SANDIA REPORT

SAND2010-0040
Unlimited Release
Printed January 2010

Lightweight Storage and Overlay Networks for Fault Tolerance

LDRD Final Report for Project 105799

Ron A. Oldfield

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-0040
Unlimited Release
Printed January 2010

Lightweight Storage and Overlay Networks for Fault Tolerance

LDRD Final Report for Project 105799

Ron A. Oldfield

Abstract

The next generation of capability-class, massively parallel processing (MPP) systems is expected to have hundreds of thousands to millions of processors, In such environments, it is critical to have fault-tolerance mechanisms, including checkpoint/restart, that scale with the size of applications and the percentage of the system on which the applications execute. For application-driven, periodic checkpoint operations, the state-of-the-art does not provide a scalable solution. For example, on today's massive-scale systems that execute applications which consume most of the memory of the employed compute nodes, checkpoint operations generate I/O that consumes nearly 80% of the total I/O usage. Motivated by this observation, this project aims to improve I/O performance for application-directed checkpoints through the use of lightweight storage architectures and overlay networks. Lightweight storage provide direct access to underlying storage devices. Overlay networks provide caching and processing capabilities in the compute-node fabric. The combination has potential to signifcantly reduce I/O overhead for large-scale applications. This report describes our combined efforts to model and understand overheads for application-directed checkpoints, as well as implementation and performance analysis of a checkpoint service that uses available compute nodes as a network cache for checkpoint operations.

Acknowledgement

This project collaborated with researchers from Sandia, Oak Ridge National Laboratory (ORNL), the University of Texas El Paso (UTEP), IBM, and the Georgia Institute of Technology. Much of the results and text in this document are contributions from these collaborators. To all of these collaborators I would like extend my sincere thanks for your hard work, enthousiasm, technical expertise, and friendship.

For the analytic modeling effort, featured in Chapter 3, Patricia Teller, Sarala Arunigiri, Maria Ruiz Varela, all from UTEP, revised and improved the analytic models; Rolf Riesen (SNL), Philip Roth (ORNL), and Seelam Seetharami (IBM) provided guidance as well as system parameters for the various systems studied.

The initial implementation of the parallel netCDF caching service, discussed in Chapter 5, was performed by GA Tech students Gerald Lofstead and Charles Reiss during a summer internship at Sandia.

I would like to extend special thanks to Todd Kordenbrock for his contribution to design, implementation, tuning, and performance analysis of nearly all of the software components required for this project. His technical expertise and attention to detail were invaluable assets to this project.

Contents

Nomenclature **9**

1 Introduction **11**

2 Background and Related Work **15**

2.1 Checkpoint/Restart Methods 15

2.2 Models of Checkpoint/Restart Mechanisms 16

2.3 Reducing Checkpoint Overhead 16

3 Approximating Overheads of Traditional Checkpoint Methods **19**

3.1 A Model for the Optimal Checkpoint Interval 19

3.2 An Improved Model for Checkpoint Latency 20

3.3 Model Parameters 21

3.3.1 SNL Red Storm Parameters 22

3.3.2 LLNL BlueGene/L Parameters 22

3.3.3 ORNL Jaguar Parameters 23

3.3.4 Petaflop System 23

3.3.5 MTTI for Large-scale Applications 23

3.4 Model Results 24

4 Reducing Checkpoint Latency **27**

4.1 Using Lightweight Filesystems for Checkpoint Data 27

4.2 Overlay Networks 28

4.3 Model Results 31

5 Parallel NetCDF Caching Service **35**

5.1 Implementation 35

5.1.1 Parallel netCDF Semantics 35

5.1.2 RPC Library 36

5.2 Evaluation 37

5.2.1 Platform Descriptions 37

Red Storm 37

Thunderbird 38

5.2.2 Results 39

Application to Caching Service 39

End-To-End Performance 39

5.2.3 Placement for Compute-Node Servivces 42

6 Summary and Conclusions **47**

6.1 Research Contributions 47

6.1.1 Modeling the Impact of Checkpoints 47

6.1.2 The PnetCDF Caching Service 48

6.2 Future Work 49

6.2.1 Validation of Models 49

6.2.2 Enhanced I/O Models 49

6.2.3 Alternative Methods to Reduce Checkpoint Overhead 49

6.2.4 Extensions to Support Continuous Computing 50

6.2.5 Other Application-Level Services 50

6.3 Conclusions 50

References **52**

List of Figures

3.1 Red Storm Validation 22

3.2 Optimal Checkpoint Interval 24

3.3 Checkpoint Throughput 25

3.4 Aggregate Checkpoint Overhead 25

4.1 The LWFS core architecture. 28

4.2 Comparison of LWFS and Lustre 29

4.3 Optimal Checkpoint Interval 32

4.4 LWFS+Overlay Throughput on Red Storm 33

4.5 LWFS+Overlay Overhead 33

4.6 Relative Difference in Checkpoint Overheads 34

5.1 Data structure used for data-movement experiments 39

5.2 Scaling of bandwidth to caching service 40

5.3 IOR Performance using PnetCDF Cache 41

5.4 FLASH I/O Performance using PnetCDF Cache 42

5.5 Naive allocation effects 43

5.6 Random versus scatter allocation 44

5.7 Performance variance from allocation choice 45

5.8 Example allocation maps 46

List of Tables

3.1 Parameter values for the studied MPPs 21

Nomenclature

Checkpoint: An operation performed by an application or system to save an application's state to persistent storage

Checkpoint Interval (τ): The time spent doing production work between checkpoint operations.

