Extending the Domain of Transparent Checkpoint-Restart for Large-scale HPC

A dissertation presented

by

Rohan Garg

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

May, 2019

Northeastern University
College of Computer and Information Science

PhD Thesis Approval

THESIS TITLE: EXTENDING THE DOMAIN OF CHECKPOINT-RESTART FOR LARGE-SCALE HPC

AUTHOR: ROHAN GARG

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor	*Date* Mar. 25, 2019
Thesis Reader	*Date* Mar 25, 2019
Thesis Reader	*Date* March 25, 2019
Thesis Reader	*Date* MARCH 26, 2019
Thesis Reader	*Date* March 27, 2019

GRADUATE SCHOOL APPROVAL:

Director, Graduate School	*Date* 3/28/2019

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature	*Date* 4/2/2019

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Abstract

While large-scale HPC systems are critical for expediting progress in many scientific fields, exascale computing will face severe resilience challenges. Checkpoint-restart is an important technology that large-scale applications continue to rely on to make forward progress in the presence of failures.

Previous work in this domain has failed to address two key challenges: (a) support for transparent checkpointing of modern hardware accelerator-based HPC systems, such as those using GPUs and newer RDMA networks; and (b) reducing the large I/O overhead that leads to reduced productivity and contention.

To address the first challenge, this dissertation presents a new transparent checkpointing framework, based on *split processes*. The framework uses the hardware virtual memory subsystem of the host CPU to decouple computation state from the external subsystem context. This isolation between the application process and the external subsystem context enables transparent checkpointing for two diverse, well-known problems: checkpointing of modern CUDA-based programs; and transparent checkpointing of MPI applications, running over a variety of RDMA networks and using different MPI implementations with a single code base.

To address the second challenge, this dissertation demonstrates that system reliability and application resilience characteristics can be used to improve system (and individual application) throughput and reduce the checkpointing I/O overhead.

Acknowledgments

This dissertation wouldn't be complete without the help and support of many people.

First, I'd like to express my sincere gratitude to my advisor, Gene Cooperman, who taught me everything about research and perseverance. His patience and dedication to his students are qualities that I'd be glad to emulate, even if by a small amount. Despite my many failed pursuits during my time at graduate school, he continued to place tremendous trust and confidence in me and gave me the freedom to explore and learn.

I'd like to especially thank my (unofficial) co-advisor, Devesh Tiwari, whose guidance and mentoring have strongly influenced my research work (in the most positive of ways) over the last 2 years of my graduate school. His passion for the field of HPC and quality research is infectious and has been a source of inspiration for me. Our impromptu converstations about technical and non-technical aspects have benefitted me tremendously.

I'm also grateful to my thesis committee members: Michael Sullivan, Peter Desnoyers, and Franck Cappello for their valuable time and feedback on various aspects of this dissertation. Peter's PhD Computer Systems course during my first semester, and deep understanding and appreciation for systems motivated me to deepen my own knowledge and pursue a career in systems. Discussions with Mike on various aspects of GPUs and his resourcefulness were really helpful. My interactions with Franck during my visit to Argonne and his feedback on my work have helped improved the quality of this thesis significantly.

The administrative staff – especially Bryan, Meg, Sarah, and Nicole – at the College of Computer and Information Science has been really kind, helpful and gone the extra mile for many of my last-minute requests. Their behind-the-scenes work made my stay at CCIS a breeze.

I'd also like to thank Kapil Arya for his guidance, friendship, and help throughout my time in Boston and graduate school.

I would be remiss if I missed to express my gratitude for my labmates: Jiajun, Apoorve, Ones, Twinkle, and Tirthak; and friends from graduate school: Jaideep, Mitesh, Anand, and Dronika. Their friendship, encouragement, and support have made this dissertation possible. The dinners, outings, and celebrations made my time here much more meaningful and enjoyable.

Finally, I'd like to thank my family without whose sacrifices and encouragement I wouldn't be here.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Overview

HPC systems are critical for progress for scientific fields that rely upon large-scale simulations and data analysis, such as, energy, biotechnology, materials science, and so on.

However, computing at the exascale will face severe resiliency challenges. As the number of nodes increases, both hard and soft faults are expected to occur with higher frequencies than previously seen.

Checkpoint-restart is, and is likely to remain, an important fault tolerance mechanism for current and future large-scale HPC systems. An application writes out its important state every so often to preserve its current state to stable storage. In case of a failure, or resource revocation, the application can restore its state from the saved data on the storage system.

For reasons of performance and energy efficiency, modern large-scale HPC systems are rapidly adopting heterogeneous architectures for example, many-core accelerators and GPUs. GPUs are thought to be one of the key enablers of future exascale systems. Similar advances in networking technology has seen the advent and usage of diverse interconnection networks, such as Cray's GNI network, InfiniBand, and Intel's OmniPath.

Yet, enabling support for transparent checkpoint-restart for GPUs and newer interconnection networks remains an open question.

Note also that the time spent in checkpointing is wasted time and resources for an application, since it cannot do any real, useful work during the time it is checkpointing. In fact, recent work shows that applications might spend up to 40% of their execution time simply doing checkpoint-restart ([31, 40, 54, 72]). This also exerts a severe pressure on the shared I/O and network infrastructure and bandwidth.

This thesis addresses the two challenges listed above: enabling transparent checkpoint-restart for modern HPC applications using hardware accelerators such as GPUs and a variety of RDMA networks; and improving system throughput and reducing I/O overhead due to checkpointing.

A solution to the first problem relies on splitting the address space of an application process into two: either through two processes, or through splitting the address space of one process into two halves. This approach is applied to to two diverse domains: transparent checkpointing for modern GPU applications; and a transparent, MPI-agnostic, network-agnostic checkpoint-restart service for MPI-based HPC applications.

A solution to the second problem exploits system reliability characteristics and variation in application checkpointing overheads in order to intelligently schedule large-scale HPC applications. This is shown to improve the system (and application) throughput and reduce checkpointing I/O overhead.

1.2 Thesis Statement

In the past, transparent checkpointing has been analyzed as a single, monolithic task. By splitting checkpointing into two loosely coupled tasks, greater performance and more flexible architectures can be achieved. This is demonstrated through proxy processes in the case of CUDA, through split processes in the case of MPI, and through improvement of system throughput in the case where two jobs (a large and small one) compete for resources on a single cluster.

1.3 Contributions

This dissertation demonstrates progress in three closely related domains: checkpoint-restart for modern CUDA (Chapter 4); transparent checkpoint-restart for MPI (Chapter 5); and improved system throughput for large-scale HPC centers in the presence of different HPC applications taking checkpoints (Chapter 6).

The first two contributions are based on a general framework for transparent checkpointing using *split processes*, which enables isolating the access to the resource context from the application context. This reduces the problem of checkpointing application processes interacting with non-reentrant external subsystems to the trivial problem of checkpointing a single, isolated process.

The third contribution is based on insights about the variation in checkpointing overheads of applications running in an HPC center and the failure recurrence behavior in the HPC center. These observations are combined to intelligently schedule applications in order to improve the system throughput and to reduce the checkpointing overhead on the backend storage.

1.3.1 Transparent Checkpointing for CUDA using Proxy Processes

Unified Virtual Memory (UVM) was recently introduced on recent NVIDIA GPUs. Through software and hardware support, UVM provides a coherent shared memory across the entire heterogeneous node, migrating data as appropriate. The older CUDA programming style is akin to older large-memory UNIX applications which used to directly load and unload memory segments before virtual memory became available. Newer CUDA programs have started taking advantage of UVM for the same reasons of superior programmability that UNIX applications long ago switched to assuming the presence of virtual memory. Therefore, checkpointing of UVM will become increasingly important, especially as NVIDIA CUDA continues to gain wider popularity: 87 of the top 500 supercomputers in the latest listings are GPU-accelerated, with a current trend of the number of GPU-based supercomput-

ers on the list growing by ten each year.

This thesis demonstrates a new scalable checkpointing mechanism, CRUM (Checkpoint-Restart for Unified Memory), for hybrid CUDA/MPI computations across multiple computer nodes. CRUM supports a fast, forked checkpointing, which mostly overlaps the CUDA computation with storage of the checkpoint image in stable storage. CRUM uses a separate proxy process for isolating the CUDA library from the application process, while using a novel *shadow page synchronization* algorithm for propagating UVM pages across the two processes.

1.3.2 Transparent Checkpointing for MPI using Split Processes

Transparently checkpointing of MPI for fault tolerance and load balancing is a long-standing problem in HPC. The problem has been complicated by the need to provide checkpoint-restart services for all combinations of an MPI implementation over all network interconnects. This thesis presents a single solution based on a single code base, MANA (MPI-Agnostic Network-Agnostic transparent checkpointing), which works for all such combinations. The agnostic properties imply that one can checkpoint an MPI application under one MPI implementation and perhaps over TCP, and then restart under a second MPI implementation over InfiniBand on a cluster with a different number of CPU cores per node. MANA is based on the *split process* approach, which enables two separate programs to co-exist within a single process with a single address space. This work also overcomes the limitations of the two previous most widely used approaches to transparent checkpointing, BLCR and DMTCP/InfiniBand, which require separate code bases for each MPI implementation and/or underlying network API.

1.3.3 Improving Large-scale HPC Throughput by Exploiting Variations in Application Checkpointing Overheads and System Reliability Behavior

Although checkpoint-restart mechanisms can keep scientific simulations moving forward, writing and reading application state incurs large I/O overhead, which impedes scientific productivity.

There have been numerous efforts to derive the optimal checkpointing interval (OCI) for an application, given the mean time between failures and the application's checkpointing overhead [32, 147]. Essentially, OCI attempts to maximize the amount of useful work done per failure for a given application. There have been several other studies that propose further refinements to OCI estimates. However, previous work has not explored how to maximize the useful work done per failure from the system's point of view, where multiple applications with different checkpointing overheads are available.

This work demonstrates a novel approach, Shiraz, to improve the system throughput of a large-scale HPC system by leveraging variations in OCIs of HPC applications and knowledge of temporal characteristics of system failures. A novel variant of Shiraz, called *Shiraz+*, is also demonstrated. Shiraz+ reduces the overall checkpointing overhead of the system while improving the system throughput and maintaining individual application performance levels. Shiraz+ can help alleviate the I/O pressure due to checkpointing on the storage backend and mitigate storage contention, potentially improving the effective I/O performance for other applications running on the shared HPC system.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows.

Chapter 2 provides a general background and literature review for the rest of the chapters. (More specific literature review for the specific domains literature is

covered in the corresponding chapters.)

Chapter 3 describes the general transparent checkpointing framework based on the split-process approach. The problem of checkpointing HPC applications interacting with non-reentrant external subsystems is also described in further detail.

Chapter 4 provides details of the checkpoint-restart framework for modern CUDA-based applications. The chapter describes the challenges, the design of the framework, and the limitations of the approach.

A transparent checkpointing framework for MPI-based HPC applications using a variety of interconnection networks is described in Chapter 5.

Chapter 6 presents the solution for improving system throughput and reducing the checkpointing overhead in large-scale HPC.

Chapter 7 presents the detailed evaluation results for the techniques described in Chapters 4, 5, and 6.

Chapter 8 describes some of the new directions and possibilities opened up by this thesis work.

Finally, Chapter 9 presents the conclusion.

CHAPTER 2

Background

2.1 Checkpointing for Fault Tolerance

Checkpoint-restart is the ability to save the state of a running process (or set of processes) to stable storage and later, restore the state from stable storage. While it's traditionally been used as a fault-tolerance strategy, there are other use cases, such as, debugging [47], accelerating process start-up times [46], and so on.

There are primarily three methodologies for implementing checkpoint-restart: (a) application-level checkpointing; (b) system-level checkpointing; and (c) virtual machine snapshotting.

Application-level Checkpointing With application-level checkpointing, the author of an application is responsible for identifying the key/relevant elements of the application state to save in order to successfully restore the application after a failure. On other other hand, system-level checkpointing offers the flexibility of transparently saving all of the application process state, without any programmer intervention.

While application-level checkpointing is considered to be more efficient than system-level checkpointing, the application is restricted to taking a checkpoint at certain "synchronization" points in the code. The synchronization points are often at the "outer-most" loop for iterative applications, where the program stack is shallow. A major part of the reason is that this allows the program code in the inner

loops to safely call into libraries that are not "checkpoint-able" and not worry about identifying and capturing relevant state for a successful restart.

Note that this could potentially force the synchronization points to be far apart in the application's runtime. Given the current failure rates and projections for future exascale systems, this would lead to severe degradation in the application and system throughput because of a large amount of wasted work and re-computation.

System-level Checkpointing On the other hand, system-level checkpointing does not require any modifications to the application. It does not suffer from the disadvantages of application-level checkpointing, viz., being able to checkpoint at fixed synchronization points. However, system-level checkpointing can impose a significant checkpointing I/O overhead. Since the system software cannot determine the important data structures of an application process, it is forced to save all of the process memory. This degrades the throughput of an application (since it spends significantly longer in checkpointing) and also exerts larger pressure on the shared network and I/O back-end.

Virtual Machine Snapshotting This involves saving the state of an entire virtual machine. This is the most general and the slowest of the three techniques. While recent work has demonstrated the ability to checkpoint distributed computations running on a network of virtual machines [49], the high checkpointing overhead and performance jitter associated with virtualization has prevented the widespread adaptation of virtual machines on HPC systems [2, 28, 67, 132, 133, 134].

2.2 System-level and Transparent Checkpointing

There are three widely used system-level checkpointing packages: (a) BLCR [58]; (b) CRIU [135]; and (c) DMTCP [7].

BLCR modifies the Linux kernel for checkpoint-restart of single-node applications. It requires the application processes to disconnect their network connections at the time of checkpoint. Later, the connections need to be rebuilt when the appli-

cation processes resume or restart from the checkpoint. While this is transparent to the end-user application, the MPI library needs to implement this service for disconnecting and rebuilding the network connections [65, 151].

CRIU is a user-space checkpoint-restart package. It uses the Linux kernel internal details that are exposed through the */proc* interface. The main focus is on being able to checkpoint and migrate microservices running in data-centers, and it does not support distributed applications, which is essential for HPC.

Distributed MultiThreaded CheckPointing (DMTCP) allows checkpoint-restart of distributed applications. It implements coordinated checkpointing via a centralized coordinator process that enforces global barriers across application processes in order to capture a consistent state of the distributed system. DMTCP pre-loads a user-space checkpointing library (using `LD_PRELOAD`) in the application processes. The library is responsible for saving and restoring the state of the process. Additional plugin libraries [8] enable virtualization of system id's being used by the application processes.

2.3 MPI Checkpointing

The use of transparent or system-level checkpointing for MPI is facing a crisis today. The most common transparent checkpointing packages for MPI in recent history are either declining in usage, or abandoned entirely. These checkpointing packages include: the checkpoint-restart service of Open MPI [66], the checkpoint-restart service of MVAPICH2 [45], DMTCP for MPI [7], and MPICH-V [21], as well as a fault-tolerant BLCR-based "backplane", CIFTS [55]. We argue existing approaches to transparent or system-level checkpointing share common complexity issues that makes long-term maintenance impractical. In particular, the HPC community needs support for checkpoint-restart services for any of m popular MPI implementations over n different network interconnects. This creates a burden to maintain $m \times n$ distinct code bases.

Currently, the two most widely used approaches to transparent checkpointing

today are BLCR [58] (used by several MPI implementations) and DMTCP/InfiniBand [58]. BLCR incurs an $m \times n$ maintenance penalty: each of m MPI implementations must implement a custom checkpoint-restart service for each of n network interconnects. DMTCP/InfiniBand is *MPI-agnostic* in that it transparently saves and restores the underlying MPI libraries along with many other constructs. However, like BLCR, DMTCP requires separate plugins [8] to save and restore different network interconnects [23, 24]. Thus, DMTCP still incurs a maintenance penalty of n plugins for each of n network interconnects.

Next, we present three case studies to demonstrate the declining usage of transparent checkpointing. We first consider Open MPI. Open MPI developers had created a novel and elegant checkpoint-restart service that made MPI applications to be *network-agnostic* [65] (checkpointing under network A and restarting under network B). But the n-interconnect penalty was still present, as Open-MPI's maintainers were required to individually support checkpointing for each n possible network interconnects supported by Open MPI. This burden caused the maintainers to drop official checkpointing support as of Open MPI version 1.7 (introduced in 2013). As of April 2019, the Open MPI FAQ continues to say: "Note: The checkpoint/restart support was last released as part of the v1.6 series. ... This feature is looking for a maintainer. Interested parties should inquire on the developers mailing list." [96].

The second case study concerns BLCR. The MPICH reference implementation, along with several other MPI implementations, adopted BLCR for checkpointing. BLCR is based on a kernel module that checkpoints the local MPI rank. In practice, the use of BLCR is severely limited due to lack of support for the System V shared memory construct (widely used for intra-node communication among MPI ranks). As of this writing, BLCR 0.8.5 (released in 2013) was the last officially supported version [16], and formal testing of the BLCR kernel module appear to have stopped with Linux 3.7 (Dec., 2012) [17]. Here, again, we speculate that BLCR declined not due to any fault with BLCR, but due to the difficulty of maintaining $m \times n$ checkpoint-restart services based on top of BLCR.

The third case study concerns DMTCP. As discussed above, DMTCP/Infini-

Band is *MPI-agnostic*, but not *network-agnostic*, requiring n plugins for each of n network interconnects. While it supports InfiniBand [24], it only partially supports Intel Omni-Path [23, Chapter 6], and does not support Cray GNI Aries network, the Mellanox extensions to InfiniBand (UCX and FCA), the libfabric API [79], and many others.

2.4 GPUs: Accelerators for HPC

As of September 2018, 87 of the top 500 supercomputers use NVIDIA GPUs, with a current trend of ten additional NVIDIA-based computers each year. CUDA is the de facto programming framework for GPUs. Hence, hybrid CUDA/MPI computations across multiple nodes have become critical for scalability.

The advent of virtual memory automated the task of managing a program's memory segments. Hence, for large, complex programs, the use of virtual memory becomes *more efficient in practice*, since few programmers wish to spend development time manually squeezing out the most efficient memory management.

In much the same way, NVIDIA has introduced *Unified Virtual Memory* (UVM) into CUDA. CUDA UVM is analogous to the virtual memory with hardware support found on traditional computers. This was recently introduced for Pascal-class GPUs using CUDA-8.

A unified virtual memory system shares a single address space between the device and the host. Newer CUDA programs are now beginning to take advantage of this. The UVM feature is particularly attractive for programs requiring more memory than resides on the GPU, since the alternative to UVM is for the application to directly copy memory between device and host. Furthermore, the use of a unified virtual address space enables deployment of complex data structures for GPU-based computation, with the same pointers being valid on the host as well on the GPU.

The use of NVIDIA GPUs continues to grow as seen in recent TOP500 lists [139], and the advent of a unified shared address space is expected to further lower the en-

try barrier and widen the adoption of GPUs in HPC systems.

Unfortunately, GPUs have been shown to suffer from a high rate of *Detected Unrecoverable Errors* (DUEs) [33, 57, 119, 122, 137, 138]. The mean time between failures (MTBF) is expected to become much worse as the number of GPU-accelarated compute nodes (and GPUs per compute node) increases in the exascale generation.

Thus, efficient checkpointing for the UVM model is considered particularly important for the future exascale generation. Unfortunately, previous checkpointing research [52, 56, 90, 120, 128, 130] assumes the older (non-UVM) memory model.

2.5 Checkpointing at Exascale: A Potential Crisis

Continued increase in computing power has enabled computational scientists to expedite the scientific research and discovery process in the past. Unfortunately, significant rise in the failure rates and a widening gap between compute and I/O system will significantly limit the usability of parallel computing systems in the future [12, 14, 18, 25, 40, 84, 118].

Computational science applications rely on resilience mechanisms such as checkpoint-restart to make forward progress in the presence of failures. Although checkpoint-restart mechanisms can keep scientific simulations moving forward, writing and reading application state incurs large I/O overhead, which impedes scientific productivity. Current large-scale scientific applications spend more than 15% of the total execution time on resilience mechanisms (e.g., checkpoint-restart) [25, 40]. At exascale, computational science applications will need to spend more than 40% of execution time on resilience mechanisms, due to orders of magnitude higher failure rate at exascale [40, 41, 136].

2.5.1 Characteristics of Failures on Large-scale Systems

A naïve strategy for improving system throughput would be to identify periods when the system is distinctly more stable (or less stable) compared to the aver-

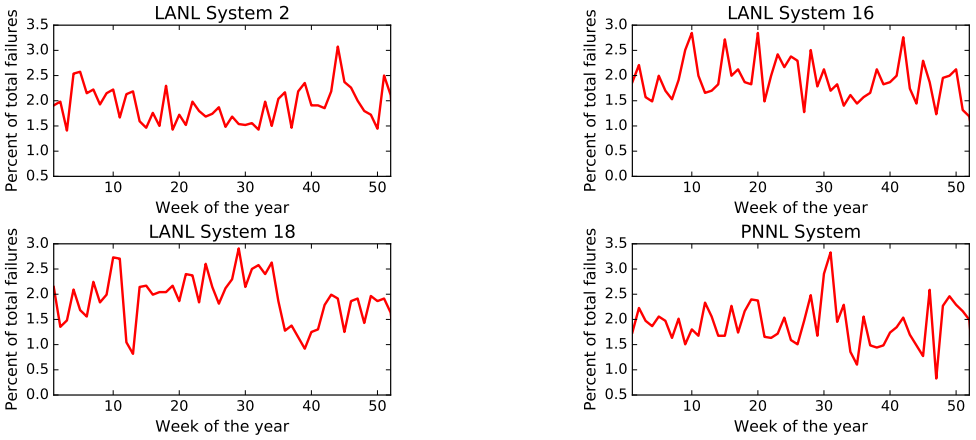


Figure 21: *Temporal failure distribution on weekly basis for multiple HPC systems.*

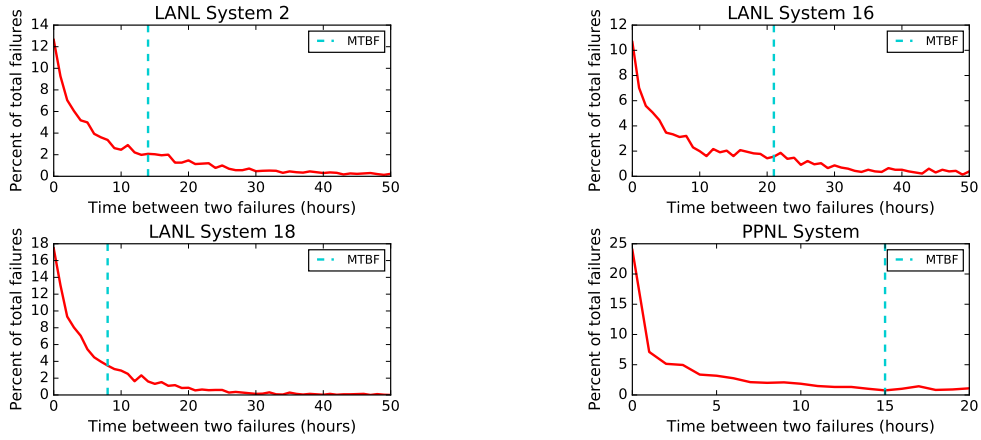


Figure 22: *Inter-arrival failure distribution for multiple HPC systems (time between two failures).*

age period and schedule applications with higher checkpointing overhead (or lower checkpointing overhead). Figure 21 shows that for large-scale HPC systems such distinct periods of stability may not exist and brief stable periods are followed by long periods of fluctuation [26]. We also note that waiting for a period when the system is more reliable can lead to starvation for applications with large checkpointing overheads.

Fortunately, we can find changing failure rate characteristics when we analyze failure characteristics at finer granularity (i.e., inter-arrival times between two failures). Note that failures considered in this study are ones that cause an application to crash and recover from last checkpoint. Figure 22 shows that a large fraction of failures are likely to occur much before the MTBF. We refer to this as the temporal

recurrence behavior of failures. This has been shown and modeled extensively for many other current and past HPC systems [10, 39, 117, 125, 136]. This property is captured by the hazard rate of the Weibull distribution which changes between consecutive failures (instead of being constant in case of the exponential distribution). The shape of the hazard rate is primarily characterized by the shape parameter (β). For $\beta < 1$, the hazard rate is high right after a failure, but it decreases over time until the next failure [103].

Multiple prior studies have shown that β varies from 0.4 to 0.7 for HPC systems [10, 117, 125, 136]. We find similar results, but since determining shape parameter is not a main contribution of this work, we omit those results. In summary, one can schedule applications within two failures to exploit changing reliability characteristics.

2.5.2 The Costs of Application Resilience at Large Scale

Table 21: *Differences in checkpointing cost among large-scale HPC applications.*

Machine	**Application Domain**	**Checkpointing Duration (sec.)**
Titan (OLCF)	Climate Change Simulation with the Community Earth System Model	1.5
Hopper (NERSC) Franklin (NERSC)	20th Century Reanalysis	2
Jaguar (ORNL) Hopper (NERSC)	Molecular Simulation in Energy Biosciences	6
Carver and Euclid (NERSC)	Computational Predictions of Trans. Factor Binding Sites	50
Cori (NERSC)	Chombo-crunch	70
Hopper (NERSC)	Climate Science for a Sustainable Energy Future	150
Hopper (NERSC)	Laser Plasma Interactions	1800
Hopper (NERSC)	Plasma Based Accelerators	2000
Hopper (NERSC)	Plasma Science Studies	2700

Next, we show evidence that large-scale scientific applications have significant variations in their checkpointing overhead. Table 21 shows the checkpointing cost of applications from different scientific domains running at different large-scale HPC centers [76, 77]. The checkpoint durations of the applications in the table

range from a few seconds to more than half an hour. Other researches have also noted a difference of orders of magnitude in the checkpointing traffic among large-scale HPC applications [81].

We observed that (1) different applications have widely varying checkpointing overheads (up to a difference of more than 40x), and (2) even the same application can exhibit different checkpointing overheads, depending on the input parameters. These variations in checkpointing overheads open up opportunities for new optimizations in the presence of multiple applications performing checkpointing on large-scale systems that experience system failures.

CHAPTER 3

Isolation of Application and Resources: A Split-process Approach to Transparent Checkpoint-Restart

3.1 Overview

This thesis presents a generic transparent checkpointing framework for HPC applications that interact with external subsystems. Historically, an HPC job in the cluster occupied a fixed number of single-core nodes with no accelerators. Modern subsystems rely on shared-memory subsystems (to optimize MPI communication between cores on a node), device drivers (e.g., for GPU and other accelerators), and a variety of low-latency, high-throughput networks (e.g., Cray's GNI, InfiniBand, and so on).

A tight coupling of application and resources creates problems for checkpointing. It is difficult to save and restore the state of tightly coupled application and resources, such as an InfiniBand library [24], or an NVIDIA CUDA library [127], or the MPI checkpoint-restart service for the network [65]. At the full system level, virtual machines have been proposed as a solution, since the same hypervisor can virtualize the access to system resources and provide isolation for applications. At a more fine-grained level, dynamic shared libraries can also virtualize access to

an external device for different processes on the same host operating system and provide isolation.

In this thesis, I propose a third, more finer-grained paradigm of isolating an application from its resources within the same process. Two variations of this concept will be discussed: one host, two processes and two address spaces; and one host, one process, but two programs.

The proposed approach enables checkpointing for external subsystems that were not designed with checkpoint-restart as one of the requirements. The external subsystem here could be a device driver, a shared-memory object, or a pinned memory region.

A key problem that arises, related to checkpointing of applications using these subsystems, is that of re-initializing these subsystem's state on restart. The state of these subsystems is manifested in the process's address space often through a helper user-space library. As an example, consider what happens when an application requires the allocation of a new pinned memory region. This is difficult to undo, and very few device driver writers will put in the effort to write this undo feature. But this undo feature is essential in order to re-initialize the subsystem state on restart, without leaving the helper user-space library in an inconsistent state.

The key insight here is that an application in this scenario executes two separate state machines, which can be decoupled into two separate address spaces. The first state machine corresponds to the application logic, and the second state machine corresponds to the logic associated with the external subsystem. The two state machines are tightly coupled and executed in the context of the application process's address space. The problem of the device driver manifests as a library that is non-reentrant. Now, if the second state machine happens to be non-reentrant, the application process cannot be restored correctly. This is because the restored application process inherits the data structures and memory from the checkpointed process's image.

