Noname manuscript No.
(will be inserted by the editor)

A Fault-Tolerant Strategy for Virtualized HPC Clusters

John Paul Walters · Vipin Chaudhary

Abstract Virtualization is a common strategy for improving the utilization of existing computing resources, particularly within data centers. However, its use for high performance computing (HPC) applications is currently limited despite its potential for both improving resource utilization as well as providing resource guarantees to its users. In this article we systematically evaluate three major virtual machine implementations for computationally intensive HPC applications using various standard benchmarks. Using VMWare Server, Xen, and OpenVZ we examine the suitability of full virtualization (VMWare), paravirtualization (Xen), and operating system-level virtualization (OpenVZ) in terms of network utilization, SMP performance, file system performance, and MPI scalability. We show that the operating system-level virtualization provided by OpenVZ provides the best overall performance, particularly for MPI scalability. With the knowledge gained by our VM evaluation, we extend OpenVZ to include support for checkpointing and fault-tolerance for MPI-based virtual server distributed computing.

Keywords Virtualization · Benchmark · Fault-Tolerance · Checkpointing · MPI

1 Introduction

The use of virtualization in computing is a well-established idea dating back more than 30 years [18]. Traditionally, its use has meant accepting a sizable performance penalty in exchange for the convenience of the virtual machine. Now, however, the performance penalties have been reduced. Faster processors as well as more efficient virtualization solutions now allow even modest desktop computers to host powerful virtual machines.

Soon large computational clusters will be leveraging the benefits of virtualization in order to enhance the utility of the cluster as well as to ease the burden of

J. P. Walters and V. Chaudhary
University at Buffalo, The State University of New York
Department of Computer Science and Engingeering
E-mail: {waltersj, vipin}@buffalo.edu

administering such large numbers of machines. Indeed, Amazon's Elastic Compute Cloud (EC2) already uses the Xen hypervisor to provide customers with a completely tailored environment on which to execute their computations [42]. Virtual machines allow administrators to more accurately control their resources while simultaneously protecting the host node from malfunctioning user-software. This allows administrators to provide "sandbox-like" environments with minimal performance reduction from the user's perspective, and also allows users the flexibility to customize their computing environment.

However, to date, a comprehensive examination of the various virtualization strategies and implementations has not been conducted, particularly with an eye towards its use in HPC environments. We begin by conducting an evaluation of three major virtualization technologies: full virtualization, paravirtualization, and operating system-level virtualization which are represented by VMWare Server, Xen, and OpenVZ, respectively. We show that OpenVZ's operating system-level virtualization provides the lowest overhead, and in most cases outperforms both VMWare Server and Xen for distributed computations, such as MPI.

However, with the increased use of virtualized HPC clusters, issues of fault-tolerance must be addressed in the context of distributed computations. To address the challenges faced in checkpointing current and future virtualized distributed systems, we propose a fault-tolerant system based on OpenVZ [36]. To do so, we leverage the existing checkpoint/restart mechanism within OpenVZ, and enhance its utility through a checkpoint-enabled LAM/MPI implementation and a lightweight checkpoint/replication daemon, *Ovzd*. Our system allows OpenVZ's virtual private servers (VPS) to initiate system checkpoints and to replicate those checkpoints to additional host machines for added fault resiliency.

We make the following contributions in this article:

1. **Virtualization Evaluation:** We evaluate several virtualization solutions for single node performance and scalability. We focus our tests on industry-standard scientific benchmarks including SMP tests through the use of OpenMP implementations of the NAS Parallel Benchmarks (NPB) [4]. We examine file system and network performance (using IOZone [25] and Netperf [26]) in the absence of MPI benchmarks in order to gain insight into the potential performance bottlenecks that may effect distributed computing. We then extend our evaluation to the cluster-level and benchmark the virtualization solutions using the MPI implementation of NPB and the High Performance LINPACK benchmark (HPL) [13].
2. **VM Checkpointing of MPI Computations:** Building on the results of our virtualization evaluation, we describe and evaluate a fully checkpoint-enabled fault-tolerance solution for MPI computations within the OpenVZ virtualization environment. The system supports the checkpointing of both the running computation, as well as incremental file system checkpointing to ensure data consistency upon restoring a failed computation. We use local disk checkpointing with replication in order to minimize overhead while providing high reliability.

Using our system, additional fault-tolerance work can be easily developed and deployed with support for full system fault-tolerance. Further, our system can be

extended to include support for job schedulers such as PBS/Torque, and system monitors such as Ganglia [29]. With such functionality, preemptive checkpointing and migration can be used to minimize checkpoint overhead while still providing maximal fault-resilience.

The rest of this article is organized as follows: in Section 2 we discuss the background of virtualization systems and checkpointing, while in Section 3 we briefly describe Xen, VMWare Server, and OpenVZ. In Section 4 we present the results of our performance comparison. In Section 5 we detail our MPI-enabled checkpointing implementation. In Section 6 we provide a brief theoretical framework used for analyzing our performance-related data. In Section 7 we demonstrate the performance of our implementation. In Section 8 we detail the work related to our project, and in Section 9 present our conclusions.

2 VM and Checkpointing Background

Both checkpointing and virtualization are well-studied in the scientific literature. In this section we provide a brief overview of the major issues relating to virtual machines and checkpointing and how the two relate to one another. We describe the major types of virtualization strategies that are currently in use, as well as the three main levels at which checkpointing can be accomplished. This background is necessary in order to understand the differences and performance implications between the evaluated virtual machines.

2.1 Existing Virtualization Technologies

To accurately characterize the performance of different virtualization technologies we begin with an overview of the major virtualization strategies that are in common use for production computing environments. In general, most virtualization strategies fall into one of four major categories:

1. **Full Virtualization:** Also sometimes called hardware emulation. In this case an unmodified operating system is run using a hypervisor to trap and safely translate/execute privileged instructions on-the-fly. Because trapping the privileged instructions can lead to significant performance penalties, novel strategies are used to aggregate multiple instructions and translate them together. Other enhancements, such as binary translation, can further improve performance by reducing the need to translate these instructions in the future [3, 35].
2. **Paravirtualization:** Like full virtualization, paravirtualization also uses a hypervisor. However, unlike full virtualization, paravirtualization requires changes to the virtualized operating system. This allows the VM to coordinate with the hypervisor, reducing the use of the privileged instructions that are typically responsible for the major performance penalties in full virtualization. The advantage is that paravirtualized virtual machines traditionally outperform fully virtualized

virtual machines. The disadvantage, however, is the need to modify the paravirtualized operating system in order to make it hypervisor-aware. This has implications for operating systems whose source code is unavailable.

3. **Operating System-level Virtualization:** The most intrusive form of virtualization is operating system-level virtualization. Unlike both paravirtualization and full virtualization, operating system-level virtualization does not rely on a hypervisor. Instead, the operating system is modified to securely isolate multiple *instances* of an operating system within a single host machine. A single kernel manages the resources of all instances. The guest operating system instances are often referred to as virtual private servers (VPS). The advantage to operating system-level virtualization lies mainly in its high performance. No hypervisor/instruction trapping is necessary. This typically results in system performance of near-native speeds. Further, because a single kernel is used for all operating system instances, fewer resources are required to support the multiple instances. The primary disadvantage, however, is that if the single kernel crashes or is compromised, all VPS instances are compromised.
4. **Native Virtualization:** Native virtualization leverages hardware support for virtualization within a processor to aid in the virtualization effort. It allows multiple unmodified operating systems to execute alongside one another, provided that all operating systems are capable of executing directly on the host processor without emulation. This is unlike the full virtualization technique where it is possible to run an operating system on a fictional (or emulated) processor, though typically with poor performance. In x86_64 series processors, both Intel and AMD support virtualization through the Intel-VT and AMD-V virtualization extensions.

For the remainder of this article we use the word "guest" to refer to the virtualized operating system utilized within any of the above virtualization strategies. Therefore a guest can refer to a VPS (OS-level virtualization), or a VM (full virtualization, paravirtualization).

In order to evaluate the viability of the different virtualization technologies, we compare VMWare Server version 1.0.2[1], Xen version 3.0.4.1, and OpenVZ based on kernel version 2.6.16. These choices allow us to compare full virtualization, paravirtualization, and OS-level virtualization for their use in HPC scenarios, and were the most recent versions available at the time of our testing. We do not include a comparison of native virtualization in our evaluation as previous studies have already shown native virtualization to perform comparably to VMWare's freely available VMWare Player in software mode [1].

2.2 Checkpointing Overview

Virtualization has historically provided an effective means towards fault-tolerance [33]. IBM mainframes, for example, have long used hardware virtualization to achieve processor, memory, and I/O fault-tolerance. With more powerful hardware, virtualization

[1] We had hoped to test VMWare ESX Server, but hardware incompatibilities prevented us from doing so.

can now be used in non-mainframe scenarios with commodity equipment. This has the potential to allow much higher degrees of fault-tolerance than have previously been seen in computational clusters.

