaluation of a version of VNET/P for lightweight kernel hosts, particularly the Kitten LWK.

Through the use of low-overhead overlay-based virtual networking in high-bandwidth, low-latency environments such as current clusters and supercomputers, and future data centers, we seek to make it practical to use virtual networking at all times, even when running tightly-coupled applications on such high-end environments. This would allow us to seamlessly and *practically* extend the already highly effective adaptive virtualization-based IaaS cloud computing model to such environments.

B.1 Related work

VNET/P is related to NIC virtualization, overlays, and virtual networks, as we describe below.

NIC virtualization: There is a wide range of work on providing VMs with fast access to networking hardware, where no overlay is involved. For example, VMware and Xen support either an emulated register-level interface [117] or a paravirtualized interface to guest operating system [83]. While purely software-based virtualized network interface has high overhead, many techniques have been proposed to support simultaneous, direct-access network I/O. For example, some work [79, 104] has demonstrated the use of self-virtualized network hardware that allows direct guest access, thus provides high per-

formance to untrusted guests. Willmann et al have developed a software approach that also supports concurrent, direct network access by untrusted guest operating systems [114]. In addition, VPIO [143] can be applied on network virtualization to allow virtual passthrough I/O on non-self-virtualized hardware. Virtual WiFi [141] is an approach to provide the guest with access to wireless networks, including functionality specific to wireless NICs. In contrast with such work, VNET/P provides fast access to an overlay network, which includes encapsulation and routing. It makes a set of VMs appear to be on the same local Ethernet regardless of their location anywhere in the world and their underlying hardware. Our work shows that this capability can be achieved without significantly compromising performance when the VMs happen to be very close together.

Overlay networks: Overlay networks implement extended network functionality on top of physical infrastructure, for example to provide resilient routing (e.g, [7]), multicast (e.g. [18]), and distributed data structures (e.g., [116]) without any cooperation from the network core; overlay networks use end-systems to provide their functionality. VNET is an example of a specific class of overlay networks, namely virtual networks, discussed next.

Virtual networking: Virtual networking systems provide a service model that is compatible with an existing layer 2 or 3 networking standard. Examples include VIOLIN [56], ViNe [125], VINI [10], SoftUDC VNET [59], OCALA [57], WoW [36], and the emerging VXLAN standard [81]. Like VNET, VIOLIN, SoftUDC, WoW, and VXLAN are specifically designed for use with virtual machines. Of these, VIOLIN is closest to VNET (and contemporaneous with VNET/U), in that it allows for the dynamic setup of an arbitrary private layer 2 and layer 3 virtual network among VMs. The key contribution of VNET/P is to show that this model can be made to work with minimal overhead even in extremely low latency, high bandwidth environments.

Connections: VNET/P could itself leverage some of the related work described above. For example, effective NIC virtualization might allow us to push encapsulation directly into the guest, or to accelerate encapsulation via a split scatter/gather map. Mapping unencapsulated links to VLANs would enhance performance on environments that support them. There are many options for implementing virtual networking and the appropriate choice depends on the hardware and network policies of the target environment. In VNET/P, we make the choice of minimizing these dependencies.

B.2 VNET model and VNET/U

The VNET model was originally designed to support adaptive computing on distributed virtualized computing resources within the Virtuoso system [27], and in particular to support the adaptive execution of a distributed or parallel computation executing in a collection of VMs potentially spread across multiple providers or supercomputing sites. The key requirements, which also hold for VNET/P, were as follows.

- VNET would make within-VM network configuration the sole responsibility of the VM owner.
- VNET would provide location independence to VMs, allowing them to be migrated between networks and from site to site, while maintaining their connectivity, without requiring any within-VM configuration changes.
- VNET would provide hardware independence to VMs, allowing them to use diverse networking hardware without requiring the installation of specialized software.
- VNET would provide minimal overhead, compared to native networking, in the contexts in which it is used.

The VNET model meets these requirements by carrying the user's VMs' traffic via a configurable overlay network. The overlay presents a simple layer 2 networking abstraction: a user's VMs appear to be attached to the user's local area Ethernet network, regardless of their actual locations or the complexity of the VNET topology/properties. Further information about the model can be found elsewhere [119].

The VNET overlay is dynamically reconfigurable, and can act as a locus of activity for an adaptive system such as Virtuoso. Focusing on parallel and distributed applications running in loosely-coupled virtualized distributed environments (e.g., "IaaS Clouds"), we demonstrated that the VNET "layer" can be effectively used to:

1. monitor application communication and computation behavior [43, 42]),
2. monitor underlying network behavior [44],
3. formulate performance optimization problems [121, 118], and
4. address such problems through VM migration and overlay network control [120], scheduling [76, 77], network reservations [70], and network service interposition [68].

These and other features that can be implemented within the VNET model have only marginal utility if carrying traffic via the VNET overlay has significant overhead compared to the underlying native network.

The VNET/P system described in this chapter is compatible with, and compared to, our previous VNET implementation, VNET/U. Both support a dynamically configurable general overlay topology with dynamically configurable routing on a per MAC address basis. The topology and routing configuration is subject to global or distributed control (for example, by the VADAPT [120]) part of Virtuoso. The overlay carries Ethernet packets encapsulated in UDP packets, TCP streams with and without SSL encryption, TOR privacy-preserving streams, and others. Because Ethernet packets are used, the VNET

abstraction can also easily interface directly with most commodity network devices, including virtual NICs exposed by VMMs in the host, and with fast virtual devices (e.g., Linux virtio network devices) in guests.

While VNET/P is implemented within the VMM, VNET/U is implemented as a user-level system. As a user-level system, it readily interfaces with VMMs such as VMware Server and Xen, and requires no host changes to be used, making it very easy for a provider to bring it up on a new machine. Further, it is easy to bring up VNET daemons when and where needed to act as proxies or waypoints. A VNET daemon has a control port which speaks a control language for dynamic configuration. A collection of tools allows for the wholesale construction and teardown of VNET topologies, as well as dynamic adaptation of the topology and forwarding rules to the observed traffic and conditions on the underlying network.

The last reported measurement of VNET/U showed it achieving 21.5 MB/s (172 Mbps) with a 1 ms latency overhead communicating between Linux 2.6 VMs running in VMware Server GSX 2.5 on machines with dual 2.0 GHz Xeon processors [68]. A current measurement, described in Section B.4, shows 71 MB/s with a 0.88 ms latency. VNET/U's speeds are sufficient for its purpose in providing virtual networking for wide-area and/or loosely-coupled distributed computing. They are not, however, sufficient for use within a cluster at gigabit or greater speeds. Making this basic VM-to-VM path competitive with hardware is the focus of this chapter. VNET/U is fundamentally limited by the kernel/user space transitions needed to handle a guest's packet send or receive. In VNET/P, we move VNET directly into the VMM to avoid such transitions.

B.3 Design and implementation

We now describe how VNET/P has been architected and implemented in the context of Palacios as embedded in a Linux host. Section B.5.3 describes how VNET/P is implemented in the context of a Kitten embedding. The nature of the embedding affects VNET/P primarily in how it interfaces to the underlying networking hardware and networking stack. In the Linux embedding, this interface is accomplished directly in the Linux kernel. In the Kitten embedding, the interface is done via a service VM.

B.3.1 Palacios VMM

VNET/P is implemented in the context of our Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available embeddable VMM designed as part of the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University, the University of New Mexico, Sandia National Labs, and Oak Ridge National Lab. Detailed information about Palacios can be found elsewhere [72, 69]. Palacios is capable of virtualizing large scale (4096+ nodes) with $< 5\%$ overheads [71]. Palacios's OS-agnostic design allows it to be embedded into a wide range of different OS architectures.

The Palacios implementation is built on the virtualization extensions deployed in current generation x86 processors, specifically AMD's SVM [6] and Intel's VT [127]. Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that comprise the PC platform. Due to the ubiquity of the x86 architecture Palacios is capable of operating across many classes of machines. Palacios has successfully virtualized commodity desktops and servers, high end Infiniband clusters, and Cray XT and XK supercomputers.

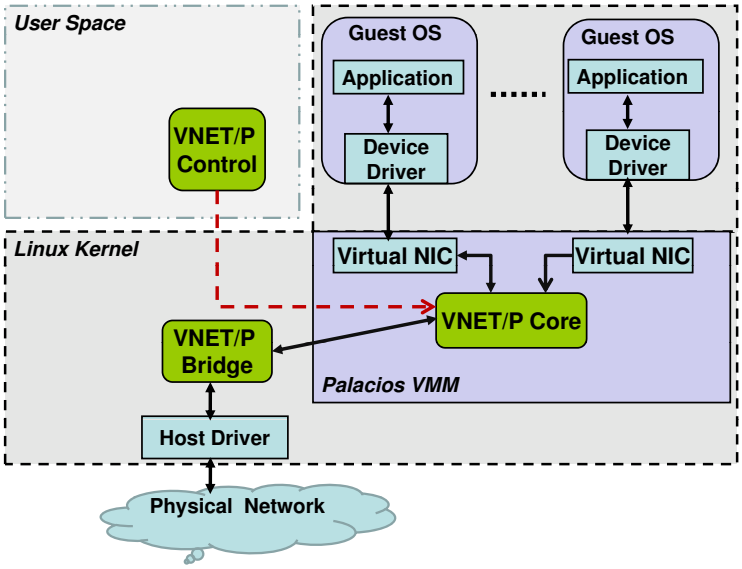


Figure B.1: VNET/P architecture

B.3.2 Architecture

Figure B.1 shows the overall architecture of VNET/P, and illustrates the operation of VNET/P in the context of the Palacios VMM embedded in a Linux host. In this architecture, *guests* run in *application VMs*. Off-the-shelf guests are fully supported. Each application VM provides a virtual (Ethernet) NIC to its guest. For high performance applications, as in this chapter, the virtual NIC conforms to the virtio interface, but several virtual NICs with hardware interfaces are also available in Palacios. The virtual NIC conveys Ethernet packets between the application VM and the Palacios VMM. Using the virtio virtual NIC, one or more packets can be conveyed from an application VM to Palacios with a single VM exit, and from Palacios to the application VM with a single VM exit+entry.

The *VNET/P core* is the component of VNET/P that is directly embedded into the

Palacios VMM. It is responsible for routing Ethernet packets between virtual NICs on the machine and between this machine and remote VNET on other machines. The VNET/P core's routing rules are dynamically configurable, through the control interface by the utilities that can be run in user space.

The VNET/P core also provides an expanded interface that the control utilities can use to configure and manage VNET/P. The *VNET/P control* component uses this interface to do so. It in turn acts as a daemon that exposes a TCP control port that uses the same configuration language as VNET/U. Between compatible encapsulation and compatible control, the intent is that VNET/P and VNET/U be interoperable, with VNET/P providing the "fast path".

To exchange packets with a remote machine, the VNET/P core uses a *VNET/P bridge* to communicate with the physical network. The VNET/P bridge runs as a kernel module in the host kernel and uses the host's networking facilities to interact with physical network devices and with the host's networking stack. An additional responsibility of the bridge is to provide encapsulation. For performance reasons, we use UDP encapsulation, in a form compatible with that used in VNET/U. TCP encapsulation is also supported. The bridge selectively performs UDP or TCP encapsulation for packets destined for remote machines, but can also deliver an Ethernet packet without encapsulation. In our performance evaluation, we consider only encapsulated traffic.

The VNET/P core consists of approximately 2500 lines of C in Palacios, while the VNET/P bridge consists of about 2000 lines of C comprising a Linux kernel module. VNET/P is available via the V3VEE project's public git repository, as part of the "devel" branch of the Palacios VMM.

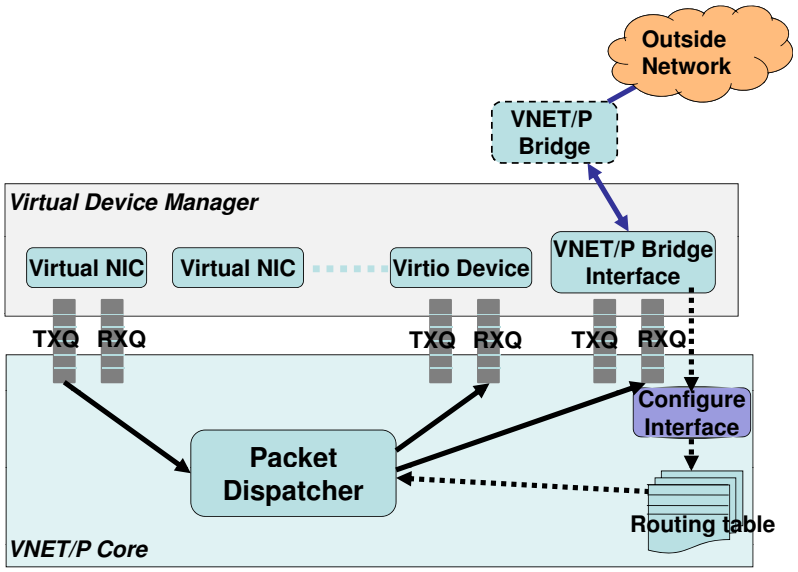


Figure B.2: VNET/P core's internal logic.

B.3.3 VNET/P core

The VNET/P core is primarily responsible for routing and dispatching raw Ethernet packets. It intercepts all Ethernet packets from virtual NICs that are associated with VNET/P, and forwards them either to VMs on the same host machine or to the outside network through the VNET/P bridge. Each packet is routed based on its source and destination MAC addresses. The internal processing logic of the VNET/P core is illustrated in Figure B.2.

Routing: To route Ethernet packets, VNET/P maintains routing tables indexed by source and destination MAC addresses. Although this table structure only provides linear time lookups, a hash table-based routing cache is layered on top of the table, and the common case is for lookups to hit in the cache and thus be serviced in constant time.

A routing table entry maps to a destination, which is either a *link* or an *interface*. A link is an overlay destination—it is the next UDP/IP-level (i.e., IP address and port) destination of the packet, on some other machine. A special link corresponds to the local network. The local network destination is usually used at the "exit/entry point" where the VNET overlay is attached to the user's physical LAN. A packet routed via a link is delivered to another VNET/P core, a VNET/U daemon, or the local network. An interface is a local destination for the packet, corresponding to some virtual NIC.

For an interface destination, the VNET/P core directly delivers the packet to the relevant virtual NIC. For a link destination, it injects the packet into the VNET/P bridge along with the destination link identifier. The VNET/P bridge demultiplexes based on the link and either encapsulates the packet and sends it via the corresponding UDP or TCP socket, or sends it directly as a raw packet to the local network.

Packet processing: Packet forwarding in the VNET/P core is conducted by *packet dispatchers*. A packet dispatcher interacts with each virtual NIC to forward packets in one of two modes: *guest-driven mode* or *VMM-driven mode*. The operation of these modes is illustrated in the top two timelines in Figure B.3.

The purpose of guest-driven mode is to minimize latency for small messages in a parallel application. For example, a barrier operation would be best served with guest-driven mode. In the guest-driven mode, the packet dispatcher is invoked when the guest's interaction with the NIC explicitly causes an exit. For example, the guest might queue a packet on its virtual NIC and then cause an exit to notify the VMM that a packet is ready. In guest-driven mode, a packet dispatcher runs at this point. Similarly, on receive, a packet dispatcher queues the packet to the device and then immediately notifies the device.