Based on this insight, the proposal is to decouple the two state machines into two separate address spaces. This relies on the use of a separate proxy process in

the case of CRUM (See Chapter 4), or splitting the address space of a single process into two separate halves in the case of MANA (See Chapter 5).

Thus, an application process's state machine is segregated into two separate processes: the main application process, and a proxy process. The main application process executes the application logic, and *drives* the external subsystem's state machine, which is executed in the context of the proxy process.

At checkpoint time, the problem of checkpointing the application trivially reduces to checkpointing a single-process application with no connections to any external subsystem, as no state associated with the external subsystem is part of the process's address space.

At the time of restart, one can restore the process memory (containing the main application logic) and context, start a new proxy process and reinitialize the external subsystem by replaying the initialization commands specific to the external subsystem.

3.2 The User-space View of the Problem: Libraries are Non-reentrant

Transparent checkpointing of an application process requires saving the following three important aspects to a checkpoint file on the disk: (a) the process memory, which includes the state (text and data) of all the libraries; (b) the process context [1]; and (c) the open connections to external resources, such as, files, sockets, and so on. Then, on restart, a new process is created and the three aspects are restored by reading the contents from the checkpoint file.

Ideally, a single-process approach toward checkpointing would seem simpler. But this approach fails for several reasons. The core problem is this:

- For an application process that's interfacing with an external subsystem, through a library in the process's memory, the connection to the external

[1] There can be multiple contexts in a process corresponding to multiple threads.

subsystem and the state of the external subsystem needs to be restored on restart.

- This requires relying on the library API to allow for re-initialization, since the external subsystem, including the library, may be a closed-source system.
- However, the state of the library that's saved and restored as part of the process memory (i.e., the library data segment) can prevent the library from re-initializing. This is because the library "remembers" that it had been initialized prior to checkpoint time. Furthermore, the state of the library is encapsulated not just in its data segment, it could manifest over several memory regions spread throughout the process's memory.

  We refer to this problem as the problem of *non-reentrant libraries*.

Note that this is not a simple software engineering problem, where, with enough effort, a library could be authored to be reentrant. In fact, we argue that there is a lack of clear semantics about what it means to re-initialize the library that's interfacing with external subsystem in a process's address space.

For example, in the case of CUDA's unified memory, libraries (and the program) itself may retain pointers to unified memory regions. One must choose either to free the host memory (thus sabotaging any CUDA application that retains a pointer to the unified memory region), or else to leave the host memory region intact (thus sabotaging any application assumptions about unification of host and device memory). Note that a fresh restart will restore all host memory, but any unification of host with device memory has already been lost.

The CUDA unified memory model was developed for standard CUDA applications — and naturally did not include extensions for transparent checkpointing. An alternative workaround would have been, at restart time, to overwrite the text and data memory segments of any CUDA libraries with a fresh, uninitialized CUDA library (matching a freshly booted GPU), and then to call `cudaInit()`. Unfortunately, the CUDA library/driver appeared to have additional state, which made this workaround infeasible.

Similarly, in the case of MPI, an MPI library using shared memory for intra-node communication can suffer from the same lack of clear semantics as the CUDA user-space library with unified memory. Just as with the device-backed unified memory regions, there are no clear semantics of cleaning up the shared memory regions (or other side effects) of the MPI library, when the library is unloaded.

3.3 The Solution: Throwaway Libraries

To address the problems described in Section 3.2, we propose a novel transparent checkpointing framework: split-process, which decouples the library state from the application state.

Recall that the key problem is that state of a library that's interfacing with an external subsystem is often spread over multiple memory regions, including the application process's stack and heap. With a closed-source library, these "side effects" are especially difficult to track and clean up.

Therefore, the key idea is *to restrict the side effects of the library to known memory regions*. Since the library regions are known, these regions can be "thrown away" and replaced on restart.

The library (and any associated state) is isolated in a separate address space, which can be thrown away between checkpoint and restart. We demonstrate the feasibility of this approach by enabling transparent checkpointing for two key HPC subsystems: NVIDIA GPUs and MPI through two different implementations.

3.3.1 A First Attempt: The Two-process Approach

In the case of NVIDIA GPUs, the approach is implemented through the use of a separate proxy process. The proxy process isolates the NVIDIA library and the GPU state from the main application process. All access to the GPU is done through the proxy process, over remote procedure calls (RPC). For the case of NVIDIA's Unified Virtual Memory (UVM), where there is no API between the ap-

plication and the NVIDIA library, we demonstrate an approach using shadow pages and remote page synchronization to propagate state between the two processes.

While this solution works, it suffers from several limitations. The use of RPC can impose prohibitively high runtime overhead in extreme cases. Furthermore, RPC requires developing code for serialization and de-serialization of data, which can be non-trivial for functions with complex arguments. Finally, there is no way to support simultaneous access of (UVM) memory regions by the GPU device and the application process. Note that the use of this approach for checkpointing MPI application would suffer from similar problems.

This brings us to a second possible implementation, which we demonstrate for checkpointing the MPI library.

3.3.2 A Second Attempt: The Split-process Approach

In the case of MPI, the approach is implemented by "splitting" the application process's address space into two halves: an upper half (which contains the application); and a lower half (which contains the non-reentrant MPI library). The library executing in the lower half (although in the same process's address space) is not allowed to affect the memory of the upper half. The lower half acts as an external proxy process – akin to running two processes in a single address space – with its own heap and stack.

Since a significant portion of the lower half is comprised only of a small proxy program, the non-reentrant (MPI) library and its dependencies, this approach imposes a small, constant memory overhead. See Section 7.2.2.2 for further details.

While MPI is used as an example to demonstrate the feasibility of this approach, the split-process approach could also be used to implement throwaway libraries for other cases such as NVIDIA CUDA (as an alternative to the two-process approach), OpenGL, InfiniBand, and other HPC subsystems.

Ideally, we would simply load two programs in the same address space for a single process, and we would introduce a concept for context switching between the

two programs, but the kernel does not support loading two programs. An alternative is to load two separate programs as two separate threads [63]. (See Section 5.4 for details on why this approach is not viable for checkpointing.)

Instead of the above approach, we use the following approach for isolating a non-reentrant library in an application process. The key idea is to have two separate runtime loaders, each with its separate heap and stack. Recall that there is just one runtime loader per process in a Linux process, which is loaded at process startup time by the Linux kernel. The loader is responsible for loading in all the dependencies of the target executable and for runtime symbol resolution (using the PLT section in the ELF standard).

So, we emulate what the kernel does at process startup: set up a new stack segment, a new heap segment, load in a second runtime linker/loader, and finally jump into the second copy of the runtime linker/loader. The second runtime linker/loader loads in the non-reentrant library and any dependencies by relying on the new stack and heap, while the application stack and heap remain unaffected.

We keep track of the memory regions created by the second runtime linker/loader by interposing on its `mmap` calls. Its `sbrk` calls are intercepted to ensure that it continues to use the second heap.

This enables us to easily "throw away" all of the known memory regions across checkpoint-restart without any memory leaks, since the memory allocations by the non-reentrant library or any of its dependencies cannot creep into the application's heap. On restart, MANA loads in a new runtime loader with a new heap and stack and initializes it again.

At runtime, calls from the application (in the upper half) for the non-reentrant library are intercepted through wrapper functions. Wrapper functions are used to keep track of the external library state and also to call into the corresponding function in the lower half. This call into the function in the lower half involves a "lightweight" context switch. In particular, it requires switching the "FS" segment base register on x86-64 Linux [2]. Since the two copies of the runtime linker/loader

[2]The FS register is used on x86-64 to implement thread-local storage and also to implement

operate in the same process's address space, they are free to set up their own separate thread-local storage regions, and hence, the wrapper function in the upper half must do this lightweight context switch before jumping into the lower half, and then must restore the context when returning back from the lower half.

Note that this is completely completely transparent to the end-user application and requires no modifications to the runtime loader, the Linux kernel, the application, or any library.

This approach achieves a balance between two conflicting goals: a shared address space to allow for simple function calls across memory regions without having to transmit any data; and isolation against the side effects of the contamination of the user address space by the kernel device driver.

See Section 3.3.3 for further discussion on the relative merits of the two implementations.

3.3.3 Discussion

While the isolation provided by the two approaches suffices for enabling checkpointing, each has its own benefits.

The two-process approach lends itself naturally to copy-on-write based forked checkpointing, even for libraries that are not fork-compatible. For example, both the CUDA library and the InfiniBand library use pinned memory, which is not compatible with fork.

On the other hand, with the two-process approach, one has to deal with additional runtime overhead for data serialization, de-serialization and inter-process communication across two processes: the application process and the proxy process.

Furthermore, the use of two processes can also lead to runtime performance jitter due to scheduling of the two processes as they compete for CPU resources.

stack-smashing detection. The FS base register points to the base of the current thread context, providing access to all the thread-local variables (e.g., `errno`, and other `__thread` global variables). Other architectures, such as ARM, used unprivileged addressing modes that do not depend on special constructs, such as the x86 segments.

The split-process approach does not suffer from either of the two problems: the performance overhead, or the runtime jitter, but requires additional work to support forked checkpointing.

Another assumption with the split-process approach is that there must be a strict API between the two halves. So, the two halves must communication only through this API. However, this is not a major limitation since even the two-process approach has the same requirement.

Finally, there is a question of determinism in the log-and-replay part. Both approaches rely on recording the library API calls at runtime and replaying the relevant calls on restart to reconstruct the state of the library and the external subsystem. Thus, a correct restart is possible only if the library has some determinism, where replaying the same set of calls results in the same end state. We note that this not a major limitation, since it is possible to serialize the library calls with an interposition library, albeit at the cost of performance. The problems of determinism could also be addressed at the device driver level with little difficulty, once transparent checkpointing is seen as viable.

Note that logging is only required for calls that affect the state of the external subsystem (control path), and that logging of API calls on the data path is not required. This is important for efficiency, as logging of large, complex parameters can impose significant runtime overhead.

CHAPTER 4

CRUM: Checkpoint-Restart For CUDA's Unified Memory

4.1 Overview

Efficient checkpointing for the CUDA's UVM model is important for the future exascale generation. Unfortunately, previous checkpointing research [52, 56, 90, 120, 128, 130] assumes the older (non-UVM) memory model.

A naïve approach to support checkpoint-restart would be to: (a) introspect and save the application process state (including the CUDA user-space library) and the GPU device driver; and (b) restore the process memory (including the CUDA user-space library) and restore the GPU device driver state. Unfortunately, the CUDA user-space library, which is checkpointed and restored as part of the process memory, is non-reentrant. Thus, it cannot restore the GPU device driver state.

To address these challenges, this work presents a novel framework, CRUM (Checkpoint-Restart for Unified Memory), which decouples the application process state from the device driver state (see Section 4.3) by using a proxy process. To allow transparent sharing of UVM memory regions between the two processes – the application process and the proxy process – CRUM uses a novel algorithm for *shadow page synchronization* (see Algorithm 1).

Thus, CRUM can transparently checkpoint the application without involving any active driver state. (This could potentially allow a CUDA application to be

checkpointed on one version of CUDA and GPU hardware, and restarted on another CUDA/GPU version.)

To optimize checkpointing of applications with large memory footprints, CRUM uses a fork-based, copy-on-write mechanism. There are two phases. The first, and relatively fast, phase is the transfer of data resident on the GPU hardware to the application process through a proxy process. In the second phase, the application process disconnects from the proxy and forks a child process that writes the checkpoint data to stable storage. Meanwhile, the application process re-connects to the proxy, which resumes using the GPU for computation.

Section 4.2 presents the background and motivation, including both the need for UVM support and the need for greater GPU reliability as we approach the exascale generation. Section 4.3 describes the design of CRUM, while Section 7.1 presents an experimental evaluation. Section 4.4 presents an analysis of the current limitations of the current approach, and the potential impact on future generations of NVIDIA GPUs. Finally, Section 4.5 describes the related work.

4.2 Background and Motivation

4.2.1 History and Motivation for Unified Virtual Memory (UVM)

Unified Virtual Memory (UVM) and its predecessor, Unified Virtual Addressing (UVA), are major CUDA features that are incompatible with prior CUDA checkpointing approaches. Yet, UVM is an important innovation for future CUDA applications.

Through software and hardware support, UVM provides a coherent shared memory across the entire heterogeneous node [61, 92]. The use of UVM-managed memory greatly simplifies data sharing and movement among multiple GPUs. This is especially useful given that the most energy-efficient supercomputers place multiple compute accelerators per node—for instance, TSUBAME3.0 [141], Coral

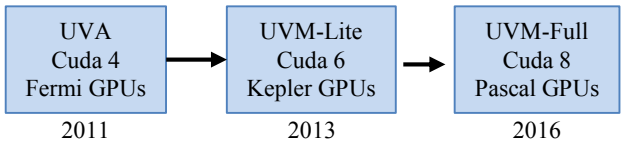


Figure 41: *The technology advancement of CUDA unified virtual memory.*

Summit [43], and the NVIDIA SATURNV [71] supercomputer use 4, 6, and 8 GPUs per node, respectively. The features and progression of UVM are briefly described below.

Historically, in CUDA 4 (2011), Fermi-class GPUs added support for Unified Virtual Addressing (UVA) with *zero-copy memory*. UVA allows transparent zero-copy accesses to memory across a heterogeneous node using a partitioned address space. UVA never migrates data, and so non-local memory accesses suffer from less bandwidth and longer latency.

To reduce the performance penalty of non-local zero-copy memory accesses, first-generation Unified Virtual Memory (UVM-Lite) was introduced in CUDA 6 (2013) for Kepler-class GPUs [59]. UVM-Lite shares a single memory space across a heterogeneous node, and it transparently migrates all memory pages that are attached to the CUDA streams associated with each kernel. This simplifies deep copies with pointer-based structures and it allows GPUs to transparently migrate UVM-managed memory to the device, nearly achieving the performance of CUDA programs using explicit memory management. Due to hardware restrictions, however, UVM-Lite does not allow concurrent access to the same memory from both CPU and GPU—host-side access is only allowed once all GPU-side accesses to a CUDA stream have completed. Concurrent access to UVM-managed memory from different GPUs is allowed, but data are never migrated between devices and non-local memory is accessed in a zero-copy fashion.

Second-generation UVM (UVM-Full) was introduced in CUDA 8 (2016) for Pascal-class GPUs [60]. It eliminates the concurrent-access constraints of the prior UVM generation and adds support for system-wide atomic memory operations, providing an unrestricted coherent shared memory across the heterogeneous node. On-demand data migration is supported by UVM-Full across all CPUs and GPUs

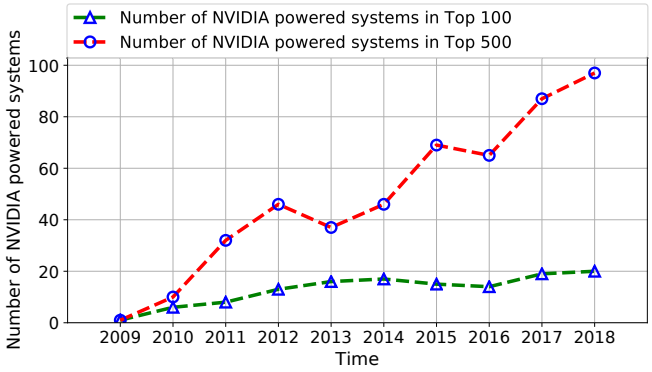


Figure 42: *NVIDIA GPUs in Top 500 list.*

in a node, with the placement of any piece of data being determined by a variety of heuristics [61].

Pascal-era UVM also adds support for memory over-subscription, meaning that UVM-managed regions that are larger than the GPU device memory can be accessed without explicit data movement. This is important for applications with large data. In particular, it greatly simplifies the programming of large-memory jobs, and avoids the need to explicitly marshal data to and from the GPU [114]. For instance, GPU-capacity-exceeding deep neural network training has been accomplished in the past through explicit data movement [108], but it can also be performed with less programmer effort by UVM over-subscription [116].

4.2.2 GPUs for Exascale: DUEs and GPU Reliability

The advantages of using GPUs for high-performance computing have been realized and a steep rise in their use in large-scale HPC systems has been observed (see Figure 42). Eighty-seven (87) systems in the Top500 list were reported to be powered by NVIDIA GPUs in November 2017, as compared to one (1) in November 2009 [139]. Thus, it is important that both hardware and the software stack (pertaining to the use of GPUs) should be highly available and reliable to maximize large-scale HPC systems productivity.

While this makes GPUs attractive for exascale computing, the high GPU detectable-

uncorrectable error rate (as compared to CPUs) remains an issue. Checkpointing plays an important role in mediating this issue. Various studies have been conducted for understanding the reliability aspects of using GPU's in large-scale HPC systems. The studies suggest that the newer generation GPU's are more reliable, as are the large-scale HPC systems using them (i.e., the observed MTBF of systems using newer GPU's is much longer than their estimated MTBF) [33, 57, 119, 122, 137, 138].

However, one factor that motivates efficient checkpoint-restart on GPU-accelerated systems is that GPU memory currently tends to have more DUEs (Detected Unrecoverable Errors) per GB than CPU memory. Memory in CPU nodes is composed of narrow 4-bit or 8-bit wide DRAM devices that are grouped together into DIMMs, meaning certain ECC codes (often called chipkill ECC) can correct the data that comes from an entire DRAM device. In contrast, GPU memory is much wider (32-bit wide for GDDR5/GDDR5X and 128-bit for HBM2) such that chipkill-level protection is not possible without a prohibitively large memory access granularity; accordingly, current GPUs use single-bit correcting SEC-DED ECC for DRAM [91, 95]. These lesser correction capabilities lead to a relative increase in detected errors. For example, a field study of the Blue Waters system [36] found that the DUE rate per GB of Kepler-era GDDR5 was roughly 5 times that of the chipkill-protected CPU memory.

Given the high rate of DUEs expected in the future exascale systems, checkpoints will be more frequent, and so it is imperative to design checkpointing mechanisms that can reduce the time that applications spend in checkpointing.

4.2.3 Checkpointing Large-memory CUDA-UVM Applications

UVM acts as an enabler for easily developing large-memory CUDA applications. UVM enables a GPU to transparently access host CPU and remote GPU memory, and hence solves the problem of otherwise manually managing data transfers. All of the host CPU's memory is available, on-demand, by the GPU device. Con-

versely, all of the UVM memory on the GPU device is available to the CPU.

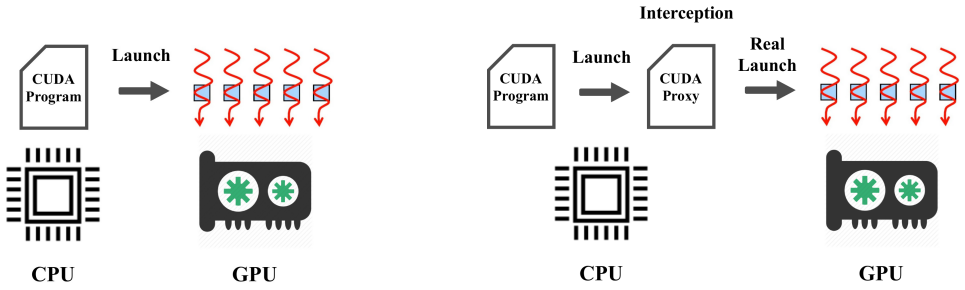
In this situation, the CUDA application may use much more memory than is present on the device. The capacity of GPU memory is currently from 16 to 32 GB for a high-end GPU, while CPU memory often ranges from 128 to 256 GB. In the past, this forced GPU application developers to choose between: scaling out to many nodes and GPUs (hence incurring communication overhead); or manually managing the data transfers on a single GPU. Later, UVM made possible a third choice: transparently transferring data on a single GPU via UVM. However, the ease of developing such large-memory CUDA-UVM applications now places a larger burden on transparent checkpointing to support this large-memory overhead.

4.3 CRUM: Design and Implementation

To address the challenges described in Section 4.2, we demonstrate CRUM, a novel framework that provides a checkpointing-based fault-tolerance mechanism. CRUM enables transparent, system-level checkpointing for CUDA and CUDA UVM applications.

Figure 43 shows a high-level schematic of CRUM's architecture. Note especially the organization into two processes: a CUDA program (the user's application), and a CUDA proxy (the only process that uses the CUDA library to communicate with the GPU). The flow of control is: (i) to interpose on CUDA library calls made by the application process; (ii) to forward the requests to the proxy process; (iii) which then executes the calls via its CUDA library and GPU, on behalf of the application; and (iv) finally returns the results back to the application.

In this section, we present the key subsystems in the design of CRUM. The first research challenge is the propagation of UVM memory pages (already shared between GPU hardware and proxy process) to make them visible to the application process. Section 4.3.2 describes a shadow page scheme (summarized in Algorithm 1) for this purpose. The second research challenge is to extend this scheme to overlap checkpointing and computation for the sake of fast, forked checkpoint and



(a) *CUDA Original* (b) *CUDA Proxy*

Figure 43: *High-level architecture of CRUM*

future exascale needs. This is discussed in Section 4.3.3. Finally, the implementation details of integrating CRUM with proxy processes is discussed in 4.3.4.

4.3.1 Post-CUDA 4: The Need for a Proxy Process

Ideally, a single-process approach toward checkpointing seems simpler. But this approach for CUDA became non-viable with CUDA 4 and beyond, when NVIDIA implemented unified virtual addressing with zero-copy, an antecedent of unified memory [116]). At that point, it was no longer possible to re-initialize the CUDA library at the time of restart. We assume that this is due to the lack of clear semantics about what it means to re-initialize a CUDA library that still retains pointers to unified memory regions on host and device. One must choose either to free the host memory (thus sabotaging any CUDA application that retains a pointer to the unified memory region), or else to leave the host memory region intact (thus sabotaging any application assumptions about unification of host and device memory). Note that a fresh restart will restore all host memory, but any unification of host with device memory has already been lost.

The core issue is that the CUDA unified memory model was developed for standard CUDA applications — and naturally did not include extensions for transparent checkpointing. An alternative workaround would have been, at restart time, to overwrite the text and data memory segments of any CUDA libraries with a fresh, uninitialized CUDA library (matching a freshly booted GPU), and then to call `cudaInit()`. Unfortunately, the CUDA library/driver appeared to have ad-

ditional state, which made this workaround infeasible.

4.3.2 Shadow Pages for the Support of UVM

Recall the use of a proxy process, as seen in Figure 43b. The core research challenge in this architecture is that UVM dictates that pages are transparently shared between the GPU hardware and the proxy process, but these shared UVM pages are not visible to the application process.

The zero-copy memory of CUDA 4 implies that there are no CUDA calls on which to interpose. In direct-mapped memory, the device may read or write to the host mapped pinned memory of the proxy process at any time. But the separate application process remains unaware of modifications to memory in the proxy process. Thus, an approach using CUDA proxies is unable to support the newer and potentially more efficient zero-copy memory for UVA. To overcome this situation, a new, transparent checkpointing approach for CUDA's zero-copy memory is proposed, in which proxy and application reflect a single application with two "personalities".

The CUDA application process and the CUDA proxy process invoke the same application binary but execute two different state machines. The application process goes through three different states: CUDA call, read from device-mapped UVM memory, write to device-mapped UVM memory. Note that the state transitions are not dictated by the CRUM framework, but rather by the application logic. On the other hand, the CUDA proxy process is simply a passive listener for requests from the application process and executes the CUDA calls and the memory reads and writes as dictated by the application.

Based on these observations, we introduce the concept of "shadow UVM pages". For every CUDA UVM allocation request by the application, CRUM creates a corresponding shadow UVM region in the context of the application process. At the same time, the CUDA proxy process requests a "real" UVM region from the device driver. The two processes, the *application* and the *proxy*, see two different views of

the memory and data at any given point.

Since there are no API calls to interpose on, this opens up the requirement for tracking the changes to the application process's memory in order to keep the two sets of pages in sync. CRUM relies on the use of user-space page-fault tracking to accomplish this. There are currently two available mechanisms for page-fault tracking in Linux: `userfaultfd`; and segfault handler and page protection bits. While there are certain performance benefits with the use of `userfaultfd`, the current work uses a segfault handler and page protection bits to allow for evaluation on clusters employing older Linux kernels.

The algorithm for synchronizing the data on shadow and real UVM pages is described in Algorithm 1.

Algorithm 1 Shadow page synchronization algorithm

```
upon event Page Fault do
    if addr ∈ AllShadowPages then
        if isReadFault() then
            ReadDataFromRealPage()
        else
            MarkPageAsDirty()
        end if
    end if
upon event CUDA call do
    if hasDirtyPages then
        SendDataToRealPages()
        ClearDirtyPages()
    end if
upon event CUDA Create UVM region do
    uvmAddr ← CreateUvmRegionOnProxy()
    reg ← CreateShadowPage(uvmAddr)
    AllShadowPages ← AllShadowPages ∪ reg
```

When an application process requests for a new UVM region, a new shadow UVM region is created in the process's memory (using the `mmap` system call). The shadow UVM region is given read-write permissions initially, and all the pages in the regions are marked as "dirty".

When the application makes a CUDA call where the device could potentially read or modify the UVM data (for example, a CUDA kernel launch), the data from

dirty pages is "flushed" to the real UVM pages on the proxy process, the dirty flag is cleared for the UVM region, and the read-write permissions are removed (using the `mprotect` system call).

This allows CRUM to interpose on either a read or write to unified memory. Standard Linux code for segfault handlers allows CRUM to detect an attempt to read or to write, and to distinguish the two cases. In the case of a read, PROT_READ permission is set for all of the memory in the application process corresponding to unified memory. In the case of a write, PROT_WRITE permission is set for all of the memory in the application process corresponding to unified memory. (See Section 4.3.2.1 for further discussion.)

At a later time, when the application process tries to read the results of the GPU computation back from the shadow UVM regions, a read page fault is triggered; the permissions of the shadow UVM region are changed to read-only, and the results are read in from the corresponding real UVM region on the proxy.

4.3.2.1 Page permissions on Linux

Note that write to shadow UVM memory region requires PROT_WRITE permission. Unfortunately, on Linux, PROT_WRITE permission implies PROT_READ permission also. Linux does not support *write-only* permission, but rather *read-write* permission instead.

This has consequences for the three-state algorithm to support unified memory in CRUM. We make the assumption here that most applications will cycle through the three states in order (possibly omitting the read-only or write-only phase). Hence, a typical cycle would be invoked: CUDA-call/read-unified-memory/write-unified-memory.

In fact, CRUM also supports overlapped execution of a CUDA call with reading and writing unified memory. The essential assumption is that read access must precede write access and a read-write cycle cannot be followed with a second read unless there is an intervening CUDA kernel. Normal CUDA calls such as

`cudaMemcpy` are allowed at all times.

As discussed earlier, unfortunately, Linux's write-only permission for memory actually grants read-write permission. It is for this reason that a transition from write-unified-memory directly to read-unified-memory cannot be detected efficiently. Possible solutions are discussed at the end of this section.

This assumption has been found to hold in each of the real-world applications that we have found for testing CRUM with unified memory. Nevertheless, it is important to also build in a (possibly slower) *verified execution mode* that will test an application to see if it violates this assumed cycle of CUDA-call/read-unified-memory/write-unified-memory.

There is more than one way to implement a verified execution mode.

One of the difficulties is that a Linux segfault handler does not allow us to reset the page permission to allow only the pending read or write, and then reset the permission back to PROT_NONE. Linux's user-space page fault handling, `userfaultfd`, introduced with Linux 4.3, can fix this, but that introduces other technical difficulties. (For example, it was only with Linux 4.11 that this was extended partially to support fork and shared memory.) Another alternative is to parse the pending read or write (load or store assembly instruction), temporarily allow read-write permission to the desired memory page, and then use the parsed information to read or write the data between register and memory, and finally to restore the previous memory permission. This might be more efficient than user-space page faulting since it might have fewer transitions to a kernel system call. This is similar to the dynamic binary translation scheme described by Adams et al. [6].

Linux kernel modification to support write-only permissions for UVM shadow pages is another possibility.

4.3.3 Fast, Forked Checkpoints

UVM enables CUDA applications to use all of the host and GPU device memory transparently. This can make checkpointing, which is dominated by the time

to write to the disk, prohibitively expensive. So while one could employ copy-on-write-based asynchronous checkpointing, UVM memory is incompatible with shared memory and fork on Linux.

Fortunately, CRUM's proxy-based architecture can be used to address this challenge. Note that the device state and the UVM memory regions are not directly a part of the application process's context, but rather they are associated with the proxy process. This frees up the application process to use forked checkpointing for copy-on-write-based associated checkpointing for the application process.