Checkpoint Latency (δ): The time to complete a single checkpoint operation.

Checkpoint Overhead: The portion of application runtime associated with checkpoint-related operations.

Communication-Induced Checkpointing: A method of checkpointing where communication patterns trigger checkpoint operations.

Coordinated Checkpoint: A method of checkpointing which requires the application to arrive at a globally consistent state before saving state to persistent storage.

MTTI (M): This is the mean time between system failures for a particular application.

Overlay Network: A logical computer network built on top of another network. For this paper, checkpoint data passes through an overlay network of compute nodes on its way to the destination storage devices.

Restart File: A file in persistent storage that contains sufficient application state to recover a parallel application.

Restart Time: The time required to restart an application after a failure.

Rework Time: The amount of time lost after an interruption. It is the elapsed wallclock time since the last successful checkpoint.

Solve Time (T_s): The time spent performing productive work toward the final solution.

Strong Scaling: A term for parallel applications that solve a fixed-sized problem with a variable number of processors.

Uncoordinated Checkpoint: A method of checkpointing where each application process independently saves its state to stable storage.

Wallclock Time (T_ω): The total time spent on the application, including productive work, checkpoints, rework, and restarts.

Weak Scaling: A term for parallel applications that scale the problem size with the number of processors used the solve the problem.

Chapter 1

Introduction

Today's high-end massively parallel processing (MPP) systems have tens of thousands of compute nodes. For example, consider the following MPP systems currently in use at three Department of Energy (DOE) laboratories. The "Red Storm" system, a Cray XT3 machine at Sandia National Laboratories (SNL) [9], has over ten thousand compute nodes; "Jaguar", a Cray XT3/4 hybrid machine at Oak Ridge National Laboratory (ORNL), has more than eleven thousand compute nodes; and the IBM BlueGene/L [44] at Lawrence Livermore National Laboratory (LLNL), has over sixty-four thousand compute nodes. All of these machines are expected to be the computation target of large-scale applications that consume large fractions of the system. For example, 80% of the compute-node hours of Red Storm are allocated to applications that use a minimum of 40% of the compute nodes.

The massive scale of current and next-generation MPP systems and their supported applications present significant challenges related to fault tolerance. Some challenges arise because current "in-practice" approaches to fault tolerance do not match well with the expected demands or usage models of these systems. For example, the most commonly used approach to fault tolerance is "checkpoint-to-disk". Using this approach, an application (or system) periodically outputs to disk an amount of data that is sufficient to restart the application after a failure. As the size of applications grow with the number of compute nodes on which they execute, the cost of the checkpoint-to-disk approach increases due to a combination of three reasons. First, scientific applications (in particular, applications at the DOE laboratories) often use a large fraction of the memory available on each of the compute nodes, preventing the use of client-side caching to overlap computation and I/O. Second, without caching, the rate at which checkpoint data can be output is, at best, the speed of the storage system, which is typically an order of magnitude slower than the interconnection network. Third, as the number of employed compute nodes increases, so does the probability of application failure; this causes the application to checkpoint more frequently. The compounding effect of the increase in checkpoint data and the increase in the frequency of checkpoint operations with the number of employed compute nodes results in checkpoint operations for large-scale applications that generate bursts of I/O that can overwhelm an I/O system and severely impact the execution time of the application. The significance of these points is illustrated by the fact that even on today's systems, the I/O generated by checkpoints consumes nearly 80% of the total I/O usage [33]. Given the expense of the checkpoint-to-disk approach and the trends to develop systems of ever-increasing size, the community must seriously evaluate alternatives to traditional disk-based checkpointing.

The checkpoint-to-memory [42, 19, 38] approach is one such alternative. Its goal is to reduce the application execution time associated with checkpoint operations, i.e., the perceived checkpoint latency or checkpoint overhead, as we refer to it in this paper. In contrast, the checkpoint latency is the time required to write checkpoint data to persistent storage. When some of the checkpoint latency can be hidden, the impact on application execution time, i.e., the checkpoint overhead, is reduced. For the checkpoint-to-memory approach, this is done by having compute nodes use their local memories to manage the state of an application executed on other compute nodes. Because network and memory (network/memory) bandwidths are typically much faster than storage system bandwidths, this approach significantly reduces the checkpoint overhead. However, checkpoint-to-memory approaches may have problems with parity computation [37]. If an application node computes the parity, the advantage gained by using the network/memory bandwidth, instead of the storage bandwidth, decreases significantly [34]. But the biggest problem with checkpoint-to-memory approaches for large-scale scientific applications is the amount of memory resources required by the compute nodes. Even in today's systems, many large-scale scientific applications are constrained by the size of the memory on the compute nodes. Consequently, checkpoint-to-memory approaches are impractical for large-scale systems.