The purpose of VMM-driven mode is to maximize throughput for bulk data transfer in a parallel application. Unlike guest-driven mode, VMM-driven mode tries to handle multiple packets per VM exit. It does this by having VMM poll the virtual NIC. The

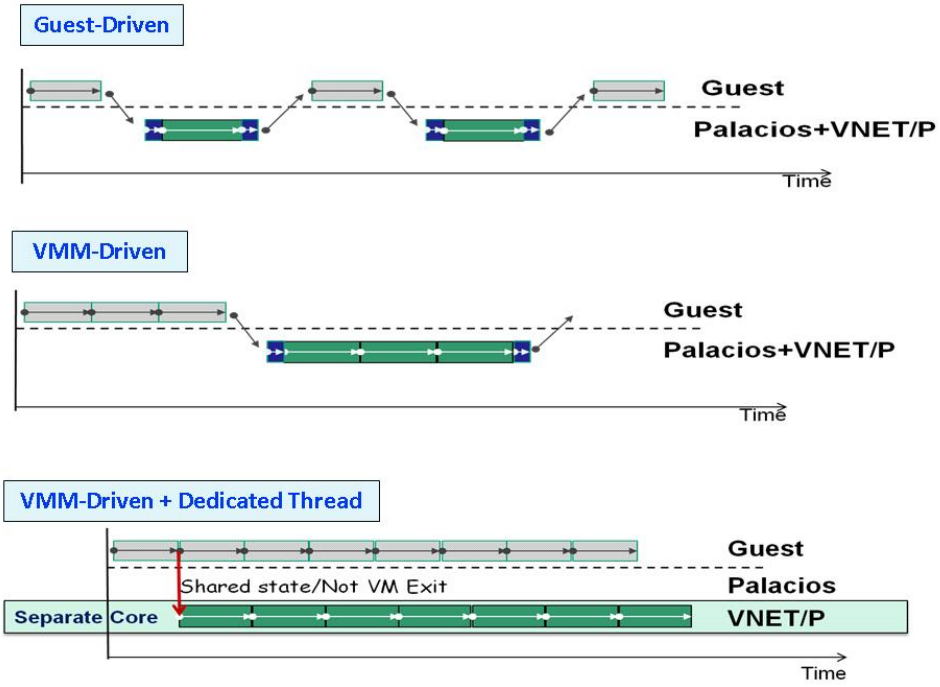


Figure B.3: The VMM-driven and guest-driven modes in the virtual NIC. Guest-driven mode can decrease latency for small messages, while VMM-driven mode can increase throughput for large messages. Combining VMM-driven mode with a dedicated packet dispatcher thread results in most send-related exits caused by the virtual NIC being eliminated, thus further enhancing throughput.

NIC is polled in two ways. First, it is polled, and a packet dispatcher is run, if needed, in the context of the current VM exit (which is unrelated to the NIC). Even if exits are infrequent, the polling and dispatch will still make progress during the handling of timer interrupt exits.

The second manner in which the NIC can be polled is in the context of a packet dispatcher running in a kernel thread inside the VMM context, as shown in Figure B.4. The packet dispatcher thread can be instantiated multiple times, with these threads running on different cores in the machine. If a packet dispatcher thread decides that a virtual NIC

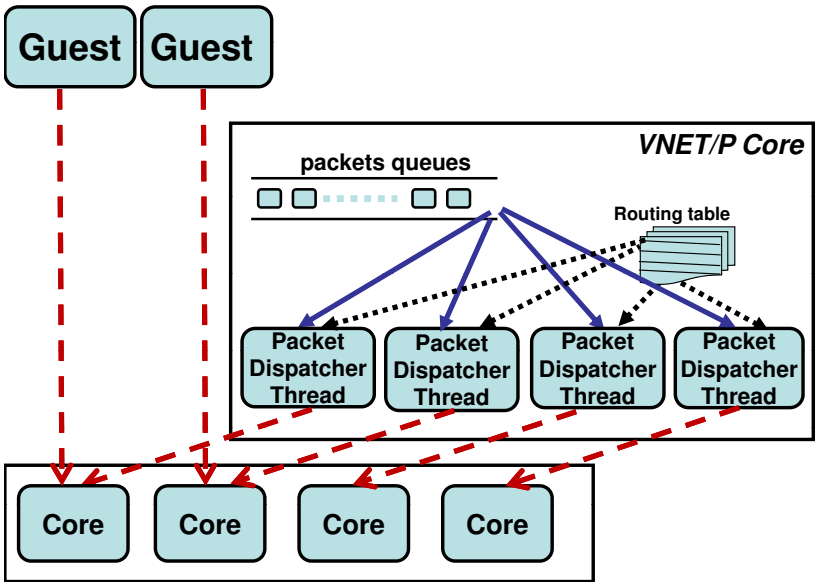


Figure B.4: VNET/P running on a multicore system. The selection of how many, and which cores to use for packet dispatcher threads is made dynamically.

queue is full, it forces the NIC's VM to handle it by doing a cross-core IPI to force the core on which the VM is running to exit. The exit handler then does the needed event injection. Using this approach, it is possible to dynamically employ idle processor cores to increase packet forwarding bandwidth. Combined with VMM-driven mode, VNET/P packet processing can then proceed in parallel with guest packet sends, as shown in the bottommost timeline of Figure B.3.

Influenced by Sidecore [65], an additional optimization we developed was to offload in-VMM VNET/P processing, beyond packet dispatch, to an unused core or cores, thus making it possible for the guest VM to have full use of its cores (minus the exit/entry costs when packets are actually handed to/from it). Figure B.5 is an example of the benefits of doing so for small MTU communication.

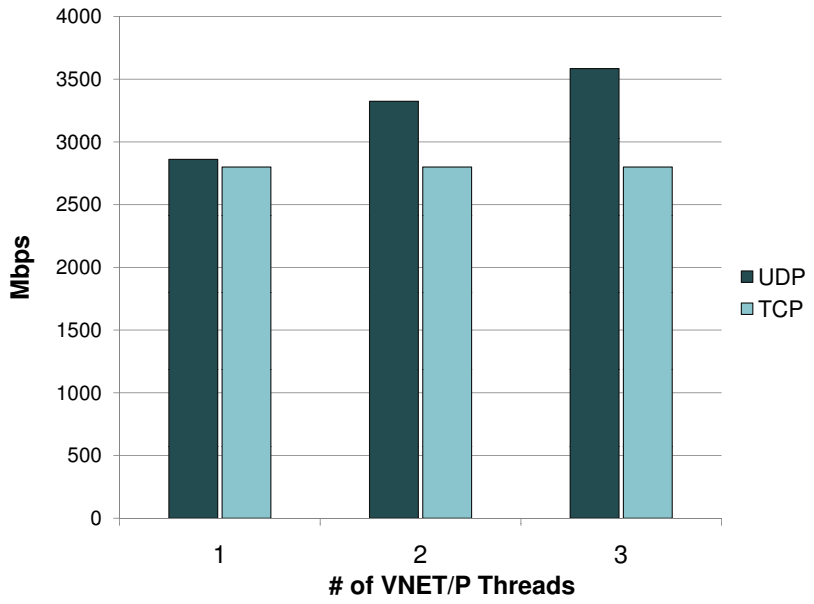


Figure B.5: Early example of scaling of receive throughput by executing the VMM-based components of VNET/P on separate cores, and scaling the number of cores used. The ultimate on-wire MTU here is 1500 bytes.

VNET/P can be configured to statically use either VMM-driven or guest-driven mode, or *adaptive operation* can be selected. In adaptive operation, which is illustrated in Figure B.6, VNET/P switches between these two modes dynamically depending on the arrival rate of packets destined to or from the virtual NIC. For a low rate, it enables guest-driven mode to reduce the single packet latency. On the other hand, with a high arrival rate it switches to VMM-driven mode to increase throughput. Specifically, the VMM detects whether the system is experiencing a high exit rate due to virtual NIC accesses. It recalculates the rate periodically. The algorithm employees simple hysteresis, with the rate bound for switching from guest-driven to VMM-driven mode being larger than the rate bound for switching back. This avoids rapid switching back and forth when the rate falls between

```
num_packets = {number of exits caused by virtual NIC
                  from last period of time window}
rate = num_packets/window_time

if (rate > upper_rate_limit  &&  current-mode == Guest-driven)
      then:
            switch_mode(VMM-Driven)
            current-mode= VMM-Driven
else if (rate < lower_rate_limit &&  current-mode == VMM-driven)
      then:
            switch_mode(Guest-driven)
            current-mode= Guest-driven
else
      do-nothing
endif
```

Figure B.6: Adaptive mode dynamically selects between VMM-driven and guest-driven modes of operation to optimize for both throughput and latency.

these bounds.

For a 1 Gbps network, guest-driven mode is sufficient to allow VNET/P to achieve the full native throughput. On a 10 Gbps network, VMM-driven mode is essential to move packets through the VNET/P core with near-native throughput. Generally, adaptive operation will achieve these limits.

B.3.4 Virtual NICs

VNET/P is designed to be able to support any virtual Ethernet NIC device. A virtual NIC must, however, register itself with VNET/P before it can be used. This is done during the initialization of the virtual NIC at VM configuration time. The registration provides additional callback functions for packet transmission, transmit queue polling, and packet reception. These functions essentially allow the NIC to use VNET/P as its backend, instead

of using an actual hardware device driver backend.

Linux virtio virtual NIC: Virtio [108], which was recently developed for the Linux kernel, provides an efficient abstraction for VMMs. A common set of virtio device drivers are now included as standard in the Linux kernel. To maximize performance, our performance evaluation configured the application VM with Palacios's virtio-compatible virtual NIC, using the default Linux virtio network driver.

MTU: The maximum transmission unit (MTU) of a networking layer is the size of the largest protocol data unit that the layer can pass onwards. A larger MTU improves throughput because each packet carries more user data while protocol headers have a fixed size. A larger MTU also means that fewer packets need to be processed to transfer a given amount of data. Where per-packet processing costs are significant, larger MTUs are distinctly preferable. Because VNET/P adds to the per-packet processing cost, supporting large MTUs is helpful.

VNET/P presents an Ethernet abstraction to the application VM. The most common Ethernet MTU is 1500 bytes. However, 1 Gbit and 10 Gbit Ethernet can also use "jumbo frames", with an MTU of 9000 bytes. Other networking technologies support even larger MTUs. To leverage the large MTUs of underlying physical NICs, VNET/P itself supports MTU sizes of up to 64 KB.[1] The application OS can determine the virtual NIC's MTU and then transmit/receive accordingly. VNET/P advertises the appropriate MTU.

The MTU used by virtual NIC can result in encapsulated VNET/P packets that exceed the MTU of the underlying physical network. In this case, fragmentation has to occur, either in the VNET/P bridge or in the host NIC (via TCP Segmentation Offloading (TSO)). Fragmentation and reassembly is handled by VNET/P and is totally transparent to the application VM. However, performance will suffer when significant fragmentation occurs.

[1]This may be expanded in the future. Currently, it has been sized to support the largest possible IPv4 packet size.

Thus it is important that the application VM's device driver select an MTU carefully, and recognize that the desirable MTU may change over time, for example after a migration to a different host. In Section B.4, we analyze throughput using different MTUs.

B.3.5 VNET/P Bridge

The VNET/P bridge functions as a network bridge to direct packets between the VNET/P core and the physical network through the host NIC. It operates based on the routing decisions made by the VNET/P core which are passed along with the packets to be forwarded. It is implemented as a kernel module running in the host.

When the VNET/P core hands a packet and routing directive up to the bridge, one of two transmission modes will occur, depending on the destination. In a *direct send*, the Ethernet packet is directly sent. This is common for when a packet is exiting a VNET overlay and entering the physical network, as typically happens on the user's network. It may also be useful when all VMs will remain on a common layer 2 network for their lifetime. In an *encapsulated send* the packet is encapsulated in a UDP packet and the UDP packet is sent to the directed destination IP address and port. This is the common case for traversing a VNET overlay link. Similarly, for packet reception, the bridge uses two modes, simultaneously. In a *direct receive* the host NIC is run in promiscuous mode, and packets with destination MAC addresses corresponding to those requested by the VNET/P core are handed over to it. This is used in conjunction with direct send. In an *encapsulated receive* UDP packets bound for the common VNET link port are disassembled and their encapsulated Ethernet packets are delivered to the VNET/P core. This is used in conjunction with encapsulated send. Our performance evaluation focuses solely on encapsulated send and receive.

B.3.6 Control

The VNET/P control component allows for remote and local configuration of links, interfaces, and routing rules so that an overlay can be constructed and changed over time. VNET/U already has user-level tools to support VNET, and, as we described in Section B.2, a range of work already exists on the configuration, monitoring, and control of a VNET overlay. In VNET/P, we reuse these tools as much as possible by having the user-space view of VNET/P conform closely to that of VNET/U. The *VNET/P configuration console* allows for local control to be provided from a file, or remote control via TCP-connected VNET/U clients (such as tools that automatically configure a topology that is appropriate for the given communication pattern among a set of VMs [120]). In both cases, the VNET/P control component is also responsible for validity checking before it transfers the new configuration to the VNET/P core.

B.3.7 Performance-critical data paths and flows

Figure B.7 depicts how the components previously described operate during packet transmission and reception. These are the performance critical data paths and flows within VNET/P, assuming that virtio virtual NICs (Section B.3.4) are used. The boxed regions of the figure indicate steps introduced by virtualization, both within the VMM and within the host OS kernel. There are also additional overheads involved in the VM exit handling for I/O port reads and writes and for interrupt injection.