Forked checkpointing allows CRUM to invoke a minimal checkpointing delay in order to "drain the GPU device" of its data, after which, a child process of each MPI process can write to stable storage. This allows the system to overlap the CUDA computation with storage of the checkpoint image in stable storage.

4.3.4 Checkpoint-Restart Methodology and Integration with Proxies

Finally, for completeness, we discuss how CRUM integrates proxy concepts into the CUDA implementation requirements. Proxies have also been used by previous authors (see Section 4.5-d).

At checkpoint time, CRUM suspends the user application threads, and "drains" the GPU kernel queue. It issues a device synchronize call (`cudaDeviceSynchronize`) to ensure that the kernels have finished execution and the memory state is consistent. Then, for all the active CUDA-MALLOC and CUDA-UVM memory regions, data is read in from the GPU to the host. The data is first transferred from the GPU into the proxy process's memory, and then from the memory of the proxy process into the memory of the user application process. The user application process then disconnects from the proxy process. This ensures that the problem reduces to the trivial problem of checkpointing a single-process application. Finally, the state of the process is saved to a checkpoint image file on stable storage.

At the time of restart, CRUM starts a new process and recreates the user appli-

cation threads. Then, the memory of the new process gets replaced by the saved state from the checkpoint image file. CRUM, then, starts a new proxy process, which starts a new GPU context. It recreates the active CUDA-MALLOC and CUDA-UVM memory regions by replaying the allocation calls. CUDA streams and events are similarly handled. (See Section 4.4 for further discussion.) Finally, CRUM transfers the data into the actual CUDA and CUDA-UVM regions through the proxy process and resumes the application threads.

4.4 Discussion

Driver support for restart: In order to restart a computation, CRUM must re-allocate memory in the same locations as during the original execution—otherwise the correctness of pointer-based code cannot be guaranteed during re-execution. The current CRUM prototype relies on deterministic CUDA memory allocation, which we verify to work with the CUDA driver libraries via experimentation (for both explicit device memory and UVM-managed memory allocation). The assumption of deterministic memory re-allocation is shared by previous GPU checkpointing efforts [90].

Memory Overhead: In a CUDA program with large data resident on the host, the memory overhead due to an additional proxy process could be a concern. In the special case of asynchronous checkpointing, the overhead could be even higher, although copy-on-write does prevent it from going too high. This could be ameliorated by future support for shared memory UVM pages between application and a proxy running CUDA.

Advanced CUDA language features: Dynamic parallelism allows CUDA kernels to recurse and launch nested work; it is supported by CRUM without change. Device-side memory allocation allows kernel code to allocate and de-allocate memory. It is partially supported by CRUM, with one important distinction—no live device-side allocations are allowed at a checkpoint time. Thus, device-side memory allocations are to be freed before the system is considered quiesced and ready

for a checkpoint. We do not anticipate this constraint to be particularly difficult to satisfy, since device-side mallocs tend to be used to store temporary thread-local state within a single kernel, whereas host-side CUDA memory allocation (which is supported by CRUM without restriction) is more often used for persistent storage.

Using `mprotect`: Currently, in a Linux kernel, `PROT_WRITE` protection for a memory region implies read-write memory permission rather than write-only memory permission. Because of this, some compromises were needed in the implementation. This work has demonstrated the practical advantages of a write-only memory permission for ordinary Linux virtual memory.

Another issue with an mprotect-based approach is that when kernel-space code page faults on a read/write protected page, it returns an error code to the user, `EFAULT`, rather than a segfault. This forces the implementation to be extended to handle such failures; the implementation cannot rely solely on a segfault handler [22, 98, 111, 143].

Other APIs and Languages: This work provides checkpoint-restart capabilities for programs written in C/C++ with the CUDA runtime library. In our experience, the CRUM prototype should support the majority of GPU-accelerated HPC workloads; however, there are other APIs to that may be valuable for some users. Given the current framework of code auto-generation for CRUM, we believe that it will be straightforward to extend the implementation to support other APIs, such as OpenACC. The ability of CRUM to support UVM-managed memory would be especially useful for OpenACC programs, as PGI's OpenACC compiler provides native and transparent support for high-performance UVM-managed programs, making UVM-accelerated OpenACC programs a low-design-effort route to performant GPU acceleration [113].

Future Versions of CUDA: Just as prior checkpointing methods for GPUs were unable to cope with versions of CUDA since CUDA 4 (released in 2011), it is likely that CRUM will need to be updated to support language features after CUDA 8. One such development is Heterogeneous Memory Management (HMM) [64], which is a kernel feature introduced in Linux 4.14 that removes the need for explicit

cudaMallocManaged calls (or use of the __managed__ keyword) to denote UVM-managed data. Rather, with HMM the GPU is able to access any program state, including the entire stack and heap. Because the current CRUM prototype relies on wrapping cudaMallocManaged calls, it will need to be redesigned to support HMM.

4.5 Related Work

Use of proxy process Zandy et al. [149] demonstrated the use of a "shadow" process for checkpointing currently running application processes that were not originally linked with a checkpointing library. This allows the application process to continue to access its kernel resources, such as open files, via RPC calls with the shadow process.

Kharbutli et al. [70] use a proxy process for isolation of heap accesses by a process and for containment of attacks to the heap.

GPU virtualization A large number of previous HPC studies have focused on virtualizing the access to the GPU [51, 56, 74, 90, 120, 128, 129, 130]. Here we describe some of those studies, with an emphasis on the use for GPU checkpointing and GPU-as-a-Service in the cloud and HPC environments.

Lagar-Cavilla et al. [74], Shi et al. [120], Gupta et al. [56], and Giunta et al. [51] focus on providing access to the GPU for processes running in a virtual machine (VM), as an alternative to PCI pass-through. The access is provided by forwarding GPU calls to a proxy process that runs outside the VM and has direct access to the GPU.

GPU-as-a-Service Two other efforts, DS-CUDA [93] and rCUDA [38], have focused on providing access to a remote GPU for the purposes of GPU-as-a-Service [100, 104, 105, 106, 107, 121, 142]. They also rely on a proxy process. Using the proxy process is similar to the one described in this work; however, the focus is on efficient remote access by using the InfiniBand's RDMA API. To the best of our

knowledge, none of the previous studies solve the problem of efficient checkpointing of modern CUDA applications that use UVM. We note that the optimizations described in these works can be used in conjunction with CRUM for providing efficient access to remote GPUs.

GPU Checkpointing Early work on virtualizing or checkpointing GPUs was based on CUDA 2.2 and earlier [52, 56, 90, 120, 130]. Those approaches stopped working with CUDA 4 (introduced in 2011), which introduced Unified Virtual Addressing (UVA). Presumably, it is the introduction of UVA that made it impossible to re-initialize CUDA 4.

In 2016, CRCUDA [128], employed a proxy-based approach, similar to the 2011 approach of CheCL [129] that targeted OpenCL [124] (as opposed to CUDA) for GPUs. OpenCL does not support unified memory, and so CheCL and CRCUDA do not support NVIDIA's unified memory [116] targeted here.

VOCL-FT [97] aims to provide resilience against soft errors. VOCL-FT leverages the OpenCL programming model to reduce the amount of data movement: both to/from the device from/to the host, and to/from the disk. This allows them to do fast checkpointing and recovery.

HiAL-Ckpt [146], HeteroCheckpoint [68], and cudaCR [99] use application-specific approaches for providing GPU checkpointing.

None of the approaches described above work for CUDA UVM. CRUM focuses on providing efficient runtime and checkpointing support for CUDA and CUDA-UVM based programs. We note that the techniques described in above approaches are complementary to CRUM and can be used to further optimize the runtime and checkpointing overheads.

CHAPTER 5

MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing

5.1 Overview

This work presents MPI-Agnostic Network-Agnostic transparent checkpointing (MANA), a single code base to support all of the $m \times n$ combinations of MPI implementation and underlying network (Section 2.3). The new approach, based on *split processes*, is fully transparent to the underlying MPI, network, and even the particular libc library or underlying Linux kernel. (Transparent checkpointing supports standard system-level checkpointing, but it can alternatively be customized in an application-specific manner.)

The new *split processes* approach provides a solution to the $m \times n$ maintenance penalty, and supports checkpointing on all $m \times n$ combinations with a single implementation. With split processes, a single process contains two independent programs in its memory address space. The two programs are an MPI proxy application (denoted the lower-half program) and the original MPI application code (denoted the upper-half program). MANA tracks which memory regions belong to the upper half and which belong to the lower half. Only the upper-half memory regions are saved at checkpoint time. As stated earlier, MANA is also fully transparent to the specific MPI implementation, network, libc library and Linux kernel.

At restart time, MANA initializes a new MPI library along with a new underlying interconnect network in the lower half of a process. The checkpointed MPI application code and data is then copied in and restored into the upper half from the checkpoint image file. By initializing a new MPI library at the time of restart, MANA provides excellent load-balancing support without the need for additional logic. The fresh initialization inherently detects the correct number of CPU cores per node, optimizes the topology as MPI ranks from the same node may now be split among distinct nodes (or vice verse), re-optimizes the rank-to-host bindings for any MPI topology declarations in the MPI application, and so on. (See Section 8.2 for further discussion.)

MANA maintains low overhead at runtime by taking advantage of split processes to directly make library calls from the upper half to an MPI routine in the lower half. Prior to this work, related, proxy-based approaches had been used for checkpointing in the framework of checkpointing general applications [149] and in the framework of CUDA (GPU-accelerated) applications [48, 128, 129]. However, such approaches incur a significant overhead due to both context switching and the need to copy buffers (e.g., the MPI_send buffers) from an MPI application process to an MPI proxy.

Finally, there is a latency between when the coordinator invokes a checkpoint and when the MPI ranks actually begin executing the checkpoint. However, this latency is bounded by a time proportional to the time to send an MPI message and the time to execute a collective communication call in the underlying MPI implementation. In particular, see Sections 5.2.4 and 5.2.5 for the algorithm and a formal proof of the latency in the case of collective communication.

Next in this Chapter, Section 5.2 describes the the subtle design issues in fitting the split process concept to an MPI implementation. In particular, the question of how to checkpoint when there are ongoing MPI point-to-point or collective communications is discussed there. Section 7.2 then presents the experimental evaluation. Section 5.3 discusses the limitations of this work.

5.2 MANA: Design and Implementation

Multiple aspects of the design of MANA are covered in this section. Section 5.2.1 discusses the design for supporting split processes themselves. Section 5.2.2 discusses the need to save and restore persistent MPI opaque objects, such as communicators, groups and topologies. Section 5.2.3 presents Algorithm 2 for draining any point-to-point MPI messages in transit prior to initiating a checkpoint. Sections 5.2.4 and 5.2.5 present a two-phase algorithm (Algorithm 4) for completing any MPI collective communication calls in progress prior to initiating a checkpoint. And finally, Section 5.2.6 presents details of the overall implementation of MANA.

5.2.1 Upper and Lower Half: Checkpointing with an Ephemeral MPI Library

In this section, we define the *lower half* of a split process as the memory associated with the MPI libraries and all of their dependencies, including any network libraries. The *upper half* is the remaining Linux process memory, associated with the MPI application's code, data, stack, and other regions (e.g., environment variables). The definitions are by analogy with the upper and lower half of a device driver in an operating system kernel. Nevertheless, the process continues to have only a single thread, a single stack (in the upper half), and a single flow of control. (More precisely, the process continues to include each thread of the original MPI application, each with its own stack. Section 5.2.6 describes an additional "helper thread", but that thread is active only during a checkpoint or a restart from a checkpoint image file.)

Note that libc and other system libraries are a special case here. There will be one copy of libc appearing in the lower half as a dependency of the MPI libraries, and there will be a second copy of libc appearing in the upper half as an independent dependency of the MPI application code.

This split-process approach allows MANA to balance two conflicting objectives: a shared address space; and isolation of upper and lower halves. The isola-

tion allows MANA to omit the lower half memory (and omit an "ephemeral" MPI library) when it creates a checkpoint image file. The shared address space allows the flow of control to pass efficiently from the upper-half MPI application to the lower-half MPI library through the standard C/Fortran calling conventions, including call by reference: passing pointer parameters from the upper half to the lower half, where those pointers continue to refer to data in the upper half.

The *isolation* is needed so that at checkpoint time, the old lower half can be omitted from the checkpoint image, and at the time of restart, replaced with a small "bootstrap" MPI program with new MPI libraries. The bootstrap program calls `MPI_Init()` and each MPI process discovers its MPI rank via a call to `MPI_Rank()`. The memory present at this time becomes the lower half. The MPI process then restores the upper-half memory from a checkpoint image file corresponding to the correct MPI rank. Control is then transferred back to the upper-half MPI application, and the original stack in the lower half is never used again.

The *shared address space* is needed for efficiency. Earlier work in the checkpointing literature had used a separate proxy process instead of a lower half within the same thread. This approach was explored in [48, Section IV.B] and in [129, Section IV.A], where the former work reported a typical 6% runtime overhead for real-world CUDA applications, and the latter reported runtime overheads of 20% or higher in some of the OpenCL examples from the NVIDIA SDK 3.0. In contrast, Section 7.2 reports typical overheads less than 2% for MANA under older Linux kernels, and less than 1% runtime overhead for recent Linux kernels.

Finally, note that the ability to discard the lower half when creating a checkpoint image file greatly simplifies the task of checkpointing. When the lower half is discarded, the MPI application in the upper half appears as an isolated process with no inter-process communication. Hence, the upper half does not involve network communication or shared memory between processes. So, a single-process checkpointing package suffices to create this checkpoint image.

A minor inconvenience of the current approach is that a call to `sbrk()` will cause the kernel to extend the process heap in the data segment. Calls to `sbrk()`

can be caused by invocations of `malloc()`. Since the kernel has no concept of the upper and lower halves of a process, the kernel will choose, for example, to extend the lower half data segment after restart since that corresponds to the original program seen by the kernel before the upper-half memory is restored. So, MANA interposes on calls to `sbrk()` in the upper-half libc, and then insert calls to `mmap()` to extend the heap of the upper half.

An alternative approach of using `dlopen/dlmopen` was considered earlier and then discarded. See Section 5.2.1.1 for a related discussion of why a process-in-process approach using dlmopen is not feasible in this context.

Finally, MANA employs coordinated checkpointing, and a checkpoint coordinator sends messages to each MPI rank at the time of checkpoint (see Sections 5.2.3, 5.2.4 and 5.2.5). MPI opaque objects (communicators, groups, topologies) are detected on creation and restored on restart (see Section 5.2.2). This is part of a broader record-replay strategy by which MPI calls with persistent effects (such as creation of these opaque objects) are recorded during runtime and replayed on restart.

5.2.1.1 Discussion: Checkpointing the MPI Library

One approach to transparent checkpoint-restart of an MPI application would be to save the state of the MPI library along with the application state to a file on the disk; then, restore the MPI process, with the MPI library state and the application, from the checkpoint file.

However, this requires the checkpointing package to recreate the underlying network connections that the MPI library was using and, in some cases, to also virtualize MPI library's access to the communication layer [23].

While one could extend the checkpointing package to support the wide variety of HPC communication networks, this requires significant development efforts and is not extensible to future HPC environments.

Since an MPI rank only communicates with other MPI ranks through the MPI

API, regardless of the underlying network, one could think of another transparent checkpointing approach that tracks and virtualizes at the MPI API layer (for example, by interposing on the MPI library calls).

However, this attempt also fails for several reasons. The MPI library state that's saved and restored from the checkpoint file does not allow for re-initialization (i.e., the MPI library initialization, `MPI_Init()`, is non-reentrant). Furthermore, the communicators and topologies that existed at checkpoint time, cannot be recreated and restored, since the MPI library still retains pointers and state to those. While one could virtualize these opaque identifiers, this leads to memory leaks, since there's no way to reclaim or reuse these library internal structures.

Thus, MANA takes a different approach: virtualizing and checkpointing at the MPI API layer, while *not* saving and restoring the MPI library. This is important to allow for replacement of an initialized MPI library with a fresh, uninitialized one on restart.

This is achieved by MANA through the use of the split-process approach, which enables an ephemeral MPI library in the MPI application process's address space.

Ephemeral MPI Library To throw away regions of linked code at runtime (or across checkpoint-restart), one requires the ability to robustly track and isolate the memory regions being used by the different code segments, including any side effects in a process's address space.

Dynamic linking enables one to dynamically load and link in arbitrary pieces of code at runtime. The GNU toolchain provides API's, such as, `dlopen`, `dlmopen`, to enable runtime, dynamic linking.

A first attempt to solve this problem using the runtime linker/loader's dynamic loading (e.g., `dlopen` and `dlmopen`) and unloading (e.g., `dlclose`) features fails for several reasons.

The fundamental issue is that these features were not intended for the purpose of isolating different data or code segments, but rather for allowing different, dynamically-loaded pieces of code to share the same address space, including the

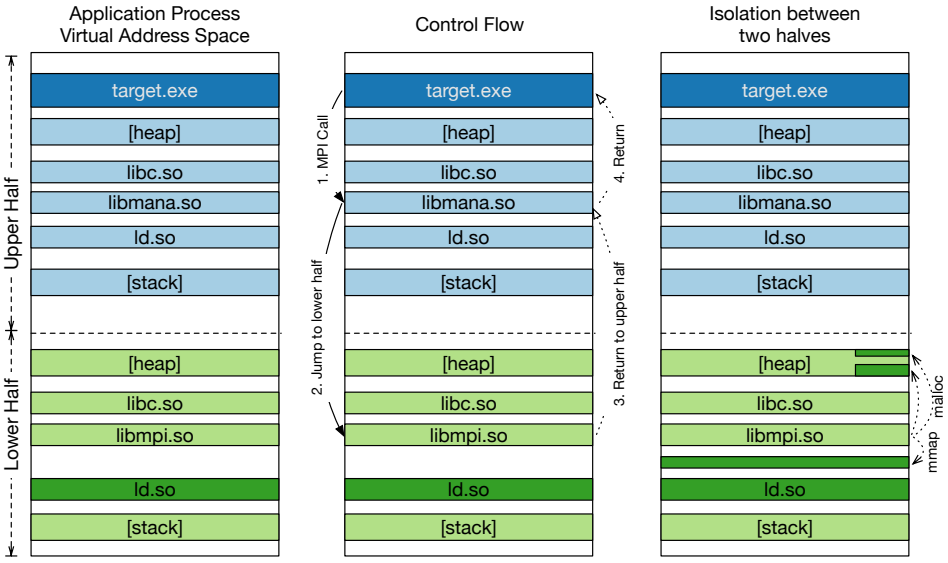


Figure 51: *Split process approach used by MANA. Note especially the second copy of the runtime linker/loader. Libraries in the lower half use the lower half's separate heap and stack segments. The side effects of libraries in the lower half are tracked and restricted to the lower half memory regions.*

process's heap and stack segments. Thus, it is very difficult to robustly track and isolate memory regions being used by the different code segments.

To get around this issue, one could try to use the `dlclose` function of the runtime linker/loader to unload the region of code that one wants to throw away. But this fails to provide the isolation we want. Note that dynamic loading and linking depends on a destructor function implemented in a library to clean up any remnants at unload time. A destructor that can clean up all of the library's side effects in memory is often difficult to write and fundamentally impossible in many cases [48]. Thus, this approach not only leads to memory leaks but a freshly loaded library can fail to initialize if it finds any inconsistent state in process's memory [48].

Finally, even with a destructor that can clean up a library's side effects in process's memory, it's difficult to handle saving and restoring of the state of an "external" subsystem that the library may be interacting with. For example, the MPI library often talks to an MPI process manager to enquire about global state, such as, the MPI rank of the current process.

Therefore, MANA uses the following approach for isolating the MPI library in

an MPI process. The key idea is to have two separate runtime loaders, each with its separate heap and stack. Recall that there's just one runtime loader per process in a Linux process, which is loaded at process startup time by the Linux kernel. The loader is responsible for loading in all the dependencies of the target executable and runtime symbol resolution (using the PLT).

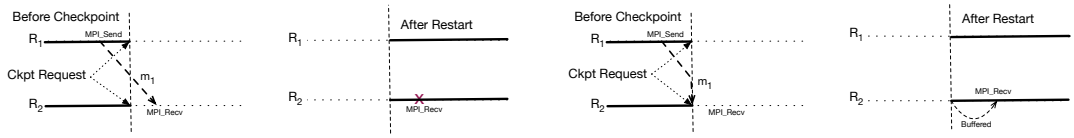
So, MANA emulates what the kernel does at process startup: sets up a new stack segment, a new heap segment, loads in a second runtime linker/loader, and finally "asks" the second copy of the runtime linker/loader to load in the MPI library. MANA keeps track of memory regions of the second runtime linker/loader creates by interposing on its `mmap` calls. Also, since the second ld.so is only aware of the new heap and stack, this is what it and the libraries it loads will continue to use.

Since the side effects of the MPI library are now restricted to "known" memory regions, there are no memory leaks and these side effects cannot creep and pollute the application's memory regions. This allows MANA to easily "throw away" all of these known memory regions across checkpoint-restart. On restart, MANA loads in a new runtime loader with a new heap and stack and initializes it again.

Another advantage of this approach is that this is completely transparent to the end-user application and requires no modifications to the runtime loader, the Linux kernel, the application, or any library.

This approach allows MANA to efficiently interpose and virtualize at the MPI API layer, and at the same time, allow for replacement of an initialized MPI library with a fresh, uninitialized one on restart. Figure 51 shows the high-level architecture of the split-process approach used by MANA.

A typical MPI application consists of distributed processes (MPI ranks) that can create subgroups for communication, send messages to each other, and invoke global barriers and do collective communication. Next, we discuss how MANA saves and restores these three important components of an MPI computation.



(a) *No draining of in-flight messages at checkpoint time.*

(b) *Failed* `MPI_Recv` *on restart.*

(c) *MANA drains in-flight messages.*

(d) *MANA returns the buffered in-flight messages on restart.*

Figure 52: *Checkpointing MPI point-to-point communication. (see Section 5.2.3)*

5.2.2 Checkpointing MPI Communicators, Groups, and Topologies

An MPI application can create communication subgroups and topologies for connecting groups of processes for ease of programmability and for efficient communication. MPI implementations provide opaque handles to the application as a reference to a communicator object or group.

MANA interposes on all calls that refer to these opaque identifiers, and virtualizes the identifiers. At runtime, MANA records any MPI calls that can modify the MPI communication state, such as `MPI_Comm_create`, `MPI_Group_incl`, etc. On restart, MANA recreates the MPI communicator state by replaying the MPI calls using a new MPI library. So, while the new MPI library may provide different opaque identifiers, the runtime virtualization allows the application to continue to run with consistent handles across checkpoint-restart.

A similar checkpointing strategy also works for other opaque identifiers, such as, MPI derived datatypes, etc.

5.2.3 Checkpointing MPI Point-to-Point Communication

Capturing the state of MPI processes requires quiescing the process threads, and preserving the process memory to a file on the disk. However, this alone is not sufficient to capture a consistent state of the computation. Any MPI messages that were already sent but not yet received at the time of quiescing all the processes must also be saved as part of the checkpoint.

Figure 52 shows the state of a system with two MPI ranks, R_1 and R_2. Rank R_1 sends a message, m_1, to rank R_2 and then, receives a checkpoint request from the user. On the other hand, rank R_2 receives the checkpoint request before it can receive the message, m_1. If the state of the network channel, with the message, m_1, is not saved, this can lead to a deadlock on restart. This is because when rank R_1 is restored from the same point on restart, it remembers that it had sent the message in the "past" and it is not going to send the message again; and since rank R_2 had not received the message at checkpoint time, it is going to get stuck when it resumes to do the receive on restart.

Thus, MANA uses an MPI message draining algorithm to save all *in-flight* MPI messages to capture the state of the network. The main idea is to publish local information (send/receive counts, and pending sends) to a centralized key-value database, and iterate until all unreceived messages have been received and buffered. A similar approach was used in [23, Chapter 5.4.2]. Algorithm 2 describes the approach in more detail.

At runtime, MANA keeps track of message send and receive requests to MPI. In particular, it tracks the blocking, synchronous, and asynchronous send and receive requests. Note that the only information that is tracked is the number of such requests and some additional metadata (like the communicator and datatype); the message contents are not tracked. This allows MANA to compute the global set of pending sends at each rank at any given point in time.

At checkpoint time, first, each rank publishes its local counts and a set of pending (unreceived) sends to a centralized key-value database. Second, each rank receives the information for all the ranks through the centralized database. Third, each rank locally probes and tries to receive the unreceived sent messages. If the receive is successful at a rank (meaning that some rank had sent it a message before the checkpointing request arrived), the rank updates its local receive count. Finally, each rank loops through the same set of these three steps, until the global send count becomes less than or equal to the global receive count and there are no more unreceived sent messages. (Note that the receive count can be higher than the send

Algorithm 2 Algorithm for MPI message draining.

```
1: upon event Checkpoint request do
2:     for all R ∈ MPI Ranks do
3:         pendingSends ← GetLocalPendingSends()
4:         PublishLocalSendAndRecvCounts(pendingSends)
5:         GlobalBarrier()
6:         <totalSends, totalRecvs> ← GetGlobalCounts()
7:         while totalSends > totalRecvs do
8:             DrainMessages(pendingSends)
9:             GlobalBarrier()
10:            pendingSends ← GetLocalPendingSends()
11:            PublishLocalSendAndRecvCounts(pendingSends)
12:            <totalSends, totalRecvs> ← GetGlobalCounts()
13:        end while
14:    end for
15: function GETLOCALPENDINGSENDS
16:    pendingSends ← ∅
17:    for all s ∈ MPI Asynchronous Send Requests do
18:        if MPI_Test(s) = Success then
19:            pendingSends ← pendingSends ∪ {s}
20:        end if
21:    end for
22:    return pendingSends
23: end function
24: procedure DRAINMESSAGES(pendingSends)
25:    for all s ∈ pendingSends do
26:        if MPI_Iprobe(s) = Success then
27:            MPI_Recv(s, localBuffer)
28:            UpdateLocalRecvCount()
29:        end if
30:    end for
31: end procedure
```

count, since a rank might post receive requests even before any message has been sent.)

After resuming from a checkpoint, any receive requests by the application are matched against the receive requests that were completed during the execution of Algorithm 2 (at checkpoint time) and returned locally.

Theorem 5.2.1. *Algorithm 2 finishes without deadlocks, assuming a network with reliable messaging.*

Proof. If the total receive count is higher than the total send count, and there are no pending sends, then there are no in-flight messages to drain.

None of the MPI calls used in draining of messages can block indefinitely (assuming a network with reliable messaging). Also, the call to `MPI_Recv()` in the `DrainMessages()` function must also not block, since the only reason it was executed was because the earlier `MPI_Iprobe()` call was successful. The MPI standard guarantees that the receive will not block if the probe was successful.

Since at each iteration of the main loop the total receive count can only increase, eventually, the control must break out of the loop. □

5.2.4 Checkpointing MPI Collectives: Overview

The next challenge in checkpointing of MPI applications is about handling the MPI barriers and collective communication.

MPI collective communication primitives involve communication amongst all or a program-defined subset of MPI ranks (as specified by the MPI communicator argument to the collective communication call). MANA's support for collective communication requires it to maintain the following invariant:

> *No checkpoint must take place while a rank is inside a collective communication call.*

Three subtle challenges exist in taking a consistent snapshot during a collective communication. Recall that MANA employs a centralized checkpoint coordinator

process (for synchronous checkpointing). The checkpoint coordinator communicates with the MPI ranks through a protocol that will guarantee that no rank is inside an MPI collective communication call at the time when the coordinator requests a checkpoint.

Challenge I (consistency): In the case of a single MPI collective communication call, there is a danger that rank A will see a request to checkpoint before entering the collective call, and rank B will see the corresponding request to checkpoint after entering the collective call, in violation of MANA's invariant. Both ranks might report that they are ready to checkpoint, and the resulting inconsistent snapshot would create problems during restart. This situation could arise, for example, if the message from the checkpoint coordinator to rank B is excessively delayed in the network. In order to resolve this, MANA introduces a two-pass protocol in which the coordinator makes a request (sends an intend-to-checkpoint message), each MPI rank acknowledges with its current state, and finally the coordinator posts a checkpoint request (possibly preceded by extra iterations).