A number of other approaches have been proposed, but have not been adopted by the HPC community. Some of these approaches are described in the following survey papers [11, 15, 16]. This lack of interest among developers of large-scale applications is likely due to the fact that at the time of the research, the scale of the systems was not yet large enough to justify a change. As we show in Section 3.4, through mathematical modeling and analysis, checkpoint operations only become an I/O problem and, thus, an application performance problem, when MPP systems reach scales at and beyond the largest existing systems. Because of this, the application-directed, checkpoint-to-disk approach still is the most widely used fault-tolerance approach in high-performance computing (HPC). Another reason application-directed approaches appeal to developers is because the application generally knows which memory is critical to the application and can make the best decision about how to checkpoint that data. Applications also generally use restart data for more than just restarts, for example, some applications use restart data to measure progress and/or correctness of the job.

This report work done from FY'06 to FY'09 on a Sandia-funded LDRD to investigate lightweight storage architectures and overlay networks to improve application-directed checkpoint performance. Our project leveraged technology developed by the Lightweight File Systems project, a joint collaboration between the University of New Mexico and Sandia National Laboratories, and over the duration of the project involved students, staff, and faculty from the University of New Mexico, Georgia Institute of Technology, Oak Ridge National Laboratories, and IBM Research.

The technical contributions of this project include:

- a general analytical model for checkpoint overhead that takes into account per-node link bandwidth, bi-section network bandwidth, and storage system bandwidth;
- a method to reduce checkpoint overhead by using lightweight storage architectures and overlay networks to buffer checkpoint data between the application and the storage system;

- a refined analytical model that approximates the overhead associated with checkpoints using an overlay network to buffer checkpoint data;
- an approximation, using derived analytic models, of checkpoint overheads for three existing MPP systems and one theoretical Petaflop system;
- an implementation of a "checkpoint-caching service" that implements the PnetCDF API[27] that buffers data to an intermediate set of compute nodes;
- a detailed performance analysis of the caching PnetCDF library on the Cray XT3 and a large InfiniBand cluster.

This report describes these efforts in detail. Chapter 3 describes the analytic modeling and results of a study of four important systems; Chapter 5 describes the design, implementation, and performance of a checkpoint-caching service that implements the PnetCDF API; and Chapter **??** concludes the report with discussion of future work.

Chapter 2

Background and Related Work

This chapter provides background and related work to justify our choice of models, assumptions, and research direction. The included sections describe checkpoint/restart mechanisms, models of some of these mechanisms, and research that targets the reduction of checkpoint overhead and latency.

2.1 Checkpoint/Restart Methods

Checkpoint-based methods provide fault tolerance by periodically saving the state of an application to a *restart file* on persistent storage so that when a failure occurs, the system or user can restart the parallel application from the point of the last checkpoint. The most prominant checkpoint-based methods are broadly categorized as *coordinated*, *uncoordinated*, or *communication-induced*.

For large-scale applications, coordinated checkpointing is the most widely used fault-tolerance mechanism. It is favored over uncoordinated checkpointing, for which better performance comes at the cost of increased restart complexity, memory overhead, and undesirable side effects, such as the domino effect [16]. Checkpointing methods also are classified based on the type of stable storage that is used to save application state, e.g., persistent or volatile storage, and the initiator of checkpoint operations, e.g., application-directed or system-initiated checkpointing. This work focuses on large-scale applications and MPP systems. In this context, we study coordinated, application-directed, periodic, checkpoint-based, fault-tolerance methods that write application state to persistent storage.

Uncoordinated checkpointing is a method where each processes perform checkpoint operations independent of one another, thus, the restart operation is more complex. However, since there is no coordination with other processes, there is no associated synchronization overhead and, as a result, uncoordinated checkpointing is faster than coordinated checkpointing. However, uncoordinated checkpointing is prone to rollback propagation, which requires processes to store multiple checkpoint states, and, consequently, requires garbage collection in order to eliminate unnecessary checkpoint data.

In communication-induced checkpointing [16], communication patterns trigger checkpoint operations. At times during application execution, checkpointing is uncoordinated, while at other

times, communication patterns trigger processes to perform required checkpoint operations.

2.2 Models of Checkpoint/Restart Mechanisms

Several models that define the optimal checkpoint interval have been proposed in the literature. Young proposed a first-order model that defines the optimal checkpoint interval in terms of checkpoint overhead and mean time to interrupt (MTTI). Young's model does not consider failures during checkpointing and recovery [50], while Daly's extension of Young's model, a higher-order approximation, does [14]. In addition to considering checkpoint overhead and MTTI, the model discussed in [43] includes sustainable I/O bandwidth as a parameter and uses Markov processes to model the optimal checkpoint interval. The model described in [32] uses useful work, i.e., computation that contributes to job completion, to measure system performance. The authors claim that Markov models are not sufficient to model useful work and propose the use of Stochastic Activity Networks (SAN) to model coordinated checkpointing for large-scale systems. Their model considers synchronization overhead, failures during checkpointing and recovery, and correlated failures. This model also defines the optimal number of processors that maximize the amount of total useful work. Vaidya models the checkpointing overheadc of a uniprocess application. This model also considers failures during checkpointing and recovery [47]. To evaluate the performance and scalability of coordinated checkpointing in future large-scale systems, [18] simulates checkpointing on several configurations of a hypothetical petaflop system. Their simulations consider the node as the unit of failure and assume that the probability of node failure is independent of its size, which is overly optimistic.