Transmission: The guest OS in the VM includes the device driver for the virtual NIC. The driver initiates packet transmission by writing to a specific virtual I/O port after it puts the packet into the NIC's shared ring buffer (TXQ). The I/O port write causes an exit that gives control to the virtual NIC I/O handler in Palacios. The handler reads the packet from the buffer and writes it to VNET/P packet dispatcher. The dispatcher does

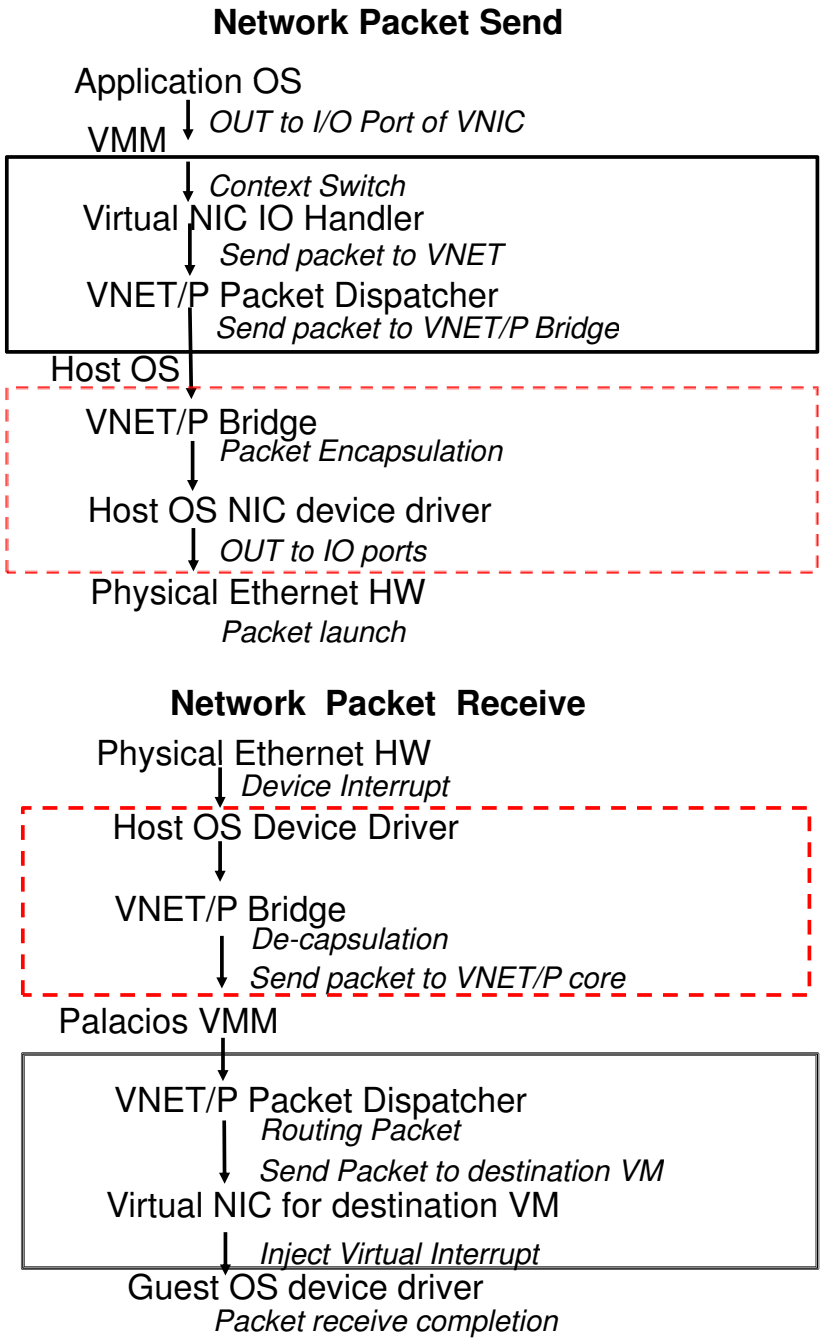


Figure B.7: Performance-critical data paths and flows for packet transmission and reception. Solid boxed steps and components occur within the VMM itself, while dashed boxed steps and components occur in the host OS.

a routing table lookup to determine the packet's destination. For a packet destined for a VM on some other host, the packet dispatcher puts the packet into the receive buffer of the VNET/P bridge and notify it. Meanwhile, VNET/P bridge fetches the packet from the receive buffer, determines its destination VNET/P bridge, encapsulates the packet, and transmits it to the physical network via the host's NIC.

Note that while the packet is handed off multiple times, it is copied only once inside the VMM, from the send buffer (TXQ) of the receive buffer of the VNET/P bridge. Also note that while the above description and the diagram suggest sequentiality, packet dispatch can occur on a separate kernel thread running on a separate core, and the VNET/P bridge itself introduces additional concurrency. From the guest's perspective, the I/O port write that initiated transmission returns essentially within a VM exit/entry time.

Reception: The path for packet reception is essentially symmetric to that of transmission. The host NIC in the host machine receives a packet using its standard driver and delivers it to the VNET/P bridge. The bridge unencapsulates the packet and sends the payload (the raw Ethernet packet) to the VNET/P core. The packet dispatcher in VNET/P core determines its destination VM and puts the packet into the receive buffer (RXQ) of its virtual NIC.

Similar to transmission, there is considerably concurrency in the reception process. In particular, packet dispatch can occur in parallel with the reception of the next packet.

B.3.8 Performance tuning parameters

VNET/P is configured with the following parameters:

- Whether guest-driven, VMM-driven, or adaptive mode is used.
- For adaptive operation, the upper and lower rate bounds (α_l, α_u) for switching between guest-driven, and VMM-driven modes, as well as the window size ω over

which rates are computed.

- The number of packet dispatcher threads $n_{dispatchers}$ that are instantiated.
- The yield model and parameters for the bridge thread, the packet dispatch threads, and the VMM's halt handler.

The last of these items requires some explanation, as it presents an important tradeoff between VNET/P latency and CPU consumption. In both the case of packets arriving from the network and packets arriving from the guest's virtual NIC, there is, conceptually, a thread running a receive operation that could block or poll. First, consider packet reception from the physical network. Suppose a bridge thread is waiting for UDP packet to arrive. If no packets have arrived recently, then blocking would allow the core on which the thread is running to be yielded to another thread, or for the core to be halted. This would be the ideal for minimizing CPU consumption, but on the next arrival, there will be a delay, even if the core is idle, in handling the packet since the thread will need to be scheduled. There is a similar tradeoff in the packet dispatcher when it comes to packet transmission from the guest.

A related tradeoff, for packet reception in the guest, exists in the VMM's model for handling the halt state. If the guest is idle, it will issue an HLT or related instruction. The VMM's handler for this case consists of waiting until an interrupt needs to be injected into the guest. If it polls, it will be able to respond as quickly as possible, thus minimizing the packet reception latency, while if it blocks, it will consume less CPU.

For all three of these cases VNET/P's and the VMM's *yield strategy* comes into play. Conceptually, these cases are written as polling loops, which in their loop bodies can yield the core to another thread, and optionally put the thread to sleep pending a wake-up after a signaled event or the passage of an interval of time. Palacios currently has selectable

yield strategy that is used in these loops, and the strategy has three different options, one of which is chosen when the VM is configured:

- Immediate yield. Here, if there is no work, we immediately yield the core to any competing threads. However, if there are no other active threads that the core can run, the yield immediately returns. A poll loop using the immediate yield has the lowest latency while still being fair to competing threads.

- Timed yield: Here, if there is no work, we put the thread on a wake up queue and yield CPU. Failing any other event, the thread will be awakened by the passage of time T_{sleep}. A poll loop using the timed yield strategy minimizes CPU usage, but at the cost of increased latency.

- Adaptive yield: Here, the poll loop reports how much time has passed since it last did any work. Until that time exceeds a threshold T_{nowork}, the immediate yield strategy is used, and afterwards the timed yield strategy is used. By setting the threshold, different tradeoffs between latency and CPU usage are made.

In our performance evaluations, we use the following parameters to focus on the performance limits of VNET/P:

Parameter	Value
mode	adaptive
α_l	10^3 packets/second
α_u	10^4 packets/second
ω	5ms
$n_{dispatchers}$	1
yield strategy	immediate yield
T_{sleep}	not used
T_{nowork}	not used

B.4 Performance evaluation

The purpose of our performance evaluation is to determine how close VNET/P comes to native throughput and latency in the most demanding (lowest latency, highest throughput) hardware environments. We consider communication between two machines whose NICs are directly connected in most of our detailed benchmarks.

In the virtualized configuration the guests and performance testing tools run on top of Palacios with VNET/P carrying all traffic between them using encapsulation. In the native configuration, the same guest environments run directly on the hardware.

Our evaluation of communication performance in this environment occurs at three levels. First, we benchmark the TCP and UDP bandwidth and latency. Second, we benchmark MPI using a widely used benchmark. Finally, we evaluated the performance of the HPCC and NAS application benchmarks in a cluster to see VNET/P's impact on the performance and scalability of parallel applications.

B.4.1 Testbed and configurations

Most of our microbenchmark tests are focused on the end-to-end performance of VNET/P. Therefore our testbed consists of two physical machines, which we call host machines. Each machine has a quadcore 2.4 GHz X3430 Intel Xeon processor, 8 GB RAM, a Broadcom NetXtreme II 1 Gbps Ethernet NIC (1000BASE-T), and a NetEffect NE020 10 Gbps Ethernet fiber optic NIC (10GBASE-SR) in a PCI-e slot. The Ethernet NICs of these machines are directly connected with twisted pair and fiber patch cables.

A range of our measurements are made using the cycle counter. We disabled DVFS control on the machine's BIOS, and in the host Linux kernel. We also sanity-checked the measurement of larger spans of time by comparing cycle counter-based timing with a separate wall clock.

All microbenchmarks included in the performance section are run in the testbed described above. The HPCC and NAS application benchmarks are run on a 6-node test cluster described in Section B.4.4.

We considered the following two software configurations:

- *Native:* In the native configuration, neither Palacios nor VNET/P is used. A minimal BusyBox-based Linux environment based on an unmodified 2.6.30 kernel runs directly on the host machines. We refer to the 1 and 10 Gbps results in this configuration as *Native-1G* and *Native-10G*, respectively.

- *VNET/P:* The VNET/P configuration corresponds to the architectural diagram given in Figure B.1, with a single guest VM running on Palacios. The guest VM is configured with one virtio network device, 2 cores, and 1 GB of RAM. The guest VM runs a minimal BusyBox-based Linux environment, based on the 2.6.30 kernel. The kernel used in the VM is identical to that in the Native configuration, with the exception that the virtio NIC drivers are loaded. The virtio MTU is configured as 9000 Bytes. We refer to the 1 and 10 Gbps results in this configuration as *VNET/P-1G* and *VNET/P-10G*, respectively.

To assure accurate time measurements both natively and in the virtualized case, our guest is configured to use the CPU's cycle counter, and Palacios is configured to allow the guest direct access to the underlying hardware cycle counter. Our 1 Gbps NIC only supports MTUs up to 1500 bytes, while our 10 Gbps NIC can support MTUs of up to 9000 bytes. We use these maximum sizes unless otherwise specified.

B.4.2 TCP and UDP microbenchmarks

Latency and throughput are the fundamental measurements we use to evaluate the VNET/P system performance. First, we consider these at the IP level, measuring the round-trip

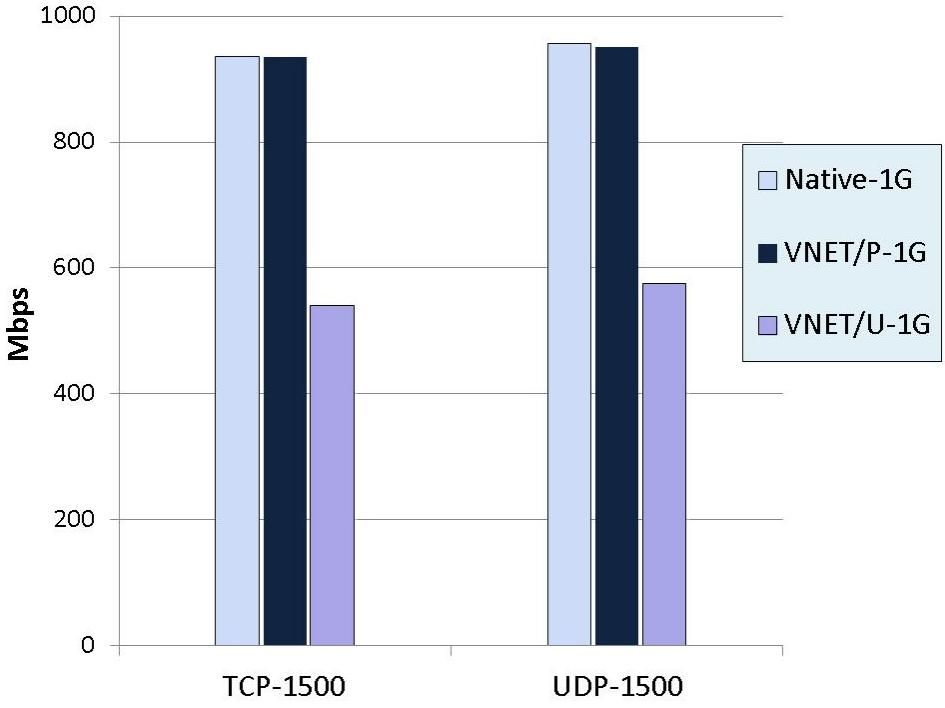
latency, the UDP goodput, and the TCP throughput between two nodes. We measure round-trip latency using *ping* by sending ICMP packets of different sizes. UDP and TCP throughput are measured using *ttcp-1.10*.

UDP and TCP with a standard MTU: Figure B.8 shows the TCP throughput and UDP goodput achieved in each of our configurations on each NIC. For the 1 Gbps network, host MTU is set to 1500 bytes, and for the 10 Gbps network, host MTUs of 1500 bytes and 9000 bytes are both tested. For 1 Gbps, we also compare with VNET/U running on the same hardware with Palacios. Compared to previously reported results (21.5 MB/s, 1 ms), the combination of the faster hardware we use here, and Palacios, leads to VNET/U increasing its bandwidth by 330%, to 71 MB/s, with a 12% reduction in latency, to 0.88 ms. We also tested VNET/U with VMware, finding that bandwidth increased by 63% to 35 MB/s, with no change in latency. The difference in performance of VNET/U on the two VMMs is due to a custom tap interface in Palacios, while on VMware, the standard host-only tap is used. Even with this optimization, VNET/U cannot saturate a 1 Gbps link.

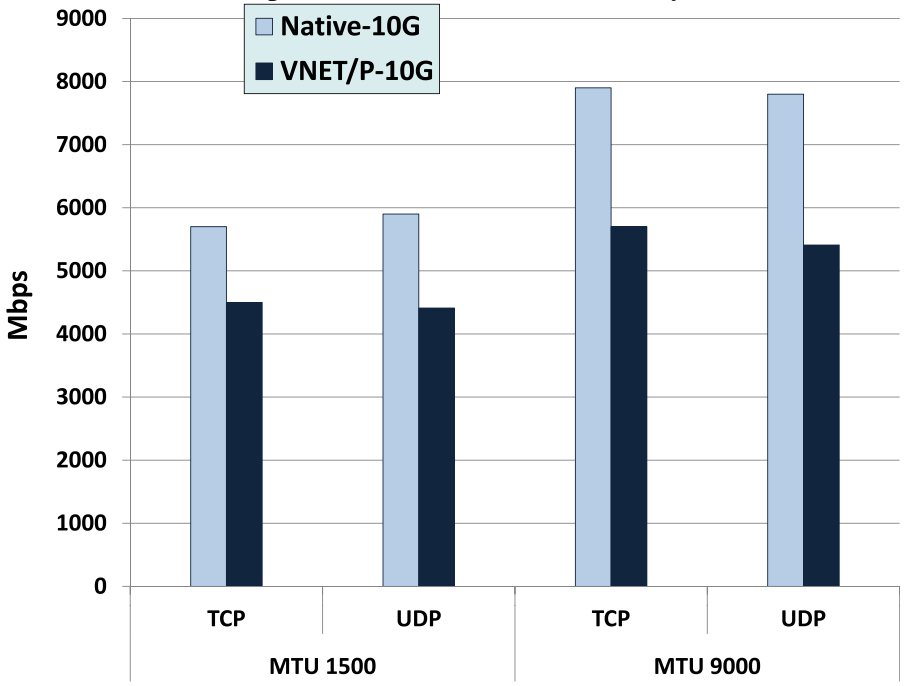
We begin by considering UDP goodput when a standard host MTU size is used. For UDP measurements, ttcp was configured to use 64000 byte writes sent as fast as possible over 60 seconds. For the 1 Gbps network, VNET/P easily matches the native goodput. For the 10 Gbps network, VNET/P achieves 74% of the native UDP goodput.

For TCP throughput, ttcp was configured to use a 256 KB socket buffer, and to communicate 40 MB writes were made. Similar to the UDP results, VNET/P has no difficulty achieving native throughput on the 1 Gbps network. On the 10 Gbps network, using a standard Ethernet MTU, it achieves 78% of the native throughput. The UDP goodput and TCP throughput that VNET/P is capable of, using a standard Ethernet MTU, are approximately 8 times those we would expect from VNET/U given the 1 Gbps results.