Challenge II (progress and checkpoint latency): Given the previous solution for consistency, there can still be long delays before a checkpoint request can be sent. It may happen that rank A has already entered the barrier, and rank B will require several more hours to finish its task before entering the barrier. Hence, the two-pass protocol may create unacceptable delays before a checkpoint can be taken. Algorithm 4) addresses this by introducing an additional, *trivial barrier*: a call to MPI_Barrier() prior to the original collective communication call. We refer to this as a *two-phase algorithm* since each collective call is now replaced by a wrapper function that invokes a trivial barrier call (phase 1) followed by the original collective call (phase 2). The "trivial" barrier call produces no side effects on the MPI rank, and so it can be safely interrupted during checkpoint and the call can even be restarted at the time of restarting the MPI application. This works because the split process archi-

tecture of MANA means that only the upper half of an MPI rank (process) is saved during checkpoint, and so there will be no inconsistent state associated with the trivial barrier call in the lower half.

Challenge III (multiple collective calls): Until now, it was assumed that at most one MPI collective communication call was in progress at the time of checkpoint. It may happen that there are multiple ongoing collective calls. During the time that some MPI ranks are exiting from one of the collective calls, it may happen that there are MPI ranks associated with an independent collective call that were formerly in the MPI trivial barrier (phase 1) and have now entered the actual collective call (phase 2). To solve this, as will be seen in Algorithm 4, after an intend-to-checkpoint message, no ranks will be allowed to enter phase 2, the actual collective call, and extra iterations will be inserted into the request-acknowledge protocol between coordinator and MPI rank.

5.2.5 Checkpointing MPI Collectives: Detailed Algorithm

Next, we present a single algorithm (Algorithm 4) for checkpointing MPI collectives that contains the elements described in Section 5.2.4: a multi-iteration protocol; and a two-phase algorithm incorporating an additional call to a trivial barrier before the main collective communication call. From the viewpoint of the MPI application, any call to an MPI collective communication function is interposed on by a wrapper function, as shown in Algorithm 3.

Algorithm 3 Two-Phase collective communication wrapper. (This wrapper function interposes on all calls of an MPI application to the corresponding MPI collective communication function.)

```
1: function COLLECTIVE COMMUNICATION WRAPPER
2:                                                  ▷ Begin Phase 1
3:     Call MPI_Barrier()                           ▷ trivial barrier
4:                                                  ▷ Begin Phase 2
5:     Call original MPI collective communication function
6: end function
```

Recall that the *trivial barrier* is an extra call to `MPI_Barrier()` prior to the collective call. The cost of the extra or trivial barrier is dominated by the collective communication call, and hence it is usually negligible.

The key to this algorithm is to ensure the following extended statement of the invariant in the previous section:

> *No checkpoint must take place while a rank is inside the collective communication call (Phase 2) of a wrapper function for collective communication (Algorithm 3).*

We formalize this with the following theorem, which guarantees that the protocol of Algorithm 4 satisfies this invariant.

Theorem 5.2.2. *Under Algorithm 4, an MPI rank is never inside a collective communication call when a checkpoint message is received from the checkpoint coordinator.*

The proof of this theorem is deferred until the end of this subsection. We begin the path to this proof by stating an axiom that serves to define the concept of a barrier.

Axiom 1. *For a given invocation of an MPI barrier, it never happens that a rank A exits from the barrier before another rank B enters the barrier under the "happens-before" relation.*

Next, we present the following two lemmas.

Lemma 5.2.3. *For a given MPI barrier, if the checkpoint coordinator sends a message to each MPI rank participating in the barrier, and if at least one of the reply messages from the participating ranks reports that its rank has exited the barrier, then the MPI coordinator can send a second message to each participating rank, and each MPI rank will reply that it has entered the barrier (and perhaps also exited the barrier).*

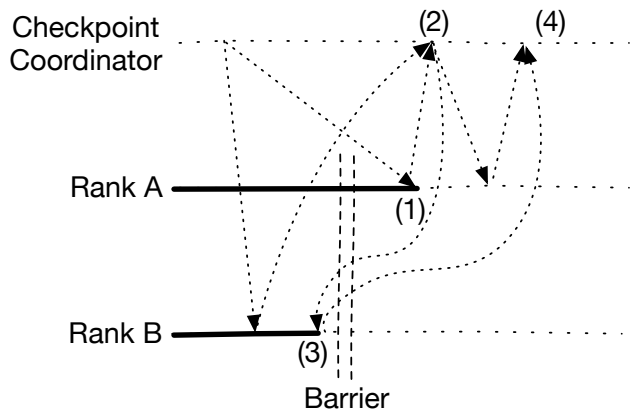


Figure 53: *Fundamental "happens-before" relation in communication between the checkpoint coordinator and the MPI ranks involved in an MPI barrier.*

Proof. We prove the lemma by contradiction. Suppose that the lemma does not hold. Figure 53 shows the general case in which this happens. At event 4, the checkpoint coordinator will conclude that event 1 (rank A has exited the MPI barrier) happened before event 2 (the first reply by each rank), which happened before event 3 (in which rank B has not yet entered the barrier). But this contradicts Axiom 1. Therefore, our assumption is false, and the lemma does indeed hold. □

Lemma 5.2.4. *Recall that an MPI collective communication wrapper makes a call to a trivial barrier and then makes an MPI collective communication call. For a given invocation of an MPI collective communication wrapper, we know that one of four cases must hold:*

(a) an MPI rank is in the collective communication call, and all other ranks are either in the call or have exited;

(b) an MPI rank is in the collective communication call, and no rank has exited, and every other rank has at least entered the trivial barrier (and possibly proceeded further);

(c) an MPI rank is in the trivial barrier and no other rank has exited (but some may not yet have entered the trivial barrier);

(d) either no MPI rank has entered the trivial barrier, or all MPI ranks have exited the MPI collective communication call.

Proof. The proof is by repeated application of Lemma 5.2.3. For case a, note that if an MPI rank is in the collective communication call and another rank has exited the collective call, then Lemma 5.2.3 says that there cannot be any rank that has not yet entered the collective call. For case b, note that if an MPI rank is in the collective communication call, then that rank has exited the trivial barrier. Therefore, by Lemma 5.2.3, all other ranks have at least entered the trivial barrier. Further, we can assume that there are no ranks that have exited the collective call, since we would otherwise be in case a, which is already accounted for. For case c, not that if an MPI rank is in the trivial barrier and no rank has exited the trivial barrier, then Lemma 5.2.3. says that there cannot be any rank that has not yet entered the trivial barrier. Finally, if we are not in case a, b, or c, then the only remaining possibility is case d: all ranks have not yet entered the trivial barrier or all ranks have exited the collective call. □

We now continue with the proof of the main theorem (Theorem 5.2.2), which was deferred earlier.

Proof. (Proof of Theorem 5.2.2 for Algorithm 4). Lemma 5.2.4 states that one of four cases must hold in a call by MANA to an MPI collective communication wrapper. We wish to exclude the possibility that an MPI rank is in the collective communication call (case a or b of the lemma) when the checkpoint coordinator invokes a checkpoint.

In Algorithm 4, assume that the checkpoint coordinator has sent an *intend-to-ckpt* message, and has not yet sent a *do-ckpt* message. An MPI rank will either reply with state *ready* or *in-phase-1* (showing that it is not in the collective communication call and that it will stop before entering the collective communication call), or else it must be in Phase 2 of the wrapper (potentially within the collective communication call), and it will not reply to the coordinator until exiting the collective call. □

Algorithm 4 Two-Phase algorithm for deadlock-free checkpointing of MPI collectives

```
1: Messages: {intend-to-checkpoint, extra-iteration, do-ckpt}
2: MPI states: {ready, in-phase-1, exit-phase-2}
3: Process Checkpoint Coordinator do
4:     function BEGIN CHECKPOINT
5:         send intend-to-ckpt msg to all ranks
6:         receive responses from each rank
7:         while some rank in state exit-phase-2 do
8:             send extra-iteration msg to all ranks
9:             receive responses from each rank
10:        end while
11:        send do-ckpt msg to all ranks
12:    end function
13: Process MPI Rank do
14:    upon event intend-to-ckpt msg or extra-iteration msg do
15:        if not inCollectiveWrapper then
16:            reply to ckpt coord:  state ← ready
17:        end if
18:        if inCollectiveWrapper and in Phase 1 then
19:            reply to ckpt coord:  state ← in-phase-1
20:        end if
21:        if inCollectiveWrapper and in Phase 2 then
22:                                  ▷ guaranteed ckpt coord won't request ckpt here
23:            finish executing coll. comm. call
24:            reply to ckpt coord:  state ← exit-phase-2
25:                                  ▷ ckpt coord can request ckpt after this
26:            set state ← ready
27:        end if
28:        continue, but wait before next coll. comm. call
29:    upon event do-ckpt msg do
30:                          ▷ guaranteed now that no rank is in phase 2 during ckpt
31:        do local checkpoint for this rank
32:                                  ▷ all ranks may now continue executing
33:        if this rank is waiting before coll. comm. call then
34:            unblock this rank and continue executing
35:        end if
```

Theorem 5.2.5. *Under Algorithm 4, deadlock will never occur. Further, the delay between the time when all ranks have received the* intend-to-checkpoint *message and the time when the* do-ckpt *message has been sent is bounded by the maximum time for any individual MPI rank to enter and exit the collective communication call, plus the usual network message latency.*

Proof. The algorithm will never deadlock, since each rank must either makes progress based on the normal MPI operation or else it stops *before* the collective communication call. If any rank replies with the state *exit-phase-2*, then the checkpoint coordinator will send an additional *extra-iteration* message. So, at the time of checkpoint, all ranks will have state *ready* or *in-phase-1*.

Next, the delay between the time when all ranks have received the *intend-to-checkpoint* message and the time when the *do-ckpt* message has been sent is clearly bounded by the maximum time for an individual MPI rank to enter and exit the collective communication call, plus the usual network message latency. This is the case since once the *intend-to-checkpoint* message is received, no MPI rank may begin to enter the collective communication call. So, upon receiving the *intend-to-checkpoint* message, either the rank is already in Phase 2 or else it will remain in Phase 1 and will not enter the call. □

Implementation of Algorithm 4: At the time of process launch for an MPI rank, a separate checkpoint helper thread is also injected into each rank. This thread is responsible for listening to incoming checkpoint-related messages from a separate coordinator process and then responding. This allows the MPI rank to asynchronously process events based on messages received from the checkpoint coordinator. Furthermore at the time of checkpoint, the existing threads of the MPI rank process are quiesced (paused) by the helper thread, and the helper thread carries out the checkpointing requirements, such as copying to stable storage the upper-half memory regions. The coordinator process does not participate in the checkpointing directly. In the implementation, a DMTCP coordinator and DMTCP

checkpoint thread [7] are modified to serve as checkpoint coordinator and helper thread, respectively.

5.2.6 Implementation and Verification with TLA+/PlusCal

The MANA prototype was implemented by extending DMTCP [7] and by developing a DMTCP plugin [8]. We used DMTCP version 3.0 for developing the prototype. DMTCP uses a helper thread inside each application process, and a coordinated checkpointing protocol by using a centralized coordinator daemon. Since this was close to the design requirements of MANA, we leveraged this infrastructure and extended the DMTCP coordinator to implement the two-phase algorithm.

However, one could equally well have modified an existing MPI coordinator process to communicate with a custom helper thread in each MPI rank that then invokes BLCR when it is required to execute the checkpoint. In particular, all sockets and other network communication objects are inside the lower half, and so any single-process checkpointing package suffices for this work.

To gain further confidence in our implementation for handling collective communication (Section 5.2.5), we developed a model for the protocol in TLA+ [75] and then used the PlusCal model checker of TLA+ based on TLC [148] to verify Algorithm 4. Specifically, PlusCal was used to verify the algorithm invariants of deadlock-free execution and consistent state when multiple concurrent MPI processes are executing. The PlusCal model checker did not report any deadlocks or broken invariants for our implementation.

5.3 Limitations

Next, we discuss the limitations of this work.

While the split-process approach for checkpointing and process migration is quite flexible, it does include some limitations inherited by any approach based on transparent checkpointing. Naturally, when restarting on a different architecture, the CPU instruction set must be compatible. In particular, on the x86 architec-

ture, the MPI application code must be compiled to the oldest x86 sub-architecture among those remote clusters where one might consider restarting a checkpoint image. (However, the MPI libraries themselves may be fully optimized for the local architecture, since restarting on a remote cluster implies using a new lower half.) Similarly, the MPI application must be limited to the oldest MPI version on which one might wish to restart. But on the brighter side, a *very* long-running application can use MANA to survive a systems upgrade that installs a newer MPI library, a newer Linux kernel, or upgrades the x86 CPU.

Similarly, while MPI implies a standard API, any local extensions to MPI must be avoided. The application *binary* interface (ABI) used by the compiled MPI application must either be compatible or else a "shim" layer of code must be inserted in the wrapper functions for calling from the upper half to the lower half. Similarly, the constant values of the MPI constants must be the same on all MPI implementations being used, or else MANA must add process virtualization code [8] to virtualize each of the MPI constants.

And of course, the use of a checkpoint coordinator implies coordinated checkpointing. If a single MPI rank crashes, MANA must restore the entire MPI computation from an earlier checkpoint.

MPI version 3 has added nonblocking collective communication calls (e.g., MPI_Igather). In future work, we propose to extend the two-phase algorithm for collective communication of Section 5.2.5 to the nonblocking case. The approach to be explored would be to employ a first phase that uses a nonblocking trivial barrier (MPI_Ibarrier), and to then convert the actual asynchronous collective call to a synchronous collective call (e.g., MPI_Gather to MPI_Igather) for the second phase. Nonblocking variations of collective communication calls are typically used as performance optimizations in an MPI application. If an MPI rank reaches the collective communication early, then instead of blocking, it can continue with an alternate compute task while occasionally testing (via MPI_Test/MPI_Wait) to see if the other ranks have all reached the barrier. In the two-phase analog, a wrapper around the nonblocking collective communication causes MPI_Ibarrier to be

invoked. When the ranks have all reached the nonblocking trivial barrier and the MPI_Test/MPI_Wait calls of the original MPI application reports completion of the MPI_Ibarrier call of phase 1, then this implies that the ranks are all ready to enter the actual collective call of phase 2. A wrapper around MPI_Test/MPI_Wait can then invoke the actual collective call of phase 2.

5.4 Related Work

Hursey et al. [65] developed a network-agnostic checkpointing service for Open MPI. It relies on BLCR for checkpointing a single isolated process and relies on the MPI implementation to handle network connections. This requires disconnecting network services prior to checkpointing and then reconnecting at resume time. This not only imposes a large checkpointing overhead, but also suffers because BLCR does not support checkpointing of SysV shared memory, which is typically used for intra-node communication internal to MPI.

Separate proxy processes for high- and low-level operations have been proposed both by CRUM (for CUDA) and McKernel (for the Linux kernel). CRUM [48] showed that by running a non-reentrant library in a separate process, one can work around the problem of a library "polluting" the address space of the application process — i.e., creating and leaving side-effects in the application process's address space. This decomposition of a single application process into two processes, however, forces the transfer of data between two processes via RPC, which can cause a large overhead.

McKernel [50] runs a "lightweight" kernel along with a full-fledged Linux kernel. The HPC application runs on the lightweight kernel, which implements time-critical system calls. The rest of the functionality is offloaded to a proxy process running on the Linux kernel. The proxy process is mapped in the address space of the main application, similar to MANA's concept of a lower half, to minimize the overhead of "call forwarding" (argument marshalling/un-marshalling).

In general, a proxy process approach is problematic for MPI, since it can lead

to additional jitter as the operating system tries to schedule the extra proxy process alongside the application process. The jitter harms performance since the MPI computation is constrained to complete no faster than its slowest MPI rank.

Process-in-process [63] has in common with MANA that both approaches load multiple programs into a single address space. However, the goal of process-in-process was intra-node communication optimization, and not checkpoint-restart. Process-in-process loads *all* MPI ranks co-located on the same node as separate threads within a single process, but in different logical "namespaces", in the sense of the `dlmopen` namespaces in Linux. (This is in contrast to MANA, which loads a lower-half and upper-half program into the address space for a single MPI rank that runs as a single thread.)

It would be difficult to adapt process-in-process for use in checkpoint-restart since that approach implies a single "ld.so" run-time linker library that managed all of the MPI ranks. In particular, difficulties occur when restarting with fresh MPI libraries while "ld.so" retains pointers to destructor functions in the pre-checkpoint MPI libraries.

In the special regime of application-specific checkpointing for bulk synchronous MPI applications, Sultana et al. [126] supported checkpointing by separately saving and restoring MPI state (MPI identifiers such as communicators, and so on). This is combined with application-specific code to save the application state. Thus, when a live process fails, it is restored using these two components, without the need restart the overall MPI job.

CHAPTER 6

Coexistence of Big and Little Jobs: Shiraz for Improving Large-scale System Throughput

6.1 Overview

At exascale, computational science applications will need to spend more than 40% of execution time on resilience mechanisms, due to orders of magnitude higher failure rate at exascale [40, 41, 136].

To address this challenge, this work presents a novel approach, Shiraz, that uses variations in checkpointing overhead among scientific applications and knowledge of temporal characteristics of failures to improve the overall system throughput (defined as total useful work done per unit time). The key idea is to schedule applications with higher checkpointing overhead during periods of relatively high reliability (with a lower failure rate), while applications with lower checkpointing overhead are scheduled during periods with relatively low reliability (with a higher failure rate). The intuition behind this idea comes from the following insight. Applications with higher checkpointing overhead have a relatively large optimal checkpointing interval and hence, the amount of average lost work per failure is also higher. Therefore, scheduling an application with higher checkpointing overhead during periods of relatively higher reliability is likely to result in lower overall lost work. By scheduling those applications having lower checkpointing overhead during periods of lower reliability (higher system failure rate), the amount of lost

work per failure can be decreased. Therefore, these schemes combined together can increase the useful work done per failure occurrence. But, it is challenging to effectively design a scheme based on this idea for several reasons.

First, the scheme relies on timely and accurate identification of time periods with varying failure rates. Second, while the scheme improves the system throughput, it also needs to ensure that the performance of individual applications is not degraded. Third, the system failure rate continually changes over time. Therefore, it is critical to adapt to the changing failure rate by switching between applications with different checkpointing overheads.

To this end, this work answers the following questions: *(1) How to accurately identify and quantify changing reliability characteristics of a system? (2) How to leverage the above information to schedule applications with different checkpointing overheads, such that the overall system throughput is improved without hurting individual applications?* This work is based on real system experiments, analytical models, and statistical techniques.

This work also presents a novel variant of Shiraz, called *Shiraz+*, which reduces the overall checkpointing overhead of the system while improving the system throughput and maintaining individual application performance levels. Shiraz+ trades the throughput gains obtained by Shiraz for a reduction in the checkpointing overhead of the application with the high checkpointing overhead. The intuition behind this idea comes from the insight that the high checkpointing overhead application is already running in a relatively higher reliability zone, and thus, it can afford to reduce its checkpointing frequency, without suffering a throughput degradation.

6.2 Shiraz: Design and Analytical Model

In a multi-application environment, a fair scheduler switches the applications at every failure, as shown in Figure 61. By switching at every failure, the scheduler provides each application an equal chance to do useful work. This traditional

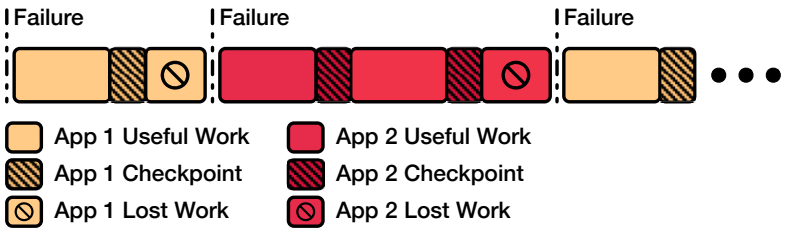


Figure 61: *Conventional scheduling (Baseline): Switch between applications after every failure.*

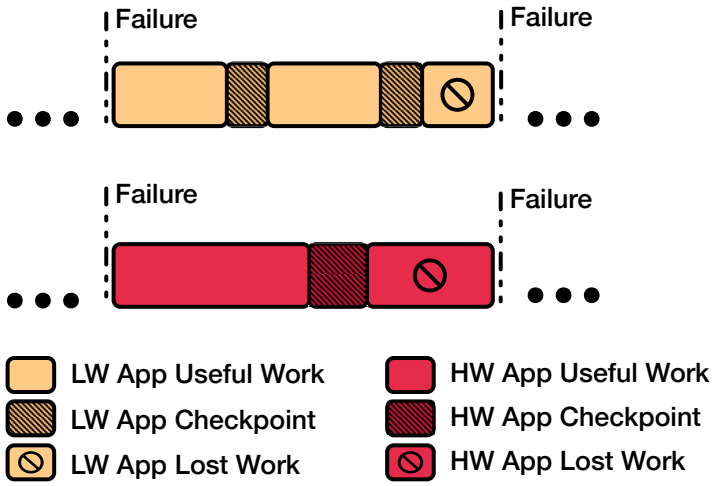


Figure 62: *Heavyweight application is likely to have higher average lost work per failure.*

approach does not exploit the two key factors discussed in Section 2.5: temporal recurrence characteristics of failures, and variation in checkpointing cost among applications.

First, we point out that the average lost work due to a failure is different for different types of applications. Figure 62 shows that an application with higher checkpointing overhead (referred as heavyweight application) is likely to have higher average lost work compared to an application with relatively lower checkpointing overhead (referred as lightweight application). This is because the optimal checkpointing interval (OCI) for the heavyweight application is larger than the OCI of lightweight application, according to Daly's formula: $\sqrt{2M\delta} - \delta$, where M is the system MTBF and δ is the checkpoint overhead of the application. Thus, larger OCI leads to higher average lost work due to a failure (Figure 62).

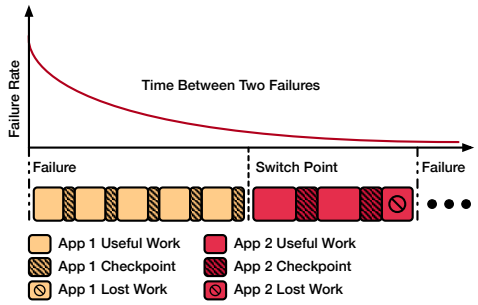


Figure 63: *Shiraz switches two applications in between two failures to reduce the overall lost work per failure by scheduling the heavyweight application during periods with relatively lower system failure rate.*

Implication: It is beneficial to schedule the heavyweight application when the system MTBF is higher. Unfortunately, it is hard to find consistent higher MTBF periods during the operational time of a system and a sub-optimal choice may result in performance degradation for the heavyweight application (as discussed in Section 2.5). To address this challenge, we leverage the non-constant failure rate between two failures. The hazard rate decreases between two failures and hence, statistically, the probability of a failure is higher right after a failure and it decreases over time. This observation can be exploited by scheduling the lightweight application before scheduling the heavyweight application.

Shiraz Key Idea: The key idea is to intelligently schedule applications with different checkpointing overheads between two failures. Shiraz schedules a heavyweight application during periods with relatively lower system failure rate, while a lightweight application is scheduled during periods with relatively higher system failure rate (as demonstrated in Figure 63). Scheduling an application with high checkpointing overhead (i.e., larger OCI) during the later part of the failure rate curve is likely to result in lower overall lost work. Similarly, scheduling a lightweight application (i.e., smaller OCI) during the earlier part of the failure rate curve decreases the amount of lost work per failure. Therefore, it increases the useful work done per failure occurrence. However, this creates new challenges.

As Figure 64 shows, while switching late, in order to avoid failures, may potentially save large amount of average lost work per failure for the heavyweight

application, it can also degrade the performance for the heavyweight application. This is because the application cannot produce the same amount of useful work as in the baseline, where each application gets a fair share of the runtime. On the other hand, switching too soon (1) exposes the heavyweight application to a higher failure rate, and (2) degrades the performance of the lightweight application. Therefore, Shiraz encapsulates an analytical model that determines the optimal switching point to dynamically adapt to the failure rate.

The formulation and details of this model are presented below. We refer to the lightweight application as *LW* and the heavyweight application as *HW*. Using Daly's formula, the OCI's for the two applications can be expressed as:

$$\text{OCI}_{\text{LW}} = \sqrt{2M\delta_{\text{LW}}} - \delta \quad \text{and} \quad \text{OCI}_{\text{HW}} = \sqrt{2M\delta_{\text{HW}}} - \delta \tag{6.1}$$

Where system MTBF, checkpoint overhead for light weight application and heavy weight application are denoted by M, δ_{LW}, and δ_{HW}, respectively.

First, we need to estimate the baseline performance for the two given applications. Recall, that in the conventional scheme, the applications are switched at every failure. Let us suppose that both the applications are executed for a total of T_{total} time. We note that switching at a failure boundary is equivalent to switching after an infinite amount of time since the last failure. This helps in developing a unified framework for modeling both baseline and Shiraz.

Estimating different components of the execution (useful work, checkpoint overhead, and lost work) requires knowing the number of failures. The number of failures between two time instances (t_{start} and t_{end}) can be estimated as follows:

$$\text{Fail}^{\text{num}}_{(t_{\text{start}},t_{\text{end}})} = \frac{T_{\text{total}}}{M} \times (e^{-(\frac{t_{\text{start}}}{\lambda})^{\beta}} - e^{-(\frac{t_{\text{end}}}{\lambda})^{\beta}}) \tag{6.2}$$

Where λ and β are the scale and shape parameter for Weibull distribution, respectively (Section 2.5). We note that the scale parameter can be derived from the MTBF: $\lambda = \frac{M}{\Gamma(1+\frac{1}{\beta})}$. Eq. 6.2 can be used to derive the total number of failures

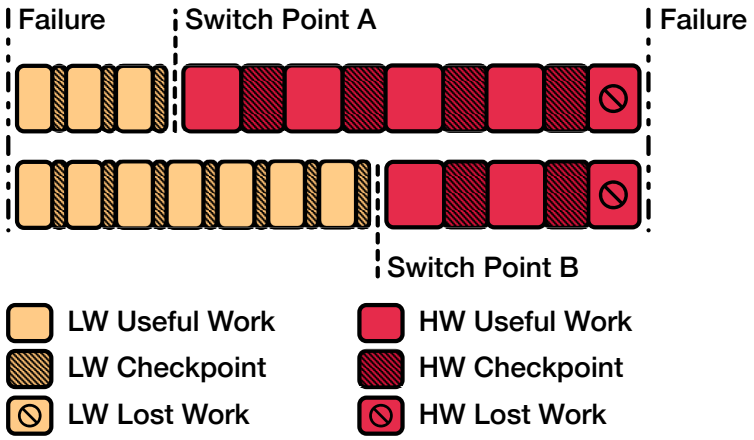


Figure 64: *Effect of different switch points between failures.*

in time T_{total} as follows.

$$\text{Fail}_{\text{total}}^{\text{num}} = \frac{T_{\text{total}}}{M} \times (1 - e^{-(\frac{T_{\text{total}}}{\lambda})^{\beta}}) \tag{6.3}$$

In the baseline case, where the application gets switched at every failure, each of the two applications essentially gets to run for $\frac{T_{\text{total}}}{2}$ time (in the baseline case $T_{\text{total}} = \frac{T_{\text{total}}}{2}$). Thus, the total lost work in the baseline case for both applications can be estimated as:

$$T_{\text{lost-base}}^{\text{LW}} = \varepsilon \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \times \text{Fail}_{\text{total}}^{\text{num}} \tag{6.4}$$

$$T_{\text{lost-base}}^{\text{HW}} = \varepsilon \times (\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \times \text{Fail}_{\text{total}}^{\text{num}} \tag{6.5}$$

Where ε is the average fraction of lost work per failure. For estimating useful work and checkpointing overhead, we can divide the time segment between two failures in chunks of optimal checkpointing interval plus checkpointing overhead ($\text{OCI} + \delta$). For probabilistic modeling, one can imagine that there are infinite such segments and calculate the probability of failure after each segment. Note that the average number of such segments is $\frac{M}{(\text{OCI}+\delta)}$. As discussed previously, the number of failures between time segments i and $i+1$ is given by $\text{Fail}_{(i\times(\text{OCI}_{\text{LW}}+\delta_{\text{LW}}),(i+1)\times(\text{OCI}_{\text{LW}}+\delta_{\text{LW}}))}^{\text{num}}$. As a short hand notation, we denote this as $\text{Fail}_{i,i+1}^{\text{num}}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$. Successful completion of a segment results in useful work

equivalent to the optimal checkpointing interval. Therefore, the useful work for the two applications in the baseline case can be mathematically expressed as:

$$T^{\text{LW}}_{\text{useful-base}} = \sum_{i=1}^{\infty} i \times \text{OCI}_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \tag{6.6}$$

$$T^{\text{HW}}_{\text{useful-base}} = \sum_{i=1}^{\infty} i \times \text{OCI}_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \tag{6.7}$$

Similarly, the checkpointing overhead per successful segment of $(OCI + \delta)$ is equal to the cost of one checkpoint. Therefore, the I/O overhead in the baseline case is:

$$T^{\text{LW}}_{\text{io-base}} = \sum_{i=1}^{\infty} i \times \delta_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \tag{6.8}$$

$$T^{\text{HW}}_{\text{io-base}} = \sum_{i=1}^{\infty} i \times \delta_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \tag{6.9}$$

This approach of modeling leads to an elegant formulation for the Shiraz case as well. The index for the summation terms does not range from 1 to ∞ now. Instead, for the lightweight application, the index will range from 1 to the switching point (k). We refer to the switching point as the number of checkpoints (say, k) the lightweight application takes before yielding to the heavyweight application. Note that the total time period the lightweight application gets to run is $k \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$. For the heavyweight application, the index will range from k to ∞. Note that for the heavyweight application, each of the segments $(i, i+1, \ldots)$ are still $(\text{OCI}_{\text{HW}} + \delta_{\text{HW}})$ long, but the first such segment starts after $k \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$ time since the last failure. Now, we can write the expressions for useful work, checkpointing overhead, and lost work for the Shiraz case as follows:

$$T^{\text{LW}}_{\text{useful-shiraz}} = \sum_{i=1}^{k} i \times \text{OCI}_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \tag{6.10}$$

$$T^{\text{HW}}_{\text{useful-shiraz}} = \sum_{i=k}^{\infty} i \times \text{OCI}_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \tag{6.11}$$

$$T^{\text{LW}}_{\text{io-shiraz}} = \sum_{i=1}^{k} i \times \delta_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \tag{6.12}$$

$$T^{\text{HW}}_{\text{io-shiraz}} = \sum_{i=k}^{\infty} i \times \delta_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \tag{6.13}$$

$$T^{\text{LW}}_{\text{lost-shiraz}} = \varepsilon \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \times \text{Fail}^{\text{num}}_{\text{LW-fraction}} \tag{6.14}$$

$$T^{\text{HW}}_{\text{lost-shiraz}} = \varepsilon \times (\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \times \text{Fail}^{\text{num}}_{\text{HW-fraction}} \tag{6.15}$$

We note that the failure can still occur before k checkpoints of the lightweight application. Our model is probabilistic and hence, sums up the probabilities over all the segments. $\text{Fail}^{\text{num}}_{\text{LW-fraction}}$ refers to the number of failures observed during the time lightweight application runs (i.e., after a failure until k checkpoints, summed over all such periods). Similarly, $\text{Fail}^{\text{num}}_{\text{HW-fraction}}$ refers to the number of failures observed during the time heavyweight application gets to runs (i.e., after k checkpoints of the lightweight application until the next failure, summed over all such periods).