Although the models presented in this section differ in many respects, all but [32] assume that system and processor failures are independent and exponentially distributed; however, a recent study of failures on systems at Los Alamos National Laboratory suggests that emperical evidence may not match this assumption [41]. The analysis in our paper is based on Daly's model, which also makes this assumption.

2.3 Reducing Checkpoint Overhead

Several techniques target the reduction of checkpoint overhead, i.e., the time added to application execution time as a result of checkpointing. Some of these techniques are meant to hide some of the checkpoint latency and, thus, reduce checkpoint overhead. *Copy-on-write* checkpoint algorithms take advantage of the low-latency of memory; they copy checkpoint data to a separate memory address space via virtual-memory, page-protection hardware. Once a memory-to-memory transfer is complete, the checkpoint data are saved to stable storage while application execution continues. Copy-on-write algorithms can be improved by adding a buffering capability to enable the overlapping of memory-to-memory transfers of checkpoint data and the writing of the data to stable storage [28]. Although copy-on-write implementations slightly increase checkpoint latency, they decrease checkpoint overhead [17]. Since applications executing on MPP systems use large frac-

tions of the available memory, copy-on-write and checkpoint-to-memory approaches [36] are not suitable for such systems.

The following techniques explicitly target the reduction of checkpoint latency. The use of RAID techniques has been proposed to store coordinated checkpoint data more efficiently [35]. RAID-inspired techniques, such as checkpoint mirroring, N+1 parity, and Reed-Solomon coding, are aimed at minimizing the impact of checkpointing on shared resources, e.g., I/O and network bandwidth, and on reducing checkpoint latency and recovery time [46]. Incremental checkpointing aims at reducing the size of checkpoint data by saving only the memory that has been touched since the last checkpoint operation. Page-based incremental checkpointing requires paging support from hardware and the operating system. Page-based techniques might not scale well on large MPP systems since even if only one bit in a page changes, the entire page must be saved; also, paging is not made use of on many MPP systems. Hash-based, as opposed to page-based, techniques are able to identify bytes changed in a page. This feature is used in [2] to propose an adaptive incremental checkpointing algorithm that aims at minimizing the amount of checkpoint data saved to stable storage. This algorithm uses a secure hashing function to dynamically identify a block corresponding to the approximate number of bytes changed in memory.

The LWFS+overlay approach to checkpointing, described in Chapter 4, reduces checkpoint overhead by buffering checkpoint data at network bandwidths rather than storage bandwidths. An important benefit of this approach for applications targeted at MPP systems is that it does not require additional memory resources on the compute nodes, a limitation that makes checkpoint-to-memory, asynchronous checkpoint, and incremental checkpoint approaches impractical for many large-scale applications [34]. The techniques described above that specifically target checkpoint latency and are applicable to MPP systems can be incorporated into the LWFS+overlay approach.

Chapter 3

Approximating Overheads of Traditional Checkpoint Methods

In this chapter, we use analytic modeling to derive a lower bound approximation for the overhead associated with traditional disk-based checkpointing. The goal is to provide insight into the real performance impact of using traditional disk-based checkpoint approaches on current and next generation systems.

3.1 A Model for the Optimal Checkpoint Interval

Much of the related work around checkpoint modeling for applications that do periodic checkpoints has been focused on finding the *optimal checkpoint interval*, or the period between checkpoints that minimizes the total time to solution [50, 12]. We based our model for application runtime on the detailed mathematical model from John Daly [13], which makes improvements on Young's earlier work [50]. Daly and Youngs models for application running time assume that the computer system exhibits Poisson single component failures [12, 13, 14]. In Daly's model the total execution time is

$$T_\omega(\tau) = M\mathrm{e}^{R/M}\left(\mathrm{e}^{\frac{(\tau+\delta)}{M}} - 1\right)\left(\frac{T_s}{\tau} - \frac{\delta}{\tau+\delta}\right), \tag{3.1}$$

where T_s is the total time spent doing application computation, also called the *solve time*; δ is the time to perform a single checkpoint operation, also called the *checkpoint latency*; R is the time to restart an application after a failure; and M is the the mean time to interrupt (MTTI) for the application. Although not explained in this paper, Daly also accounts for the fractional *rework time*, or the amount of application computation lost after a failure that has to be re-computed after a restart.

To calculate τ_{opt}, the *optimal checkpoint interval*, Daly calculates the minimum of T_ω with respect tau to produce

$$\tau_{opt-D} = \begin{cases} \sqrt{2\delta M}\left[1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{\frac{1}{2}} + \frac{\delta}{9 \cdot 2M}\right] - \delta & \delta < 2M \\ M & \delta \geq 2M \end{cases} \tag{3.2}$$

3.2 An Improved Model for Checkpoint Latency

Daly's analysis uses a fixed value for checkpoint latency; however, in real systems, checkpoint latency could vary dramatically depending on the size of the job, the supporting hardware, and the performance of the file system. A significant contribution of our project is an analytic model for checkpoint latency that takes into account per-node link bandwidth, bi-section network bandwidth, and storage system bandwidth. Our model attempts to approximate a lower bound on the cost of a checkpoint operation to measure the minimum impact of checkpoints on current and next-generation systems. In particular, it does not consider network or storage contention, which may have a significant contribution to application performance. We use the following formula to approximate checkpoint latency:

$$\delta = \alpha_c + \frac{n \cdot d}{\beta_{chkpt}}, \tag{3.3}$$

where α_c is the start-up cost (e.g., creating files) associated with a checkpoint operation, n is the number of processors used by the application, d is the amount of data written by each processor, and β_{chkpt} is the "perceived" bandwidth of the checkpoint operation.