UDP and TCP with a large MTU: We now consider TCP and UDP performance with 9000 byte jumbo frames our 10 Gbps NICs support. We adjusted the VNET/P MTU so



(a) 1 Gbps network (host MTU=1500 Bytes)



(b) 10 Gbps network (host MTU=1500, 9000 Bytes)

Figure B.8: End-to-end TCP throughput and UDP goodput of VNET/P on 1 and 10 Gbps network. VNET/P performs identically to the native case for the 1 Gbps network and achieves 74–78% of native throughput for the 10 Gbps network.

that the ultimate encapsulated packets will fit into these frames without fragmentation. For TCP we configure ttcp to use writes of corresponding size, maximize the socket buffer size, and do 4 million writes. For UDP, we configure ttcp to use commensurately large packets sent as fast as possible for 60 seconds. The results are also shown in the Figure B.8. We can see that performance increases across the board compared to the 1500 byte MTU results. Compared to the VNET/U performance we would expect in this configuration, the UDP goodput and TCP throughput of VNET/P are over 10 times higher.

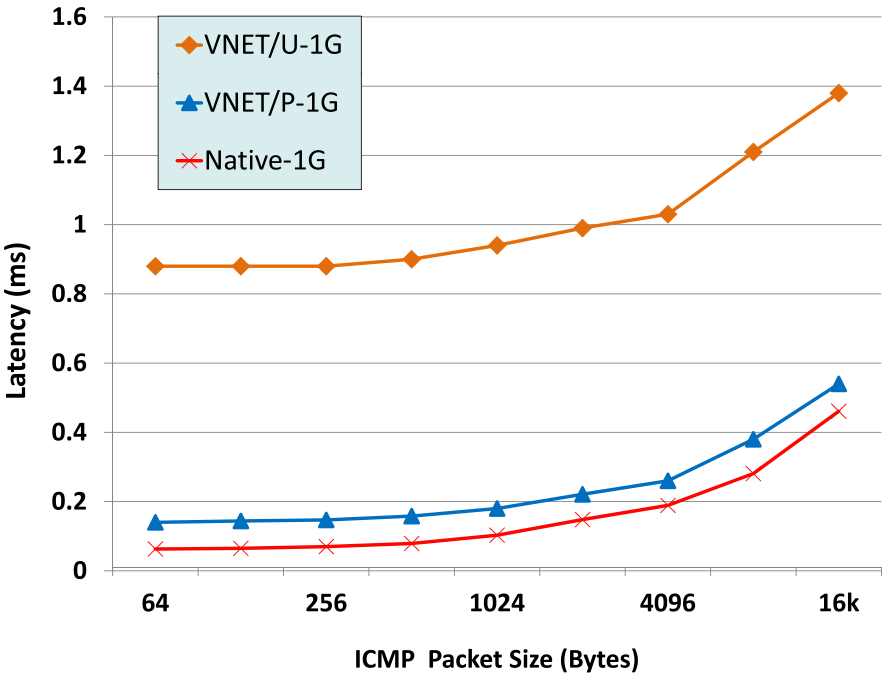
Latency: Figure B.9 shows the round-trip latency for different packet sizes, as measured by ping. The latencies are the average of 100 measurements. While the increase in latency of VNET/P over Native is significant in relative terms (2x for 1 Gbps, 3x for 10 Gbps), it is important to keep in mind the absolute performance. On a 10 Gbps network, VNET/P achieves a 130 μs round-trip, end-to-end latency. The latency of VNET/P is almost seven times lower than that of VNET/U.

B.4.3 MPI microbenchmarks

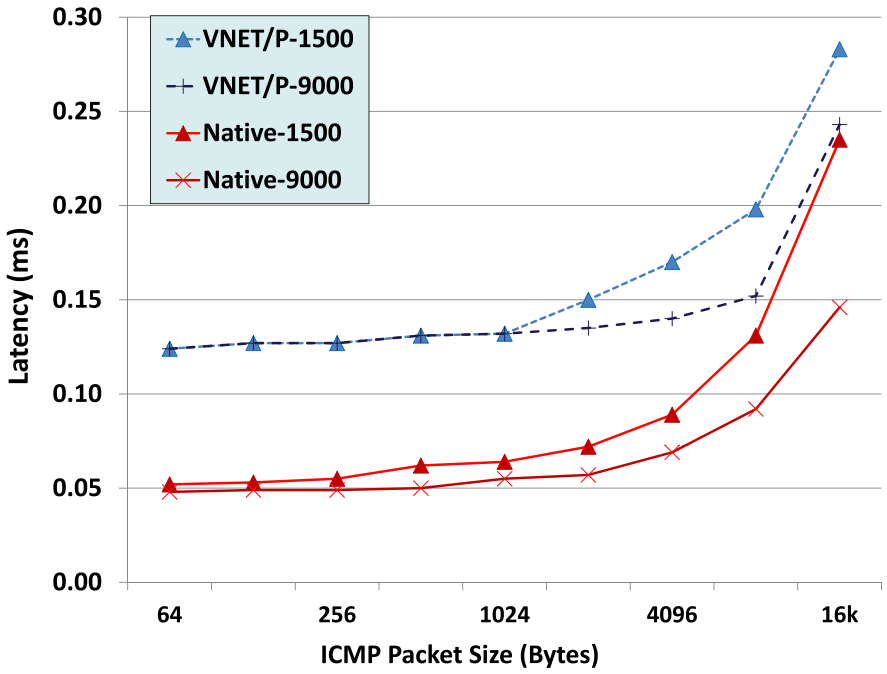
Parallel programs for distributed memory computers are typically written to the MPI interface standard. We used the OpenMPI 1.3 [35] implementation in our evaluations. We measured the performance of MPI over VNET/P by employing the widely-used Intel MPI Benchmark Suite (IMB 3.2.2) [54], focusing on the point-to-point messaging performance. We compared the basic MPI latency and bandwidth achieved by VNET/P and natively.

Figures B.10 and B.11(a) illustrate the latency and bandwidth reported by Intel MPI PingPong benchmark for our 10 Gbps configuration. Here the latency measured is the one-way, end-to-end, application-level latency. That is, it is the time from when an MPI send starts on one machine to when its matching MPI receive call completes on the other machine. For both Native and VNET/P, the host MTU is set to 9000 bytes.

VNET/P's small message MPI latency is about 55 μs, about 2.5 times worse than the



(a) 1 Gbps network (Host MTU=1500 Bytes)



(b) 10 Gbps network (Host MTU=1500, 9000 Bytes)

Figure B.9: End-to-end round-trip latency of VNET/P as a function of ICMP packet size. Small packet latencies on a 10 Gbps network in VNET/P are $\sim$130 μs.

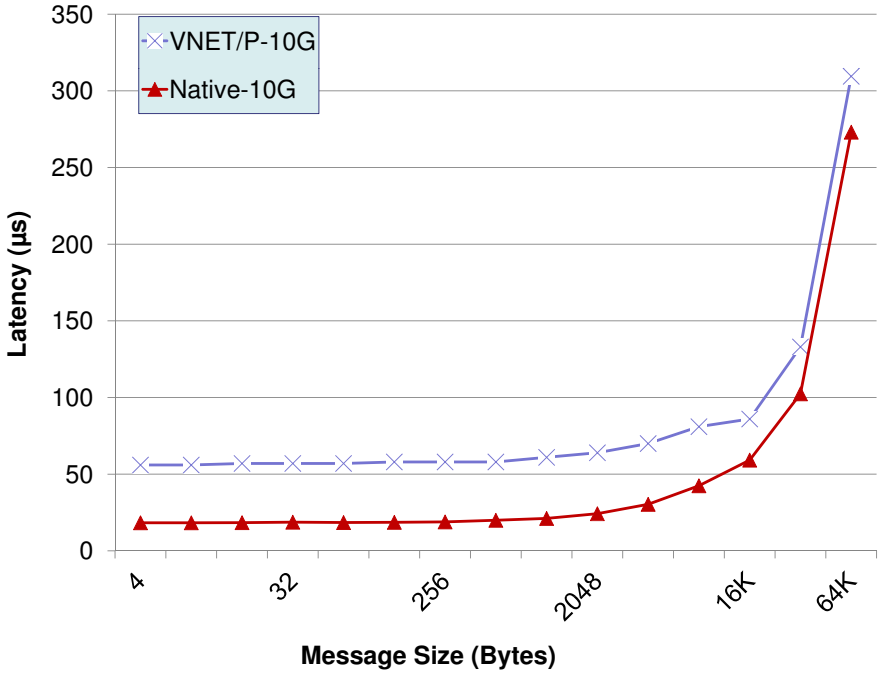
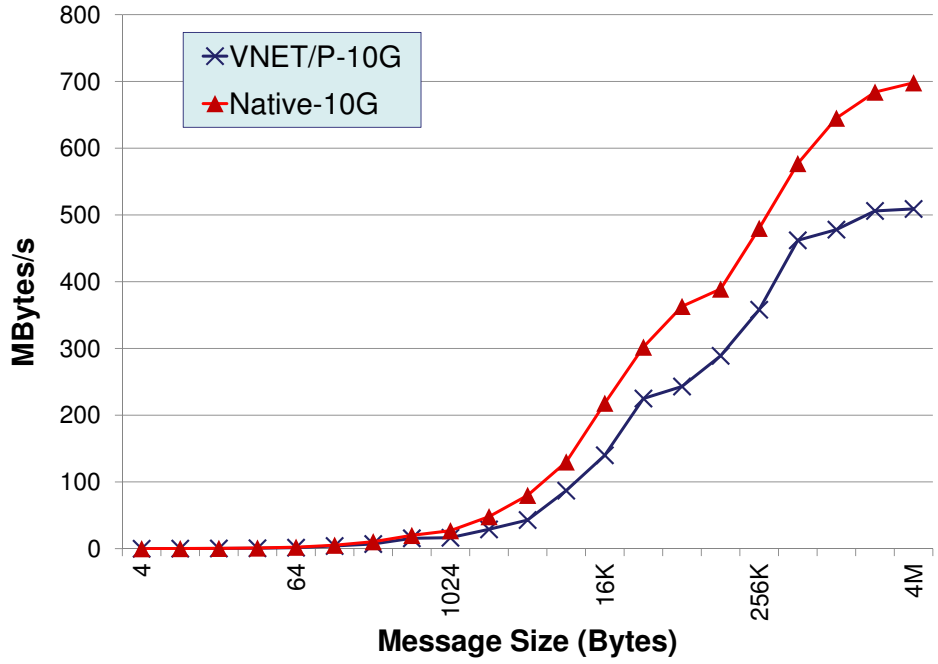


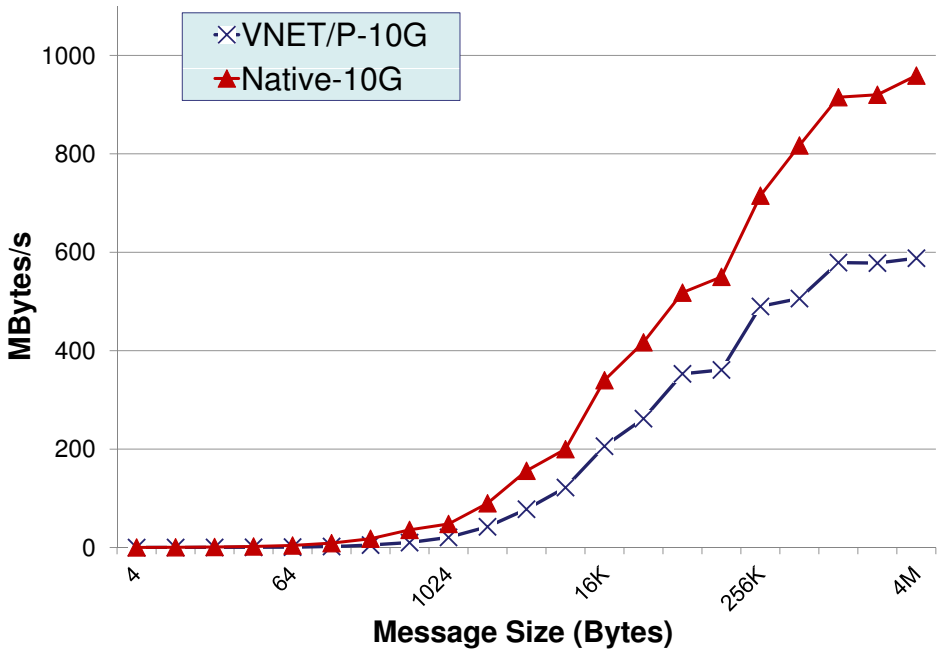
Figure B.10: One-way latency on 10 Gbps hardware from Intel MPI PingPong microbenchmark

native case. However, as the message size increases, the latency difference decreases. The measurements of end-to-end bandwidth as a function of message size show that native MPI bandwidth is slightly lower than raw UDP or TCP throughput, and VNET/P performance tracks it similarly. The bottom line is that the current VNET/P implementation can deliver an MPI latency of 55 μs and bandwidth of 510 MB/s on 10 Gbps Ethernet hardware.

Figure B.11(b) shows the results of the MPI SendRecv microbenchmark in which each node simultaneously sends and receives. There is no reduction in performance between the bidirectional case and the unidirectional case. For both figures, the absolute numbers are shown, but it is important to note that the overhead is quite stable in percentage terms once the message size is large: beyond 256K, the one-way bandwidth of VNET/P is around 74% of native, and the two-way bandwidth is around 62% of native. A possible interpretation



(a) One-way bandwidth



(b) SendRecv Bandwidth

Figure B.11: Intel MPI PingPong microbenchmark showing (a) one-way bandwidth and (b) bidirectional bandwidth as a function of message size on the 10 Gbps hardware.

is that we become memory copy bandwidth limited.

B.4.4 HPCC benchmarks on more nodes

To test VNET/P performance on more nodes, we ran the HPCC benchmark [53] suite on a 6 node cluster with 1 Gbps and 10 Gbps Ethernet. Each node was equipped with two quad-core 2.3 GHz 2376 AMD Opterons, 32 GB of RAM, an nVidia MCP55 Forthdeth 1 Gbps Ethernet NIC and a NetEffect NE020 10 Gbps Ethernet NIC. The nodes were connected via a Fujitsu XG2000 10Gb Ethernet Switch. Similar to our A range of our measurements are made using the cycle counter. We disabled DVFS control on the machine's BIOS, and in the host Linux kernel. We also used the cycle counter on these machines, and we applied the same techniques (deactivated DVFS, sanity-checking against an external clock for larger time spans) described in Section B.4.1 to ensure accurate timing.

The VMs were all configured exactly as in previous tests, with 4 virtual cores, 1 GB RAM, and a virtio NIC. For the VNET/P test case, each host ran one VM. We executed tests with 2, 3, 4, 5, and 6 VMs, with 4 HPCC processes per VM (one per virtual core). Thus, our performance results are based on HPCC with 8, 12, 16, 20 and 24 processes for both VNET/P and Native tests. In the native cases, no VMs were used, and the processes ran directly on the host. For 1 Gbps testing, the host MTU was set to 1500, while for the 10 Gbps cases, the host MTU was set to 9000.