Where is optimal point (optimal value of k)?: If the goal is to simply maximize the system throughput (useful work done per unit time), one can simply set k to ∞ . However, this results in starvation of the heavyweight application. In this approach, the system throughput improvement comes from favoring the lightweight application over the heavyweight application at all times. The key constraint is that both applications should not see any performance degradation compared to the baseline. That is,

$$T^{\text{LW}}_{\text{useful-shiraz}} \geq T^{\text{LW}}_{\text{useful-base}} \quad \text{and} \quad T^{\text{HW}}_{\text{useful-shiraz}} \geq T^{\text{HW}}_{\text{useful-base}} \tag{6.16}$$

Note that a range of values for k will satisfy Eq. 6.16. The highest of the values of k in this range will be the theoretical optimal switching point. It will result in the

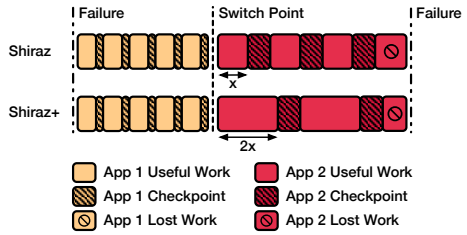


Figure 65: *Shiraz+: Reducing the checkpointing overhead.*

maximum useful work done per unit time for the whole system. However, it will not necessarily be fair to both the applications. Recall that increasing k improves the lightweight application's performance (useful work done per unit time), however, it also decreases the heavyweight application's performance. Therefore, choosing the highest such value of k that satisfies Eq. 6.16 will result in zero improvement for heavyweight application. To address this issue, Shiraz choose a sub-optimal value of k that provides fairness, i.e., equal benefits to both the applications. Therefore, Shiraz uses the following constraints to derive optimal value of k:

$$
\begin{aligned}
& T^{\text{LW}}_{\text{useful-shiraz}} - T^{\text{LW}}_{\text{useful-base}} = T^{\text{HW}}_{\text{useful-shiraz}} - T^{\text{HW}}_{\text{useful-base}} \\
\text{s.t.} \quad & (T^{\text{LW}}_{\text{useful-shiraz}} - T^{\text{LW}}_{\text{useful-base}}) \geq 0 \\
\text{and} \quad & (T^{\text{HW}}_{\text{useful-shiraz}} - T^{\text{HW}}_{\text{useful-base}}) \geq 0
\end{aligned}
$$

Shiraz solves this optimization problem numerically to determine the optimal switching point (k) such that the system throughput is maximized but both applications are treated fairly. Shiraz will return $k = \infty$ if no system throughput improvement can be achieved in the above equation.

Shiraz+ for reducing I/O overhead: Shiraz model demonstrates that choosing optimal switching point can lead to an improvement in system throughput without performance degradation for individual applications, but it does not specifically address the problem of high data movement caused by checkpointing. Checkpointing causes excessive pressure and contention on the I/O subsystem. Therefore, reducing checkpointing overhead leads to alleviating the I/O pressure, reduction in data movement (i.e., higher energy efficiency), and potentially better performance

for other applications too. Shiraz+ works on top of Shiraz and trades the additional performance gain obtained by Shiraz to reduce the checkpointing overhead.

The key idea is to increase the checkpointing interval of heavyweight application (Fig 65). The intuition behind this idea is simple: heavyweight application observes effectively higher MTBF and hence, can afford to run at a checkpointing interval that is larger than its OCI (and thus, reduce the I/O cost), though at the risk of losing performance.

Determining the new checkpointing interval for heavyweight application is a new optimization problem that Shiraz and Shiraz+ open up. But, for this work, Shiraz+ takes a relatively simpler approach and explores increasing the OCI_{HW} by an integer factor ($2\times, 3\times, \ldots$) and evaluating its impact on performance and checkpointing overhead (Section 7.3). We also note that this is a more practical approach since it does not require the application programmers to change the checkpointing interval to some new value (e.g., $2\times$ stretch in OCI can be emulated at the system level).

Interestingly, Shiraz's choice of a sub-optimal value of k helps Shiraz+. The sub-optimal value of k ensures that the heavyweight application's performance also improves. This allows Shiraz+ to trade this performance improvement for a lower checkpointing overhead. If theoretically optimal value of k was chosen, no such opportunity would exist, and increasing OCI for the heavyweight application will lead to performance degradation.

Shiraz+ does not alter the checkpointing interval of lightweight application for two reasons: (1) it has a lower impact on the overall checkpointing overhead (due to the lower cost of taking one checkpoint); and (2) it requires a re-adjustment to the the optimal switch point, which would further complicate the determination of optimal switching point. Note that Shiraz+ has no impact on the performance or checkpointing overhead of lightweight application. We evaluate Shiraz+ thoroughly under different scenarios, and analyze its impact on I/O overhead and performance in Section 7.3.

6.3 Shiraz: Analytical Model Validation

In this section, we validate the Shiraz model with a discrete-event simulator that simulates multiple applications with different characteristics running on an HPC system.

The goal of the validation is demonstrate that our probability theory based model has accurate estimations when compared to the discrete event simulation. This validation exercise will also form the basis for demonstrating that the optimal switch point predicted by Shiraz model matches with the optimal switch point obtained via extensive simulation (Section 7.3).

Our discrete-event simulator executes an application with a given checkpointing overhead on a system with a given MTBF. The application takes periodic checkpoints at optimal checkpointing interval. The application restarts from the latest checkpoint when a failure strikes. The failures are generated from a Weibull distribution. Since we are interested in analyzing the switching points between two different types of applications, we simulate two scenarios: (1) *first application*: an application is executed first and after some specified time, it is switched out. From the validation perspective, it is irrelevant what happens after the application of interest is switched out. We need to validate the application characteristics for the time frame for which the application of interest was run; and (2) *second application*: in this case, after some specified time, the application of interest is scheduled and run. From the validation perspective, it is irrelevant what happens before the application of interest is scheduled. Essentially, the goal is to not make any assumptions about relationship between the two applications being run (for example, one lightweight and other heavyweight).

We simulated a wide range of scenarios and validated our model against the simulation. For brevity, we show validation results only for representative parameters (Figure 66). We ran both the cases described earlier. An application is switched out or started at different times (expressed as fraction of MTBF for easy interpretation; results were similar for longer time periods.). We simulated an application for

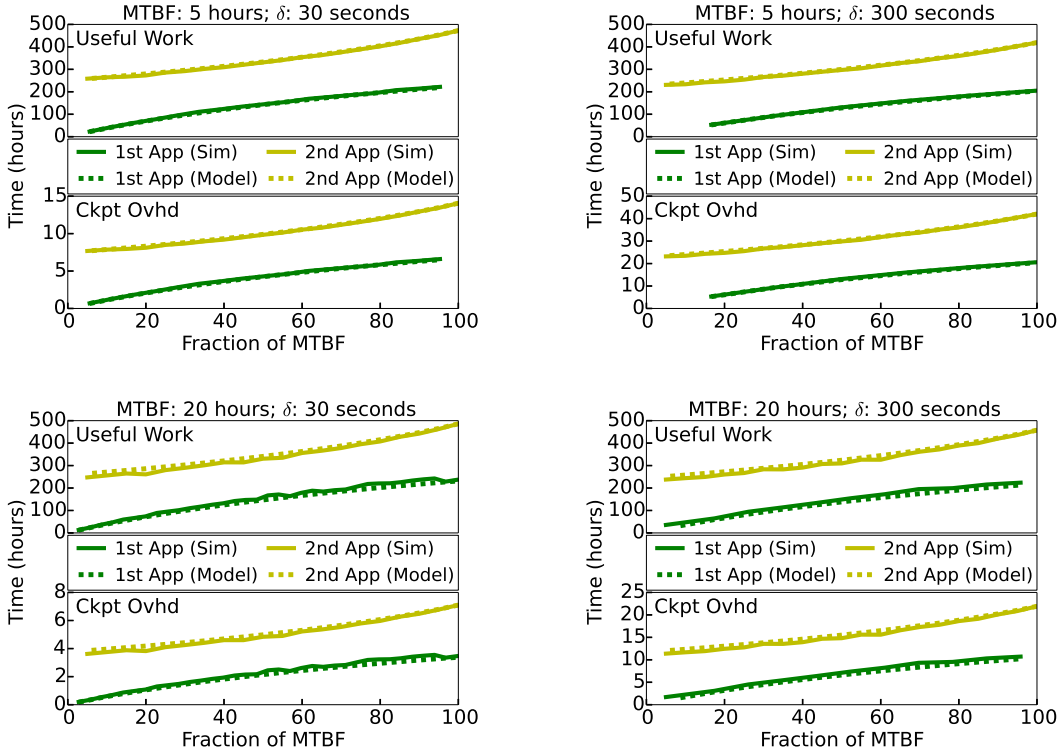


Figure 66: *Shiraz model matches with the discrete-event based simulator for a wide range of parameters and scenarios.*

a total of 1000 hours under different scenarios: MTBF of 20 hours and 5 hours (representative of peta- and exa-scale systems, respectively), and 30 seconds and 300 seconds checkpointing overhead. Failure events were generated from the Weibull distribution using the shape parameter β of 0.6. The scale parameter is determined by the MTBF and the shape parameter. We only show the useful work and the checkpointing overhead here, since the lost work, by definition, will be validated, if these other two components match. The average fraction of lost work ε parameter was estimated to be 0.45. These parameter values match with other studies and are true for many systems [10, 25, 112, 125, 136]. We obtained similar results with other values in the range.

From Figure 66, we make several observations. First, Shiraz model matches closely with the simulation across a number of parameters and scenarios. We observed similar results for the time longer than the MTBF and other parameters. For

example, in the case of the second application, the average difference in the checkpointing overhead between the model and the simulation for petascale and exascale systems is 0.06 hours and 0.14 hours, respectively.

Second, Shiraz model matches the simulation for both the application execution cases. For example, the average difference in the useful work between model and simulation for first and second applications is 2.1 and 2.2 hours, respectively. One can note the lack of data points in the case of the first application in Figure 66. This is because the data points are limited to integer multiples of application's OCI. Recall that when an application is executed first, it can be switched out only after a checkpoint, and it can invoke a maximum of $\frac{\text{MTBF}}{\text{OCI}}$ checkpoints before getting switched out. However, for second application case, we can assume any arbitrarily small OCI for the application that ran first and was switched out. This implies that we can switch in the second application at any point resulting in a smoother validation curve.

Finally, the model matches well for both applications for both useful work and checkpoint overhead. For example, application with 300 sec checkpointing overhead for petascale system observes less than 3 hours and 0.5 hours of average difference between model and simulation. This is critical as the choice of an optimal switching point relies on accurate estimation of both components.

6.4 Related Work

A large number of previous HPC studies have focused on performing failure analysis and developing checkpointing methods for fault tolerance. HPC system and application logs are extensively studied to extract information about the characteristics of failures [15, 39, 44, 101, 112, 117]. More recent studies have used neural networks, statistical learning and big-data analytics to model failure characteristics and provide potential root causes [109, 110].

These works complement our study in identifying the temporal behavior of failures in HPC systems. However, they neither provide the reason behind the

temporal locality, nor do they utilize it to reduce checkpoint overhead and improve useful work.

In order to improve the checkpointing overhead, past studies have provided different derivations for application-specific OCI [32, 82, 94, 125, 136, 147]. Our work relies on and is complementary to these studies as it schedules jobs using the proposed OCI values to improve system throughput. Some recent studies have also proposed to use multi-level checkpointing: a strategy that checkpoints at different levels (memory, SSD, PFS) to tolerate different types of failures, based on the temporal and spacial distribution of the failures [11, 34, 35, 86]. Another method called incremental checkpointing proposes to only store the state of the data which has been modified since the last checkpoint, thus potentially reducing I/O overhead [42, 89].

On the other hand, several studies have sought to use faster storage options such as SSD-based burst buffers to reduce the overhead of writing the checkpoint files [9, 81, 123]. Guler and Ozkasap [53] explore different compression methods to determine the most competent one to store a checkpoint image.

All of the above optimizations, which target different methods of reducing checkpointing overhead, can be used in conjunction with Shiraz, which targets efficient scheduling as a way to improve system throughput and decrease the checkpointing overhead.

Bouguera et al. [20] propose an application-oriented resilience scheme that combines predictive, proactive, and preventive checkpointing, by tracking and drawing correlation graphs between faults and failures. Several other works have developed reliability-aware task scheduling strategies that optimize the degree of job replication to reduce communication interference and/or energy consumption [87, 131, 144, 150]. However, replication for increased reliability also increases consumption of valuable compute and energy resources. Tiwari et al. [136] introduced Lazy checkpointing that uses temporal locality of failures to dynamically adjust the checkpointing frequency of an application to reduce I/O overhead.

Lazy checkpointing results in non-equidistant checkpoints because the rate of

increase of interval depends on the hazard rate. Unfortunately, non-equidistant checkpoints are unattractive for many applications because some domain scientists may use checkpoints to monitor the progress of the simulation; non-equidistant checkpoints make it difficult to monitor such progress. On the contrary, both Shiraz and Shiraz+ provide equidistant checkpoints. Shiraz+ shows that it is possible to increase the OCI by a factor, reduce I/O overhead and still achieve significant performance improvement unlike Lazy checkpointing. Therefore, techniques proposed in this work are more practical strategies, which improve performance, I/O overhead and work even when scheduling multiple applications unlike Lazy checkpointing [136].

CHAPTER 7

Evaluation

This chapter presents the results of experimental evaluation of the transparent checkpointing techniques described in the previous chapters. The evaluation is driven by experiments with real-world HPC application benchmarks using different prototype implementations, and guided by real-world HPC system parameters.

7.1 CRUM: Experimental Evaluation

The goal of this section is to present a detailed analysis of the performance of CRUM. In particular, this section answers the following questions:

Q1 *What's the overhead of running a CUDA (or a CUDA UVM) application under CRUM?*

Q2 *Does CRUM provide the ability to checkpoint CUDA (and CUDA UVM) applications?*

Q3 *Can CRUM improve a CUDA UVM based application's throughput by reducing the checkpointing overhead?*

Q4 *Is the approach scalable?*

7.1.1 Setup

To answer the above questions, we first briefly describe our experimental setup and methodology.

7.1.1.1 Hardware

The experiments were run on a local cluster with 4 nodes. Each node is equipped with 4 NVIDIA PCIe-attached Tesla P100 GPU devices, each with 16 GB of RAM. The host machine is running a 16-core Intel Xeon E5-2698 v3 (2.30 GHz) processor with 256 GB of RAM. Each node runs CentOS-7.3 with Linux kernel version 3.10.

7.1.1.2 Software

Each GPU runs NVIDIA CUDA version 8.0.44 with driver 396.26. Experiments use DMTCP [7] version 3.0. We developed a CRUM-specific DMTCP plugin [8] for checkpoint-restart of NVIDIA CUDA UVM applications.

The DMTCP CRUM plugin (referred to as the CRUM plugin from here onwards) interposes on the CUDA calls made by the application. The interposition code is generated in a semi-automated way, where a user specifies the prototype of a CUDA function, and whether the call needs to be logged. This not only allows us to cover the extensive CUDA API, but also allows for ease of maintainability and for future CUDA extensions.

The plugin forwards the requests, over a SysV shared memory region, to a proxy process running on the same node. The forwarded request is then executed by the proxy process, which then returns the results back to the application. To improve the performance, we use well-studied concepts from pipelining of requests, to allow the application to send requests without blocking. Blocking requests, such as, `cudaDeviceSynchronize`, result in a pipeline flush. For data transfers (both for UVM shadow page data and for `cudaMalloc` data) we use Linux's Cross Memory Attach (CMA) to allow for data transfers using a single copy operation.

7.1.1.3 Application Benchmarks

We use Rodinia 3.1 [27] benchmarks for evaluating CRUM for CUDA applications. Note that the Rodinia benchmarks do *not* use UVM, and can be run even with

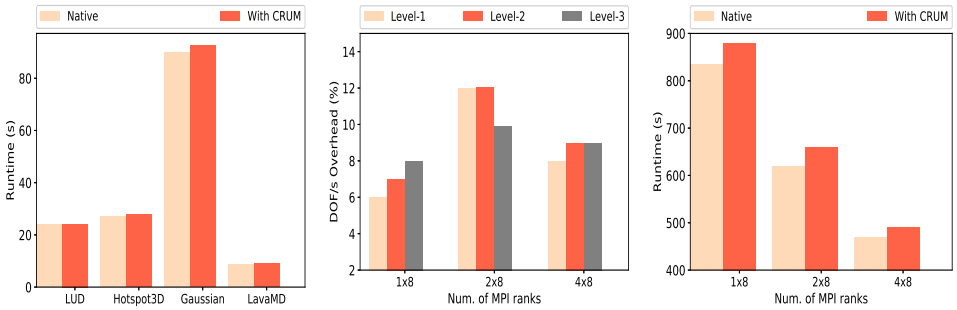
CUDA 2.x. They are included here to show comparability of the new approach with the older work from 2011 and earlier using CUDA 2.x [90, 129, 130].

We note that CheCUDA [130] does not work for modern CUDA (i.e., CUDA version 4 and above) because it relies on a single-process checkpoint-restart approach. CheCL [129] only supports OpenCL and does not work with CUDA. We tried compiling the CRCUDA [128] version available online [127], but it failed to compile with CUDA version 8. It didn't work for the benchmarks used in our experiments, after applying our compilation fixes.

To evaluate CRUM using UVM-managed memory allocation, we run a GPU-accelerated build of two DOE benchmarks: a high-performance geometric multi-grid proxy application (HPGMG-FV [78]), and a test application using a production linear system solver library (HYPRE [83]). For the HYPRE library, we run the test driver for an unstructured matrix interface using the AMG-PCG solver. For HPGMG-FV, we evaluate two versions: the standard HPGMG-FV benchmark with one grid (the *master* branch, as described in [115]), and an AMR proxy modification with multiple AMR levels (the *amr_proxy* branch, as described in [114]).

We focus on HPGMG-FV and HYPRE because they are scientific applications and libraries with potential importance in future exascale computing [73], and they have publicly available ports to UVM-enabled multi-GPU CUDA. HPGMG-FV has also been used as a benchmark for ranking the speeds of the top supercomputers [5].

To evaluate the relative performance of HPGMG-FV runs, we quote its throughput in degrees-of-freedom per second — the same metric used to rank supercomputer speeds [5]. Thus, larger numbers indicate higher performance. To evaluate the relative performance of HYPRE runs, we measure the wall clock time taken by each program execution.



(a) *Rodinia.* (b) *HPGMG-FV.* (c) *HYPRE.*

Figure 71: *Runtime overheads for different benchmarks under CRUM.*

7.1.2 Runtime Overhead

While the ability to checkpoint is important for improving the throughput of an application on a system with frequent failures, a checkpointing system that imposes excessive runtime overhead can render the framework ineffective, and in the worst case, reduce the throughput. We, therefore, benchmark and analyze the sources of runtime overhead. For these experiments, no checkpoint or restart was invoked during the run of the application.

The results demonstrate that CRUM is able to run the CUDA application with a worst case overhead of 12%, and a 6% overhead on average. We note that this is a prototype implementation and a production system could incorporate many optimizations to further reduce the overhead.

Table 71: *Runtime parameters for Rodinia applications.*

Application	Configuration Parameter
LUD	"-s 2048 -v"
Hotspot3D	"512 8 1000 power_512x8 temp_512x8"
Gaussian	"-s 8192"
LavaMD	"-boxes1d 40"

Figure 71a shows the runtimes for four applications (LUD, Hotspot3D, Gaussian, and LavaMD) from the Rodinia benchmark suite with and without CRUM. The applications mostly use the CUDA API's from CUDA 2.x: `cudaMalloc`, `cudaMemcpy`, and `cudaLaunch`. Table 71 shows the configuration parameters

used for the experiments. We observe that the runtime overhead varies from 1% (for LUD) to 3% (in the case of LavaMD). The runtime overhead is dominated by the cost of data transfers from the application process to the proxy process. In a different experiment, using Unix domain sockets for data transfer, we observed overheads varying from 1.5% to 16.5%. The use of CMA reduces the overhead significantly.

Figure 71b shows the runtime results for the HPGMG-FV benchmark with increasing number of nodes and MPI ranks. As noted in Section 7.1.1.3, we use the HPGMG-FV throughput metric DOF/s as a proxy for performance. We note that the DOF/s reported by the application running under CRUM are less than the native numbers by 6% to 12%. We present a more in-depth analysis below.

In our experiments, we observed that a single MPI rank of the HPGMG-FV benchmark runs about 9 million CUDA kernels during its runtime of 3 minutes. This implies that each CUDA kernel runs for approximately 20 microseconds on average. Note that the cost of executing a `cudaLaunch` call itself can be up to 5 microseconds. The program allocates many CUDA UVM regions, sets up the data, and runs a series of kernels to operate on the data. Each MPI rank then exchanges the results with its neighbors. While the size of the UVM regions vary from 12 KB to 128 KB, the frequent reads and writes the application process, stresses the CRUM framework in two dimensions: (a) frequent interrupts and data transfer; and (b) frequent context switches and the need to synchronize with proxy process (because of the many CUDA calls that need to be executed).

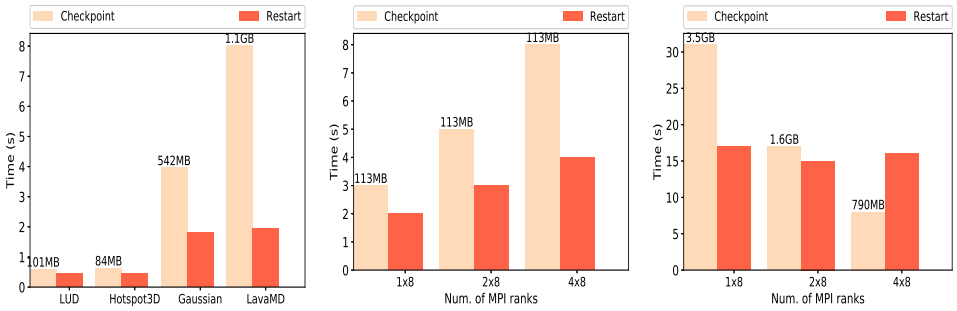
While the use of CMA (cross-memory attach) reduces the cost of data transfers, interestingly, we observed a lot of variability in the cost of a single CMA operation for the same data transfer size. The cost of a single page transfer varies from 1 microsecond to 1 millisecond, a difference of three orders of magnitude. We attribute this to two sources: (a) O/S jitter; (b) the pre-fetching algorithm employed by the UVM driver. In many cases, reading a UVM page is slowed down because of a previous read on a large UVM region, spanning several pages, because the driver gets busy pre-fetching the data for the large UVM region.

To address the second source of overhead, we optimized the CRUM implementation to: (a) use a lock-free, inter-process synchronization mechanism over shared-memory; and (b) pipeline non-blocking CUDA calls from the application. A CUDA call, such has `cudaLaunch`, `cudaMemsetAsync`, is pipelined and the application is allowed to move ahead in its execution, while the proxy finishes servicing the request. At a synchronization point, like `cudaDeviceSynchronize`, the application must wait for a pipeline flush, i.e., for the pending requests to be completed.

Figure 71c shows the runtimes for the HYPRE benchmark for a different number of MPI ranks running on a varying number of nodes. The benchmark observes up to 6.6% overhead when running under CRUM compared to native execution.

The HYPRE benchmark presents different checkpointing challenges than HPGMG-FV. While the HYPRE benchmark invokes only about 100 CUDA kernels per second (10 milliseconds on average per kernel) during its execution, it uses many large UVM regions (up to 900 MB). Thus, the overhead is dominated by the cost of data transfers between the application process and the proxy.

In addition to CMA, CRUM employs a simple heuristic to help reduce the data transfer overhead. For small shadow UVM regions, it reads in all of the data from the real UVM pages on the proxy. However, for a read fault on a large shadow UVM region, it starts off by only reading the data for just one page containing the faulting address. On subsequent read faults on the same region, while in the read phase (see Section 4.3), we exponentially increase (by powers of 2) the number of pages read in from the real UVM region on the proxy. This heuristic relies on the spatial and temporal locality of accesses. While there will be pathological cases where an application does "seemingly" random reads from different UVM regions, we have found this assumption to be valid in the two applications we tested.



(a) *Rodinia.* (b) *HPGMG-FV.* (c) *HYPRE.*

Figure 72: *Checkpoint-restart times and checkpoint image sizes for different benchmarks under CRUM.*

7.1.3 Checkpointing CUDA Applications: Rodinia and MPI

Next, we evaluate the ability of CRUM to provide fault tolerance for CUDA and CUDA UVM applications using checkpoint-restart.

Figure 72a shows the checkpoint times, restart times, and the checkpoint image sizes for the four applications from the Rodinia benchmark suite. The checkpointing overhead is dependent on the time to transfer the data from the device memory to the host memory, then transferring it from the proxy process to the application process using CMA, and then finally writing to the disk. We observe that the time to write dominates the checkpointing time.

Figure 72b shows the checkpoint times, restart times, and the checkpoint image sizes for HPGMG. The results are shown with increasing number of MPI ranks (and the number the nodes). We observe that as the total amount of checkpointing data increases from 904 MB (8×113 MB) to 3.6 GB (32×113 MB), the checkpoint time increases from 3 seconds to 8 seconds. We attribute the small checkpoint times to the buffer cache on Linux. We observed that forcing the files to be synced (by using an explicit call to `fsync` increased the checkpoint times by up to 3 times.