In addition to the benefits of fault-tolerance that are introduced by virtualization, system administration can also be improved. By using seamless task migration, administrators can migrate computations away from nodes that need to be brought down for maintenance. This can be done without a full cluster checkpoint, or even a pause in the entire cluster's computation. More importantly, it allows administrators to quickly address any maintenance issues without having to drain a node's computations.

Traditionally, checkpointing has been approached at one of three levels: kernel-level, user-level, or application-level. In kernel-level checkpointing [14], the checkpointer is implemented as a kernel module, or directly within the kernel, making checkpointing fairly straightforward. However, the checkpoint itself is heavily reliant on the operating system (kernel version, process IDs, etc.). User-level checkpointing [44] performs checkpointing using a checkpointing library, enabling a more portable checkpointing implementation at the cost of limited access to kernel-specific attributes (e.g. user-level checkpointers cannot restore process IDs). At the highest level is application-level checkpointing [9] where code is instrumented with checkpointing primitives. The advantage to this approach is that checkpoints can often be restored to arbitrary architectures. However, application-level checkpointers require access to a user's source code and do not support arbitrary checkpointing.

There are two major checkpointing/rollback recovery techniques: coordinated checkpointing and message logging. Coordinated checkpointing requires that all processes come to an agreement on a consistent state before a checkpoint is taken. Upon failure, all processes are rolled back to the most recent checkpoint/consistent state.

Message logging requires distributed systems to keep track of interprocess messages in order to bring a checkpoint up-to-date. Checkpoints can be taken in a non-coordinated manner, but the overhead of logging the interprocess messages can limit its utility. Elnozahy et al. provide a detailed survey of the various rollback recovery protocols that are in use today [15].

2.3 LAM/MPI Background

LAM/MPI [10] is a research implementation of the MPI-1.2 standard [17] with portions of the MPI-2 standard. LAM uses a layered software approach in its construction [34]. In doing so, various modules are available to the programmer that tune LAM/MPI's runtime functionality including TCP, Infiniband, Myrinet, and shared memory communication.

To enable checkpointing, LAM includes a TCP replacement module named CRTCP. The CRTCP module handles the underlying TCP communication, but adds additional byte counters to keep track of the number of bytes sent to/received from every participating MPI process. When checkpointing, these byte counters are exchanged between MPI processes and are used to ensure that all outstanding messages have been collected before checkpointing begins. LAM then uses the BLCR [14] checkpointing module to perform the actual checkpointing of each process. We extend LAM's

built-in checkpointing support to include the checkpointing of a full OpenVZ virtual server by making use of OpenVZ's save/restore (checkpoint/restart) functionality. In doing so, we do not rely on BLCR in any way.

3 Overview of Test Virtualization Implementations

Before our evaluation we first provide a brief overview of the three virtualization solutions that we will be testing: VMWare Server [37], Xen [5], and OpenVZ [36].

VMWare is currently the market leader in virtualization technology. We chose to evaluate the free VMWare Server product, which includes support for both full virtualization and native virtualization, as well as limited (2 CPU) virtual SMP support. Unlike VMWare ESX Server, VMWare Server (formerly GSX Server) operates on top of either the Linux or Windows operating systems. The advantage to this approach is a user's ability to use additional hardware that is supported by either Linux or Windows, but is not supported by the bare-metal ESX Server operating system (SATA hard disk support is notably missing from ESX Server). The disadvantage is the greater overhead from the base operating system, and consequently the potential for less efficient resource utilization.

VMWare Server supports three types of networking: bridged networking, NAT networking, and host-only networking. Bridged networking allows multiple virtual machines to act as if they are each distinct hosts, with each virtual machine being assigned its own IP address. NAT networking allows one or more virtual machines to communicate over the same IP address. Host-only networking can be used to allow the virtual machine to communicate directly with the host without the need for a true network interface. Bridged networking was used for all of our experimentation.

Xen is the most popular paravirtualization implementation in use today. Because of the paravirtualization, guests exist as independent operating systems. The guests typically exhibit minimal performance overhead, approximating near-native performance. Resource management exists primarily in the form of memory allocation, and CPU allocation. Xen file storage can exist as either a single file on the host file system (file backed storage), or in the form of partitions or logical volumes.

Xen networking is completely virtualized (excepting the Infiniband work done by Liu, et al. [22]). A series of virtual ethernet devices are created on the host system which ultimately function as the endpoints of network interfaces in the guests. Upon instantiating a guest, one of the virtual ethernet devices is used as the endpoint to a newly created "connected virtual ethernet interface" with one end residing on the host and another in the guest. The guest sees its endpoint(s) as standard ethernet devices (e.g. "eth0"). Each virtual ethernet devices is also given a MAC address. Bridging is used on the host to allow all guests to appear as individual servers.

OpenVZ is the open source version of Parallels' Virtuozzo product for Linux. It uses operating system-level virtualization to achieve near native performance for operating system guests. Because of its integration with the Linux kernel, OpenVZ is able to achieve a level of granularity in resource control that full virtualization and paravirtualization cannot. Indeed, OpenVZ is able to limit the size of an individual guest's communication buffer sizes (e.g. TCP send and receive buffers) as well as

kernel memory, memory pages, and disk space down to the inode level. Adjustments can only be made by the host system, meaning an administrator of a guest operating system cannot change his resource constraints.

OpenVZ fully virtualizes its network subsystem and allows users to choose between using a virtual network device, or a virtual ethernet device. The default virtual network device is the fastest, but does not allow a guest administrator to manipulate the network configuration. The virtual ethernet device is configurable by a guest administrator and acts like a standard ethernet device. Using the virtual network device, all guests are securely isolated (in terms of network traffic). Our tests were performed using the default virtual network device.

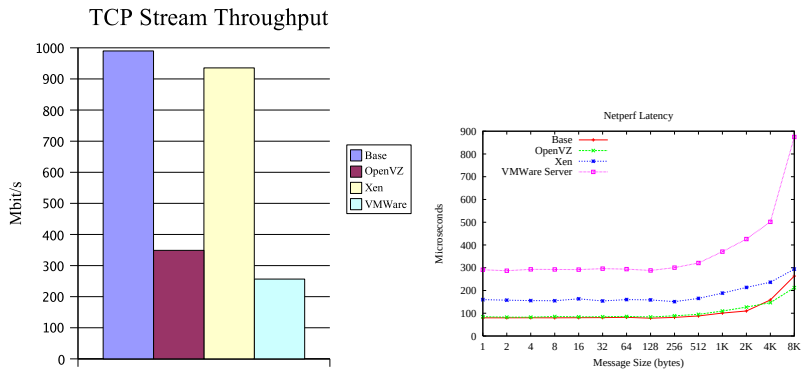
4 Performance Results

We now present the results of our performance analysis. We benchmark each system (Xen, OpenVZ, and VMWare Server) against a base x86 Fedora Core 5 install. All analysis was performed on a cluster of 64 dedicated Dell PowerEdge SC1425 servers consisting of:

- 2x3.2GHz Intel Xeon processors
- Intel 82541GI gigabit ethernet controller
- 2 GB RAM
- 7200 RPM SATA hard disk

In addition, nodes are connected through a pair of Force10 E1200 switches. The E1200 switches are fully non-blocking gigabit ethernet using 48port copper line cards. To maintain consistency, each guest consisted of a minimal install of Fedora Core 5 with full access to both CPUs. The base system and VMWare Server installs used a 2.6.15 series RedHat kernel. The OpenVZ benchmarks were performed on the latest 2.6.16 series OpenVZ kernels, while the Xen analysis was performed using a 2.6.16 series kernel for both the host and guest operating systems. All guest operating systems were allotted 1650 MB RAM, leaving 350 MB for the host operating system. This allowed all benchmarks to run comfortably within the guest without any swapping, while leaving adequate resources for the host operating system as well. In all cases, unnecessary services were disabled in order to maximize the guest's resources.

Each system was tested for network performance using Netperf [26], as well as file system-read/re-read and file system-write/re-write performance using IOZone [25]. These tests serve as microbenchmarks, and will prove useful (particularly the network benchmarks) in analyzing the scalability and performance of the distributed benchmarks. Our primary computational benchmarks are the NAS Parallel Benchmark suite [4] and the High Performance LINPACK (HPL) benchmark [13]. We test both serial, parallel (OpenMP), and MPI versions of the NPB kernels. All guests are instantiated with a standard install, and all performance measurements were obtained with "out-of-the-box" installations. The LAM MPI implementation was used for all MPI performance analysis.



(a) Throughput (b) Latency

Fig. 1 Our network performance comparison. Xen closely matches the native bandwidth performance, while OpenVZ demonstrates nearly native latency. VMWare Server suffers in both bandwidth and latency.