Latency-bandwidth benchmark: This benchmark consists of the ping-pong test and the ring-based tests. The ping-pong test measures the latency and bandwidth between all distinct pairs of processes. The ring-based tests arrange the processes into a ring topology and then engage in collective communication among neighbors in the ring, measuring bandwidth and latency. The ring-based tests model the communication behavior of multi-dimensional domain-decomposition applications. Both naturally ordered rings and randomly ordered rings are evaluated. Communication is done with MPI non-blocking

sends and receives, and MPI SendRecv. Here, the bandwidth per process is defined as total message volume divided by the number of processes and the maximum time needed in all processes. We reported the ring-based bandwidths by multiplying them with the number of processes in the test.

Figure B.12 and B.13 show the results for different numbers of test processes. The ping-pong latency and bandwidth results are consistent with what we saw in the previous microbenchmarks: in the 1 Gbps network, bandwidth are nearly identical to those in the native cases while latencies are 1.2–2 times higher. In the 10 Gbps network, bandwidths are within 60-75% of native while latencies are about 2 to 3 times higher. Both latency and bandwidth under VNET/P exhibit the same good scaling behavior of the native case.

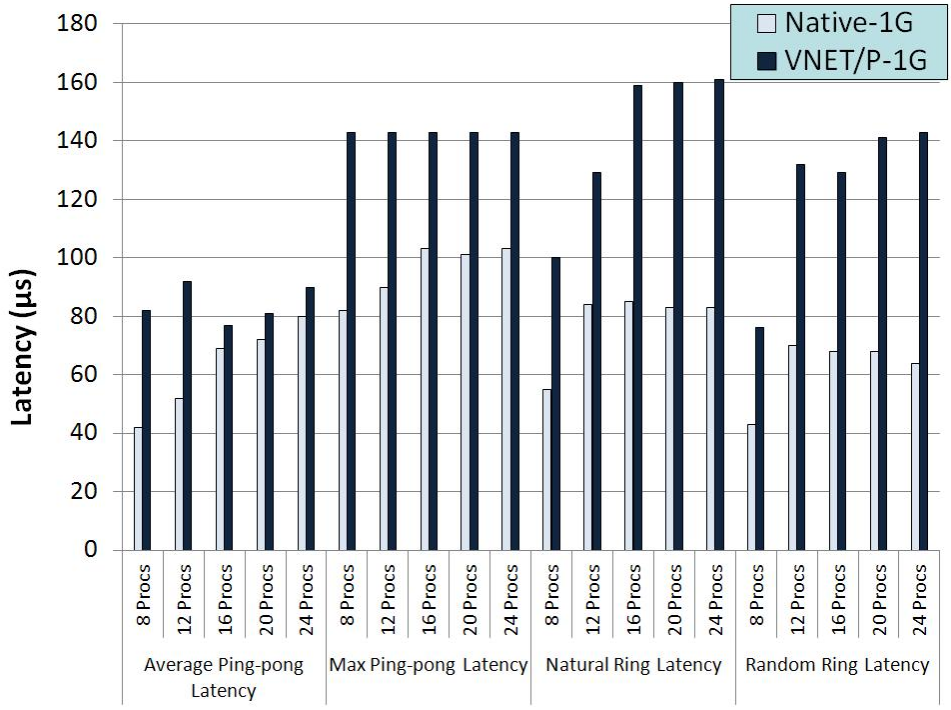
B.4.5 Application benchmarks

We evaluated the effect of a VNET/P overlay on application performance by running two HPCC application benchmarks and the whole NAS benchmark suite on the cluster described in Section B.4.4. Overall, the performance results from the HPCC and NAS benchmarks suggest that VNET/P can achieve high performance for many parallel applications.

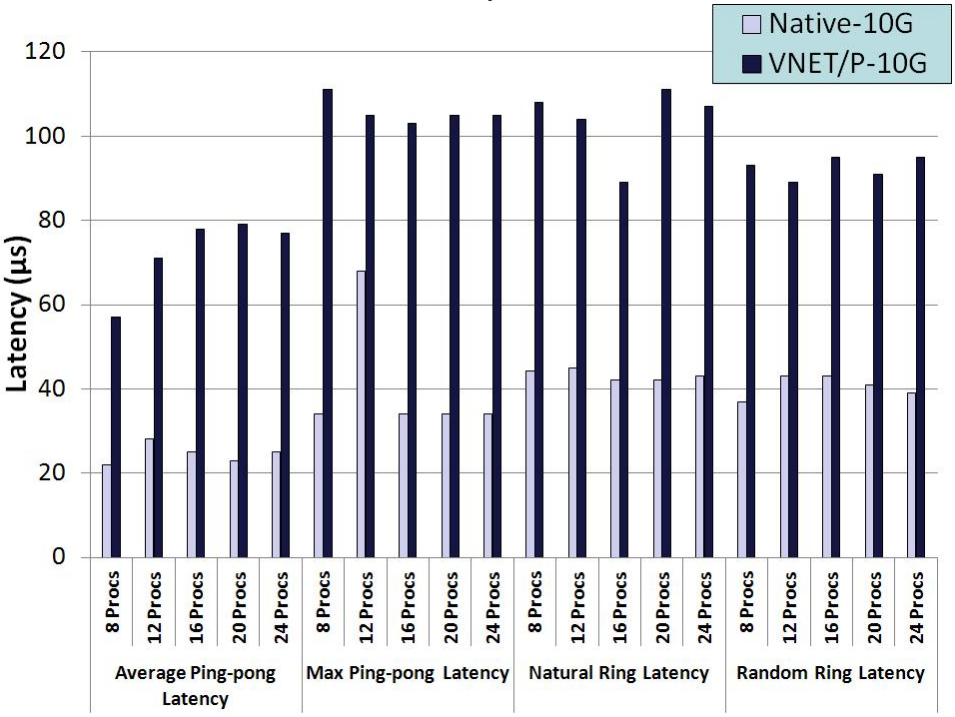
HPCC application benchmarks: We considered the two application benchmarks from the HPCC suite that exhibit the large volume and complexity of communication: MPIRandomAcceess and MPIFFT. For 1 Gbps networks, the difference in performance is negligible so we focus here on 10 Gbps networks.

In MPIRandomAccess, random numbers are generated and written to a distributed table, with local buffering. Performance is measured by the billions of updates per second (GUPs) that are performed. Figure B.14(a) shows the results of MPIRandomAccess, comparing the VNET/P and Native cases. VNET/P achieves 65-70% application performance compared to the native cases, and performance scales similarly.

MPIFFT implements a double precision complex one-dimensional Discrete Fourier

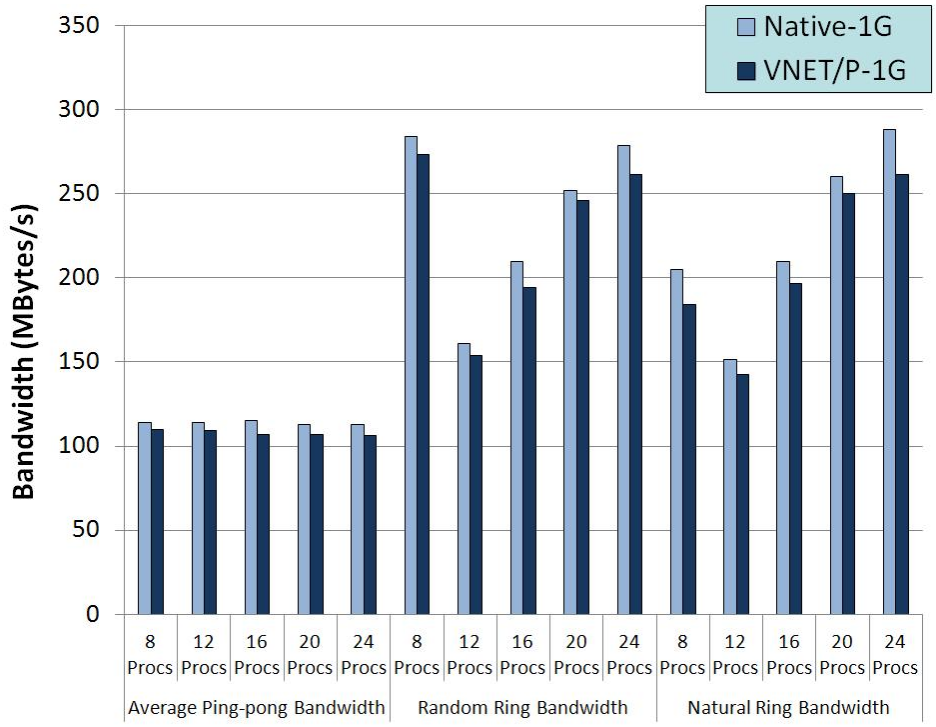


(a) Latency on 1G

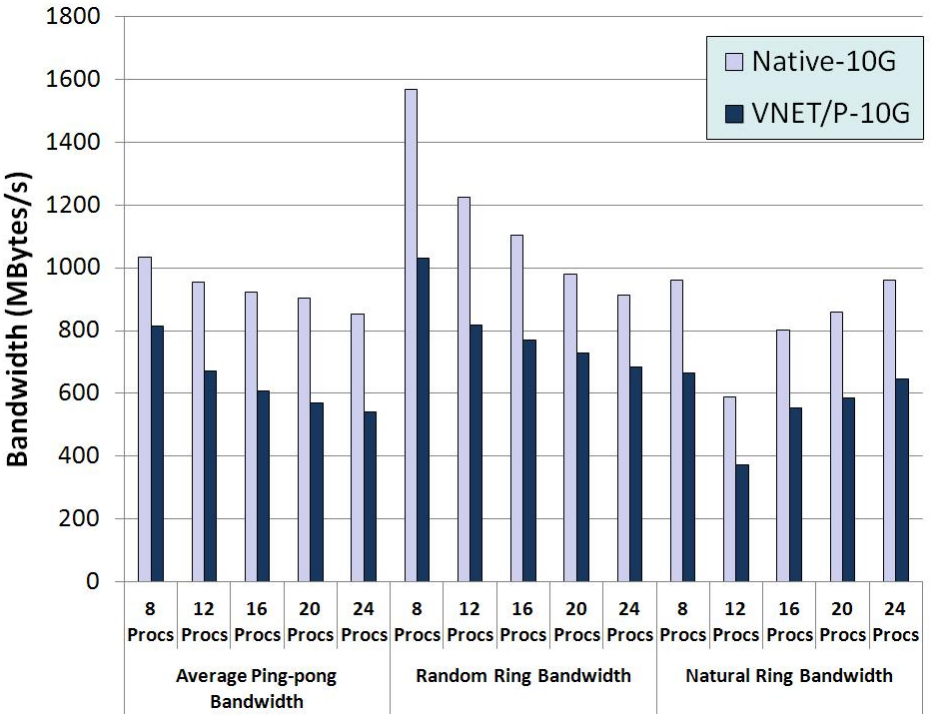


(b) Latency on 10G

Figure B.12: Latencies from HPCC Latency-bandwidth benchmark for both 1 Gbps and 10 Gbps. The ping-pong latency tests show results that are consistent with the previous microbenchmarks, while the ring-based tests show that latency of VNET/P scale similarly to the native cases.

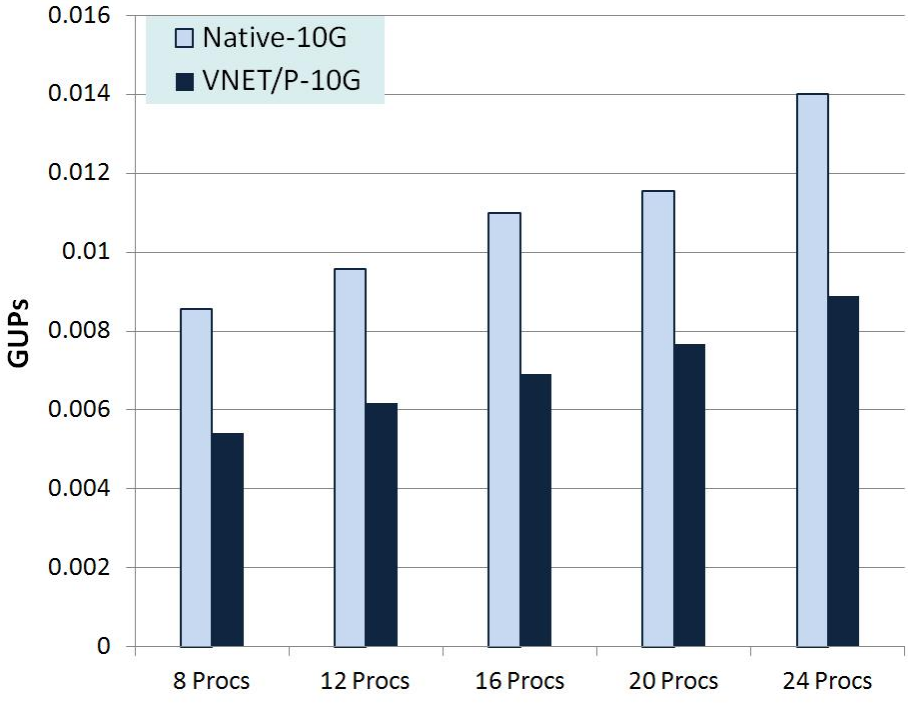


(a) Bandwidth on 1G

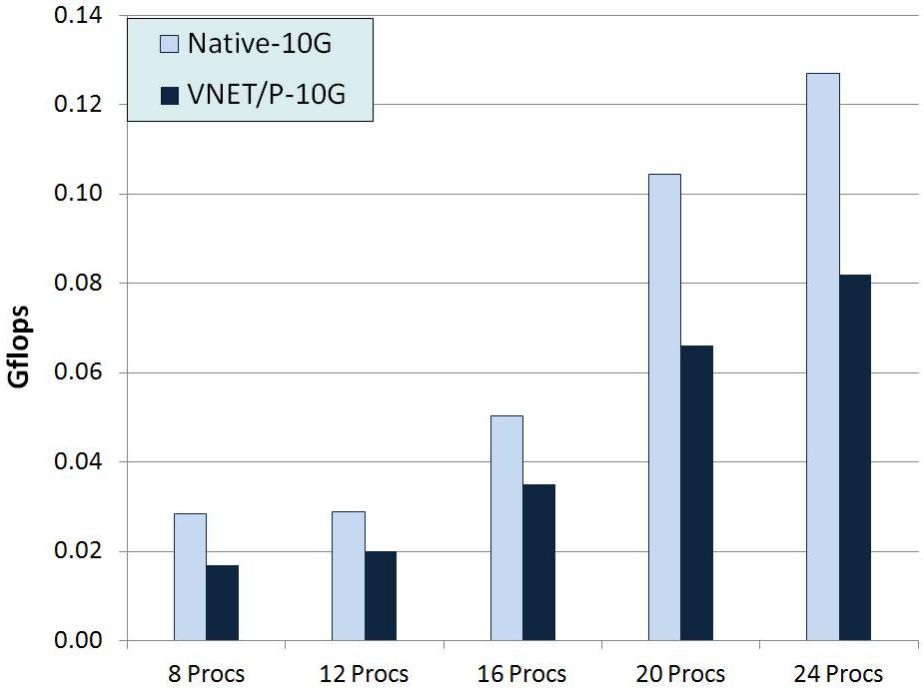


(b) Bandwidth on 10G

Figure B.13: Bandwidths from HPCC Latency-bandwidth benchmark for both 1 Gbps and 10 Gbps. Ring-based bandwidths are multiplied by the total number of processes in the test. The ping-pong bandwidth tests show results that are consistent with the previous microbenchmarks, while the ring-based tests show that bandwidth of VNET/P scale similarly to the native cases.