The results for HYPRE are shown in Figure 72c. The application divides a fixed amount of data (approx. 28 GB in total) equally among its ranks. So, we observe that the checkpoint image size reduces by almost half every time we double the number of ranks. This helps improve the checkpoint cost especially with smaller

process sizes, as the Linux buffer caches the writes, and the checkpoint times reduce from 31 seconds (for 8 ranks on 1 node) to 8 seconds (for 32 ranks over 4 nodes).

7.1.4 Reducing the Checkpointing Overhead: A Synthetic Benchmark for a Single GPU

To showcase the benefits of using CRUM to reduce checkpointing overhead for CUDA UVM applications, we develop a CUDA UVM synthetic benchmark. The synthetic benchmark allocates two vectors of 2^{32} 4-byte floating point numbers (32 GB in total) and computes the dot product of the two vectors. The floating point numbers are generated at random. Note that the total memory requirements are double of what is available on the GPU device (16 GB). However, UVM allows an application to use more than the available memory on the GPU device. The host memory, in this case, acts as "swap storage" for the device and the pages are migrated to the device or to the host on demand.

Table 72: *Checkpoint times using different strategies for the synthetic benchmark.*

Strategy	Ckpt Time	Ckpt Size	Data Migration Time
Naïve	45 s	33 GB (100% random)	4 s
Gzip	1296 s	29 GB (100% random)	4 s
Parallel gzip	86 s	29 GB (100% random)	4 s
LZ4	62 s	33 GB (100% random)	4 s
Forked Ckpting	4.1 s	32 GB (100% random)	4 s
Gzip	749 s	15 GB (50% random)	4 s
Parallel gzip	56 s	15 GB (50% random)	4 s
LZ4	45 s	17 GB (50% random)	4 s

Table 72 shows the checkpoint times for three different cases: (a) using a naïve checkpointing approach; (b) using three different compression schemes, Gzip, Parallel Gzip, and LZ4, before writing to the disk; and (c) using CRUM's forked checkpointing approach. The first two approaches, naïve and compression, use CRUM's CUDA UVM checkpointing framework. The third approach adds the forked checkpointing optimization to the base CUDA UVM checkpointing framework. The

three compression schemes use Gzip's lowest compression level (`-1` flag). While parallel Gzip uses the same compression algorithm as Gzip, it launches as many threads as the number of cores on a node to compress input data.

We observe that the forked checkpointing approach outperforms the other two approaches by up to three orders of magnitude. Since the program uses random floating point numbers, compression is ineffective at reducing the size of the checkpointing data (Table 72). We note that the time taken by the compression algorithm is also correlated with the randomness of data. As an experiment, we introduced redundancy in the two input vectors to improve the "compressibility". Of the 2^{32} floating point elements in a vector, only half (2^{16}) of the elements were generated randomly and the rest were assigned the same floating point number. This improves the compression time and reduces the checkpoint time to 749 seconds and the checkpoint image size is reduced to 15 GB by using the Gzip-based strategy.

Note that parallel Gzip may not be a practical option in many HPC scenarios, where an application often uses one MPI rank per core on a node. On the other hand, LZ4 provides a computationally fast compression algorithm at the cost of a lower compression ratio.

7.1.5 Reducing the Checkpoint Overhead: Real-world MPI Applications

Finally, we present the results from using CRUM with the forked checkpointing optimization for the real-world CUDA UVM application benchmarks. The results reported here correspond to the largest scale of 4 CPU nodes, with 16 GPU devices, running 8 MPI ranks per node (32 processes in total).

Table 73 shows the results for checkpointing time (and checkpoint image sizes) normalized to the checkpointing time using the naïve checkpointing approach (as shown in Figures 72b and 72c). The results are shown for HPGMG-FV and HYPRE.

We observe trends similar to the synthetic benchmark case. While in the naïve checkpointing approach, the checkpointing overhead is dominated by the cost of

Table 73: *Checkpoint times using different strategies for real-world CUDA UVM applications. The numbers reported corresponds to running 32 MPI ranks over 4 nodes. The checkpoint size reported is for each MPI rank. The checkpoint times are normalized to the time for the naïve checkpointing approach (1x).*

App.	Strategy	Ckpt Time	Ckpt Size
HPGMG-FV	Gzip	0.78x	14 MB
HPGMG-FV	Parallel gzip	0.60x	14 MB
HPGMG-FV	LZ4	0.30x	16 MB
HPGMG-FV	Forked ckpting	0.025x	113 MB
HYPRE	Gzip	2x	176 MB
HYPRE	Parallel gzip	1x	176 MB
HYPRE	LZ4	1x	296 MB
HYPRE	Forked ckpting	0.032x	868 MB

I/O, i.e., writing the data to the disk, under forked checkpointing, the overhead is dominated by the cost of in-memory data transfers: from the GPU to the proxy process, and from the proxy process's address space to the application process's address space. Further, the cost of quiescing the application process, quiescing the network (for MPI), and "draining" and saving the in-flight network messages is 0.01% of the total cost.

However, unlike the synthetic benchmark, using in-memory compression to reduce the size of data for writing is better in this case for both HPGMG and HYPRE. This indicates that the compression algorithm is able to efficiently reduce the size of the data, which helps lower the I/O overhead. Note that this is still worse than using forked checkpointing by an order of magnitude.

7.2 MANA: Experimental Evaluation

MANA's evaluation is driven by a prototype implementation, which was used to run and checkpoint real-world HPC applications.

This section seeks to answer the following questions:

Q1: What is the runtime overhead of running MPI applications under MANA?

Q2: What are the checkpoint and restart overheads of transparent checkpointing of MPI applications under MANA?

Q3: Can MANA allow transparent switching between MPI implementation across checkpoint-restart for MPI applications for purposes of load balancing?

7.2.1 Setup

We first describe the hardware and software setup for MANA's evaluation.

7.2.1.1 Hardware

The experiments were run on the Cori supercomputer [30] at the National Energy Research Scientific Computing Center (NERSC). As of this writing, Cori is the #12 supercomputer in the Top-500 list [140]. All experiments used the Intel Haswell nodes (Xeon E5-2698 v3) connected via Cray's Aries interconnect network. Checkpoints were saved to the backend Lustre filesystem.

7.2.1.2 Software

Cori provides modules for two implementations of MPI: Intel MPI and Cray MPICH. Cray compilers and Cray MPICH is the recommended way to use MPI, presumably for reasons of performance. Cray MPICH version 3.0 was used for the experiments.

7.2.1.3 Application Benchmarks

MANA was tested with five real-world HPC applications from different computational science domains:

1. GROMACS [13]: Versatile package for molecular dynamics, often used for biochemical molecules.

2. CLAMR [29, 88]): Mini-application for CelL-based Adaptive Mesh Refinement.

3. miniFE [62]: Proxy application for unstructured implicit finite element codes.

4. LULESH [69]: Unstructured Lagrangian Explicit Shock Hydrodynamics

5. HPCG [37] (High Performance Conjugate Gradient): Uses a variety of linear algebra operations to match a broad set of important HPC applications, and used for ranking HPC systems.

7.2.2 Runtime Overhead

7.2.2.1 Real-world HPC Applications

Next, we evaluate the performance of MANA for real-world HPC applications. It will be shown that the runtime overhead is close to 0 % for miniFE and HPCG, and as much as 2 % for the other three applications. The higher overhead has been tracked down to an inefficiency in the Linux kernel [80] in the case of many point-to-point MPI calls (send/receive) with messages of small size. This worst case is analyzed further in Section 7.2.3, where tests with an optimized Linux kernel show a worst case runtime overhead of less than 1 %. The optimized Linux kernel is based on a patch under review for a future Linux version.

Single Node: Since the tests were performed within a larger cluster where the network use of other jobs could create congestion, we first eliminate any network-related overhead by running the benchmarks on a single node with multiple MPI ranks, both under MANA and natively (without MANA). This experiment isolates the single-node runtime overhead due to MANA by ensuring that all communication among ranks is intra-node.

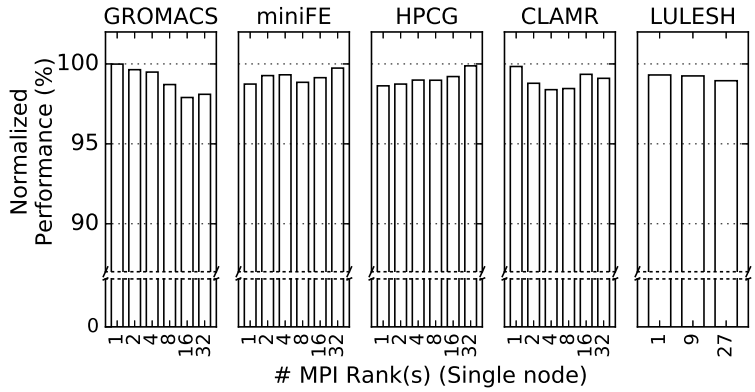


Figure 73: *Single Node: Runtime overhead under MANA for different real-world HPC benchmarks with an unpatched Linux kernel. (Higher is better.)*

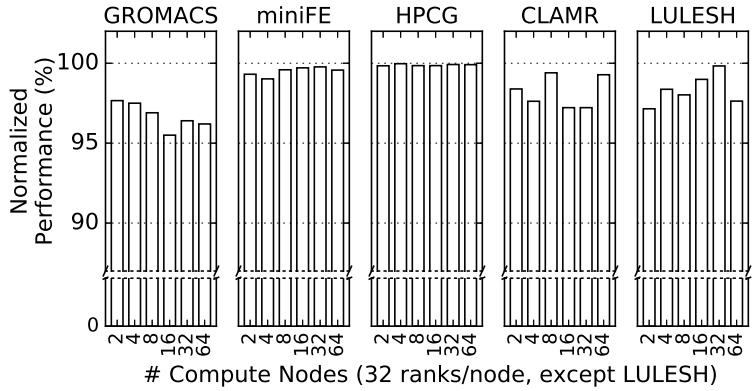


Figure 74: *Multiple Nodes: Runtime overhead under MANA for different real-world HPC benchmarks with an unpatched Linux kernel. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. (Higher is better.)*

Figure 73 shows the results for the five different real-world HPC applications running on a single node under MANA. Each run was repeated 5 times (including the native runs), and the figure shows the mean of the 5 runs. The worst case overhead incurred by MANA is 2.1 % in the case of GROMACS (with 16 MPI ranks). In most cases, the mean overhead is less than 1 %.

Multiple Nodes: Next, the scaling of MANA across the network is examined for up to 64 compute nodes and with 32 ranks per node (except for LULESH, whose configuration restricts the number of ranks/node based on the number of nodes). Hence, the number of MPI ranks ranges from 64 to 2048.

Figure 74 shows the results for the five different real-world HPC applications

running on multiple nodes under MANA. Each run was repeated 5 times, and the mean of 5 runs is reported. We observe a trend similar to the single node case. MANA imposes an overhead of typically less than 2 %. The highest overhead observed is 4.5 % in the case of GROMACS (512 ranks running over 16 nodes). However, see Section 7.2.3 where we demonstrate a reduced overhead of less than 1 % with GROMACS.

7.2.2.2 Memory Overhead

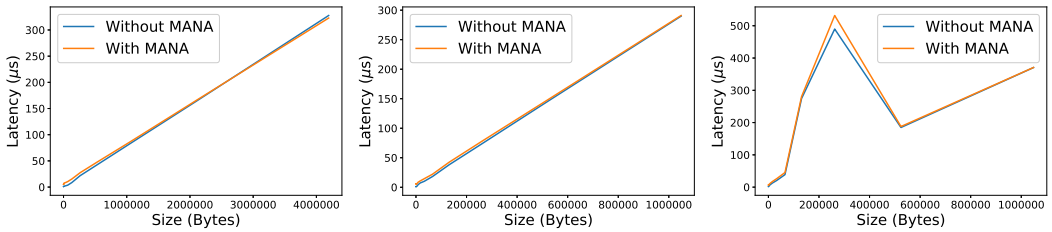
The upper half libraries were built with `mpicc`, and hence include additional copies of the MPI library that are not used. However, the upper half MPI library is never initialized, and so no network library is ever loaded into the upper half.

Since a significant portion of the lower half is comprised only of the MPI library and its dependencies, the additional copy of the libraries (with one copy residing in the upper half) imposes a constant memory overhead. This text segment (code region) was 26 MB in all of our experiments on Cori with the Cray MPI library.

In addition to the code, the libraries (for example, the networking driver library) in the lower half also allocate additional memory regions (shared memory regions, pinned memory regions, memory-mapped driver regions). We observed that the shared memory regions mapped by the network driver library grow in proportion with the number of nodes (up to 64 nodes): from 2 MB (for 2 nodes) to 40 MB for (64 nodes). We expect MANA to have a reduced checkpoint time compared to DMTCP/InfiniBand [24], as MANA omits these regions during checkpointing, reducing the amount of data that's written out to the disk.

7.2.2.3 Microbenchmarks

To dig deeper into the sources for the runtime overhead, we tested MANA with the OSU micro-benchmarks. The benchmarks stress and evaluate the bandwidth and latency of different specific MPI subsystems. Our choice of the specific micro-benchmarks was motivated by the MPI calls commonly used by our real-world MPI



(a) *Point-to-Point Latency* (b) *Collective MPI_Gather* (c) *Collective MPI_Allreduce*

Figure 75: *OSU Micro-benchmarks under MANA. (Results are for two MPI ranks on a single node.)*

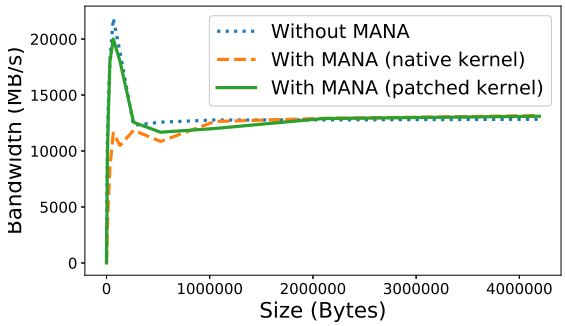


Figure 76: *Point-to-Point Bandwidth under MANA with patched and unpatched Linux kernel. (Higher is better.)*

applications.

Figure 75 shows the results with three benchmarks from the OSU micro-benchmark suite. These benchmarks correspond with the most frequently used MPI subsystems in the set of real-world HPC applications. The benchmarks were run with 2 MPI ranks running on a single compute node.

The results show that latency does not suffer under MANA, for both point-to-point and collective communication. (The latency curves for application running under MANA closely follow the curves when the application is run natively.)

7.2.3 Source of Overhead and Improved Overhead for Patched Linux Kernel

All experiments in this section were performed on a single node of our local cluster, where it was possible to directly install a patched Linux kernel in the bare machine.

Further investigation revealed two sources of runtime overhead. The larger source of overhead is due to the use of the "FS" register during transfer of flow of control between the upper and lower half and back during a call to the MPI library in the lower half. The "FS" register of the x86-64 CPU is used by most compilers to refer to the thread-local variables declared in the source code. The upper and lower half programs each have their own thread-local storage region. Hence, when switching between the upper and lower half programs, the value of the "FS" register must be changed to point to the correct thread-local region. Most Linux kernels today require a kernel call to invoke a privileged assembly instruction to get or set the "FS" register. In 2011, Intel Ivy Bridge CPUs introduced a new, unprivileged FSGSBASE assembly instruction for modifying the "FS" register, and a patch to the Linux kernel [80] is under review to allow other Linux programs to use this more efficient mechanism for managing the "FS" register.

A second (albeit smaller) source of overhead is the virtualization of MPI communicators and datatypes, and recording of metadata for MPI sends and receives. Virtualization requires a hash table lookup and locks for thread safety.

The first and larger source of overhead is then eliminated by using the patched Linux kernel, as discussed above. Point-to-point bandwidth benchmarks were run both with and without the patched Linux kernel (Figure 76). A degradation in runtime performance is seen for MANA for small message sizes (less than 1 MB) in the case of a native kernel. However, the figure shows that the patched kernel yields much reduced runtime overhead for MANA. Note that the Linux kernel community is actively reviewing this patch (currently in its third version), and it is likely to be incorporated in future Linux releases.

Finally, we return to GROMACS, since it exhibited a higher runtime overhead

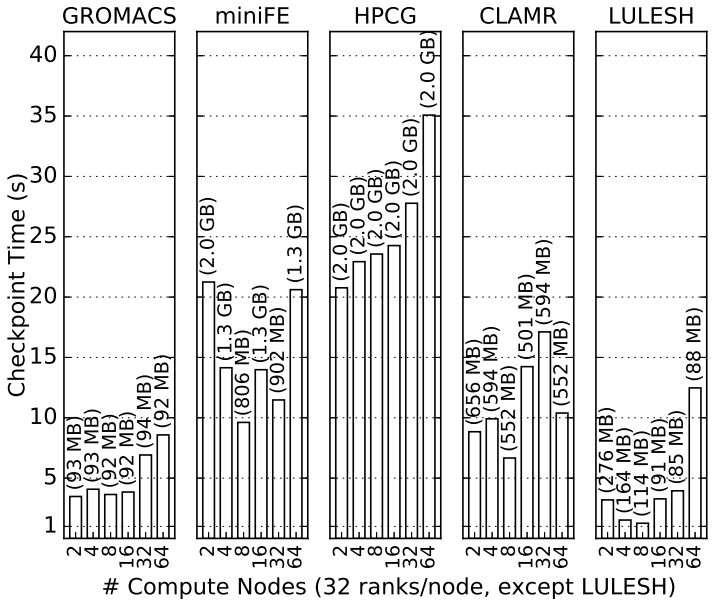


Figure 77: *Checkpointing overhead and checkpoint image sizes under MANA for different real-world HPC benchmarks running on multiple nodes. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. For LULESH, the total number of ranks was either 64 (for 2, 4, and 8 nodes), or 512 (for 16, 32, and 64 nodes). Hence, the maximum number of ranks (for 64 nodes) was 2048. The numbers above the bars (in parentheses) indicate the checkpoint image size for each MPI rank.*

(e.g., 2.1 % in the case of 16 ranks) in many cases. We did a similar experiment, running GROMACS with 16 MPI ranks on a single node with the patched kernel. With the patched kernel, the performance degradation was reduced to less than 1 %.

7.2.4 Checkpoint-restart Overhead

Next, we evaluate MANA's ability to transparently checkpoint-restart different real-world HPC applications.

Figure 77 shows the checkpointing overhead for the five different real-world HPC applications running on multiple nodes under MANA. Each run was repeated 5 times, and the mean of five runs is reported. For each run, we use the `fsync` system call to ensure the data is flushed to the Lustre backend storage.

The total checkpointing data written at each checkpoint varies from 5.9 GB (in the case of 64 ranks of GROMACS running over 2 nodes) to 4 TB (in the case of

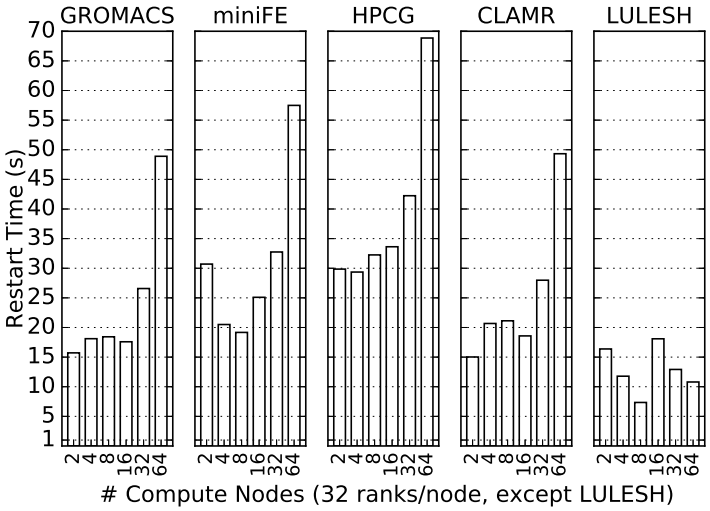


Figure 78: *Restart overhead under MANA for different real-world HPC benchmarks running on multiple nodes. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. Ranks/node is as in Figure 77.*

2048 ranks of HPCG running over 64 nodes). Note that the checkpointing overhead is proportional to the total amount of memory used by the benchmark. This is also reflected in the size of the checkpoint image per MPI rank. While Figure 77 reports the overall checkpoint time, note that there is significant variation in the write times for each MPI rank during a given run. (The time for one rank to write its checkpoint data can be up to 4 times more than that for 90 % of the other ranks.) This phenomenon of stragglers during a parallel write has also been noted by other researchers [8, 145]. Thus, the overall checkpoint time is bottlenecked by the checkpoint time of the slowest rank.

Our next set of questions were: what are the sources of the checkpointing overhead? Does the draining of MPI messages and the two-phase algorithm impose a significant overhead at checkpoint time?

Figure 79 shows the contribution of different components to the checkpointing overhead for the case of 64 nodes for the five different benchmarks. In all the cases, the time to execute the two-phase algorithm (see Section 5.2.5) to ensure that the checkpointing does not occur in the middle of an MPI collective calls was less than 1.6 s.

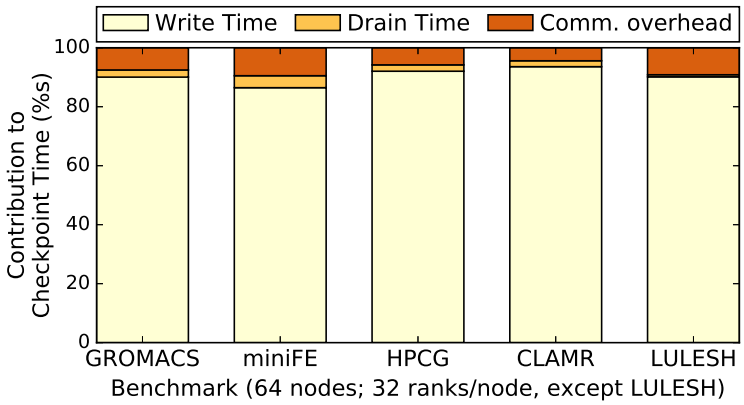


Figure 79: *Contribution of different factors to the checkpointing overhead under MANA for different real-world HPC benchmarks running on 64 nodes. Ranks/node is as in Figure 77. The "drain time" is the delay in starting a checkpoint while MPI message in transit are completed. The communication overhead is the time required in the protocol for network communication between the checkpoint coordinator and each rank.*

In all the cases, the time to drain in-flight MPI messages was less than 0.7 s. The total checkpoint time was dominated by the time to write to the storage system. The next big source of checkpointing overhead was the communication overhead. The current implementation of the checkpointing protocol in DMTCP uses TCP/IP sockets for communication between the MPI ranks and the centralized DMTCP coordinator. The communication overhead associated with the TCP layer is found to increase with the number of ranks, especially due to metadata in the case of small messages that are exchanged between the MPI ranks and the coordinator during checkpoint.

Finally, Figure 78 shows the restart overhead under MANA for the different MPI benchmarks. The restart time varies from less than 10 s to 68 s (for 2048 ranks of HPCG running over 64 nodes). The restart times increase in proportion to the total amount of checkpointing data that is read from the storage. In all the cases, the restart overhead is dominated by the time to read the data from the disk. The time to recreate the MPI opaque identifiers (see Section 5.2.2) is less than 10 % of the total restart time.

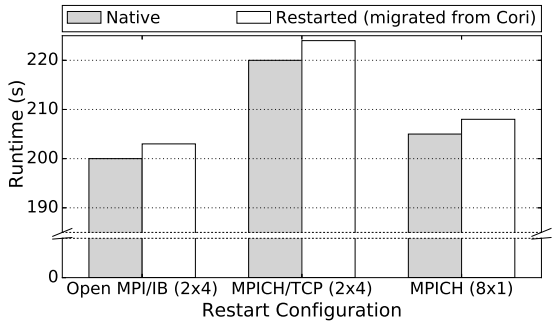


Figure 710: *Performance degradation of GROMACS after cross-cluster migration under three different restart configurations. The application was restarted after being checkpointed at the half-way mark on Cori. (Lower is better.)*

7.2.5 Transparent Switching of MPI libraries across Checkpoint-restart

This section demonstrates that MANA can transparently switch between different MPI implementations across checkpoint-restart. This is useful for debugging programs (even the MPI library) as it allows a program to switch from a production version of an MPI library to a debug version of the MPI library.

The GROMACS application is launched using the production version of CRAY MPI, and a checkpoint is taken 55 s into the run. The computation is then restarted on top of a custom-compiled debug version of MPICH (for MPICH version 3.3). MPICH was chosen because it is a reference implementation whose simplicity makes it easy to instrument for debugging.

7.2.6 Transparent Migration across Clusters

Next, we consider cross-cluster migration for purposes of wide-area load balancing either among clusters at a single HPC site or even among multiple HPC sites. This is rarely done today, since both current vehicles for transparent checkpoint (a checkpoint-restart service for a particular MPI implementation or DMTCP/InfiniBand) save the MPI library within the checkpoint image and continue to use that same MPI library on the remote cluster after migration. At each site and for each cluster, administrators will typically configure and tune a locally recommended

MPI implementation for performance. Migrating an MPI application *along with its underlying MPI library* eliminates the benefits of this local performance tuning.

This experiment showcases the benefits of MPI-agnostic, network-agnostic support for transparent checkpointing. GROMACS is run under MANA, initially running on Cori with a statically linked Cray MPI library running over the Cray Aries network. GROMACS on Cori is configured to run with 8 ranks over 4 nodes (2 ranks per node). Each GROMACS rank is single-threaded. A checkpoint was then taken exactly half way into the run. The checkpoints were then copied (migrated) to a local cluster that uses Open MPI over the InfiniBand network.

The restarted GROMACS under MANA was compared with three other configurations: GROMACS using the local Open MPI, configured to use the local InfiniBand network (8 ranks over 2 nodes); GROMACS/MPICH, configured to use TCP (8 ranks over 2 nodes); and GROMACS/MPICH, running on a single node (8 ranks over 1 node). The network-agnostic nature of MANA allowed the Cori version of GROMACS to be restarted on the local cluster with any of three network options.

We wished to isolate the effects due to MANA from the effects due to different compilers on Cori and the local cluster. In order to accomplish this, the native GROMACS on the local cluster was compiled specially. The Cray compiler of Cori (using Intel's C compiler) was used to generate object files (.o files) on Cori. Those object files were copied to the local cluster. The native GROMACS was then built using the local `mpicc`, but with the (.o files) as input instead of the (.c files). The local `mpicc` linked these files with the local MPI implementation, and the native application was then launched in the traditional way.

Figure 710 shows that GROMACS's performance degrades by less than 1.8% post restart on the local cluster for the three different restart configurations (compared to the corresponding native runs). Also, note that the performance of GROMACS under MANA post restart closely tracks the performance of the native configuration.

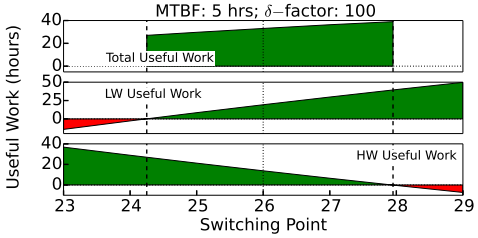


Figure 711: *Shiraz identifies optimal switching point and region of interest. Switching point k varies from 24 to 28 – region of interest (no performance degradation). Shiraz's optimal k = 26. The total runtime is 1000 hours; the δ-factor is 100×; the MTBF is 5 hours.*

7.3 Shiraz: Evaluation

This section presents the evaluation results for Shiraz. In particular, this section answers the following questions:

Q1. *Does an optimal switching point between two applications with different checkpointing overheads exist?*

Q2. *Can Shiraz determine optimal switching point accurately and improve the overall system throughput?*

Q3. *Is Shiraz effective with real-world applications and produce significant energy savings?*

Q4. *Can Shiraz+ reduce the data movement caused by checkpointing under different scenarios? If so, what is the impact on system throughput and application performance?*

Q5. *Are Shiraz and Shiraz+ effective in improving throughput and reducing I/O overhead for representative applications on a real-system?*

Optimal Switching Point: First, we show that an optimal switching point exists, given two applications with different checkpointing overhead, such that the overall useful work is increased without degrading the performance of individual applications. To demonstrate this, we use Figure 711 as an example. Figure 711 illustrates that Shiraz finds an optimal point (i.e., $k = 26$) and Shiraz improves the overall useful work by 33 hours at this optimal point for the two given applications

Table 74: *Shiraz model predicts the optimal switching point correctly across scenarios.*

System Type	δ-factor	Model Optimal Switch Point	Sim Optimal Switch Point
Exascale	5×	6	6
Exascale	25×	13	13
Exascale	100×	26	26
Exascale	1000×	81	79
Petascale	5×	12	11
Petascale	25×	26	24
Petascale	100×	51	51
Petascale	1000×	161	161

with a checkpointing overhead ratio (δ-factor) of 100×. The total runtime is 1000 hours and the MTBF is 5 hours. We use 20 hours and 5 hours MTBF to represent the failure rate of a petascale and exascale systems, respectively [25, 84]. Note that these failure rates are conservative estimates.