The start-up cost of a checkpoint operation, α_c, heavily depends on the employed checkpoint algorithm, e.g., whether a shared file or a file per process is employed. In POSIX-compliant parallel filesystems, consistency semantics and device conflicts contribute to poor performance when writing to a shared file. Considering this negative performance impact, many DOE applications create a file per process for checkpoint/restart files. This improves write performance but creates a significant overhead associated with sending thousands of simultaneous create operations through a centralized metadata server [29].

With respect to the amount of data written by each processor, i.e., d, we assume a *weak scaling* scheme where the size of the data scales with the size of the application.

The perceived bandwidth takes into account the fact that the compute nodes of most MPP systems are diskless, meaning that all I/O requests travel through a communication network to access the storage devices. Thus, a checkpoint operation is bound by the aggregate network link bandwidth $n \cdot \beta_L$, the bisection network bandwidth β_n, or the storage system bandwidth β_s.

$$\beta_{chkpt} = \min(\beta_L, \beta_n, \beta_s)$$

Table 3.1: Parameter values for the studied MPPs .

Parameter	Red Storm	BlueGene/L	Jaguar	Petaflop
$n_{max} \times cores$	$12,960 \times 2$	$65,536 \times 2$	$11,590 \times 2$	$50,000 \times 2$
d_{max}	1GB	0.25GB	2.0GB	2.5GB
M_{dev}	5 years	5 years	5 years	5 years
β_s	50GB/s	45GB/s	45GB/s	500GB/s
β_n	2.3TB/s	360GB/s	1.8TB/s	30TB/s
β_L	4.8GB/s	1.4GB/s	3.8GB/s	40GB/s

3.3 Model Parameters

We model the performance of a representative, scientific, parallel application on four MPP architectures: SNL's Red Storm, LLNL's BlueGene/L (BG/L), ORNL's Jaguar, and a theoretical petaflop system, all scaled to 128K compute-node processors, where a node is comprised of multiple cores/processors. The representative parallel application employs all of the compute-node processors in a system and checkpoints half of each processor's available memory at periodic intervals equal to the optimal checkpoint interval computed using Daly's model with our improved model for checkpoint latency. The memory available to each processor is the total memory of a compute node divided by the number of processors. Anecdotal evidence gathered from conversations with computational scientists who use DOE systems indicates that checkpoint data comprises 10-50% of the data in each compute node's memory; we chose to use 50% to cover the most data-intensive case.

Table 3.1 shows the model parameters for each of the four systems studied. The $n_{max} \times cores$ parameter is the number of compute-node processors in the system. It is the product of the number of nodes and the number of processors per node in the system[1]. d_{max} is the total memory available to a processor. M_{dev} is the expected MTTI of any compute node in the system; see Section 3.3.5 for a description of how it is calculated. Our parameters for link and bisection bandwidths (β_n, β_L) indicate hardware rates reported by vendors; they do not include the overheads associated with message headers, encoding, or integrity checks (e.g., checksums), which may be required for production use.

The following sections describe the various systems and justify the parameter values used in our model.

[1]Our model assumes that the application uses every compute-node processor as if it were independent; we do not assume SMP compute nodes. This is known as "virtual node" mode on the Cray XT systems at SNL and ORNL.

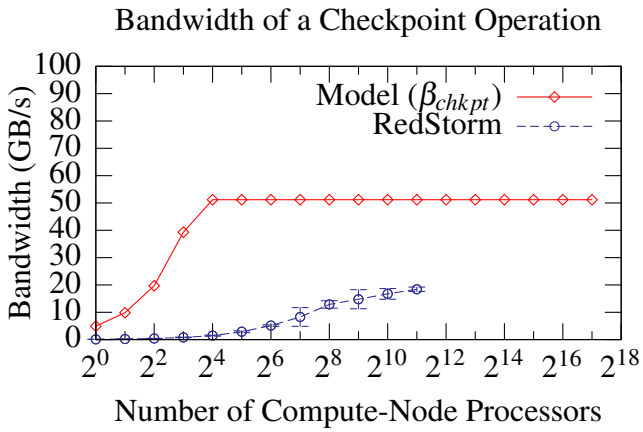

Figure 3.1: Comparison between the modeled and actual performance of checkpointing for LLNL's IOR benchmark writing to a file per process on SNL's Red Storm.