(a) MPIRandomAccess



(b)MPIFFT

Figure B.14: HPCC application benchmark results. VNET/P achieves reasonable and scalable application performance when supporting communication-intensive parallel application workloads on 10 Gbps networks. On 1 Gbps networks, the difference is negligible.

Transform (DFT). Figure B.14(b) shows the results of MPIFFT, comparing the VNET/P and Native cases. It shows that VNET/P's application performance is within 60-70% of native performance, with performance scaling similarly.

NAS parallel benchmarks: The NAS Parallel Benchmark (NPB) suite [128] is a set of five kernels and three pseudo-applications that is widely used in parallel performance evaluation. We specifically use NPB-MPI 2.4 in our evaluation. In our description, we name executions with the format "name.class.procs". For example, *bt.B.16* means to run the BT benchmark on 16 processes with a class B problem size.

We run each benchmark with at least two different scales and one problem size, except FT, which is only run with 16 processes. One VM is run on each physical machine, and it is configured as described in Section B.4.4. The test cases with 8 processes are running within 2 VMs and 4 processes started in each VM. The test cases with 9 processes are run with 4 VMs and 2 or 3 processes per VM. Test cases with 16 processes have 4 VMs with 4 processes per VM. We report each benchmark's *Mop/s total* result for both native and with VNET/P.

Figure B.15 shows the NPB performance results, comparing the VNET/P and Native cases on both 1 Gbps and 10 Gbps networks. The upshot of the results is that for most of the NAS benchmarks, VNET/P is able to achieve in excess of 95% of the native performance even on 10 Gbps networks. We now describe the results for each benchmark.

EP is an "embarrassingly parallel" kernel that estimates the upper achievable limits for floating point performance. It does not require a significant interprocessor communication. VNET/P achieves native performance in all cases.

MG is a simplified multigrid kernel that requires highly structured long distance communication and tests both short and long distance data communication. With 16 processes, MG achieves native performance on the 1 Gbps network, and 81% of native performance on the 10 Gbps network.

	1Gbps Network				10Gbps Network		
Mop/s	Native	VNET/P	$\frac{VNET/P}{Native}$ (%)		Native	VNET/P	$\frac{VNET/P}{Native}$ (%)
ep.B.8	103.15	101.94	98.8%		102.18	102.12	99.9%
ep.B.16	204.88	203.9	99.5%		208	206.52	99.3%
ep.C.8	103.12	102.1	99.0%		103.13	102.14	99.0%
ep.C.16	206.24	204.14	99.0%		206.22	203.98	98.9%
mg.B.8	4400.52	3840.47	87.3%		5110.29	3796.03	74.3%
mg.B.16	1506.77	1498.65	99.5%		9137.26	7405	81.0%
cg.B.8	1542.79	1319.43	85.5%		2096.64	1806.57	86.2%
cg.B.16	160.64	159.69	99.4%		592.08	554.91	93.7%
ft.B.16	1575.83	1290.78	81.9%		1432.3	1228.39	85.8%
is.B.8	78.88	74.61	94.6%		59.15	59.04	99.8%
is.B.16	35.99	35.78	99.4%		23.09	23	99.6%
is.C.8	89.54	82.15	91.7%		132.08	131.87	99.8%
is.C.16	84.76	82.22	97.0%		77.77	76.94	98.9%
lu.B.8	6818.52	5495.23	80.6%		7173.65	6021.78	83.9%
lu.B.16	7847.99	6694.12	85.3%		12981.86	9643.21	74.3%
sp.B.9	1361.38	1215.85	89.3%		2634.53	2421.98	91.9%
sp.B.16	1489.32	1399.6	94.0%		3010.71	2916.81	96.9%
bt.B.9	3423.52	3297.04	96.3%		5229.01	4076.52	78.0%
bt.B.16	4599.38	4348.99	94.6%		6315.11	6105.11	96.7%

Figure B.15: NAS Parallel Benchmark performance with VNET/P on 1 Gbps and 10 Gbps networks. VNET/P can achieve native performance on many applications, while it can get reasonable and scalable performance when supporting highly communication-intensive parallel application workloads.

CG implements the conjugate gradient method to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. It is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication. With 16 processes, CG achieves native performance on the 1 Gbps network and 94% of native performance on the 10 Gbps network.

FT implements the solution of partial differential equations using FFTs, and captures the essence of many spectral codes. It is a rigorous test of long-distance communication performance. With 16 nodes, it achieves 82% of native performance on 1 Gbps and 86% of native performance on 10 Gbps.

IS implements a large integer sort of the kind that is important in particle method codes and tests both integer computation speed and communication performance. Here VNET/P achieves native performance in all cases.

LU solves a regular-sparse, block (5×5) lower and upper triangular system, a problem associated with implicit computational fluid dynamics algorithms. VNET/P achieves 75%-85% of native performance on this benchmark, and there is no significant difference between the 1 Gbps and 10 Gbps network.

SP and BT implement solutions of multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations, also common in computational fluid dynamics. The salient difference between the two is the communication to computation ratio. For SP with 16 processes, VNET/P achieves 94% of native performance on 1 Gbps and around 97% of native on 10 Gbps. For BT at the same scale, 95% of native at 1 Gbps and 97% of native at 10 Gbps are achieved.

It is worth mentioning that in microbenchmarking we measured throughput and latency separately. During the throughput microbenchmarking, large packets were continuously sent, saturating VNET/P. On the other hand, for the application benchmarks, the network communication consisted of a mixture of small and large packets, and so their performance was determined both by throughput and latency. Recall that the latency overhead of VNET/P is on the order of 2–3x. This may explain why some application benchmarks cannot achieve native performance even in the 1 Gbps case (e.g. LU, FT) despite VNET/P achieving native throughput in the microbenchmarks. This also suggests that in order to gain fully native application performance, we need to further reduce the small packet la-

tency. In a separate paper from our group [21], we have described additional techniques to do so, resulting in microbenchmark latency overheads of 1.2–1.3x. This results in the latency-limited application benchmarks achieving $> 95\%$ of native performance.

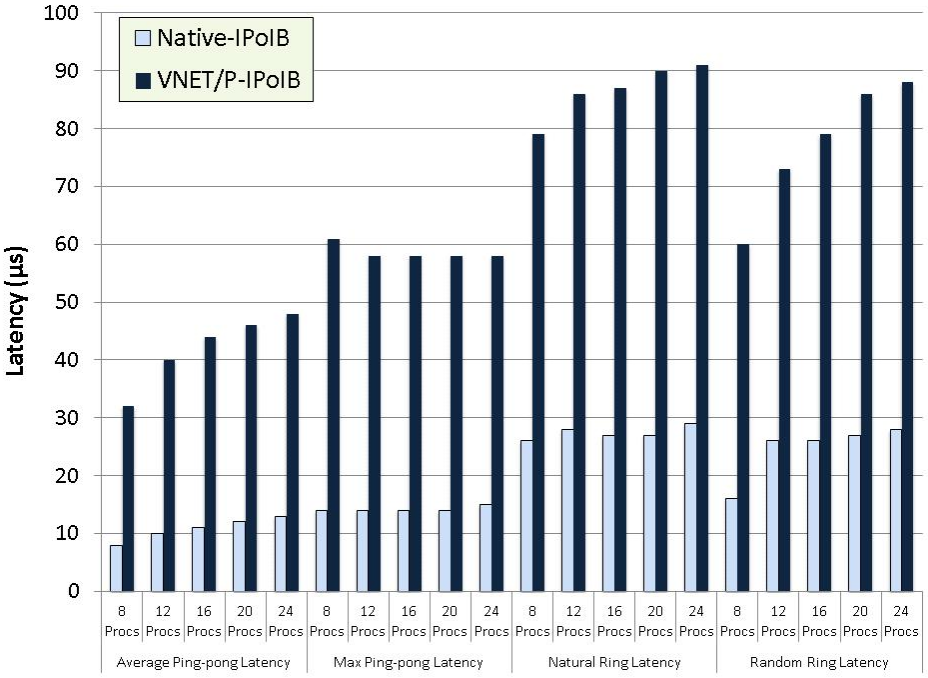
B.5 VNET/P portability

We now describe our experiences in running the Linux-based VNET/P on alternative hardware, and a second implementation of VNET/P for a different host environment, the Kitten lightweight kernel. These experiences are in support of hardware independence, the 3rd goal of VNET articulated in Section B.2. Regardless of the underlying networking hardware or host OS, the guests see a simple Ethernet LAN.

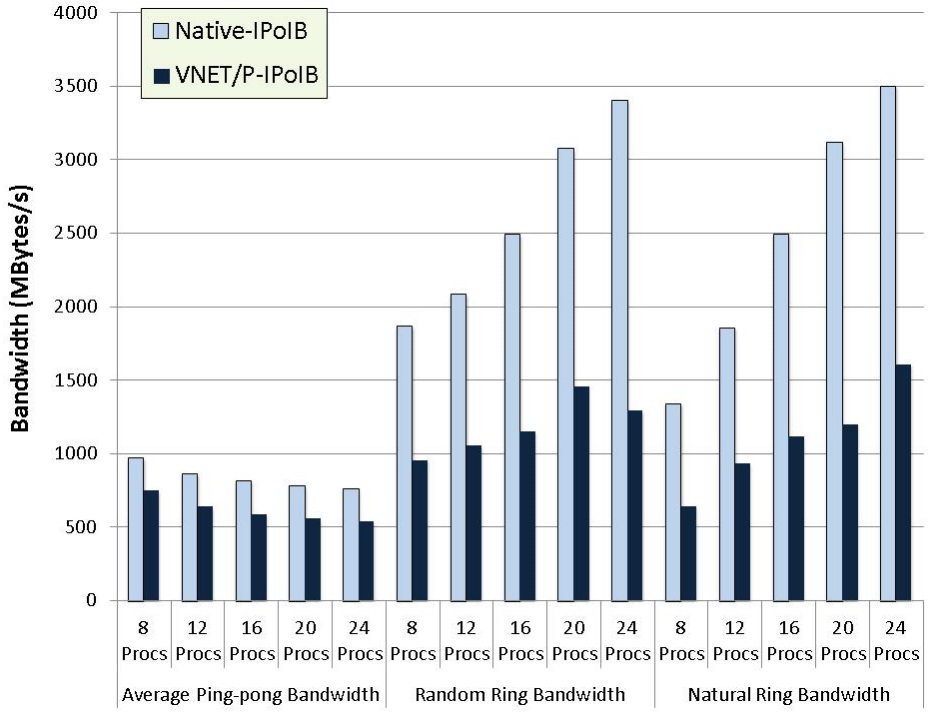
B.5.1 Infiniband

Infiniband in general, and our Mellanox hardware in particular, supports IP over Infiniband (IPoIB) [60]. The IPoIB functionality of the device driver, running in the host, allows the TCP/IP stack of the host to use the NIC to transport IP packets. Because VNET/P sends UDP packets, it is trivial to direct it to communicate over the IB fabric via this mechanism. No code changes to VNET/P are needed. The primary effort is in bringing up the Infiniband NICs with IP addresses, and establishing routing rules based on them. The IP functionality of Infiniband co-exists with native functionality.

It is important to point out that we have not yet tuned VNET/P for this configuration, but we describe our current results below. On our testbed the preliminary, "out of the box" performance of VNET/P includes a ping latency of 155 μs and a TTCP throughput of 3.6 Gbps. We also ran the HPCC benchmarks on a 6-node cluster with the same configuration as in Section B.4.4, comparing the performance of VNET/P on IPoIB with native performance on IPoIB.



(a) Latency



(b) Bandwidth

Figure B.16: Preliminary results for the HPCC latency-bandwidth benchmark, comparing the fully virtualized environment using VNET/P running on IPoIB with the purely native environment using IPoIB. Ring-based bandwidths are multiplied by the total number of processes in the test.

Figure B.16 shows the results for the HPCC latency-bandwidth benchmark with different numbers of test processes. In the pingpong test, VNET/P on IPoIB can achieve 70–75% of native bandwidth, while its latencies are about 3 to 4 times higher. In the ring-based tests, VNET/P on IPoIB can achieve on average 50–55% of the native bandwidth, while the latency is on average about 2.5 to 3 times higher.

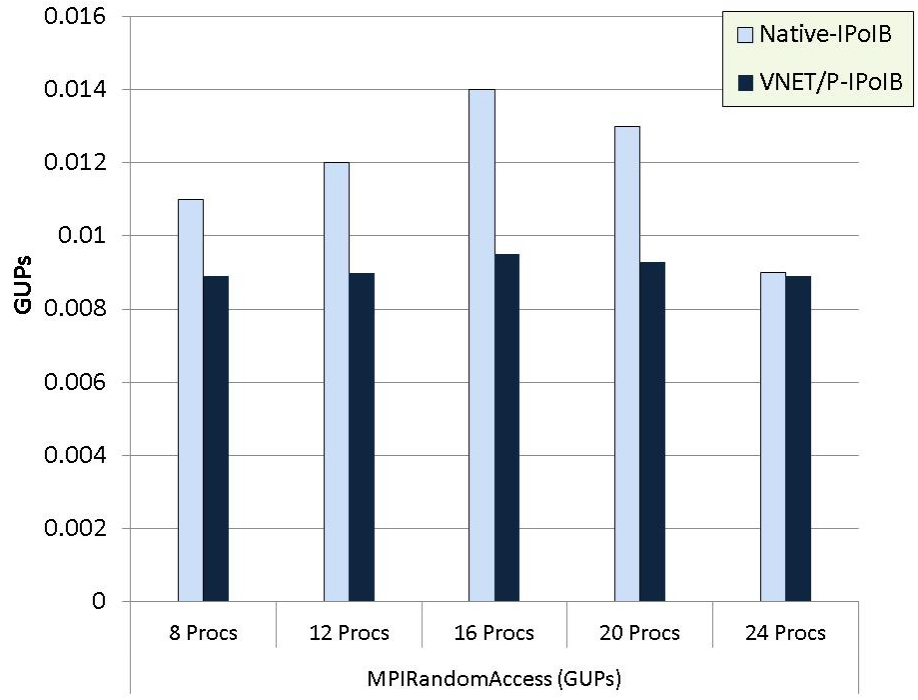
Figure B.17 shows the results of two HPCC application benchmarks comparing the VNET/P and native cases. For MPIRandomAccess, VNET/P on IPoIB achieves 75–80% of the native application performance, and performance scales similarly. On the other hand, the results of MPIFFT show that VNET/P's achieve bout 30–45% of the native performance, similar performance scaling.

Note also that there are two overheads involved in this form of mapping, the VNET/P overhead, and the IPoIB overhead. We have not yet determined the relative sizes of these overheads.