For a deeper analysis, Figure 711 also shows region of interest where none of the two applications is being hurt and there is an opportunity for improving the overall system throughput. Simulation results (which can take more than a few hours in some cases) confirm the same optimal point as the model predicts (which takes a few seconds). In fact, Table 74 shows that Shiraz model estimates the same optimal switching point as the simulation across different scenarios — the maximum difference in the estimations is 2, which results in a difference of less than 0.5% in the throughput improvement. The δ-factor is the ratio of checkpointing overheads of the heavy-weight and the light-weight applications (the heavy-weight application's checkpoint takes 30 mins).

In summary, Shiraz model can successfully identify regions of benefit and determine the optimal switching point much more quickly than extensive simulation based method. The next question to investigate is: how does the improvement vary across scenario and the reasons behind that?

Impact of Shiraz on system throughput and individual application perfor-

mance: Next, we demonstrate that Shiraz improves overall system throughput without hurting individual applications' performance for different situations. Figure 712 shows that Shiraz's optimal switching point improves system throughput (overall useful work done per unit time) (a) as the scale of the system changes (MTBF changes), and (b) as the checkpointing overhead ratio between the heavy-weight and light-weight application changes (δ-factor changes). From Figure 712, we make following observations:

(1) Shiraz improves the system throughput in all cases and does not penalize individual applications. In fact, Shiraz improves the performance of individual applications in all cases. In the exascale case, both light-weight and heavy-weight applications on an average observe approximately 14 hours of individual performance improvement on average, leading to an overall average improvement of 28 hours. Therefore, Shiraz improves both system throughput and individual performance (latency).

(2) Shiraz's overall improvement in useful work increases as the δ-factor increases. This is expected since a high δ-factor provides more potential for Shiraz to eliminate lost work. Interestingly, the overall improvement in useful work increases as the MTBF decreases. For example, the overall improvement in useful work increases from 19 hours to 33 hours as the system changes from petascale to exascale when the δ-factor is fixed at 100. Essentially, Shiraz minimizes the lost work due to failures and this opportunity is higher with a low MTBF. This demonstrates that Shiraz will continue to be effective on future systems.

(3) Shiraz's optimal switching point also increases as the checkpointing overhead ratio between the heavy-weight and light-weight application (δ-factor) increases. For example, Figure 712 shows that the switching point increases from 6 to 83 when δ-factor increases from 5 to 1000. This is because the light-weight application is able to perform more checkpoints in the same time period.

Shiraz's optimal switching point also increases with MTBF for a fixed δ-factor factor. For example, Figure 712 shows that that the switching point increases from 6 to 12 when system changes from exascale to petascale. This is because between

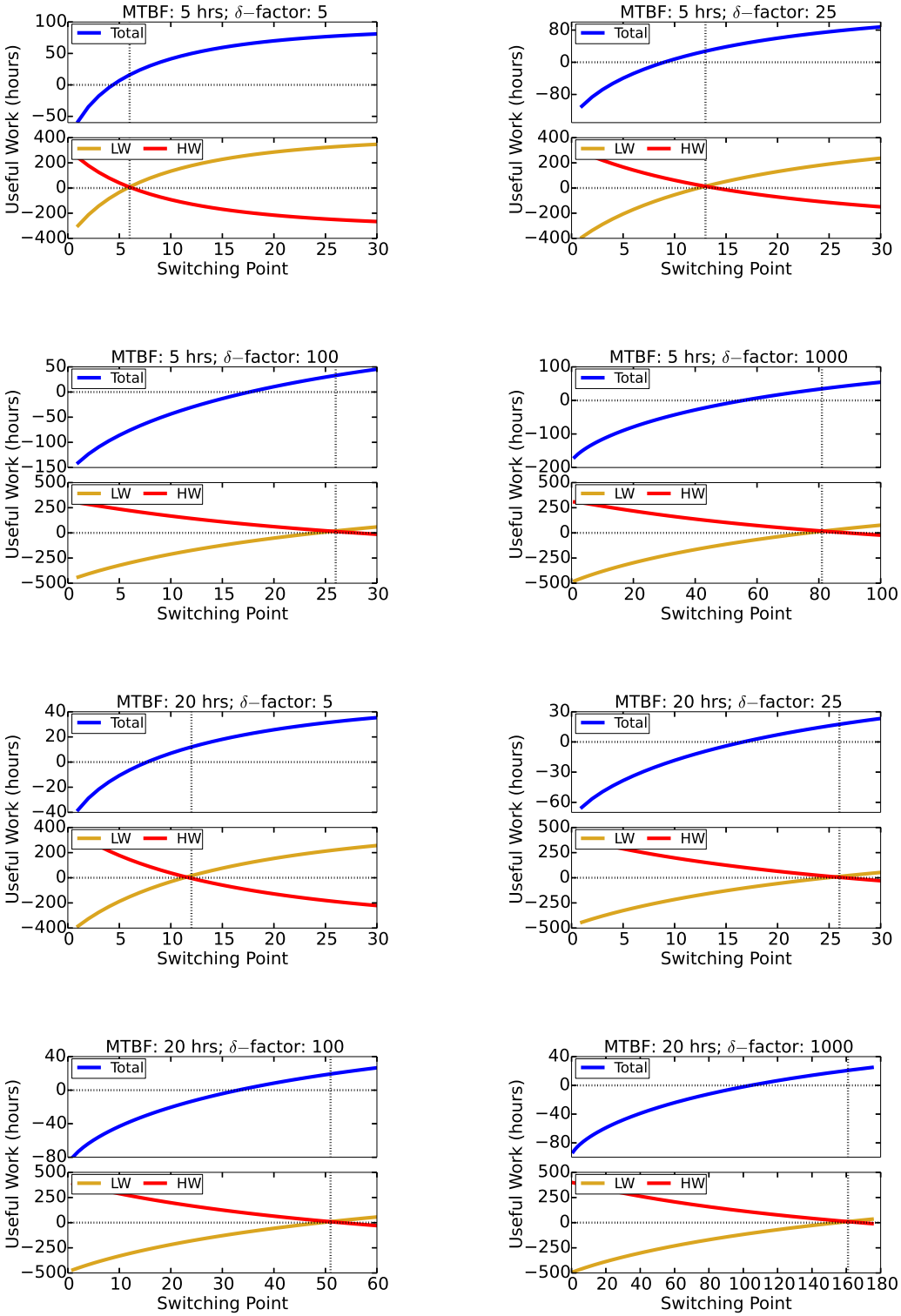


Figure 712: *Shiraz provides improvements across different scenarios. For all the cases, the total runtime is 1000 hours, and the checkpoint duration (δ) of the heavy-weight*

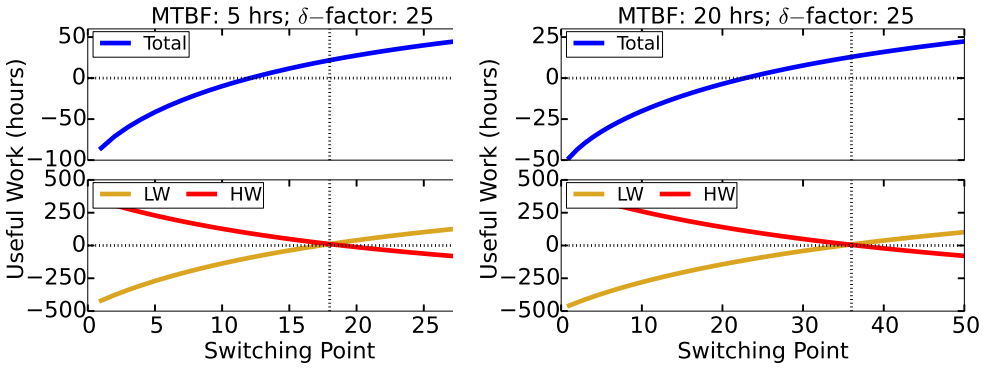


Figure 713: *Shiraz improves throughput across system scale with heavyweight application checkpoint duration (δ) of 0.25 hours.*

two failure points, the hazard rate drops less quickly with a higher MTBF and hence, it is beneficial to run the light-weight application for a longer time.

Finally, we note that Shiraz delivers improvement in overall useful work as the checkpointing overhead of the heavy-weight application varies. We reduce the checkpointing overhead of the heavy-weight application from 0.5 hours to 0.25 hours. Figure 713 shows that the total throughput improvement is 21.8 hours with 5 hours MTBF system and 12.9 hours with 20 hours MTBF system.

Interestingly, our analysis reveals that the optimal switching point is not necessarily half of the MTBF value. As an example, the optimal switch point is 6 when the δ-factor is 5$\times$ and the MTBF is 5 hours (Figure 712). This implies that the switch happens at 6.6 hours, which is higher than the MTBF. Similarly, for the 20 hours MTBF case, the switching happens after 25.2 hours. As discussed in Section 6.2, since the light-weight application observes effectively higher MTBF, it needs to run for a longer duration in the beginning to gain improvement in the overall useful work. A naïve strategy to switch applications at half of the MTBF or slightly a higher value will lead to a significant decrease in the overall useful work. This demonstrates the need and efficacy of Shiraz.

Analysis of impact of Shiraz+ on checkpointing overhead: Next, we evalu-

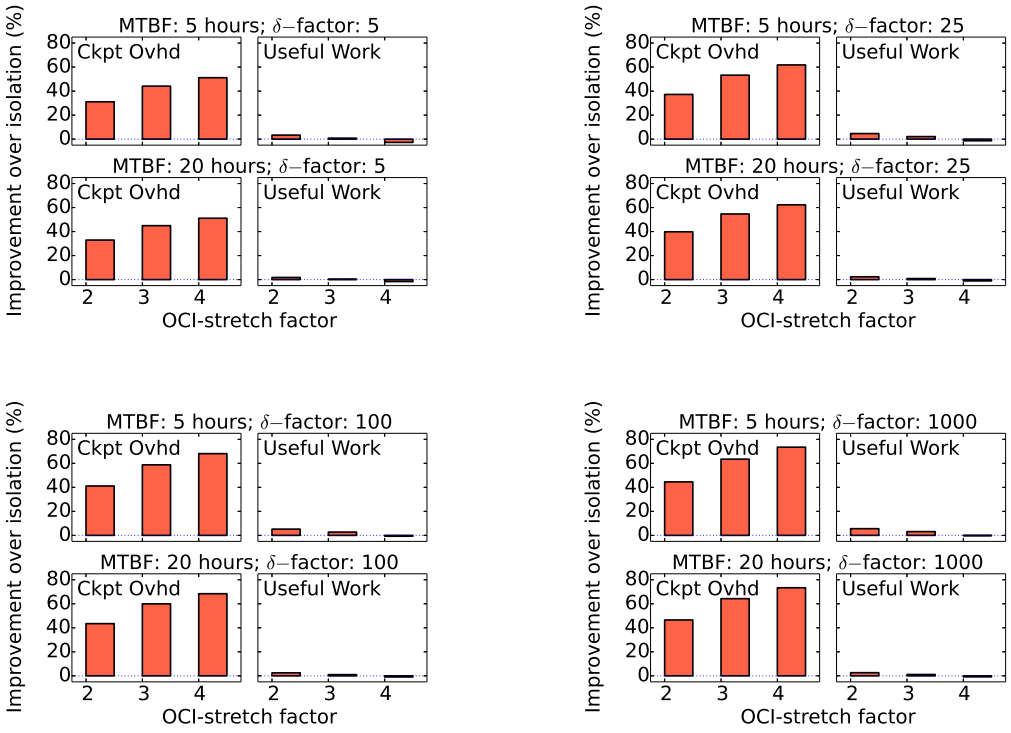


Figure 714: *Impact of Shiraz+ on checkpointing overhead and useful work: checkpointing interval is increased by different factors (2× - 4×) under varying system scale and checkpoint overhead ratios. The checkpoint duration of the heavy weight application is set to be 30 minutes. The baseline refers to switching between applications at every failure.*

ate and analyze the effect of Shiraz+ on the overall checkpointing overhead and throughput. Recall that Shiraz+ increases the checkpointing interval of the heavy-weight application (Section 6.2). Thus, it is intuitive that it will reduce the checkpointing overhead. However, this may also result in a loss of throughput, since the heavy-weight application is no longer operating at its OCI.

Figure 714 shows that when Shiraz+ is applied on top of Shiraz, it significantly reduces the overall checkpointing overhead across different scenarios. Note that Shiraz+ operates at the optimal switching point determined by Shiraz. From Figure 714, we make several observations. First, as the checkpointing interval is stretched from 2× to 4× for the heavy-weight application, the checkpointing overhead reduces drastically. This observation is true across changes in different

parameters: system MTBF, application checkpointing overhead, and δ-factor. The average reduction in checkpointing overhead is approximately 40%. When the OCI-stretch factor is $4\times$, the checkpointing overhead reduces by more than 60% in many cases.

Second, interestingly, while the checkpointing overhead drops significantly, the corresponding performance degradation is minimal. In fact, using a $2\times$ OCI-stretch always keeps a part of the performance improvement obtained by Shiraz; in some cases, the throughput improvement remains up to 5.6% (with no performance degradation for any application). Even with $3\times$ and $4\times$ OCI-stretch factors, the maximum performance degradation across petascale and exascale systems is less than 1.4% and 4.8%, respectively. The underlying insight is that Shiraz schedules the heavy-weight application in a lower failure rate region (i.e., effective higher MTBF) and hence, the effective OCI also increases. We note that Shiraz+ also has the opportunity to use the performance improvement provided by Shiraz and hence, sees no performance degradation in the $2\times$ OCI-stretch case.

In this work, we do not explicitly determine the optimal OCI-stretch factor for different situations, since application programmers and system resource managers are likely to increase the checkpointing interval by an integer factor. Due to practical constraints, many applications do not adopt techniques that alter the checkpointing interval dynamically. In other words, Shiraz+ has chosen to value practical feasibility over theoretical optimum point — which will be an interesting avenue for future work.

Shiraz in multi-application environment and energy savings: Shiraz can determine the optimal switching point between two given applications and improve the overall system throughput. The next question is: can Shiraz scale and be effective in the presence of multiple applications? Fortunately, it turns out that Shiraz can be naturally scaled to the multiple applications scenario. It can be achieved in multiple possible ways. One easy way to achieve this is to make pairs of applications with different checkpointing overheads and run one such pair between two failures

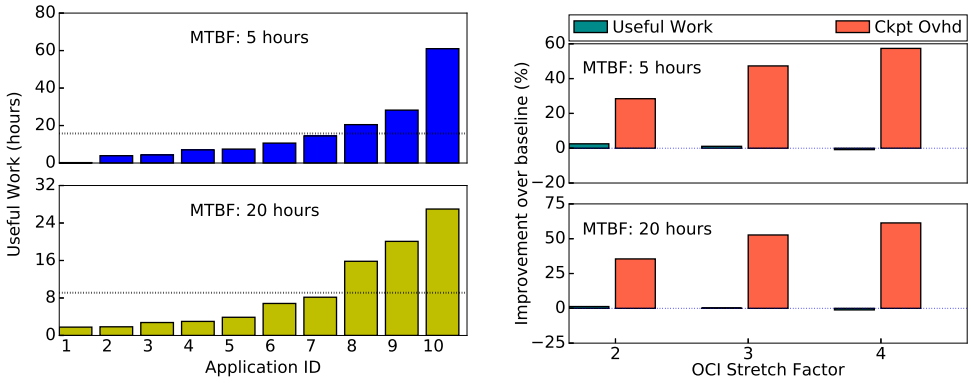


Figure 715: *Shiraz provides improvement in real-world multi-application mix selected from Table 21 and simulated for year-long time period (left). The horizontal lines denotes the average improvement in useful work per application. Shiraz+ decreases checkpointing overhead significantly for the same mix of applications (right).*

using Shiraz, and switching to a different pair after every failure. Optimal strategy to make such pairs is to combine the application with the highest checkpointing overhead with the application with the lowest checkpointing overhead, until we exhaust the available applications. The theoretical proof is not provided for brevity; the intuition behind such a strategy is simple: it maximizes the average of the ratios of checkpointing overheads. We also experimented with another strategy: making random pairs. We found that while it may not deliver the maximum possible improvement, it is relatively easier to implement.

To evaluate Shiraz in a multi-application environment, we experimented with the latter strategy using 10 applications and noted the corresponding throughput gains. The application list is composed from the real-world application characteristics from Table 21. We used the Shiraz model to obtain the optimal switch point for the different application pairs, and simulated the scenario where these applications ran for one calendar year (8,700 hours).

Figure 715 (left) shows the overall system throughput improvement and impact on individual job performance for all the 10 applications. We make a few interesting observations. First, no application suffers a performance degradation, and the average throughput improvement is 15 hours. Second, Shiraz improves the total

useful work by approx. 91 hours and 157 hours for the petascale and the exascale systems, respectively.

Our results demonstrate that Shiraz+ is also effective in the multi-application scenario. Figure 715 (right) shows that Shiraz+ (with 3× OCI stretch factor) decreases the checkpointing overhead by up to 52%, without incurring any loss in the overall system throughput for both exascale and petascale systems. When the OCI stretch factor is increased to 4×, only then the system incurs degradation (less than 1%) the total useful work, while the checkpointing overhead decreases by up to 60%.

To show the results in a conservative scenario, we conduct an experiment with 40 jobs, with 5 heavy-weight applications, and the rest 35 light-weight applications. The 35 light-weight applications are selected at random from the three least heavy applications from Table 21. Shiraz improves the total useful work done 57 hours and 89 hours for the petascale and the exascale systems, respectively.

Finally, we evaluate the potential energy savings enabled by Shiraz for the exascale (5 hours MTBF) and the petascale systems (20 hours MTBF). Since Shiraz increases the useful work done per unit time at the whole system level, it effectively saves energy that would have been spent on lost work (due to failures). In order to simplify the evaluation and interpretation, we estimate the yearly energy savings. Taking a conservative electricity rate of $0.1 per kW-Hour [1], the energy and monetary savings on the exascale (5 hours MTBF and 20MW power consumption) system would translate to 1.78 MW-Hour and $178,000 per year, respectively. For the petascale (20 hours MTBF and 10MW power consumption) system, the energy and monetary savings would translate to 0.57 MW-Hour and $57,000 per year.

These savings could be invested towards faster storage systems and more computing power in the future — which would further increase the profits due to faster completion times. For the petascale system, the cost savings due to energy expenditure cuts enabled by Shiraz translate to $285,000 over 5 years (anticipated lifetime of a system). At 0.2 GB/USD for SSD-based burst buffers [3, 4] (the total cost of infrastructure pessimistically assumed to be 3× of the hardware cost due to

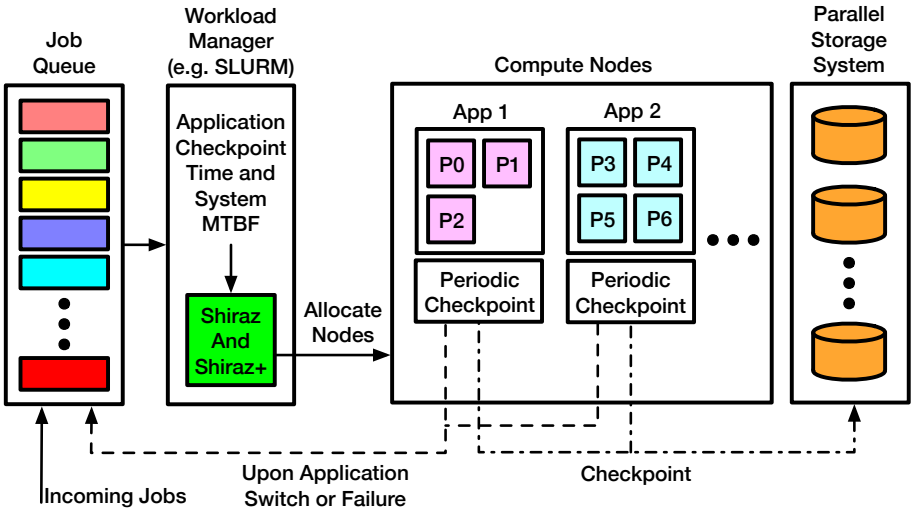


Figure 716: *Prototype of Shiraz and Shiraz+.*

packaging, assembly, firmware and integration cost), the monetary savings could pay for 5.7% of the cost of the burst buffers (0.285M USD out of 5M USD) for the petascale system, with 1 PB of storage. For the exascale system, the cost savings enabled by Shiraz would amount to $890,000 over 5 years. We note that this analysis is on the conservative side, as it does not include the energy cost reduction due to the reduction in data movement enabled by Shiraz+.

We note that in a multi-application environment, Shiraz can produce different individual performance improvements for the same application depending upon on the pairing and application-mix since the runtime improvement provided by Shiraz depends on the δ-factor. This can possibly lead to small amount of unpredictability in the runtime, although Shiraz will improve the individual runtime in all such cases. Improving predictability in a dynamic application-mix will be a worthy goal for future works.

In summary, our results show that Shiraz leads to significant energy and monetary saving for real-world applications that can act as positive feedback loop and result in compounded returns over years.

Prototype implementation and evaluation of Shiraz and Shiraz+ using system-level checkpointing: We developed a prototype of Shiraz and Shiraz+ to evalu-

ate its effectiveness on real-world applications. We developed a scheduler plug-in that implements the core scheduling algorithm of Shiraz and Shiraz+. It maintains records of the checkpointing overhead for different applications, temporal characteristics of system failures, and takes checkpoints using a system-level checkpointing package, and schedules applications based on the Shiraz model. To demonstrate the effectiveness, we evaluated the prototype using two real-world HPC applications: Co-Design Molecular Dynamics Proxy (CoMD) [85] and Finite Element Solver (miniFE) [62]. CoMD represents a variety of scientific applications including SPaSM, and miniFE is an approximation of unstructured finite element and finite volume codes including HPCCG and pHPCCG. We used DMTCP [7], a system-level checkpointing library, to perform checkpoints, and the optimal switch point was decided based on the checkpointing overhead obtained experimentally. We note that our plugin is not tied to a particular implementation of checkpointing library and can be ported across systems and resource managers (e.g., SLURM) (schematic shown in Figure 716). The ratio of the checkpointing overhead of miniFE (heavyweight application) to that of CoMD (lightweight application) is 30x, as experimentally measured using DMTCP.

Statistically sound evaluation of such a prototype implementation is challenging since it requires dedicated time (in order of months) on a large-scale supercomputer. To address this challenge, we emulated the setting by feeding a failure trace with the same characteristics as large-scale supercomputers (discussed in Section 2.5) but at a higher frequency. We also scaled down the program input size to ensure that the runs completed on a local cluster within a month. We performed an effectively 200-hour long run by scaling the failure-frequency and program size, and did this run 30 times for each point, to obtaining stable results. We injected errors in the local cluster that crash the application and used checkpoints to recover from errors without any human intervention during the experiments. At the end of run, we collected runtime statistics (useful work, checkpoint overhead, and lost work) to compare Shiraz with the baseline.

We found that Shiraz results in 10.2% more useful work system-wide using

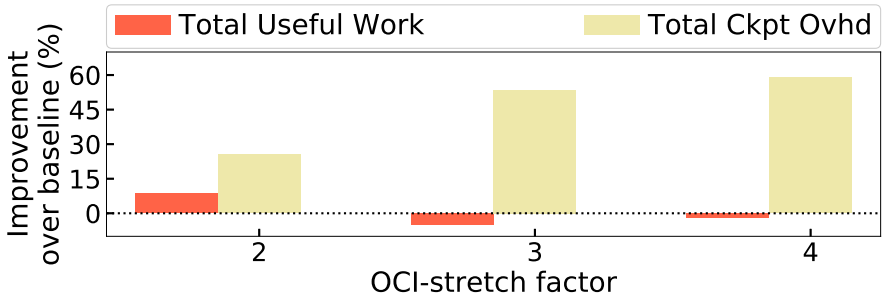


Figure 717: *Impact of Shiraz+ on CoMD and miniFE application performance and checkpointing overhead.*

CoMD and miniFE application, compared to the baseline case, where applications are switched at every failure. Since these experiments take prohibitively long, we did not explore the optimal switching point using experiments. Instead, we used the Shiraz model to obtain optimal point offline and results show improvements.

We also evaluated Shiraz+ using this prototype. Figure 717 shows that Shiraz+ reduces the checkpointing overhead significantly with minimal or no performance degradation. For example, the overall checkpointing overhead is reduced by approximately 35.8% when using a 2× OCI-stretch factor, while still maintaining the overall improvement in useful work at approximately 7%. When Shiraz+ applies 3× and 4× OCI-stretch, the overall checkpointing overhead is reduced by 69.6% and 77.6%, respectively, while the performance degradation is under 3%. Overall, the evaluation shows that when operating at the optimal switching point obtained by Shiraz model, Shiraz+ is effective in reducing the data movement caused by checkpointing and still retains some of the performance benefits provided by Shiraz.

CHAPTER 8

Impact of this Thesis for the Future

In this chapter, we discuss some of the future research work that can be pursued based on this dissertation.

8.1 Debugging of Distributed Processes

While the methodology described in Chapter 3 is based on loading two programs in one process's address space, the split-process approach can be generalized to any number of programs. The only constraint is the available virtual address space.

This can open up new ways of tracing and debugging distributed programs when combined with checkpoint-restart. A distributed application running at full, production scale could be checkpointed, migrated to a single computer, and restarted as a single-process, multi-threaded program. The restart process would restore the separate programs in different memory sections and start a single thread (or multiple threads) corresponding to the memory sections. An additional section with an ephemeral communication library could be used to simulate the network.

8.2 Dynamic Load Balancing for MPI

The split-process approach of MANA also opens up some important new features in managing long-running MPI applications. An immediately obvious feature is the

possibility of switching *in the middle of a long run* to a customized MPI implementation. Hence, one can dynamically substitute a customized MPI for performance analysis (e.g., using PMPI for profiling or tracing; or using a specially compiled "debug" version of MPI to help developers understand an unusual bug in the MPI library that occurs only in the middle of a long run).

MANA also helps support many tools and proposals for optimizing MPI applications. For example, a trace analyzer is sometimes used to discover communication hotspots and opportunities for better load balancing. Such results are then fed back by re-configuring the binding of MPI ranks to specific hosts in order to better fit the underlying interconnect topology.

Currently, such bindings of MPI ranks are chosen statically and used for the life of the MPI application run. But MANA allows one to dynamically re-bind MPI ranks in the middle of a long run to create new configurations of rank-to-host bindings (new topology mappings). This is useful either when the MPI application enters a new phase for which a different rank-to-host binding is optimal, or else when other codes that run on the same cluster begin to create contention or interference through communication hotspots. This will enable researchers to leverage tools [19, 102] for online dynamic monitoring and dynamic performance engineering by creating new topology mappings for rank-to-host bindings.

For dynamic performance engineering, MANA can also co-locate arbitrary MPI ranks onto the same host, where they will benefit from MPI library optimizations such as shared memory for improved communication. Under older approaches to transparent checkpoint-restart, this was impossible, since the older approaches were saving all of process memory, including the shared memory regions created by the MPI library.

MANA can enable new approaches to dynamically load balance by checkpointing on one cluster and restarting on a different cluster. This added flexibility allows system managers to burst current long-running applications into the Cloud during periods of heavy usage. At the same time, the restarted application benefits from the locally configured MPI on the new cluster, which has been optimized for that

cluster's topology (e.g., through topology mappings).

Finally, MANA can enable a new class of very long-running MPI applications — ones which may outlive the lifespan of the original MPI Implementation, cluster, or even the network interconnect. Such temporally complex computations might be discarded as infeasible today without the ability to migrate MPI implementations or clusters.

8.3 Transparent Checkpointing for Large-scale HPC

Traditionally, the checkpointing decisions (such as the checkpointing interval) have been made in isolation by individual applications. Previous studies have demonstrated different methodologies for making these decisions in an optimal way.