3.3.1 SNL Red Storm Parameters

The Red Storm at SNL is a Cray XT3 system comprised of 12,960 dual-core compute nodes and 256 dual-core I/O nodes. The I/O nodes are split evenly between two filesystems, one for classified work and one for unclassified work, each of which is supposed to provide 50GB/s sustained throughput to the storage devices. However, as indicated by recent experiments, the I/O nodes deliver filesystem performance that is well below the targeted throughput [49]. Using the LLNL IOR benchmark writing to a file per process, our independent experiments (shown in Figure 3.1) compare the actual performance of checkpointing on Red Storm, in terms of throughput in GB/s, with the modeled performance. The modeled performance, which reflects hardware rates, further emphasizes that Red Storm's achieved filesystem performance is well below hardware rates.

The Red Storm network consists of a 3-D mesh with a per-link (one-way) peak bandwidth of 4.8GB/s and a bisection bandwidth of 1.8TB/s. See [6] for a detailed description of the Red Storm, as well as a presentation of the employed design philosophy.

3.3.2 LLNL BlueGene/L Parameters

The IBM BlueGene/L (BG/L) at LLNL has 64K dual-core compute nodes and 1K I/O nodes. Each I/O node is a Lustre [5] client that connects to an 896TB Lustre cluster. The Lustre cluster consists of 224 "Object Storage Servers" (OSSs), each attached to a Data Direct Network 8500 RAID controller through a 2Gb/s link. LLNL's BG/L uses a separate network for storage and computation. The storage network consists of 1,024 1Gb/s interfaces and can provide a potential I/O bandwidth of 128GB/s to the storage system. Note, however, that the Lustre system was designed to provide 45GB/s theoretical bandwidth between BG/L and the storage cluster, but only about 22GB/s has been achieved using LLNL's IOR benchmark [39].

3.3.3 ORNL Jaguar Parameters

The Jaguar at ORNL is a Cray XT system with 11,590 nodes, each comprised of a 2.6 GHz dual-core AMD Opteron and 4GB memory. Of these nodes, 11,508 are used exclusively for computation, 10 are service and login nodes, and the remaining 72 are used for I/O to serve three Lustre filesystems–one 300TB system and two 150TB systems. Each I/O node has four Lustre object-storage targets (OSTs). Two OSTs serve the 300TB system, while the other two serve the 150TB systems. The expected peak block-I/O bandwidth to the 300TB system is 45GB/s.

Like the Red Storm at SNL, each node is connected to a Cray SeaStar router through HyperTransport technology. The SeaStar routers are interconnected in a 3D-torus. Each router provides six network links (one for each neighbor in the torus), each with a peak (one-way) link bandwidth of 3.8GB/s. The peak bisection bandwidth is 1.4TB/s. See [48] for a performance evaluation of the XT3 at ORNL. Details about the recent upgrade of the ORNL XT system can be found at `http://info.nccs.gov/resources/jaguar`.

3.3.4 Petaflop System

Although no true petaflop capability-class systems exist, Tomkins presents a "conservative" description of the system requirements for this next class of systems in [45]. The architecture of the Red Storm follow-on remains basically the same as its predecessor with improvements in the network, storage system, processors, and memory capacity. A petaflop Red Storm system will consist of over 50K compute nodes. Our table shows two processors per node, but we do not know exactly how many processors per compute node the system will have. If the trend toward multi-core systems continues, it would not be surprising to see up to 64 cores/node. Applications will have to execute on 25K or more compute nodes with over 50% efficiency and have an I/O throughput to the filesystem of 500GB/s. Each compute node will need at least 5GB of memory and the network will need a per-link (one-way) bandwidth of 40GB/s with a bisection bandwidth of 30TB/s.

3.3.5 MTTI for Large-scale Applications

The specifications of both Red Storm and the proposed petaflop system expect a MTTI of over 50 hours for the whole system, including software and hardware failures. Given our assumption of an exponential failure distribution, achieving 50 continuous hours of service without any failures on a 128K-processor application means that the per-processor MTTI has to be $128K \times 50$ hours, or 748 years! However, a recent paper from Schroeder, et al. [41], which documents a variety of different types of interrupts registered for MPP systems at Los Alamos National Laboratory (LANL), shows that with software interrupts caused by either the operating system or application libraries, even the most reliable systems achieve an MTTI of no more than five years/processor. Based on failure rates reported for the ASCI-Q supercomputer at LANL, Elnozahy, et al. [18] suggest an even lower value of single-node MTTI of one year, as a "conservative" estimate. Since we do not have definitive

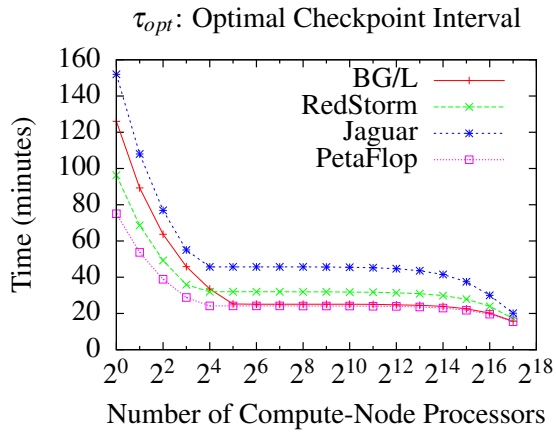

Figure 3.2: Optimal checkpoint interval as a function of the number of compute-node processors.

failure rates for the systems studied, we use a generous five-year processor failure rate (MTTI); this is optimistic, considering the failure rates reported in the literature.