4.1 Network Performance

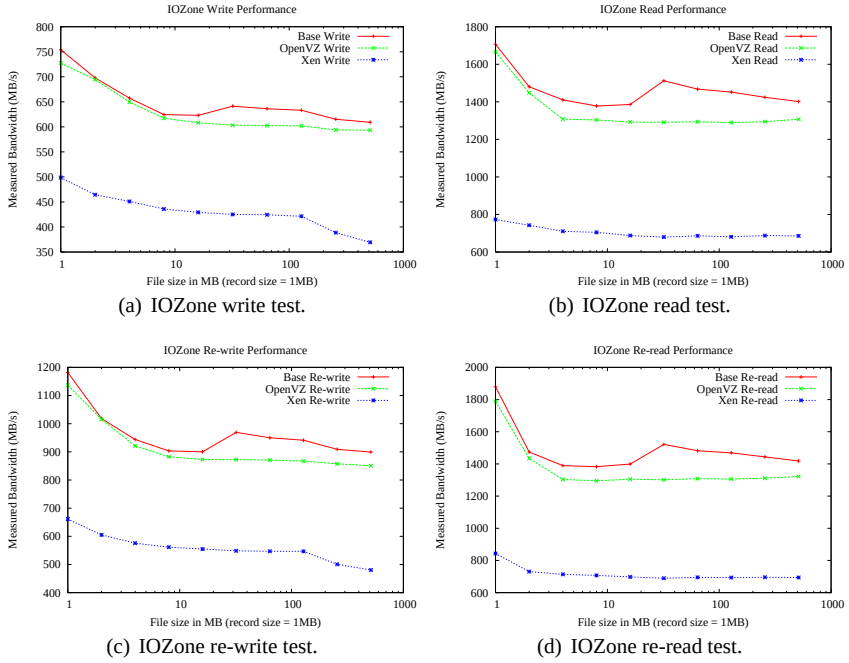
Using the Netperf [26] network benchmark tool, we tested the network characteristics of each virtualization strategy and compared it against the native results. All tests were performed multiple times and their results were averaged. We measured latency using Netperf's TCP Request/Response test with increasing message sizes. The latency shown is the half-roundtrip latency.

In Figure 1 we present a comparison of two key network performance metrics: throughput and latency. Examining Figure 1(a) we see that Xen clearly outperforms both OpenVZ and VMWare Server in network bandwidth and is able to utilize 94.5% of the network bandwidth (compared to the base/native bandwidth). OpenVZ and VMWare Server, however, are able to achieve only 35.3% and 25.9%, respectively, of the native bandwidth.

Examining Figure 1(b), however, tells a different story. While Xen was able to achieve near-native performance in bulk data transfer, it demonstrates exceptionally high latency. OpenVZ, however, closely matches the base latency with an average 1-byte one-way latency of 84.62 μs compared to the base latency of 80.0 μs. This represents a difference of only 5.8%. Xen, however, exhibits a 1-byte one-way latency of 159.89 μs, approximately twice that of the base measurement. This tells us that, while Xen may perform exceptionally well in applications that move large amounts of bulk data, it is unlikely to outperform OpenVZ on applications that require low-latency network traffic.

4.2 File System Performance

We tested each guest's file system using the IOZone [25] file system benchmark using files of varying size ranging from 64 KB to 512 MB, and record sizes from 4 KB to 16 MB. For ease of interpretation, we fix the record size at 1 MB for the graphs



(a) IOZone write test. (b) IOZone read test.

(c) IOZone re-write test. (d) IOZone re-read test.

Fig. 2 IOZone file system performance. OpenVZ closely follows the native performance, but suffers in the caching effect. Xen also misses the caching effect, but exhibits approximately half the performance of the base system.

shown in Figure 2. While in each case, the full complement of IOZone tests were run, space does not allow us to show all results. We chose the read, write, re-read, and re-write file operations as our representative benchmarks as they accurately demonstrated the types of overhead found within each virtualization technology. The read test measures the performance of reading an existing file and the write test measures the performance of writing a new file. The re-read test measures the performance of reading a recently read file while the re-write test measures the performance of writing to a file that already exists. All of our tests IOZone tests are effectively measuring the caching and buffering performance of each system, rather than the spindle speed of the disks. This is intentional, as we sought to measure the overhead introduced by each virtualization technology, rather than the disk performance itself. We omit the results of the VMWare Server IOZone tests as incompatibilities with the serial ATA disk controller required the use of file-backed virtual disks rather than LVM-backed or partition-backed virtual disks. It is well known that file-backed virtual disks suffer from exceptionally poor performance.

In Figure 2 one can immediately see a consistant trend in that OpenVZ and the base system perform similarly while Xen exhibits major performance overheads in all cases. However, even OpenVZ demonstrates an important impact of virtualization in that the effect of the buffer cache is reduced or eliminated. The same is true for Xen. A non-virtualized system should exhibit two performance plateaus for file sizes

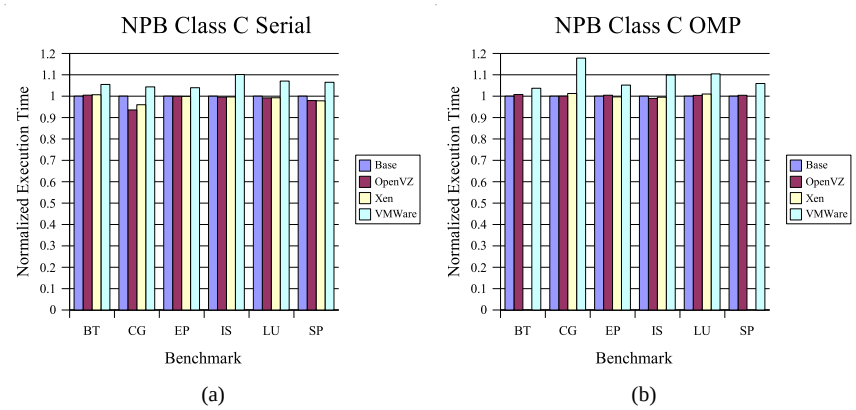


Fig. 3 Relative execution time of NPB serial and parallel (OpenMP) benchmarks. In (a) all three virtualization solutions perform similarly, with OpenVZ and Xen exhibiting near-native performance, while in (b) VMWare Server shows a decrease in performance for OpenMP tests, while OpenVZ and Xen remain near-native.

less than the system's memory. The first is the CPU cache effect, which, to a degree, all three systems exhibit. File sizes that fit entirely within the processor's cache will exhibit a sharp spike in performance, but will decline rapidly until the second plateau representing the buffer cache effect. The base system is the only system to exhibit the proper performance improvement for files fitting into the system's buffer cache.

Nevertheless, as we show in Figures 2(a) and 2(b), OpenVZ achieves reasonable file system performance when compared to the base system. The OpenVZ guests reside within the file system of the host as a directory within the host's file system. Consequently, the overhead of virtualization is minimal, particularly when compared to Xen. The results show that OpenVZ has low overhead resulting in high performance file system operations.

Xen, which uses Logical Volume Management (LVM) for guest storage, exhibits lower performance than either OpenVZ or the base system. The read performance of Xen, shown in Figure 2(b) ranges from 686-772 MB/s, and is less than half of the performance of the base system which which peeks at 1706 MB/s (see Figure 2(b)). Similar results are seen for the write, re-write, and re-read tests.

4.3 Single Node Benchmarks

While our primary objective is to test the performance and scalability of VMWare Server, Xen, and OpenVZ for distributed HPC applications we first show the baseline performance of NAS Parallel Benchmarks [4] on a single node using both the serial and OpenMP benchmarks from NPB 3.2. Some of the benchmarks (namely MG and FT) were excluded due to their memory requirements.

The results of the serial and parallel NPB tests are shown in Figure 3. We normalize the result of each test to a fraction of the native performance in order to maintain a consistent scale between benchmarks with differing run times. In Figure 3(a) we

see that the class C serial results nearly match the baseline native performance. Even the fully-virtualized VMWare Server demonstrates performance that is consistently within 10% of the normalized native run time.

The most problematic benchmark for VMWare Server, as shown in Figures 3(a), is the IS (integer sort) kernel. Indeed the IS kernel is the only benchmark that exhibits a relative execution time that is more than 10% slower than the native time. Because of the normalized execution times shown in Figure 3 the actual time component of the benchmark is removed. However, IS exhibits an exceptionally short run time for the class C problem size. Thus, the small amount of overhead is magnified due to the short run times of the IS benchmark.

However, we see no meaningful performance penalty in using either Xen or OpenVZ. Even the IS kernel exhibits near-native performance. This suggests that the CPU-bound overhead of both paravirtualization and operating system-level virtualization is quite insignificant. Indeed, in several cases we see a slight performance boost over the native execution time. These slight performance improvements have previously been shown to occur, and may be the result of the differing kernel versions between the base and guest systems.