Shiraz presents the first solution that uses the variations in application checkpointing characteristics for improving large-scale system throughput. An individual application's checkpointing interval is not changed and is kept constant. However, this can result in unpredictable performance for an individual application, since its performance relies on the optimal switching point, which, in turn, depends on the other application it gets paired with.

A future possibility would be design a solution where the HPC job scheduler can make checkpointing interval decisions for individual applications by using the global knowledge of the system (e.g., system MTBF) while guaranteeing predictable performance. This could be enabled, for example, by fixing the run time for each application and running the application during that time with an updated checkpointing interval. The checkpointing interval could be transparently updated to a constant value that depends on the global system state (failure distribution, contention scenarios, and so on).

CHAPTER 9

Conclusion

Transparent checkpointing is a critical fault-tolerance technology for large-scale HPC. Previous work in this domain has failed to address some of the key challenges for enabling transparent checkpointing for modern HPC applications, which no longer run in an isolated environment. They often rely on modern subsystem such as shared-memory, device drivers for hardware accelerators, and a variety of high-throughput, low-latency networks. HPC applications also run in a highly shared environment, and thus, require efficient scheduling schemes to manage system throughput.

This dissertation demonstrates a general framework for transparent checkpointing using split processes. This framework provides isolation between the application process and the external resource. This simplifies the problem of checkpointing to the extent that previously available single-process checkpointing solutions can be used for checkpointing modern HPC applications.

This approach was successfully applied to two different HPC domains: for checkpointing modern CUDA-based HPC applications; and for checkpointing MPI-based applications.

The solution for checkpointing CUDA supports modern CUDA with unified virtual address space. This is a critical technology for the exascale, as it relieves the programmer from the burden of explicitly managing memory on the host CPU and the GPU device.

The solution for checkpointing MPI provides an MPI-agnostic and network-agnostic checkpointing solution. The agnostic properties were shown to enable migration across HPC clusters, moving between different MPI implementations, and networking topology and technologies. This can enable flexible dynamic load balancing in the future.

Finally, a solution for improving throughput for large-scale HPC was also presented. The solution exploits the variation in checkpointing overheads of applications running in an HPC center and the failure recurrence behavior in the HPC center for improving the system throughput. The solution schedules applications based on their checkpointing overheads at different time zones of varying failure rate to improve system throughput. A variation of the solution is shown to reduce the checkpointing overhead, without sacrificing on the individual application performance.

Bibliography

[1] EIA - Electricity Data. https://tinyurl.com/ya6o3eas. [Online; accessed 04-Dec-2017]. (Cited on page 108.)

[2] Facebook: Virtualisation does not scale. https://www.zdnet.com/article/facebook-virtualisation-does-not-scale/. [Online; accessed 11-Jul-2018]. (Cited on page 8.)

[3] Intel DC P3608 SSDPECME040T401. https://tinyurl.com/ybng8ll3. [Online; accessed 04-Dec-2017]. (Cited on page 108.)

[4] Samsung PM1725a Series 1.6TB TLC. https://tinyurl.com/yd8rcy55. [Online; accessed 04-Dec-2017]. (Cited on page 108.)

[5] High-performance Geometric Multigrid, an HPC Benchmark and Supercomputing Ranking Metric, 2016. [Online; accessed 28-Mar-2018]. (Cited on page 81.)

[6] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006. (Cited on page 35.)

[7] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2009. (Cited on pages 8, 9, 60, 80, and 110.)

[8] Kapil Arya, Rohan Garg, Artem Y Polyakov, and Gene Cooperman. Design and implementation for checkpointing of distributed resources using process-level virtualization. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*, pages 402–412. IEEE, 2016. (Cited on pages 9, 10, 60, 61, 80, and 96.)

[9] L. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large-scale Systems. In *CCGrid 2010*, pages 63–72. IEEE Computer Society, 2010. (Cited on page 77.)

[10] Leonardo Bautista-Gomez et al. Reducing Waste in Extreme Scale Systems Through Introspective Analysis. In *IPDPS 2016*, pages 212–221. IEEE, 2016. (Cited on pages 14 and 75.)

[11] Anne Benoit, Aurélien Cavelan, Valentin Le Fèvre, Yves Robert, and Hongyang Sun. Towards Optimal Multi-level Checkpointing. *IEEE Trans. Comput*, 66(7):1212–1226, 2017. (Cited on page 77.)

[12] John Bent, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. Storage Challenges at Los Alamos National Lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5. IEEE, 2012. (Cited on page 12.)

[13] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1):43 – 56, 1995. (Cited on page 90.)

[14] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008. (Cited on page 12.)

[15] R. Birke, I. Giurgiu, L. Y Chen, D. Wiesmann, and T. Engbersen. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *DSN 2014*, pages 1–12. IEEE, 2014. (Cited on page 76.)

[16] Berkeley lab checkpoint/restart for Linux (BLCR) downloads. http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/berkeley-lab-checkpoint-restart-for-linux-blcr-downloads/, 2019. [Online; accessed Jan., 2019]. (Cited on page 10.)

[17] BLCR admin guide — version 0.8.5. https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html, 2019. [Online; accessed Jan., 2019]. (Cited on page 10.)

[18] Shekhar Borkar. The Exascale Challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2–3. IEEE, 2010. (Cited on page 12.)

[19] George Bosilca, Clément Foyer, Emmanuel Jeannot, Guillaume Mercier, and Guillaume Papauré. Online dynamic monitoring of MPI communications. In *European Conference on Parallel Processing (Euro-Par'18)*, pages 49–62. Springer, 2017. (Cited on page 113.)

[20] Mohamed Slim Bouguerra et al. Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing. In *IPDPS 2013*, pages 501–512. IEEE, 2013. (Cited on page 77.)

[21] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: A multiprotocol automatic fault tolerant MPI. *International Journal of High Performance Computing Applications*, 20:319–333, 2006. (web site at http://mpich-v.lri.fr/, accessed Jan., 2019). (Cited on page 9.)

[22] Edouard Bugnion, Vitaly Chipounov, and George Candea. Lightweight Snapshots and System-Level Backtracking. In *Proceedings of the 14th Workshop on Hot Topics on Operating Systems*, number EPFL-CONF-185945. USENIX, 2013. (Cited on page 38.)

[23] Jiajun Cao. *Transparent Checkpointing over RDMA-based Networks*. PhD thesis, Northeastern University, 2017. (Cited on pages 10, 11, 45, and 50.)

[24] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. Transparent Checkpoint-Restart over InfiniBand. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 13–24. ACM, 2014. (Cited on pages 10, 11, 16, and 92.)

[25] Franck Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *IJHPCA*, 23(3):212–226, 2009. (Cited on pages 12, 75, and 101.)

[26] CFDR Data. https://tinyurl.com/yd6ornwa. [Online; accessed 28-Nov-2017]. (Cited on page 13.)

[27] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the International Symposium on Workload Characterization*, pages 44–54, 2009. (Cited on page 80.)

[28] Ron C. Chiang, H. Howie Huang, Timothy Wood, Changbin Liu, and Oliver Spatscheck. IOrchestra: Supporting High-performance Data-intensive Applications in the Cloud via Collaborative Virtualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 45:1–45:12, New York, NY, USA, 2015. ACM. (Cited on page 8.)

[29] CLAMR source code. https://github.com/lanl/CLAMR, 2019. [Online; accessed Jan., 2019]. (Cited on page 90.)

[30] Cori supercomputer at NERSC. http://www.nersc.gov/users/computational-systems/cori/, 2019. [Online; accessed Jan., 2019]. (Cited on page 89.)

[31] Christopher S Daley, Devarshi Ghoshal, Glenn K Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas J Wright. Performance characterization of scientific workflows for the optimal use of burst buffers. *Future Generation Computer Systems*, 2017. (Cited on page 2.)

[32] John T Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006. (Cited on pages 5 and 77.)

[33] Nathan DeBardeleben, Sean Blanchard, Laura Monroe, Phil Romero, Daryl Grunau, Craig Idler, and Cornell Wright. GPU Behavior on a Large HPC Cluster. In *Euro-Par 2013: Parallel Processing Workshops*, pages 680–689, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (Cited on pages 12 and 29.)

[34] S. Di, M. S Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. In *IPDPS 2014*, pages 1181–1190. IEEE, 2014. (Cited on page 77.)

[35] Sheng Di, Yves Robert, Frédéric Vivien, and Franck Cappello. Towards an Optimal Online Checkpoint Solution Under a Two-level HPC Checkpoint Model. *TPDS 2017*, 28(1):244–259, 2017. (Cited on page 77.)

[36] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at Petascale: The case of Blue Waters. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014. (Cited on page 29.)

[37] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. A new metric for ranking high-performance computing systems. *National Science Review*, 3(1):30–35, 2016. (Cited on page 90.)

[38] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation*, pages 224–231, June 2010. (Cited on page 39.)

[39] N. El-Sayed and B. Schroeder. Reading Between the Lines of Failure Logs: Understanding How HPC Systems Fail. In *DSN 2013*, pages 1–12. IEEE, 2013. (Cited on pages 14 and 76.)

[40] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for Peta-scale Systems: A Look into the Future of Practical Rollback-Recovery. *TDSC 2004*, 1(2):97–108, 2004. (Cited on pages 2, 12, and 64.)

[41] Kurt Ferreira et al. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *SC 2011*, page 44. ACM, 2011. (Cited on pages 12 and 64.)

[42] Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, and Ron Brightwell. Accelerating incremental checkpointing for extreme-scale computing. *Future Generation Computer Systems*, 30:66–77, 2014. (Cited on page 77.)

[43] Fernanda Foertter. Preparing GPU-accelerated applications for the Summit supercomputer. GPU Technology Conference (GTC), May 2017. (Cited on page 27.)

[44] Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale HPC systems. In *IPDPS 2012*, pages 1168–1179. IEEE, 2012. (Cited on page 76.)

[45] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *PP '06: Proceedings of the 2006 Int. Conf. on Parallel Processing (ICPP'06)*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 9.)

[46] Rohan Garg, Kapil Arya, Jiajun Cao, Gene Cooperman, Jeff Evans, Ankit Garg, Neil A Rosenberg, and K Suresh. Adapting the DMTCP Plugin Model for Checkpointing of Hardware Emulation. *arXiv preprint arXiv:1703.00897*, 2017. (Cited on page 7.)

[47] Rohan Garg, Jiajun Cao, Kapil Arya, Gene Cooperman, and Jérôme Vienne. Extended Batch Sessions and Three-Phase Debugging: Using DMTCP to Enhance the Batch Environment. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, page 42. ACM, 2016. (Cited on page 7.)

[48] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018. (Cited on pages 42, 44, 47, and 62.)

[49] Rohan Garg, Komal Sodha, Zhengping Jin, and Gene Cooperman. Checkpoint-restart for a network of virtual machines. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013. (Cited on page 8.)

[50] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. On the scalability, performance isolation and device driver transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 1041–1050. IEEE, 2016. (Cited on page 62.)

[51] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *EUROPAR*, pages 379–391. Springer, 2010. (Cited on page 39.)

[52] L Bautista Gomez, Akira Nukada, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Transparent Low-overhead Checkpoint for GPU-accelerated Clusters, 2010. [Online; accessed 16-Mar-2018]. (Cited on pages 12, 25, and 40.)

[53] Berkin Guler and Oznur Ozkasap. Compressed Incremental Checkpointing for Efficient Replicated Key-value Stores. In *ISCC 2017*, pages 76–81. IEEE, 2017. (Cited on page 77.)

[54] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative i/o workload characterization of two leadership class storage clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015. (Cited on page 2.)

[55] R. Gupta, P. Beckman, B.H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *38th Int. Conf. on Parallel Processing (ICPP'09)*, September 2009. (web site at `https://wiki.mcs.anl.gov/cifts/index.php/CIFTS`, accessed Jan., 2019). (Cited on page 9.)

[56] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proc. of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009. (Cited on pages 12, 25, 39, and 40.)

[57] Imran S. Haque and Vijay S. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *CCGRID*, pages 691–696, May 2010. (Cited on pages 12 and 29.)

[58] Paul H Hargrove and Jason C Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006. (Cited on pages 8 and 10.)

[59] Mark Harris. Unified memory in CUDA 6. NVIDIA Blog, 2013. [Online; accessed 17-Jan-2018]. (Cited on page 27.)

[60] Mark Harris. CUDA 8 features revealed. NVIDIA Blog, 2016. [Online; accessed 17-Jan-2018]. (Cited on page 27.)

[61] Mark Harris. Unified memory for CUDA beginners. NVIDIA Blog, 2016. [Online; accessed 18-Jan-2018]. (Cited on pages 26 and 28.)

[62] M Heroux and S Hammond. MiniFE: Finite element solver. `https://tinyurl.com/y7hslf65`, 2019. [Online; accessed Jan 2019]. (Cited on pages 90 and 110.)

[63] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. Process-in-process: Techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 131–143, New York, NY, USA, 2018. ACM. (Cited on pages 22 and 63.)

[64] John Hubbard. Using HMM to blur the lines between CPU and GPU programming. GPU Technology Conference (GTC), May 2017. (Cited on page 38.)

[65] Joshua Hursey, Timothy I Mattox, and Andrew Lumsdaine. Interconnect Agnostic Checkpoint/Restart in OpenMPI. In *Proceedings of the 18th ACM*

international symposium on High performance distributed computing, pages 49–58. ACM, 2009. (Cited on pages 9, 10, 16, and 62.)

[66] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proc. of 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) / 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society, March 2007. (Cited on page 9.)

[67] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance Analysis of Cloud Computing Services for Many-tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed systems*, 22(6):931–945, 2011. (Cited on page 8.)

[68] Sudarsun Kannan, Naila Farooqui, Ada Gavrilovska, and Karsten Schwan. HeteroCheckpoint: Efficient Checkpointing for Accelerator-based Systems. In *DSN*, pages 738–743. IEEE, 2014. (Cited on page 40.)

[69] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013. (Cited on page 90.)

[70] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. In *ACM Sigplan Notices*, volume 41, pages 207–218. ACM, 2006. (Cited on page 39.)

[71] Roy Kim. NVIDIA DGX SATURNV ranked world's most efficient supercomputer by wide margin. NVIDIA Blog, 2016. (Cited on page 27.)

[72] Youngjae Kim and Raghul Gunasekaran. Understanding i/o workload characteristics of a peta-scale storage system. *The Journal of Supercomputing*, 71(3):761–780, 2015. (Cited on page 2.)

[73] Kothe, Douglas B. Exascale Applications: Opportunities and Challenges. Presentation to the Advanced Scientific Computing Advisory Committee (ASCAC), September 2016. [Online; accessed 28-Mar-2018]. (Cited on page 81.)

[74] H. Andrés Lagar-Cavilla, Niraj Tolia, Mahadev Satyanarayanan, and Eyal De Lara. VMM-independent graphics acceleration. In *Proc. of the 3rd Int. Conf. on Virtual Execution Environments*, pages 33–43. ACM, 2007. (Cited on page 39.)

[75] Leslie Lamport. Specifying concurrent systems with TLA$^+$. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:183–250, 1999. (Cited on page 60.)

[76] Large Scale Computing and Storage Requirements for Biological and Environmental Science: Target 2017. Technical Report LBNL-6256E, LBNL, 2012. (Cited on page 14.)

[77] Large Scale Production Computing and Storage Requirements for High Energy Physics: Target 2017. Technical report, LBNL, 2012. (Cited on page 14.)

[78] Lawrence Berkeley National Laboratory (LBL). Hpgmg: High-performance geometric multigrid, 2017. [Online, accessed 17-Jan-2018]. (Cited on page 81.)

[79] Libfabric. https://ofiwg.github.io/libfabric/, 2019. [Online; accessed Jan., 2019]. (Cited on page 11.)

[80] x86: Enable fsgsbase instructions. https://lwn.net/Articles/769355/, 2018. [Online; accessed Jan., 2019]. (Cited on pages 90 and 94.)

[81] Ning Liu et al. On the Role of Burst Buffers in Leadership-class Storage Systems. In *MSST 2012*, pages 1–11. IEEE, 2012. (Cited on pages 15 and 77.)

[82] Yudan Liu et al. An optimal checkpoint/restart model for a large-scale high performance computing system. In *IPDPS 2008*, pages 1–9. IEEE, 2008. (Cited on page 77.)

[83] Lawrence Livermore National Laboratory (LLNL). Hypre: Scalable linear solvers and multigrid methods, 2017. [Online, accessed 17-Jan-2018]. (Cited on page 81.)

[84] Robert Lucas. Top Ten Exascale Research Challenges. In *DOE ASCAC Subcommittee Report*, 2014. (Cited on pages 12 and 101.)

[85] Jamaludin Mohd-Yusof, S Swaminarayan, and TC Germann. Co-Design for Molecular Dynamics: An Exascale Proxy Application, 2013. (Cited on page 110.)

[86] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC 2010*, pages 1–11. IEEE, 2010. (Cited on page 77.)

[87] A. Namazi, M. Abdollahi, S. Safari, S. Mohammadi, and M. Daneshtalab. Reliability-aware task scheduling using clustered replication for multi-core real-time systems. In *NoCArc 2016*, pages 45–50. ACM, 2016. (Cited on page 77.)

[88] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. 2012. (Cited on page 90.)

[89] Bogdan Nicolae and Franck Cappello. AI-Ckpt: Leveraging Memory-access Patterns for Adaptive Asynchronous Incremental Checkpointing. In *HPDC 2013*, pages 155–166. ACM, 2013. (Cited on page 77.)

[90] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *Proceedings of the International Symposium on Parallel and Distributed Processing Workshops*

and PhD Forum, pages 104–113. IEEE, 2011. (Cited on pages 12, 25, 37, 39, 40, and 81.)

[91] NVIDIA Tesla P100—the most advanced data center accelerator ever built. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016. (Cited on page 29.)

[92] NVIDIA. CUDA C programming guide, appendix k: Unified memory programming. NVIDIA Developer Zone, 2017. PG-02829-001_v9.1 [Online; accessed 17-Jan-2018]. (Cited on page 26.)

[93] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1207–1214. IEEE, 2012. (Cited on page 39.)

[94] Ron A Oldfield et al. Modeling the impact of checkpoints on next-generation systems. In *MSST 2007*, pages 30–46. IEEE, 2007. (Cited on page 77.)

[95] Daniel AG Oliveira, Paolo Rech, Laércio L Pilla, Philippe OA Navaux, and Luigi Carro. GPGPUs ECC efficiency and efficacy. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 209–215, October 2014. (Cited on page 29.)

[96] FAQ: Fault tolerance for parallel MPI jobs. https://www.open-mpi.org/faq/?category=ft#cr-support, 2019. [Online; accessed Jan., 2019]. (Cited on page 10.)

[97] A. J. Peña, W. Bland, and P. Balaji. VOCL-FT: Introducing Techniques for Efficient Soft Error Coprocessor Recovery. In *SC*, pages 1–12, Nov 2015. (Cited on page 40.)

[98] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association. (Cited on page 38.)

[99] B. Pourghassemi and A. Chandramowlishwaran. cudaCR: An In-Kernel Application-Level Checkpoint/Restart Scheme for CUDA-Enabled GPUs. In *CLUSTER*, pages 725–732, Sept 2017. (Cited on page 40.)

[100] J. Prades and F. Silla. Turning GPUs into Floating Devices over the Cluster: The Beauty of GPU Migration. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 129–136, Aug 2017. (Cited on page 39.)

[101] Narasimha Raju, Y Liu Gottumukkala, Chokchai B Leangsuksun, Raja Nassar, and Stephen Scott. Reliability analysis in HPC clusters. In *HAPCW*, pages 673–684, 2006. (Cited on page 76.)

[102] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D. Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K. Panda. MPI performance engineering with the MPI tool interface: The integration of MVAPICH and TAU. *Parallel Computing*, 77:19–37, 2018. (Cited on page 113.)

[103] Marvin Rausand and Arnljot Hoyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-IEEE, 3 edition, November 2003. (Cited on page 14.)

[104] C. Reaño, F. Silla, and J. Duato. Enhancing the rCUDA Remote GPU Virtualization Framework: From a Prototype to a Production Solution. In *CCGRID*, pages 695–698, May 2017. (Cited on page 39.)

[105] Carlos Reaño and Federico Silla. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *CLUSTER*, pages 488–489. IEEE, 2015. (Cited on page 39.)

[106] Carlos Reaño, Federico Silla, Dimitrios Nikolopoulos, and Blesson Varghese. Intra-node memory safe GPU co-scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2017. (Cited on page 39.)

[107] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, page 4. ACM, 2015. (Cited on page 39.)

[108] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016. (Cited on page 28.)

[109] Andrea Rosà, Lydia Y Chen, and Walter Binder. Predicting and mitigating jobs failures in big data clusters. In *CCGrid 2015*, pages 221–230. IEEE, 2015. (Cited on page 76.)

[110] Andrea Rosà, Lydia Y Chen, and Walter Binder. Failure analysis and prediction for big-data systems. *TSC 2016*, 2016. (Cited on page 76.)

[111] Joseph F Ruscio, Michael A Heffner, and Srinidhi Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In *IPDPS*, pages 1–10. IEEE, 2007. (Cited on page 38.)

[112] Ramendra K Sahoo, Mark S Squillante, A Sivasubramaniam, and Yanyong Zhang. Failure Data Analysis of a Large-scale Heterogeneous Server Environment. In *DSN 2004*, pages 772–781. IEEE, 2004. (Cited on pages 75 and 76.)

[113] Nikolay Sakharnykh. Combine OpenACC and unified memory for productivity and performance. NVIDIA Blog, 2015. [Online; accessed 21-Jan-2018]. (Cited on page 38.)

[114] Nikolay Sakharnykh. Beyond GPU memory limits with unified memory on Pascal. NVIDIA Blog, 2016. [Online; accessed 17-Jan-2018]. (Cited on pages 28 and 81.)

[115] Nikolay Sakharnykh. High-performance geometric multi-grid with GPU acceleration. NVIDIA Blog, 2016. [Online; accessed 21-Jan-2018]. (Cited on page 81.)

[116] Nikolay Sakharnykh. Unified memory on Pascal and Volta. GPU Technology Conference (GTC), 2017. [Online; accessed 17-Jan-2018]. (Cited on pages 28, 31, and 40.)

[117] B Schroeder and Garth Gibson. A Large-scale Study of Failures in High-performance Computing Systems. *TDSC 2010*, 7(4):337–350, 2010. (Cited on pages 14 and 76.)

[118] John Shalf, Sudip Dosanjh, and John Morrison. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011. (Cited on page 12.)

[119] J. Y. Shi, M. Taifi, A. Khreishah, and J. Wu. Sustainable GPU Computing at Scale. In *2011 14th IEEE International Conference on Computational Science and Engineering*, pages 263–272, Aug 2011. (Cited on pages 12 and 29.)

[120] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU-accelerated High Performance Computing in Virtual Machines. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2009. (Cited on pages 12, 25, 39, and 40.)

[121] Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reaño. Remote GPU Virtualization: Is It Useful? In *High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2016 2nd IEEE International Workshop on*, pages 41–48. IEEE, 2016. (Cited on page 39.)

[122] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *ASPLOS*, pages 297–310, New York, NY, USA, 2015. ACM. (Cited on pages 12 and 29.)

[123] Jim Stevens, Paul Tschirhart, and Bruce Jacob. Fast Full System Memory Checkpointing with SSD-aware Memory Controller. In *MemSys 2016*, pages 96–98. ACM, 2016. (Cited on page 77.)

[124] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. (Cited on page 40.)

[125] Omer Subasi, Gokcen Kestor, and Sriram Krishnamoorthy. Toward a General Theory of Optimal Checkpoint Placement. In *CLUSTER 2017*, pages 464–474. IEEE, 2017. (Cited on pages 14, 75, and 77.)

[126] Nawrin Sultana, Anthony Skjellum, Ignacio Laguna, Matthew Shane Farmer, Kathryn Mohror, and Murali Emani. MPI stages: Checkpointing MPI state for bulk synchronous applications. In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI'18, pages 13:1–13:11, New York, NY, USA, 2018. ACM. (Cited on page 63.)

[127] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. CRCUDA Source, 2015. [Online; accessed 17-Jan-2018]. (Cited on pages 16 and 81.)

[128] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. Transparent Checkpoint and Restart Technology for CUDA Applications. GPU Technology Conference (GTC), 2016. [Online; accessed 17-Jan-2018]. (Cited on pages 12, 25, 39, 40, 42, and 81.)

[129] Hiroyuki Takizawa, Kentaro Koyama, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *Proceedings of the International Sym-*

posium on Parallel and Distributed Processing (IPDPS), pages 864–876. IEEE, 2011. (Cited on pages 39, 40, 42, 44, and 81.)

[130] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 408–413. IEEE, 2009. (Cited on pages 12, 25, 39, 40, and 81.)

[131] Xiaoyong Tang, Kenli Li, Renfa Li, and Bharadwaj Veeravalli. Reliability-aware Scheduling Strategy for Heterogeneous Distributed Computing Systems. *JPDC*, 70(9):941–952, 2010. (Cited on page 77.)

[132] Alain Tchana, Bao Bui, Boris Teabe, Vlad Nitu, and Daniel Hagimont. Mitigating Performance Unpredictability in the IaaS Using the Kyoto Principle. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 6:1–6:10, New York, NY, USA, 2016. ACM. (Cited on page 8.)

[133] Boris Teabe, Alain Tchana, and Daniel Hagimont. Mitigating performance unpredictability in heterogeneous clouds. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 593–600, June 2016. (Cited on page 8.)

[134] Boris Teabe, Patrick Lavoisier Wapet, Alain Tchana, and Daniel Hagimont. Dealing with Performance Unpredictability in an Asymmetric Multicore Processor Cloud. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 332–344, Cham, 2017. Springer International Publishing. (Cited on page 8.)

[135] CRIU Team. CRIU, 2018. [Online; accessed 11-07-2018]. (Cited on page 8.)

[136] D. Tiwari, S. Gupta, and S S Vazhkudai. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on

Extreme-scale Systems. In *DSN 2014*, pages 25–36. IEEE, 2014. (Cited on pages 12, 14, 64, 75, 77, and 78.)

[137] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability Lessons Learned from GPU Experience with the Titan Supercomputer at Oak Ridge Leadership Computing Facility. In *SC*, pages 38:1–38:12, New York, NY, USA, 2015. ACM. (Cited on pages 12 and 29.)

[138] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation. In *HPCA*, pages 331–342. IEEE, 2015. (Cited on pages 12 and 29.)

[139] TOP500. TOP500 supercomputer sites. https://www.top500.org/, 2018. (Cited on pages 11 and 28.)

[140] Top500 supercomputers. https://www.top500.org/list/2018/11/?page=1, 2018. [Online; accessed Jan., 2019]. (Cited on page 89.)

[141] Tiffany Trader. TSUBAME3.0 points to future HPE Pascal-NVLink-OPA server. HPC Wire, 2017. (Cited on page 26.)

[142] Blesson Varghese, Javier Prades, Carlos Reaño, and Federico Silla. Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pages 47–56. IEEE, 2015. (Cited on page 39.)

[143] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S Tanenbaum. Lightweight Memory Checkpointing. In *DSN*, pages 474–484. IEEE, 2015. (Cited on page 38.)

[144] S. Wang, K. Li, J. Mei, G. Xiao, and K. Li. A Reliability-aware Task Scheduling Algorithm Based on Replication on Heterogeneous Computing Systems. *JGC*, 15(1):23–39, 2017. (Cited on page 77.)

[145] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 2012. (Cited on page 96.)

[146] X. Xu, Y. Lin, T. Tang, and Y. Lin. Hial-Ckpt: A Hierarchical Application-level Checkpointing for CPU-GPU Hybrid Systems. In *2010 5th International Conference on Computer Science Education*, pages 1895–1899, Aug 2010. (Cited on page 40.)

[147] John W Young. A First-order Approximation to the Optimum Checkpoint Interval. *CACM*, 17(9):530–531, 1974. (Cited on pages 5 and 77.)

[148] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999. (Cited on page 60.)

[149] Victor C Zandy, Barton P Miller, and Miron Livny. Process hijacking. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 177–184. IEEE, 1999. (Cited on pages 39 and 42.)

[150] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, and K. Li. Maximizing Reliability with Energy Conservation for Parallel Task Scheduling in a Heterogeneous Cluster. *Information Sciences*, 319:113–131, 2015. (Cited on page 77.)

[151] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level Checkpoint and Recovery for LAM/MPI. *ACM SIGOPS Operating Systems Review*, 39(3):72–81, 2005. (Cited on page 9.)