3.4 Model Results

For the systems studied, using Daly's model and our model for checkpoint overhead, this section presents results regarding the optimal checkpoint interval, the throughput of the checkpoint operation, and the checkpoint overhead as a percentage of application execution time. Figure 3.2 depicts the optimal checkpoint interval as a function of the number of compute-node processors. Since the probability of application failure is directly proportional to the number of employed processors, as the figure shows, the application increases the frequency of checkpoint operations, i.e., the optimal checkpoint interval decreases, as the number of employed processors increases to account for the increased probability of failure.

Figure 3.3 illustrates the modeled throughput of a checkpoint operation of our representative parallel application executed on the MPPs studied. As shown in the figure, for any job executing on more than 32 compute-node processors, the execution time of a checkpoint operation is governed by storage system performance. In contrast, for smaller jobs, the storage system can keep up with the demands of checkpointing because the aggregate node-link bandwidth of the compute-node processors does not exceed the storage system's ability to consume data. On all systems, the execution time of a checkpoint operation is never bound by the bisection bandwidth.

Figure 3.4 shows the aggregate execution time spent performing all checkpoint operations as a percentage of the overall application execution time. The following formula is used to calculate this percentage:

$$\frac{T_\delta}{T_s + T_\delta},$$

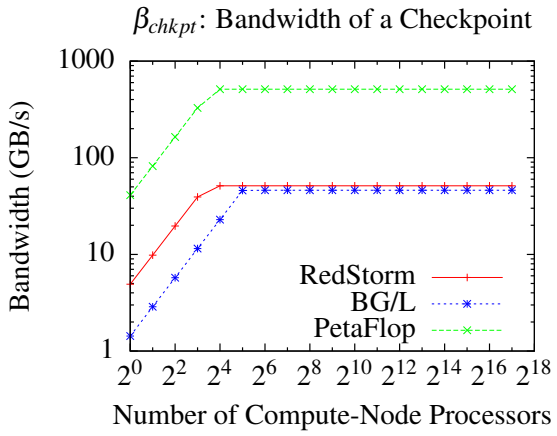

Figure 3.3: Modeled throughput of a checkpoint operation as a function of the number of compute-node processors.

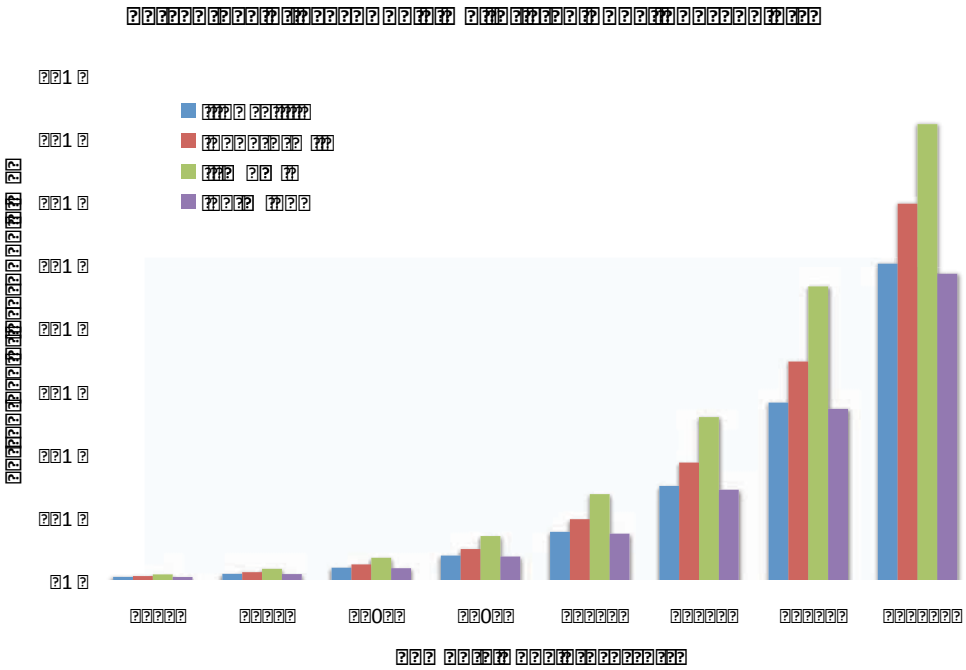

Figure 3.4: Aggregate checkpoint overhead as a percentage of application execution time.

where T_s is the amount of compute time required for the application, i.e., the solve time without checkpointing, and T_δ is the total execution time spent performing checkpoint operations. It is this fraction that matters to the computational scientist; it provides an upper bound on the scalability of the application. According to our model, even when a checkpoint operation executes at hardware rates, an application that executes on 64K processors can achieve no better than 70% efficiency, even on the hypothetical petaflop system. The dramatic increase in checkpoint overhead as the system size increases demonstrates the need to investigate alternative approaches to checkpoint/restart.

Our analysis also provides substantial evidence to explain why checkpoint-to-disk has been an acceptable solution to date. On capacity systems, with small job sizes, system-directed checkpoints, where the system checkpoints the entire memory footprint, are viable. For applications that scale to more than 4K processors, application-directed checkpoint-to-disk solutions are sufficient if the application can select the data that needs to be dumped, opting to re-calculate portions of lost data. However, as applications scale beyond 32K processors, application-directed checkpoint operations begin to dominate execution time, severely limiting application scalability.