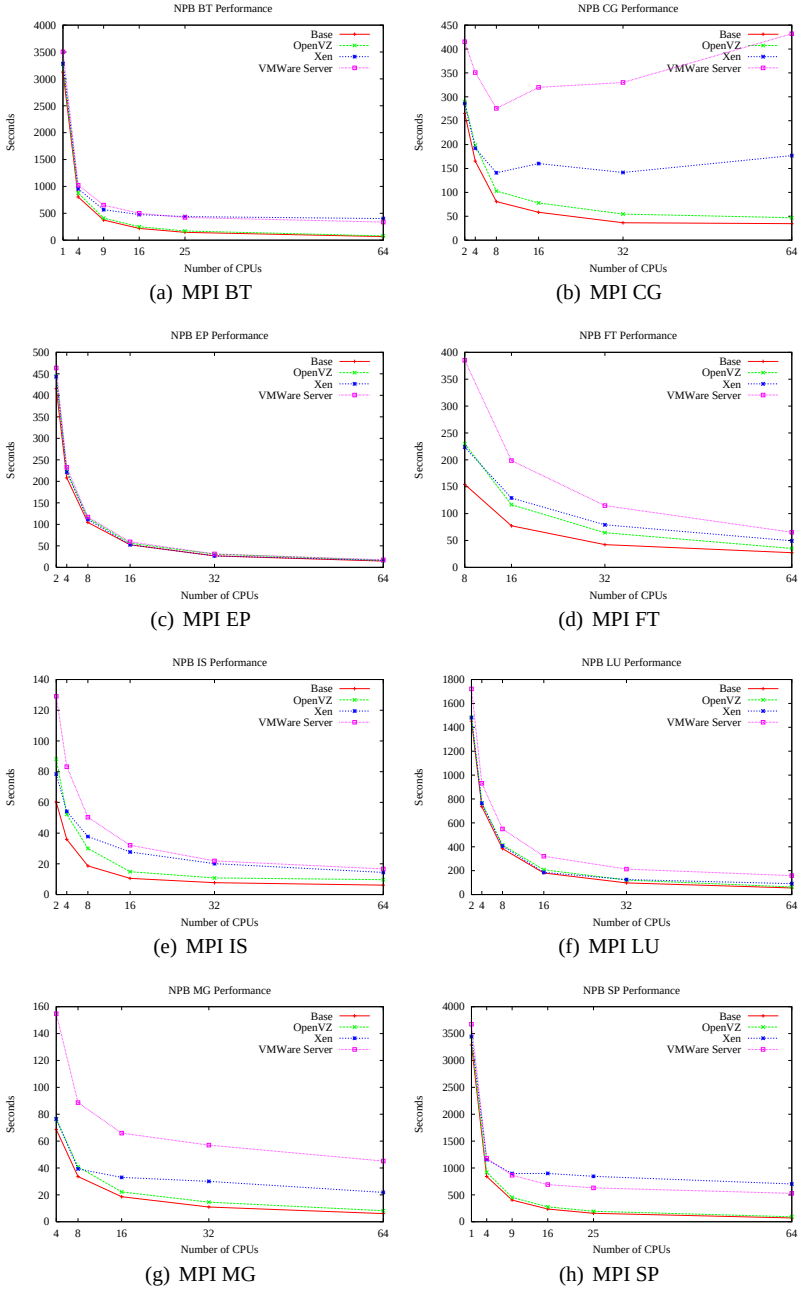
In Figure 3(b) we show the relative execution time of the OpenMP implementations of NPB. This time, however, we found that Xen was unable to execute both the BT and SP benchmarks. As a consequence, we omit Xen's results for non-working benchmarks.

In general we see from Figure 3(b) that the relative performance of the OpenMP benchmarks is on-par with that of the native SMP performance, especially in the cases of Xen and OpenVZ. Similar to Figure 3(a) we see that both OpenVZ and Xen perform at native speeds, further suggesting that the overhead of both paravirtualization and operating system-level virtualization remains low even for parallel tasks. Indeed, for both OpenVZ and Xen, no benchmarks exhibit a relative execution time that is more than 1% slower than the native execution time.

VMWare Server, however, exhibits greater SMP overhead than the serial benchmarks. Further, the number of benchmarks with runtimes of over 10% greater than the base time has also increased. Whereas the serial benchmarks see only IS exhibiting such a decrease in performance, three benchmarks (IS, LU, and CG) exhibit a decrease in performance of 10% or greater in the OpenMP benchmarks.

4.4 MPI Benchmarks

In Figure 4 we present the results of our MPI benchmark analysis, again using the Class C problem sizes of the NPB. We test each benchmark with up to 64 nodes (using 1 process per node). Unlike the serial and parallel/OpenMP results, it is clear from the outset that both VMWare Server and Xen suffer from a serious performance bottleneck, particularly in terms of scalability. Indeed, both VMWare Server and Xen exhibited exceptionally poor processor utilization as the number of nodes increased. In general, however, both Xen and VMWare Server were able to utilize, to some extent, the available processors to improve the overall run time with three notable exceptions: BT, CG, and SP.



(a) MPI BT (b) MPI CG

(c) MPI EP (d) MPI FT

(e) MPI IS (f) MPI LU

(g) MPI MG (h) MPI SP

Fig. 4 Performance of the NPB MPI tests. Strictly CPU-bound tests, such as EP exhibit near-native performance for all guests. Other benchmarks show OpenVZ exhibiting performance closest to native, while Xen and VMWare server suffer due to network overhead.

Figure 4 suggests that the greatest overhead experienced by the guest operating systems is related to network utilization. For example, in Figure 4(c) we show the results of the "embarrassingly parallel" kernel, EP. EP requires a minimum of network interaction, and as a consequence we see near-native performance for all virtualization technologies, including VMWare Server.

For benchmarks that make greater use of the network, however, the results are quite different. In fact, rather than closely following the results of OpenVZ and the base system, Xen now more accurately groups with VMWare Server. This is particularly true with regards to BT, CG, and SP, three of the most poorly performing benchmarks. OpenVZ, however, largely follows the performance of the base system, particularly for the longer running computational benchmarks. Even the more network-heavy benchmarks, such as BT and SP, achieve near-native performance despite OpenVZ's bandwidth results shown in Figure 1(a). Unlike Xen, however, OpenVZ demonstrated near-native latencies (Figure 1(b)) which we believe is the primary reason for OpenVZ's scalability.

Both BT and SP are considered "mini applications" within the NAS Parallel Benchmark suite. They are both CFD applications with similar structure. While they are not considered network-bound, they are responsible for generating the greatest amount of network traffic (SP followed by BT) as shown by Wong, et al. [43]. We believe the primary reason for the poor performance of these benchmarks is the exceptionally high latencies exhibited by Xen and VMWare Server (Figure 1(b)). Unlike CG, however, the modest performance improvement demonstrated with BT and SP is likely due to a small amount of overlap in communication and computation that is able to mask the high latencies to a limited extent.

While the BT and SP benchmarks demonstrated poor performance, the CG benchmark was unique in that it demonstrated decreasing performance on both Xen and VMWare Server. This is likely due to the CG benchmark requiring the use of blocking sends (matched with non-blocking receives). Because of the exceptionally high penalty that Xen and VMWare Server observe in latency, it comes as no surprise that their blocking behavior severely impacts their overall performance. Indeed, the single byte "ping-pong" latency test shows a difference of nearly 80 μs between Xen and the base system, while a difference of only 4 μs was observed between the base system and OpenVZ. VMWare Server exhibited a latency over 3.5x larger than the base system. This suggests that for the NPB kernels, latency has a greater impact on scalability and performance than bandwidth as we see a corresponding decrease in benchmark performance with the increase in latency.

In Figure 5 we show the results of our HPL benchmarks. Again, we see the effect of the high latencies on the benchmark performance. At 64 nodes, for example, Xen is able to achieve only 57% of the performance (Gflops) of the base system while OpenVZ achieves over 90% of the base system performance. VMWare Server, suffering from both exceptionally high latencies and low bandwidth, is able to achieve only 25% of the base performance. We believe that an improvement in the guest bandwidth within OpenVZ guests would further improve the performance of OpenVZ to nearly match the native performance.

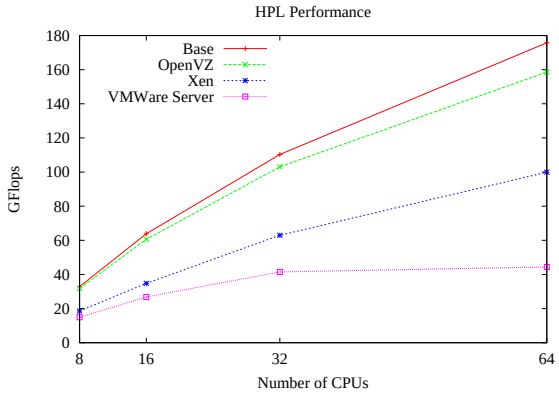


Fig. 5 OpenVZ performs closest to the native system, while Xen and VMWare Server exhibit decreased scalability.