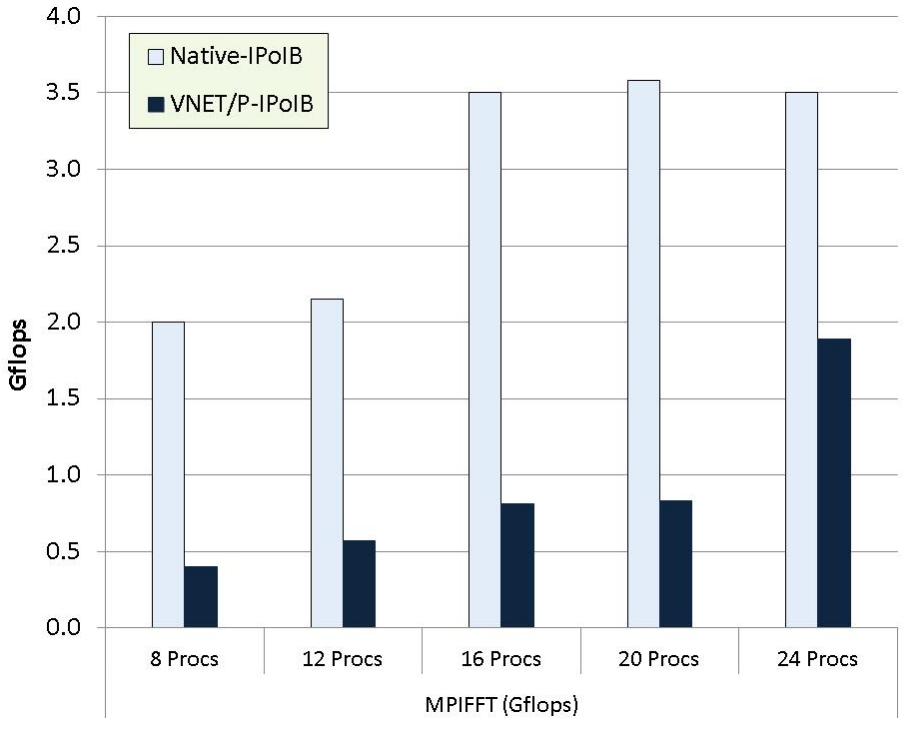
B.5.2 Cray Gemini

As a proof-of-concept we have brought Palacios and VNET/P up on a Cray XK6, the Curie testbed at Sandia National Labs. Curie consists of 50 compute nodes that each combine an AMD Opteron 6272, an nVidia Tesla M2090, and 32 GB of RAM. The nodes communicate using the Gemini network, which is a three dimensional torus. The Gemini NIC connects to a node via a HyperTransport 3 link. The theoretical peak inter-node latency and bandwidth are 1.5 μs and 5 GB/s (40 Gbps). Measured MPI throughput for large (64 KB) messages is about 4 GB/s (32 Gbps). Vaughan et al [131] give a more detailed description of Gemini in the context of a large installation.

Although the XK6 runs Cray Compute Node Linux as its host OS, the changes to Palacios needed to support this configuration have proven to be relatively minor. Gemini supports an "IPoG" layer. This creates a virtual Ethernet NIC abstraction on top of the



(a) MPIRandomAccess



(b)MPIFFT

Figure B.17: Preliminary results for HPCC application benchmarks, comparing the fully virtualized environment using VNET/P over IPoIB with the purely native environment using IPoIB.

underlying Gemini NIC, and this abstraction then supports the host OS's TCP/IP stack. Given this, VNET/P maps its encapsulated UDP traffic straightforwardly, simply by using the IP addresses assigned to the relevant Gemini NICs. Except for talking to the IPoG layer instead of talking to an Ethernet NIC, the architecture of VNET/P on Gemini is identical to that shown in Figure B.1.

We are in the process of tuning VNET/P in this environment. Our preliminary results show VNET/P achieves a TCP throughput of 1.6 GB/s (13 Gbps). We are currently addressing a likely precision-timing problem that is limiting performance. We have not yet determined the relative overheads of VNET/P and IPoG.

B.5.3 VNET/P for Kitten

The VNET/P model can be implemented successfully in host environments other than Linux. Specifically, we have developed a version of VNET/P for the Kitten Lightweight Kernel. This version focuses on high performance Infiniband support and leverages precision timing and other functionality made possible by Kitten.

Figure B.18 shows the architecture of VNET/P for for Kitten and can be compared and contrasted with the VNET/P for Linux architecture shown in Figure B.1. While the architectures are substantially different, the abstraction that the guest VMs see is identical: the guests still believe they are connected by a simple Ethernet network.

Kitten has, by design, a minimal set of in-kernel services. For this reason, the VNET/P Bridge functionality is not implemented in the kernel, but rather in a privileged service VM called the Bridge VM that has direct access to the physical Infiniband device. In place of encapsulating Ethernet packets in UDP packets for transmission to a remote VNET/P core, VNET/P's for InfiniBand on Kitten simply maps Ethernet packets to InfiniBand frames. These frames are then transmitted through an InfiniBand queue pair accessed via the Linux IPoIB framework.

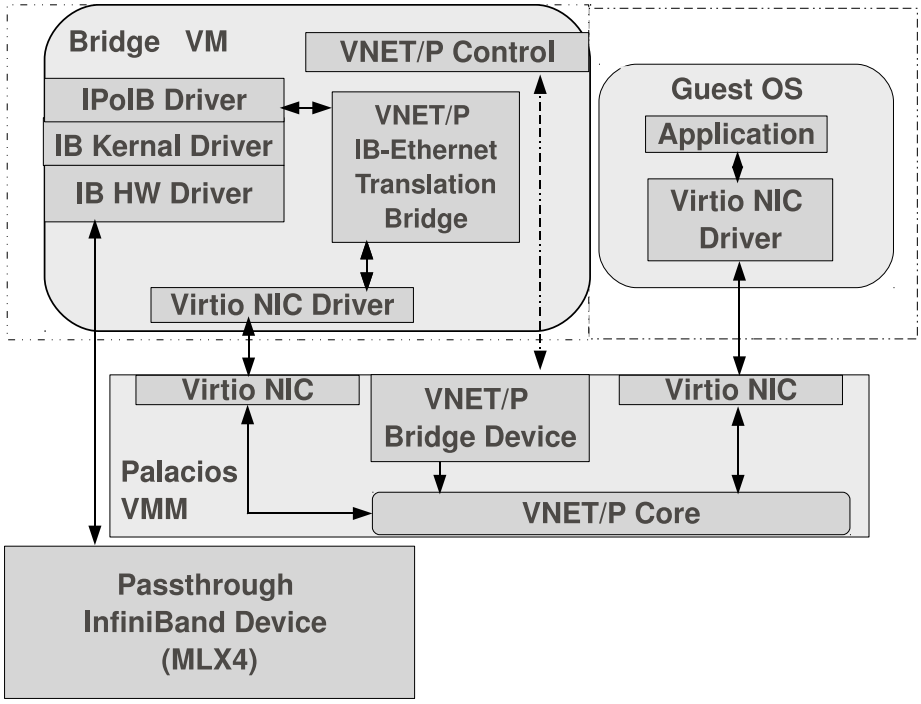


Figure B.18: The architecture of VNET/P for Kitten running on InfiniBand. Palacios is embedded in the Kitten lightweight kernel and forwards packets to/receives packets from a service VM called the bridge VM.

We conducted preliminary performance tests of VNET/P for Kitten on InfiniBand using 8900 byte TCP payloads running on ttcp on a testbed similar to the one described in Section B.4.1. Here, each node was a dual quad-core 2.3 GHz 2376 AMD Opteron machine with 32 GB of RAM and a Mellanox MT26428 InfiniBand NIC in a PCI-e slot. The Infiniband NICs were connected via a Mellanox MTS 3600 36-port 20/40Gbps InfiniBand switch. This version of VNET/P is able to achieve 4.0 Gbps end-to-end TCP throughput, compared to 6.5 Gbps when run natively on top of IP-over-InfiniBand in Reliable Connected (RC) mode.

The Kitten version of VNET/P described here is the basis of VNET/P+, which is de-

scribed in more detail elsewhere [21]. VNET/P+ uses two techniques, optimistic interrupts and cut-through forwarding, to increase the throughput and lower the latency compared to VNET/P. Additionally, by leveraging the low-noise environment of Kitten, it is able to provide very little jitter in latency compared to the Linux version. When run with a 10 Gbps Ethernet network, native throughput is achievable. We are currently back-porting the VNET/P+ techniques into the Linux version.

B.6 Conclusion

We have described the VNET model of overlay networking in a distributed virtualized computing environment and our efforts in extending this simple and flexible model to support tightly-coupled high performance computing applications running on high-performance networking hardware in current supercomputing environments, future data centers, and future clouds. VNET/P is our design and implementation of VNET for such environments. Its design goal is to achieve near-native throughput and latency on 1 and 10 Gbps Ethernet, InfiniBand, Cray Gemini, and other high performance interconnects.

To achieve performance, VNET/P relies on several key techniques and systems, including lightweight virtualization in the Palacios virtual machine monitor, high-performance I/O, and multicore overlay routing support. Together, these techniques enable VNET/P to provide a simple and flexible level 2 Ethernet network abstraction in a large range of systems no matter what the actual underlying networking technology is. While our VNET/P implementation is tightly integrated into our Palacios virtual machine monitor, the principles involved could be used in other environments as well.

Bibliography

[1] FUSE: Filesystem in userspace. http://fuse.sourceforge.net/.

[2] Memcached. http://memcached.org/.

[3] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic routing in future data centers. In *Proceedings of SIGCOMM* (August 2010).

[4] AGARWAL, S., GARG, R., AND GUPTA, M. S. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS 04)* (2004).

[5] AL-KISWANY, S., SUBHRAVETI, D., SARKAR, P., AND RIPEANU, M. VMFlock: virtual machine co-migration for the cloud. In *Proceedings of the 20th International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'11)* (2011), pp. 159–170.

[6] AMD CORPORATION. Pacifica virtualization extensions . http://enterprise.amd.com/Enterprise/serverVirtualization.aspx, 2005.

[7] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP'01)* (March 2001).

[8] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium (OLS '09)* (2009).

[9] BAUTISTA-GOMEZ, L. A., TSUBOI, S., KOMATITSCH, D., CAPPELLO, F., MARUYAMA, N., AND MATSUOKA, S. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011).

[10] BAVIER, A. C., FEAMSTER, N., HUANG, M., PETERSON, L. L., AND REXFORD, J. In vini veritas: realistic and controlled network experimentation. In *Proceedings of the ACM SIGCOMM 2006 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'2006)* (September 2006).

[11] BENT, J., GIBSON, G. A., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09)* (2009).

[12] BISWAS, S., SUPINSKI, B. R. D., SCHULZ, M., FRANKLIN, D., SHERWOOD, T., AND CHONG, F. T. Exploiting data similarity to reduce memory footprints. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)* (2011).

[13] BRONEVETSKY, G., MARQUES, D., PINGALI, K., AND RUGINA, R. Compiler-enhanced incremental checkpointing. In *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium(IPDPS 2009)* (2009).

[14] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'97)* (1997), pp. 143–156.

[15] CHEN, Y., PLANK, J. S., AND LI, K. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'97)* (San Jose, November 1997).

[16] CHEN, Z., AND DONGARRA, J. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)* (2006), pp. 97–97.

[17] CHIN, A. Locality-preserving hash functions for general purpose parallel computation. *Algorithmica 12*, 2/3 (1994), 170–181.

[18] CHU, Y., RAO, S., SHESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'01)* (August 2001).

[19] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI'05)* (2005).

[20] COHEN, E., AND SHENKER, S. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'02)* (2002).

[21] CUI, Z., XIA, L., BRIDGES, P., DINDA, P., AND LANGE, J. Optimizing Overlay-based Virtual Networking Through Optimistic Interrupts and Cut-through Forwarding. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12) (Supercomputing)* (November 2012).

[22] CULLY, B., LEFEBVRE, G., MEYER, D. T., FEELEY, M., HUTCHINSON, N. C., AND WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)* (2008).

[23] DABEK, F. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2005.

[24] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the ACM symposium on Operating systems principles (SOSP'01)* (2001).

[25] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation (OSDI 2004)* (2004).

[26] DESHPANDE, U., WANG, X., AND GOPALAN, K. Live gang migration of virtual machines. In *Proceedings of the 20th International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'11)* (2011), pp. 135–146.

[27] DINDA, P., SUNDARARAJ, A., LANGE, J., GUPTA, A., AND LIN, B. Methods and Systems for Automatic Inference and Adaptation of Virtualized Computing Environments, March 2012. United States Patent Number 8,145,760.

[28] DONGARRA, J., AND LUSZCZEK, P. Introduction to the HPC Challenge Benchmark Suite. Tech. rep., 2005.

[29] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies (FAST 2009)* (2009), pp. 197–210.

[30] EMENEKER, W., AND STANZIONE, D. HPC cluster readiness of Xen and User Mode Linux. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06)* (2006).

[31] EVANGELINOS, C., AND HILL, C. Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Proceedings of Cloud Computing and its Applications (CCA)* (October 2008).

[32] FERREIRA, K., STEARLEY, J., LAROS, III, J. H., OLDFIELD, R., PEDRETTI, K., BRIGHTWELL, R., RIESEN, R., BRIDGES, P. G., AND ARNOLD, D. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011), pp. 44:1–44:12.

[33] FERREIRA, K. B., RIESEN, R., BRIGHWELL, R., BRIDGES, P., AND ARNOLD, D. libhashckpt: hash-based incremental checkpointing using gpu's. In *Proceed-*

ings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface (EuroMPI'11) (2011).

[34] FIGUEIREDO, R., DINDA, P. A., AND FORTES, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)* (May 2003).

[35] GABRIEL, E., FAGG, G. E., BOSILCA, G., ANGSKUN, T., DONGARRA, J. J., SQUYRES, J. M., SAHAY, V., KAMBADUR, P., BARRETT, B., LUMSDAINE, A., CASTAIN, R. H., DANIEL, D. J., GRAHAM, R. L., AND WOODALL, T. S. open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting* (September 2004).

[36] GANGULY, A., AGRAWAL, A., BOYKIN, P. O., AND FIGUEIREDO, R. IP over P2P: Enabling self-configuring virtual ip networks for grid computing. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April 2006).

[37] GAVRILOVSKA, A., KUMAR, S., RAJ, H., SCHWAN, K., GUPTA, V., NATHUJI, R., NIRANJAN, R., RANADIVE, A., AND SARAIYA, P. High performance hypervisor architectures: Virtualization in HPC systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt)* (2007).

[38] GHARAIBEH, A., AL-KISWANY, S., GOPALAKRISHNAN, S., AND RIPEANU, M. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), Proceedings of the International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'10).

[39] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: Bare-Metal Performance for I/O Virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)* (March 2012).

[40] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)* (2009).

[41] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)* (August 2009).

[42] GUPTA, A. *Black Box Methods for Inferring Parallel Applications Properties in Virtual Environments*. PhD thesis, NWU-EECS-08-04, Northwestern University, Department of Electrical Engineering and Computer Science, March 2008.

[43] GUPTA, A., AND DINDA, P. A. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)* (June 2004).

[44] GUPTA, A., ZANGRILLI, M., SUNDARARAJ, A., HUANG, A., DINDA, P., AND LOWEKAMP, B. Free network measurement for virtual machine distributed com-

puting. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2006).

[45] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (2008), pp. 309–322.

[46] HALE, K. C., XIA, L., AND DINDA, P. A. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)* (2012).

[47] HERMENIER, F., LORCA, X., MENAUD, J.-M., MULLER, G., AND LAWALL, J. L. Entropy: a consolidation manager for clusters. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE 2009)* (2009).