5 Checkpointing/Restart System Design

In Section 4 we showed that virtualization is a viable choice for HPC clusters. However, in order for large scientific research to be carried out on virtualized clusters, some form of fault tolerance/checkpointing must be present. Building on our previous work in MPI checkpointing [40, 41] we propose a checkpointing solution based on OpenVZ [36], an operating system level virtualization solution. We assume that failures follow the stopping model; that is, one or more nodes crashes or otherwise stops sending or receiving messages. We then reduce the overhead of checkpointing by eliminating the SAN or other network storage as a checkpointing bottleneck. To do so, we leverage OpenVZ's existing checkpoint/restart mechanism, and enhance its utility through a checkpoint-enabled LAM/MPI implementation and a lightweight checkpoint/replication daemon, *Ovzd*. All checkpoints are stored on a node's local disk in order to eliminate any reliance on network storage. This, however, requires the use of replication in order to tolerate node failures as a checkpoint stored only to a single node's local disk will be unavailable in the event of a crash. Our system allows OpenVZ's virtual private servers (VPS) to initiate system checkpoints and to replicate those checkpoints to additional peer machines for added fault resiliency. In the sections to follow, we describe the implementation of our OpenVZ MPI-enabled checkpointing solution and demonstrate its performance using the standard NPB benchmarks.

5.1 System Startup

In order to properly facilitate a checkpoint/restart and fault-tolerance mechanism in OpenVZ we implemented a checkpointing daemon, *Ovzd*, that is responsible for taking the actual checkpoint/snapshot of the running computation and file system. *Ovzd* runs as a single instance on the host system and acts as a relay between a virtual private server (VPS) and the checkpoint/restart mechanism built into OpenVZ.

Ovzd also adds file system checkpointing and replication to the VPS to enable fault-resilience in case of node failures and to maintain a consistent file system for VPS restarts.

Upon starting a VPS, *Ovzd* overlays a FIFO into the VPS' file system in order to facilitate communication between the VPS and the host node/*Ovzd*. Through this FIFO, our checkpoint-enabled MPI implementation is able to signal *Ovzd* when a checkpoint is desired. Once the new VPS has initialized, *Ovzd* immediately begins to checkpoint the file system of the newly connected VPS. This is done while the VPS continues to run and serves as the baseline file system image for future incremental checkpoints. We use *tar* with incremental backup options to save the file system image to local disk. During the creation of the base file system image, checkpointing is suppressed in order to guarantee a consistent file system image. Once all participating *Ovzds* have created the base file system image, checkpointing is re-enabled.

5.2 Checkpointing

We are not the first group to implement checkpointing within the LAM/MPI system. Basic checkpointing support was added directly to the LAM/MPI implementation by Sankaran et al [30]. Because of the previous work in LAM/MPI checkpointing, the basic checkpointing/restart building blocks were already present within LAM's source code. This provided an ideal environment for testing our virtualization and replication strategies.

MPI checkpointing in OpenVZ is a multi-step process that begins within the VPS. Our checkpoint-enabled MPI is based on the LAM/MPI implementation with modifications to its existing checkpointing support. The major differences between our OpenVZ-based implementation and the basic checkpointing support already included within LAM are in the manner in which checkpoints are taken, and how those checkpoints are stored (discussed in Section 5.4). To perform checkpoints, we no longer rely on the use of the BLCR checkpointing module [14]. Instead we provide an OpenVZ-aware LAM implementation that coordinates with the checkpointing features of the OpenVZ-based kernel. This coordination is performed through the *Ovzd* described above.

The protocol begins when *mpirun* instructs each LAM daemon (*lamd*) to checkpoint its MPI processes. When a checkpoint signal is delivered to an MPI process, each process exchanges bookmark information with all other MPI processes. This process is known as quiescing the network, and is already provided by LAM and reused in our implementation. These bookmarks contain the number of bytes sent to/received from every other MPI process. With this information, any in-flight messages can be waited on and received before the *Ovzd* performs the checkpoint. This is critical in order to maintain a consistent distributed state upon restart.

Once all messages have been accounted for, the checkpointing of the VPS memory footprint can begin. To do so, the lowest ranking MPI process within each VPS writes a checkpoint message to its FIFO instructing the *Ovzd* to perform a checkpoint. Upon receiving a checkpoint signal, each *Ovzd* performs the following actions:

1. *Ovzd* momentarily suspends its VPS to prepare for a checkpoint.

2. The VPS' memory image is saved to local storage.
3. The VPS' file system is incrementally saved using the baseline image to minimize the number of saved files.
4. *Ovzd* then resumes the VPS to continue the computation.

Because the majority of the file system is saved prior to the first checkpoint being delivered, the primary overhead in checkpointing the VPS is in step 2 (saving the memory image) above.

5.3 Restarting

One of the advantages of using virtualization is that a virtual server/virtual machine is able to maintain its network address on any machine. With this functionality, restarting an MPI computation is greatly simplified because MPI jobs need not update their address caches, nor must they update any process IDs. Because the entire operating system instance was checkpointed, MPI sees the entire operating system exactly as it was prior to checkpointing.

The disadvantage, however, is that a great deal of data must be restored before the VPS can be reinitialized and restored. To handle the mechanics of restarting the computation on multiple nodes, we developed a set of user tools that can be used to rapidly restart the computation of many nodes simultaneously. Using our restore module, a user simply inputs a list of nodes/VPS IDs as well as the desired checkpoint to restore. The VPS list functions in almost the same manner as an MPI machine list, and performs the following:

1. Connect to each host machine in the VPS list.
2. Remove any instance of the to-be-restored VPS from the host machine.
3. Restore the file system, including all increments, to maintain a consistent file system.
4. Reload the memory image of the VPS only, do not continue computation.
5. Once all participating VPS images have been reloaded, resume computation.

Because the most time consuming portion of the recovery algorithm is the file system restoration, we designed the recovery tool to perform items 1-4 concurrently on all participating host nodes. This allows the recovery time to be reduced primarily to the time of the slowest host node. Because the virtualized network subsystem is not restarted until the VPS image is restarted, we prevent any node from resuming computation until all nodes are first reloaded. Once all nodes have indicated the completion of step 4, each VPS can be resumed without any loss of messages.

5.4 Data Resiliency to Node Failures

If checkpoints are saved only to the host node's local disk, computations will be lost due to a node failure. Common strategies for preventing the loss of data include saving to network storage and dedicated checkpoint servers. However, virtual

servers/virtual machines present additional problems to checkpointing directly to network storage or dedicated servers. In particular, checkpointing a virtual server may result in considerably larger checkpoints due to the need to checkpoint the file system of a virtual server. While this overhead can, to some extent, be mitigated by the use of incremental file system checkpoints (as we describe), large differences in the file system will still result in large checkpoints.

Our solution is to use a replication system in order to replicate checkpoint data throughout the participating cluster. Upon startup, each *Ovzd* is given a series of randomly chosen addresses (within the participating cluster) that will function as replica nodes. The user indicates the degree of replication (number of replicas), and each checkpoint is replicated to the user-defined number of nodes. The replication itself is performed after a checkpoint completes and after the VPS has been resumed. All replication is therefore performed concurrently with the VPS computation. This reduces the impact of checkpointing on shared resources such as network storage, and also reduces the impact of checkpointing on the computation itself by propagating checkpoints while computation continues. Further, by spreading the cost of checkpointing over all nodes participating in the computation, no individual network links become saturated, such as in the case of dedicated checkpoint servers. As we show in [41] we are able to drastically reduce the overhead of checkpointing even the largest computations in a scalable fashion. This is crucial for the large amount of data that may be generated by checkpointing virtual servers.

5.5 The Degree of Replication

While the replication strategy that we have described has clear advantages in terms of reducing the overhead on a running application, an important question that remains is the number of replicas necessary to reliably restart a computation. In Table 1 we present simulated data representing the number of allowed node failures with a probability of restart at 90, 99, and 99.9%. We simulate with up to 3 replicas for each cluster size. The data are generated from a simulator developed in-house to simulate a user-defined number of failures with a given number of replicas and to compute whether a restart is possible with the remaining nodes. We define a successful restart as one in which at least one replica of each virtual server exists somewhere within the remaining nodes.

From Table 1 we observe that a high probability of restart can be achieved with seemingly few replicas. More important, however, is that the effectiveness of our replication strategy is *ideal* for large clusters. Indeed, unlike the network storage and centralized server approach commonly used [41], checkpoint replication scales well with the size of the cluster (replication overhead is discussed in Section 7.1). Put differently, as the size of the cluster increases, our replication strategy can probabilistically tolerate greater node failures with fewer replicas. Such scalability is a requirement as clusters increase to thousands or hundreds of thousands of nodes.

Table 1 Probability of successful restart with 1-3 replicas.

	1 Replica			2 Replicas			3 Replicas		
	Allowed Failures for			Allowed Failures for			Allowed Failures for		
Nodes	90%	99%	99.9%	90%	99%	99.9%	90%	99%	99.9%
8	1	1	1	2	2	2	3	3	3
16	1	1	1	2	2	2	5	4	3
32	2	1	1	5	3	2	8	5	4
64	3	1	1	8	4	2	14	8	4
128	4	1	1	12	6	3	22	13	8
256	5	2	1	19	9	5	37	21	13
512	7	2	1	31	14	7	62	35	20
1024	10	3	1	48	22	11	104	58	33
2048	15	5	2	76	35	17	174	97	55

6 Checkpoint/Replication Analysis

Before presenting our numerical results we begin with a more general theoretical analysis of the overheads involved in both checkpointing and replicating a computation. This will allow us to more accurately reason with regards to the actual data collected in our studies, particularly the replication data. We begin with a general description of the distribution of time (in terms of the total run time of the application) in a distributed system. Let:

t_{comp} = Portion of the computation's running time spent computing
t_{comm} = Portion of the computation's running time spent communicating
n = Number of checkpoints taken
t_{coord} = Coordination (pre-checkpointing) time
t_{write} = Time to serialize memory and write checkpoints to disk
t_{cont} = Time to resynchronize nodes post-checkpoint
t_{tot} = The total run time of the distributed computation
α = Impact of replication on communication time

We emphasize that each of the above values are *per node*. In the absence of checkpointing we can then approximate the running time of a single node of an MPI application as:

$$t_{tot} = t_{comp} + t_{comm}$$

That is, we approximate the total running time of the distributed computation as the sum of the time spent both communicating and computing. If we then allow for checkpointing (without replication) we can approximate the total running time with:

$$t_{tot} = t_{comp} + t_{comm} + n(t_{coord} + t_{write} + t_{cont}) \quad (1)$$

For simplicity, we assume that from one checkpoint to another t_{coord}, t_{write}, and t_{cont} remain constant. The overhead of Equation 1 can be most accurately characterized

within the quantity $n(t_{coord} + t_{write} + t_{cont})$. Typically, the quantity t_{write} dominates the overhead of checkpointing solutions, particularly periodic solutions such as we describe here. Thus, applications with large memory footprints will almost always experience greater checkpointing overheads than applications which consume less memory. In cases where a large amount of memory is written to disk the choice of n, the number of checkpoints taken (directly related to the time between checkpoints), should be smaller (meaning more time between checkpoints) than an application that consumes less memory. However, in some situations it is possible for t_{coord} to dominate the checkpointing overhead. This could happen with particularly large distributed systems that consume relatively small amounts of memory. Indeed, the memory footprint (per process) of NPB reduces quite quickly for each doubling of nodes (see Table 2). Clearly then, it is important to carefully select the number of checkpoints with both the memory footprint of each node as well as the overall size of the cluster in mind.

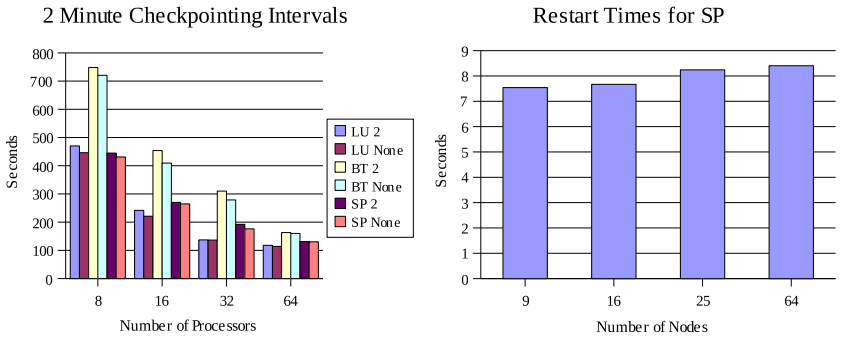
In Section 5.4 we described our replication system that we use to both increase a computation's resiliency to node failure as well as reduce contention on shared network storage. Starting with Equation 1, we can now approximate the impact of replication on the per-node running time of the application:

$$t_{tot} = t_{comp} + \alpha t_{comm} + n(t_{coord} + t_{write} + t_{cont}) \quad (2)$$

The variable α represents the impact of replication on the communication component of the application. Ultimately, α is directly related to the value of t_{write} where a larger memory footprint would reasonably result in a larger impact on the communication portion of the computation. However, from Equation 2 we can see that the impact of checkpointing with replication is not only dependent on the size of the memory footprint, but also on the individual communication characteristics of the application being checkpointed. This allows us to reason as to why an application with a small memory footprint might experience a greater checkpointing overhead than one with a larger memory footprint. For example, a computation using only a small number of nodes with a large memory footprint (t_{write}) may experience a greater overhead than a computation with a greater communication (t_{comm}) component but smaller memory footprint. However, as the number of nodes increases and the memory footprint (per node) decreases, the overhead may shift towards the computation with the greater communication component. We will see an example of this in Section 7.1.

7 Performance Results

In order to demonstrate the performance of our implementation, we used the NAS Parallel Benchmarks [43] with up to 64 nodes. The NPB contains a combination of computational kernels and "mini applications." For our analysis, we choose to use the "mini applications" LU, SP, and BT. Again, all tests were conducted on nodes from the University at Buffalo's Center for Computational Research (CCR), whose characteristics are discussed Section 4. For these tests, each VPS was allocated 1.8 GB



(a) OpenVZ checkpointing with 2 minutes intervals.

(b) Time to restart from a checkpoint.

Fig. 6 Base checkpoint and restart performance. Checkpoints are saved directly to local storage, but replication is not used.

RAM, full access to a single processor, and a CentOS 4.4 instance of approximately 270 MB (compressed to approximately 144 MB). The CentOS operating system instance was created from a standard OpenVZ template. In order to demonstrate the effectiveness of checkpointing the file system, no extraordinary measures were taken to reduce the CentOS image size. We used the most recent version of the OpenVZ testing kernel (2.6.18-ovz028test015.1) for all tests. The operating system memory footprints for each of the benchmarks is listed in Table 2.

Table 2 Checkpoint sizes for varying cluster sizes.

	8	16	32	64
LU	106 MB	57 MB	34 MB	21 MB
BT	477 MB	270 MB	176 MB	77 MB
SP	180 MB	107 MB	75 MB	38 MB

In Figure 6(a) we present the timings for our basic checkpointing implementation. All checkpoints are written directly to local disk in order to minimize the time needed for writing the checkpoint file to stable storage. We checkpoint each system at 2 minute intervals in order to gain a sense of the overhead involved in checkpointing an entire virtual server. Comparing the checkpointing times to the non-checkpointing times, we see that the overhead remains low, with a maximum overhead of 11.29% for the BT benchmark at 32 nodes. From Table 2 we can see that the BT benchmark generates checkpoint files that are consistently larger than both the LU and SP benchmarks. Thus, we expect to observe greater overheads with BT's larger checkpoints.

In Figure 6(b) we show the time needed to restore a computation for the SP benchmark with up to 64 nodes. In the case of restarting, the dominating factor is the time needed to restore the VPS' file system to its checkpointed state. Since we do this concurrently on all nodes, the time needed to restore a computation is approximately equal to the time needed to restore the computation to the slowest member of the

group. Because restoring a computation must be done by first restoring all file system checkpoints (in order) followed immediately by a coordinated reloading of all VPS memory checkpoints, a slowdown in either the memory restore phase or the file system restore phase of a single VPS will slow down the restore process for the entire cluster.