[48] HEROUX, M. A., DOERFER, D. W., CROZIER, P. S., WILLENBRING, J. M., EDWARDS, H. C., WILLIAMS, A., RAJAN, M., KEITER, E. R., THORNQUIST, H. K., AND NUMRICH, R. W. Improving Performance via Mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratory, 2009.

[49] HINES, M. R., AND GOPALAN, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '09)* (2009).

[50] HU, L., JIN, H., LIAO, X., XIONG, X., AND LIU, H. Magnet: A novel scheduling policy for power reduction in cluster with virtual machines. In *Proceedings of*

the 2008 IEEE International Conference on Cluster Computing (CLUSTER 2008) (2008), pp. 13–22.

[51] HUANG, W., GAO, Q., LIU, J., AND PANDA, D. K. High performance virtual machine migration with rdma over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing* (2007), CLUSTER '07, pp. 11–20.

[52] HUANG, W., LIU, J., ABALI, B., AND PANDA, D. K. A case for high performance computing with virtual machines. In *20th Annual International Conference on Supercomputing (ICS)* (2006), pp. 125–134.

[53] INNOVATIVE COMPUTING LABORATORY. HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[54] INTEL. Intel cluster toolkit 3.0 for linux. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[55] INTEL CORPORATION. Intel virtualization technology specification for the ia-32 intel architecture, April 2005.

[56] JIANG, X., AND XU, D. Violin: Virtual internetworking on overlay infrastructure. Tech. Rep. CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[57] JOSEPH, D. A., KANNAN, J., KUBOTA, A., LAKSHMINARAYANAN, K., STOICA, I., AND WEHRLE, K. Ocala: An architecture for supporting legacy applications over overlays. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)* (May 2006).

[58] KAI LI, JEFFREY F. NAUGHTON, J. S. P. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'90)* (1990).

[59] KALLAHALLA, M., UYSAL, M., SWAMINATHAN, R., LOWELL, D. E., WRAY, M., CHRISTIAN, T., EDWARDS, N., DALTON, C. I., AND GITTLER, F. Softudc: A software-based data center for utility computing. *IEEE Computer 37*, 11 (2004), 38–46.

[60] KASHYAP, V. IP over Infiniband (IPoIB) Architecture. IETF Network Working Group Request for Comments RFC 4392, April 2006. Current expiration: August 2012.

[61] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in seattle: a scalable Ethernet architecture for large enterprises. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'08)* (August 2008).

[62] KIRAN, R. B., TATI, K., CHUNG CHENG, Y., SAVAGE, S., AND VOELKER, G. M. Total recall: System support for automated availability management. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)* (2004), pp. 337–350.

[63] KLOSTER, J., KRISTENSEN, J., AND MEJLHOLM, A. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor. Tech. rep., Master Thesis, Department of Computer Science, Aalborg University, 2006.

[64] K.SUZAKI, T.YAGI, K.IIJIMA, N.A.QUYNH, AND Y.WATANABE. Effect of readahead and file system block reallocation for LBCAS (LoopBack Content Addressable Storage). In *Proceedings of the Linux Symposium (OLS'09)* (2009).

[65] KUMAR, S., RAJ, H., SCHWAN, K., AND GANEV, I. Re-architecting vmms for multicore systems: The sidecore approach. In *Proceedings of the 2007 Workshop on the Interaction between Operating Systems and Computer Architecture* (June 2007).

[66] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)* (2009).

[67] LANGE, J. Symbiotic virtualization. Tech. rep., Doctoral Dissertation, NWU-EECS-10-08, Department of Electrical Engineering and Computer Science, Northwestern University, 2010.

[68] LANGE, J., AND DINDA, P. Transparent network services via a virtual traffic layer for virtual machines. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (June 2007).

[69] LANGE, J., DINDA, P., HALE, K., AND XIA, L. An introduction to the Palacios virtual machine monitor—release 1.3. Tech. Rep. NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, October 2011.

[70] LANGE, J., SUNDARARAJ, A., AND DINDA, P. Automatic dynamic run-time optical network reservations. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[71] LANGE, J. R., PEDRETTI, K. T., DINDA, P. A., BRIDGES, P. G., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th International Conference on Virtual Execution Environments(VEE'2011)* (2011).

[72] LANGE, J. R., PEDRETTI, K. T., HUDSON, T., DINDA, P. A., CUI, Z., XIA, L., BRIDGES, P. G., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)* (2010).

[73] LEVY, S., FERREIRA, K. B., BRIDGES, P. G., THOMPSON, A. P., AND TROTT, C. An examination of content similarity within the memory of hpc applications. Tech. Rep. SAND2013-0055, Sandia National Laboratory, 2013.

[74] LI, K., EFFREY F. NAUGHTON, AND PLANK, J. S. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems 5*, 8 (1994), 874–879.

[75] LI, T., ZHOU, X., BRANDSTATTER, K., ZHAO, D., WANG, K., RAJENDRAN, A., ZHANG, Z., AND RAICU, I. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2013).

[76] LIN, B., AND DINDA, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'05)* (2005).

[77] LIN, B., SUNDARARAJ, A., AND DINDA, P. Time-sharing parallel applications with performance isolation and control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC)* (June 2007).

[78] LIU, H., JIN, H., LIAO, X., HU, L., AND YU, C. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing(HPDC'09)* (2009), pp. 101–110.

[79] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (May 2006).

[80] LU, C.-D. *Scalable Diskless Checkpointing for Large Parallel Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 2005.

[81] MAHALINGAM, M., DUTT, D., DUDA, K., AGARWAL, P., KREEGER, L., SRIDHAR, T., BURSELL, M., AND WRIGHT, C. VXLAN: A framework for overlaying virtualized layer 2 networks over layer 3 networks. IETF Network Working Group Internet Draft, Febuary 2012. Current expiration: August 2012.

[82] MANDAL, P. S., AND MUKHOPADHYAYA, K. Concurrent checkpoint initiation and recovery algorithms on asynchronous ring network. *Journal of Parallel and Distributed Computing 64*, 5 (2004), 649–661.

[83] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference (USENIX)* (May 2006).

[84] MERGEN, M. F., UHLIG, V., KRIEGER, O., AND XENIDIS, J. Virtualization for high-performance computing. *Operating Systems Review 40*, 2 (2006), 8–11.

[85] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: enlightened page sharing. In *Proceedings of the USENIX Annual Technical Conference (USENIX'09)* (2009).

[86] MOODY, A., BRONEVETSKY, G., MOHROR, K., AND DE SUPINSKI, B. R. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)* (2010), pp. 1–11.

[87] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'01)* (2001), pp. 174–187.

[88] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)* (2009).

[89] NAGARAJAN, A. B., MUELLER, F., ENGELMANN, C., AND SCOTT, S. L. Proactive fault tolerance for HPC with Xen virtualization. In *21st Annual International Conference on Supercomputing (ICS'07)* (2007), pp. 23–32.

[90] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd conference on Networked Systems Design and Implementation (NSDI'06)* (2006).

[91] NATHUJI, R., AND SCHWAN, K. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (2007), pp. 265–278.

[92] NISHIMURA, H., MARUYAMA, N., AND MATSUOKA, S. Virtual clusters on the fly - fast, scalable, and flexible installation. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid* (2007), CCGRID '07, pp. 549–556.

[93] NURMI, D., WOLSKI, R., GRZEGORZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)* (May 2009).

[94] OSTERMANN, S., IOSUP, A., YIGITBASI, N., PRODAN, R., FAHRINGER, T., AND EPEMA, D. An early performance analysis of cloud computing services for scientific computing. Tech. Rep. PDS2008-006, Delft University of Technology, Parallel and Distributed Systems Report Series, December 2008.

[95] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[96] PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys*

European Conference on Computer Systems 2007 (2007), EuroSys '07, pp. 289–302.

[97] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)* (1988).

[98] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[99] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter 1995 Technical Conference* (January 1995), pp. 213–223.

[100] PLANK, J. S., CHEN, Y., LI, K., BECK, M., AND KINGSLEY, G. Memory exclusion: Optimizing the performance of checkpointing systems. Tech. Rep. UT-CS-96-335, University of Tennessee, August 1996.

[101] PLANK, J. S., LI, K., AND PUENING, M. A. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (October 1998), 972–986.

[102] PLIMPTON, S. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J Comp Phys, 117, 1-19 (1995) 117* (1995), 1–19.

[103] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[104] RAJ, H., AND SCHWAN, K. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC'07)* (July 2007).

[105] REFSON, K. Moldy Users Manual, 2001.

[106] RITCHIE, D. M. A guest facility for unicos. In *Proceedings of the (UNIX) and Supercomputers Workshop* (September 1988), (USENIX), (USENIX), pp. 19–24.

[107] RITEAU, P., MORIN, C., AND PRIOL, T. Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing. In *Proceedings of the 17th International Conference on Parallel Computing (Euro-Par 2011)* (2011).

[108] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review 42*, 5 (2008), 95–103.

[109] RUTH, P., JIANG, X., XU, D., AND GOASGUEN, S. Towards virtual distributed environments in a shared infrastructure. *IEEE Computer* (May 2005).

[110] RUTH, P., MCGACHEY, P., JIANG, X., AND XU, D. Viocluster: Virtualization for dynamic computational domains. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)* (September 2005).

[111] SANCHO, J. C., PETRINI, F., JOHNSON, G., FERNÁNDEZ, J., AND FRACHTENBERG, E. On the feasibility of incremental checkpointing for scientific computing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)* (2004).

[112] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)* (2002), pp. 377–390.

[113] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Survey (CSUR) 22* (December 1990), 299–319.

[114] SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., ZWAENEPOEL, W., AND WILLMANN, P. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA'07)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 306–317.

[115] SHEN, K., YANG, T., AND CHU, L. Clustering support and replication management for scalable network services. *IEEE Transactions on Parallel and Distributed Systems 14*, 11 (2003), 1168–1179.

[116] STOICA, I., MORRIS, R., KARGER, D. R., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'01)* (2001).

[117] SUGERMAN, J., VENKITACHALAN, G., AND LIM, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[118] SUNDARARAJ, A. *Automatic, Run-time, and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments*. PhD thesis, Northwestern University, December 2006. Technical Report NWU-EECS-06-18, Department of Electrical Engineering and Computer Science.

[119] SUNDARARAJ, A., AND DINDA, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

[120] SUNDARARAJ, A., GUPTA, A., , AND DINDA, P. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[121] SUNDARARAJ, A., SANGHI, M., LANGE, J., AND DINDA, P. An optimization problem in adaptive virtual environmnets. In *Proceedings of the seventh Workshop on Mathematical Performance Modeling and Analysis (MAMA)* (June 2005).

[122] THIBAULT, S., AND DEEGAN, T. Improving performance by embedding HPC applications in lightweight Xen domains. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt)* (2008), pp. 9–15.

[123] TIKOTEKAR, A., VALLÉE, G., NAUGHTON, T., ONG, H., ENGELMANN, C., SCOTT, S. L., AND FILIPPI, A. M. Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines. In *2nd Workshop on System-Level Virtualization for High Performance Computing (HPCVirt)* (2008), pp. 16–23.

[124] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T. C., AND PERRIG, A. Opportunistic use of content addressable storage for distributed file systems. In *USENIX Annual Technical Conference, General Track* (2003), pp. 127–140.

[125] TSUGAWA, M. O., AND FORTES, J. A. B. A virtual network (vine) architecture for grid computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS)* (April 2006).

[126] UHLEMANN, K., ENGELMANN, C., AND SCOTT, S. L. Joshua: Symmetric active/active replication for highly available hpc job and resource management. In *Proceedings of IEEE Conference on Cluster Computing (CLUSTER'06)* (2006).

[127] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTIN, F., ANDERSON, A., BENNETTT, S., KAGI, A., LEUNG, F., AND SMITH, L. The architecture of virtual machines. *IEEE Computer* (May 2005), 48–56.

[128] VAN DER WIJNGAART, R. NAS parallel benchmarks version 2.4. Tech. Rep. NAS-02-007, NASA Advanced Supercomputing (NAS Division), NASA Ames Research Center, October 2002.

[129] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)* (2004).

[130] VANDERWERF, J. SuperHash. redshift.sourceforge.net/superhash/.

[131] VAUGHAN, C., RAJAN, M., BARRETT, R., DOERFLER, D., AND PEDRETTI, K. Investigating the impact of the Cielo Cray XE6 architecture on scientific application codes. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW 2011)*.

[132] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and contain-

ment in the potemkin virtual honeyfarm. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'05)* (2005).

[133] WALDSPURGER, C. A. Memory resource management in vmware esx server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)* (2002).

[134] WOLINSKY, D., LIU, Y., JUSTE, P. S., VENKATASUBRAMANIAN, G., AND FIGUEIREDO, R. On the design of scalable, self-configuring virtual networks. In *Proceedings of 21st ACM/IEEE International Conference of High Performance Computing, Networking, Storage, and Analysis (SuperComputing / SC)* (November 2009).

[135] WOOD, T., SHENOY, P. J., VENKATARAMANI, A., AND YOUSIF, M. S. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)* (2007).

[136] WOOD, T., TARASUK-LEVIN, G., SHENOY, P. J., DESNOYERS, P., CECCHET, E., AND CORNER, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE'09)* (2009).

[137] WOZNIAK, J. M., JACOBS, B., LATHAM, R., LANG, S., SON, S. W., AND ROSS., R. C-mpi: A dht implementation for grid and hpc environments. Tech. Rep. ANL/MCS-P1746-0410, Argonne National Laboratory, 2010.

[138] XIA, L., CUI, Z., LANGE, J., TANG, Y., DINDA, P., AND BRIDGES, P. VNET/P: Bridging the cloud and high performance computing through fast overlay network-

ing. In *Proceedings of the 21st ACM Symposium on High-performance Parallel and Distributed Computing (HPDC'12)* (June 2012).

[139] XIA, L., CUI, Z., LANGE, J., TANG, Y., DINDA, P., AND BRIDGES, P. Fast VMM-based Overlay Networking for Bridging the Cloud and High Performance Computing. *Journal of Cluster Computing DOI 10.1007/s10586-013-0274-7* (2013).

[140] XIA, L., AND DINDA, P. A. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing (VTDC '12)* (2012).

[141] XIA, L., KUMAR, S., YANG, X., GOPALAKRISHNAN, P., LIU, Y., SCHOENBERG, S., AND GUO, X. Virtual WiFi: bring virtualization from wired to wireless. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE'11)* (2011), pp. 181–192.

[142] XIA, L., LANGE, J., AND DINDA, P. Towards virtual passthrough I/O on commodity devices. In *Proceedings of the Workshop on I/O Virtualization at OSDI'08 (WIOV'08)* (December 2008).

[143] XIA, L., LANGE, J., DINDA, P. A., AND BAE, C. Investigating virtual passthrough I/O on commodity devices. *Operating Systems Review 43*, 3 (2009), 83–94.

[144] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 05)* (2005).

[145] YOUSEFF, L., WOLSKI, R., GORDA, B., AND KRINTZ, C. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In *2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (2006), p. 1.

[146] YU, H., GIBBONS, P. B., AND NATH, S. Availability of multi-object operations (awarded best paper). In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)* (2006).

[147] ZANDY, V. C., MILLER, B. P., AND LIVNY, M. Process hijacking. In *Proceedings of the International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'99)* (1999).