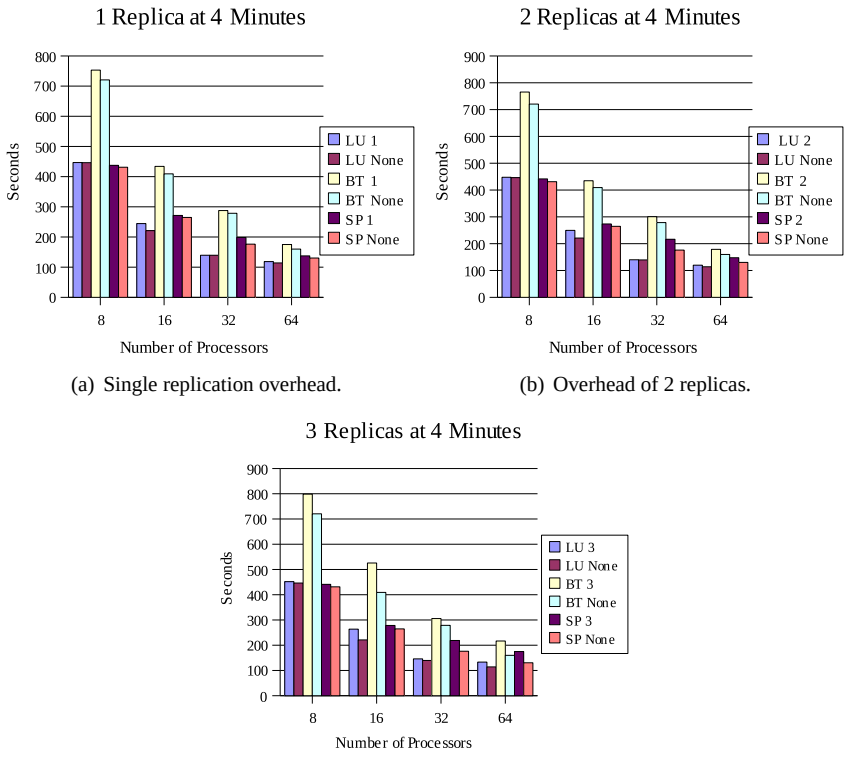
7.1 Replication Overhead

In order to increase the resiliency of our checkpointing scheme to node failures, we also include checkpoint replication. This allows us to eliminate any reliance on network storage while also increasing the survivability of the application. In this section we demonstrate the overhead of our approach with 1-3 replicas, and up to 64 nodes per computation. Each replication consists of both replicating the memory checkpoint as well as any incremental file system changes. In our experiments, the typical file system incremental checkpoints amount to less than 0.5 MB, thus contributing only a very small amount of overhead to the replication system. Each experiment assumes that checkpointing occurs at 4 minute intervals. When the entire computation is completed in fewer than 4 minutes, a single checkpoint is taken at the midpoint of the computation.

In Figure 7(a) we present the results of replicating each checkpoint to exactly one additional node. As can be seen, the BT benchmark consistently results in greater overhead than either the LU or SP benchmarks. This is particularly true with smaller cluster sizes, where the checkpoint sizes are larger (from Table 2). Nevertheless, with only a single replica being inserted into the network, the overhead due to the replication remains quite low.

In Figures 7(b) and 7(c) we present the results of replicating each checkpoint to two and three nodes, respectively. As can be seen, the computations suffer from only a minimal amount of additional overhead as the extra replications are used, with overheads as low as 2.3% for the case of the SP 8 node benchmark. As in the case of the single replica, the overhead does increase with the size of the cluster. However, we would expect to see a reduction in overhead for the larger cluster sizes with longer running computations as well as more reasonable checkpointing intervals. Because the larger cluster sizes lasted less than 4 minutes, the results show a disproportionately high overhead.

We note, however, that the impact of replication on the overall run time of the benchmark depends not only on the size of the checkpoints, but also on the benchmark's communication characteristics (recall Section 6). Wong et al. have previously characterized the scalability and communications characteristics of NPB [43]. While the BT benchmark may exhibit the largest memory footprint, the SP benchmark performs (by far) the most communication. Similarly, the LU benchmark performs the least amount of communication. Their results are for 4 CPU implementations of the class A problem sizes. More precisely, they report that the BT benchmark is responsible for a total of 1072 MB of communication data, while the SP benchmark is responsible for 1876 MB, and the LU benchmark communicates only 459 MB of data.



(a) Single replication overhead.

(b) Overhead of 2 replicas.

(c) Overhead of 3 replicas.

Fig. 7 Performance of checkpointing class C NPB with 1-3 replicas. Checkpoints are first saved to local storage. Once computation resumes, *Ovzd* performs the replication.

Of course, the class C benchmarks that we tested would result in considerably greater amounts of communication. Nevertheless, we can see a trend when we examine both the replication overhead along with the size of the checkpoints. Using the analysis developed in Section 6 we see that for small cluster sizes, particularly 8 or 9 nodes, the BT benchmark exhibits the greatest overhead for any replication level. This is due to the additional communication generated by BT's replicas that is concentrated on only 9 nodes. Thus, a greater amount of communication overhead is placed on fewer nodes. This results in an increased α (per node) contributing to a larger t_{comm} component from Equation 2. In fact, the replication of the 9 node BT benchmark adds between 1.4 GB and 4.3 GB of data (depending on the number of replicas) to the system.

However, as the number of nodes increases, the size of the checkpoints decrease, as does the total running time of the computation. This distributes the α component of Equation 2 over a greater number of nodes and reduces the number of checkpoints as shown in Figure 7. This is particularly evident in the case of the SP benchmark which initially starts with relatively little overhead, particularly compared with BT. Indeed, the overhead of replicating the SP benchmark increases much more rapidly

(with increasing cluster sizes) than either the BT or LU benchmark. At 64 nodes, for example, the overhead of replicating the SP benchmark is nearly identical to that of the BT benchmark (approximately 35%) despite checkpoint sizes that differ by a factor of two (Table 2). As would be expected, LU consistently exhibits much lower replication overhead than either BT or SP, suggesting that its smaller communication volume is less affected by the replication overhead.

8 Related Work

There have been many descriptions of virtualization strategies in the literature, including performance enhancements designed to reduce their overhead. Xen, in particular, has undergone considerable analysis [12]. In Huang, et al. [22] the idea of VMM bypass I/O is discussed. VMM bypass I/O is used to reduce the overhead of network communication with Infiniband network interfaces. Using VMM bypass I/O it was shown that Xen is capable of near-native bandwidth and latency, which resulted in exceptionally low overhead for both NPB and HPL benchmarks using the MVAPICH Infiniband-based MPI implementation. However, the VMM bypass I/O currently breaks Xen's checkpointing functionality. Raj et al. describe network processor-based self-virtualizing network interfaces that can be used to minimize the overhead of network communication [28]. Their work reduces the network latency by up to 50%, but requires specialized network interface cards. Menon et al. provide an analysis of Xen and introduce the Xenoprof [23] tool for Xen VM analysis. Using the Xenoprof tool, they demonstrated Xen's key weaknesses in the area of network overhead.

Emeneker et al. compare both Xen and User-Mode Linux (UML) for cluster peformance [16]. Their goal was to compare both a paravirtualized virtualization implementation (Xen) against an operating system-level virtualization package (UML). They showed that Xen clearly outperforms UML in terms of performance, reliability, and the impact of the virtualization technologies.

In Soltesz, et al. [32] a comparison of Xen and Linux-Vserver is discussed with special attention paid to availability, security and resource management. They show that container-based/operating system-based virtualization is particularly well suited for environments where resource guarantees along with minimal overhead are needed. However, no existing work has adequately examined the varying virtualization strategies for their use in HPC environments, particularly with regard to scalability. Our evaluation and experimentation fills this gap and provides a better understanding of the use of virtualization for cluster computing.

VMWare ESX Server has been studied in regards to the architecture, memory management and the overheads of I/O processing [2, 11, 31, 38]. Because VMWare infrastructure products implement a full virtualization technology, they have detrimental impact on the performance even though they provide easy support for unmodified operating systems.

Checkpointing at both the user-level and kernel-level has been extensively studied [14, 27]. Often such implementations are applicable only to single-process or multi-threaded checkpointing. Checkpointing a distributed system requires additional

considerations, including in-flight messages, sockets, and open files. Gropp, et al. provide a high-level overview of the challenges and strategies used in checkpointing MPI applications [20]. The official LAM/MPI implementation includes support for checkpointing using the Berkeley Linux Checkpoint/Restart (BLCR) kernel-level checkpointing library [30]. A more recent implementation by Zhang et al. duplicates the functionality of LAM's kernel-level checkpointer, but implements checkpointing at the user-level [45].

The most widely published application-level checkpointing system for MPI programs is the C^3 system by Bronevetsky et al. [9]. C^3 uses a non-blocking coordinated checkpointing protocol for programs written in the *C* programming language. Because it provides checkpointing at the application level, a pre-processor/pre-compiler is used to first transform a user's source code into checkpointable code. This allows for platform independence in that a checkpointing engine need not be created specifically for each architecture, and in many cases allows for checkpointing and migration within heterogeneous cluster architectures [7]. However, application-level checkpointing also requires more effort on the part of the programmer, where checkpointing primitives must be inserted manually.

MPICH-V [8] uses an uncoordinated message logging strategy with checkpointing provided by the Condor checkpointing library [21]. The advantage to using a message logging strategy is that nodes participating in the computation may checkpoint independently of one another. Further, upon failure, only the failed node is restarted. Messages sent to the failed node between the time of the last checkpoint and the node's failure (and subsequent restart) are replayed from stable storage while the unaffected nodes continue their computation. However, the overhead of capturing and storing all of an application's messages results in additional overhead.

Regardless of the level at which checkpointing is performed, there is typically an implicit assumption regarding the reliability of message passing in distributed systems. Graham et al. provide a detailed discription of the problem and have introduced LA-MPI in order to take advantage of network-based fault-tolerance [19]. Network fault-tolerance solutions, such as LA-MPI, complement our work.

In our previous work, in non-virtualized systems, we have shown that the overhead of checkpointing can be reduced dramatically by the use of local disk checkpointing with replication [40, 41]. Because virtual machines must maintain both a consistent memory state as well as a consistent file system, the amount of data that must be stored may increase dramatically. In this article we have shown that a similar replication strategy may be applied to virtualized compute nodes. Other strategies, such as the use of parallel file systems or NFS cluster delegation [6] may improve the overhead of checkpointing when compared to SANs. However, as we have shown in our previous work, even highly scalable commerical parallel file systems are easily overwhelmed by large-scale checkpointing.

Checkpointing within virtual environments has also been studied, though typically not for the use of HPC applications. OpenVZ [36], Xen [5], and VMWare [37] all provide mechanisms to checkpoint the memory footprint of a running virtual machine. However, to date, the use of these checkpointing mechanisms have been limited to the area of "live migration" due to the lack of complete file system check-

pointing and/or a lack of checkpoint/continue functionality. By supporting only live migration, the virtualization tools avoid file system consistency issues.

Another application is in the use of preemptive migration [24]. By integrating a monitor with Xen's live migration, Nagarajan et al. attempt to predict node failures and migrate computations away from failing nodes. However, such strategies are still susceptible to sudden and unexpected node failures. Further, the work by Nagarajan, et al. is incapable of rollback-recovery and instead relies on accurately predicting failures prior to their occurrence. Our work does not preclude such proactive migration, and would prove quite complementary.

Our work differs from the previous work in VM checkpointing and storage in that we enable periodic checkpointing and rollback recovery for MPI applications executing within an virtualized environment. This requires cooperation with the existing VM-level checkpointing support that is already provided by the VM/VPS. Moreover, we also include support for checkpointing incremental file system changes in order to provide rollback support. We have added local disk checkpointing with replication to both reduce the overhead of checkpointing as well as to improve checkpoint resiliency in the presence of multiple simultaneous node failures. Our checkpointing solution does not rely on the existence of network storage for checkpointing. The absence of network storage allows for improved scalability and also shorter checkpoint intervals (where desired) [40].

9 Conclusions and Future work

We have performed an analysis of the effect of virtualization on scientific benchmarks using VMWare Server, Xen, and OpenVZ. Our analysis shows that, while none match the performance of the base system perfectly, OpenVZ demonstrates low overhead and high performance in both file system performance and industry-standard scientific benchmarks. While Xen demonstrated excellent network bandwidth, its exceptionally high latency hindered its scalability. VMWare Server, while demonstrating reasonable CPU-bound performance, was similarly unable to cope with the MPI-based NPB benchmarks.

Drawing on these results we have shown that full checkpointing of OpenVZ-based virtualized servers can be accomplished at low-cost and near invisibility to the end user. We use both checkpointing and replication in order to ensure the lowest possible checkpointing overhead. A remaining issue that must be addressed is the integration of our checkpointing and fault-tolerance system into common cluster batch schedulers, such as PBS Pro or Torque. We have already begun work on a cluster framework that integrates our VM fault-tolerance with the commonly used Torque resource manager [39]. The goal is to extend our fault-tolerance work beyond failure management in order to enable better utilization of cluster resources

Acknowledgments

We would like to acknowledge the input of the anonymous reviewers whose suggestions have greatly improved the quality of this article. We also acknowledge Minsuk

Cha, Salvatore Guercio Jr, and Steve Gallo for their contributions to the virtual machine evaluations. Support was provided in part by NSF IGERT grant 9987598, the Institute for Scientific Computing at Wayne State University, MEDC/Michigan Life Science Corridor, and NYSTAR.

References

1. K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS-XII: Proceedings of the 12^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13. ACM Press, 2006.
2. I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija. An Analysis of Disk Performance in VMware ESX Server Virtual Machines. In *WWC '03: Proceedings of the 6^{th} International Workshop on Workload Characterization*, pages 65–76. IEEE Computer Society Press, 2003.
3. E. R. Altman, D. Kaeli, and Y. Sheffer. Guest Editors' Introduction: Welcome to the Opportunities of Binary Translation. *Computer*, 33(3):40–45, 2000.
4. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
5. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the 19^{th} Symposium on Operating Systems Principles*, pages 164–177. ACM Press, 2003.
6. A. Batsakis and R. Burns. NFS-CD: Write-Enabled Cooperative Caching in NFS. *IEEE Transactions on Parallel and Distributed Systems*, 19(3):323–333, 2008.
7. A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *J. Parallel Distrib. Comput.*, 43(2):147–155, 1997.
8. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SC '02: Proceedings of the 19^{th} annual Supercomputing Conference*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
9. G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-Level Checkpointing of MPI Programs. In *PPoPP '03: Proceedings of the 9^{th} Symposium on Principles and Practice of Parallel Programming*, pages 84–94. ACM Press, 2003.
10. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386. IEEE Computer Society Press, 1994.
11. L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *USENIX 2005 Annual Technical Conference, General Track*, pages 387–390. USENIX Association, 2005.
12. B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the Art of Repeated Research. In *USENIX Technical Conference FREENIX Track*, pages 135–144. USENIX Association, 2004.
13. J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.
14. J. Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Lab, 2002.
15. E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
16. W. Emeneker and D. Stanzione. HPC Cluster Readiness of Xen and User Mode Linux. In *CLUSTER '06: Proceedings of the International Conference on Cluster Computing*, pages 1–8. IEEE Computer Society Press, 2006.
17. The MPI Forum. MPI: A Message Passing Interface. In *SC '93: Proceedings of the 6^{th} annual Supercomputing Conference.*, pages 878–883. IEEE Computer Society Press, 1993.
18. R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.
19. R. L. Graham, S. E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A Network-Failure-Tolerant Message-Passing System for Terascale Clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.

20. W. D. Gropp and E. Lusk. Fault Tolerance in MPI Programs. *International Journal of High Performance Computer Applications*, 18(3):363–372, 2004.
21. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. Technical Report 1346,University of Wisconsin-Madison, 1997.
22. J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, pages 3–16. USENIX Association, 2006.
23. A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *VEE '05: Proceedings of the 1^{st} ACM/USENIX International Conference on Virtual Execution Environments*, pages 13–23. ACM Press, 2005.
24. A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *ICS '07: Proceedings of the 21^{st} annual International Conference on Supercomputing*, pages 23–32. ACM Press, 2007.
25. W. D. Norcott and D. Capps. The IOZone Filesystem Benchmark. `http://www.iozone.org`.
26. Hewlett Packard. Netperf. `http://www.netperf.org`.
27. J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing Under Unix. Technical Report UT-CS-94-242, 1994.
28. H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *HPDC '07: Proceedings of the International Symposium on High Performance Distributed Computing*, pages 179–188. IEEE Computer Society Press, 2007.
29. F. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide Area Cluster Monitoring with Ganglia. In *CLUSTER '03: The International Conference on Cluster Computing*, pages 289–298. IEEE Computer Society Press, 2003.
30. S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.
31. J. E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005.
32. Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.
33. L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, 1999.
34. J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings of the 10^{th} European PVM/MPI Users' Group Meeting, LNCS 2840*, pages 379–387. Springer-Verlag, 2003.
35. S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In *VEE '06: Proceedings of the 2^{nd} International Conference on Virtual Execution Environments*, pages 175–185. ACM Press, 2006.
36. SWSoft. OpenVZ - Server Virtualization, 2006. http://www.openvz.org/.
37. VMWare. VMWare, 2006. http://www.vmware.com.
38. C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.
39. J. P. Walters, B. Bantwal, and V. Chaudhary. Enabling Interactive Jobs in Virtualized Data Centers. In *CCA'08: The 1st Workshop on Cloud Computing and Its Applications*, `http://www.cca08.org/papers/Paper21-JohnPaul-Walters.pdf`, 2008.
40. J. P. Walters and V. Chaudhary. Replication-Based Fault-Tolerance for MPI Applications. *To appear in IEEE Transactions on Parallel and Distributed Systems*.
41. J. P. Walters and V. Chaudhary. A Scalable Asynchronous Replication-Based Strategy for Fault Tolerant MPI Applications. In *HiPC '07: the International Conference on High Performance Computing, LNCS 4873*, pages 257–268. Springer-Verlag, 2007.
42. A. Weiss. Computing in the Clouds. *netWorker*, 11(4):16–25, 2007.
43. F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *ICS '99: Proceedings of the 13^{th} International Conference on Supercomputing*, pages 41–58. ACM Press, 1999.
44. V. Zandy. Ckpt: User-level checkpointing. `http://www.cs.wisc.edu/~zandy/ckpt/`.
45. Y. Zhang, D. Wong, and W. Zheng. User-Level Checkpoint and Recovery for LAM/MPI. *SIGOPS Oper. Syst. Rev.*, 39(3):72–81, 2005.