Chapter 4

Reducing Checkpoint Latency

According to Daly's model, the optimal checkpoint interval depends on two variables: the application MTTI and checkpoint latency. This chapter describes two ways to reduce checkpoint latency as a means to improve overall application performance and reliability of the application. The first approach uses lightweight filesystems [29] to write direct to storage devices. The second approach uses the memory in an *overlay network* [21] of available compute nodes as a buffer for checkpoint data between the application and the storage system.

4.1 Using Lightweight Filesystems for Checkpoint Data

Lightweight filesystems [29] allow secure, direct access to storage, bypassing features of traditional filesystems that impose performance bottlenecks. Figure 4.1 illustrates the core architecture of a lightweight filesystem (LWFS). The LWFS core architecture consists of a small set of services and mechanisms to provide security, efficient data transport, and direct access to storage. It does not provide direct support for traditional filesystem services like naming, consistency/conflict management, or organizational information that describes data distribution. If the application requires these services, the user includes the necessary library services at link time.

Figure 4.2 illustrates why the LWFS architecture, a lightweight storage architecture, is well suited for application checkpoints. These results were obtained on SNL's Darkstar cluster, an I/O-development system that consists of 64 dual-core Opteron compute nodes and 16 dual-core Opteron I/O nodes configured to mimic the I/O-node configuration of Red Storm. First, referring to the figure, consider the negative effect that the consistency semantics of traditional filesystems have on the performance associated with shared-file access (labeled n-to-1 in the figure). Also, consider the alternative, i.e., the file-per-process approach (labeled n-to-n), which generates an unnecessarily large number of operations targeted at a centralized metadata server. In contrast, with LWFS, it is possible to design a library that permits each client process to allocate on a storage server its own object for checkpoint data. After all clients dump their states, the application collectively generates the necessary metadata to represent the distributed data set; then it selects one client to associate that metadata with a name in an external naming service. This approach avoids the expensive overhead of a file-per-process case, while achieving near physical bandwidths to the storage system. With respect to the checkpoint model, this only effects the start-up cost, α_c. Based on the measured results from Figure 4.2c, for LWFS, we set $\alpha_c = n/60,000$ seconds; this is

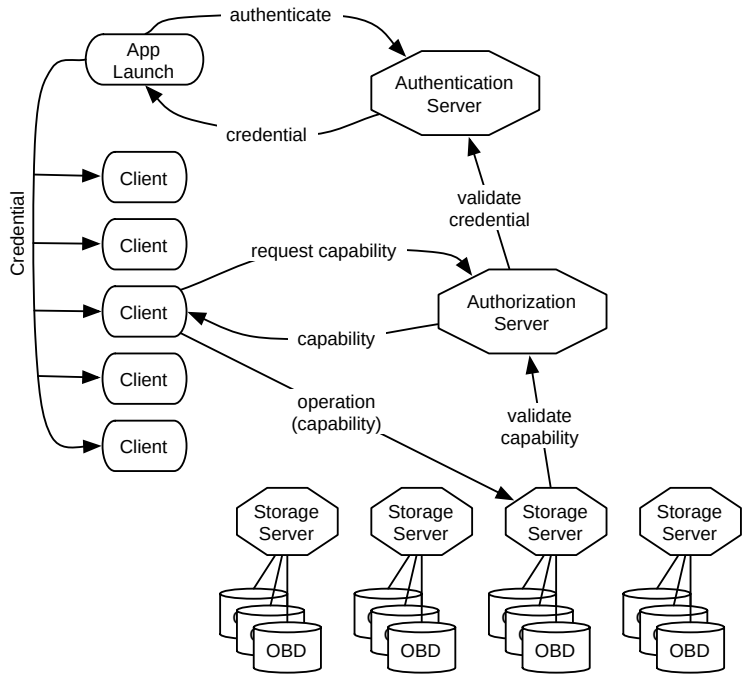

Figure 4.1: The LWFS core architecture.

a conservative estimate of the cost of allocating objects on a large system.

4.2 Overlay Networks

Lightweight storage architectures provide a direct-to-storage option for checkpoints that can improve I/O performance when compared to using a traditional "heavyweight" filesystem. However, these improvements will be modest if the data throughput is bound by the bandwidth of the storage system.

One way to relieve this storage system bottleneck is to exploit available processing and memory resources in an overlay network [21, 40]. Overlay networks provide a mechanism that allows an application to move potentially performance-limiting I/O operations off the compute nodes; this allows compute nodes to perform I/O operations unencumbered by the associated overheads and potential serialization imposed by the I/O operations. There are a number of interesting uses for overlay networks. For example, overlay networks could (1) filter input data in an application-specific way, saving network bandwidth and compute-node memory [24, 4, 20, 25]; (2) efficiently route data to compute nodes using data-dependent mapping functions, for example in applications with data-dependent decomposition of unstructured data [24]; and (3) process in-flight data sets to transform data into a format that matches the needs of the computation or a particular data distribution, for example to convert time-series data into frequency data for seismic imaging [31].

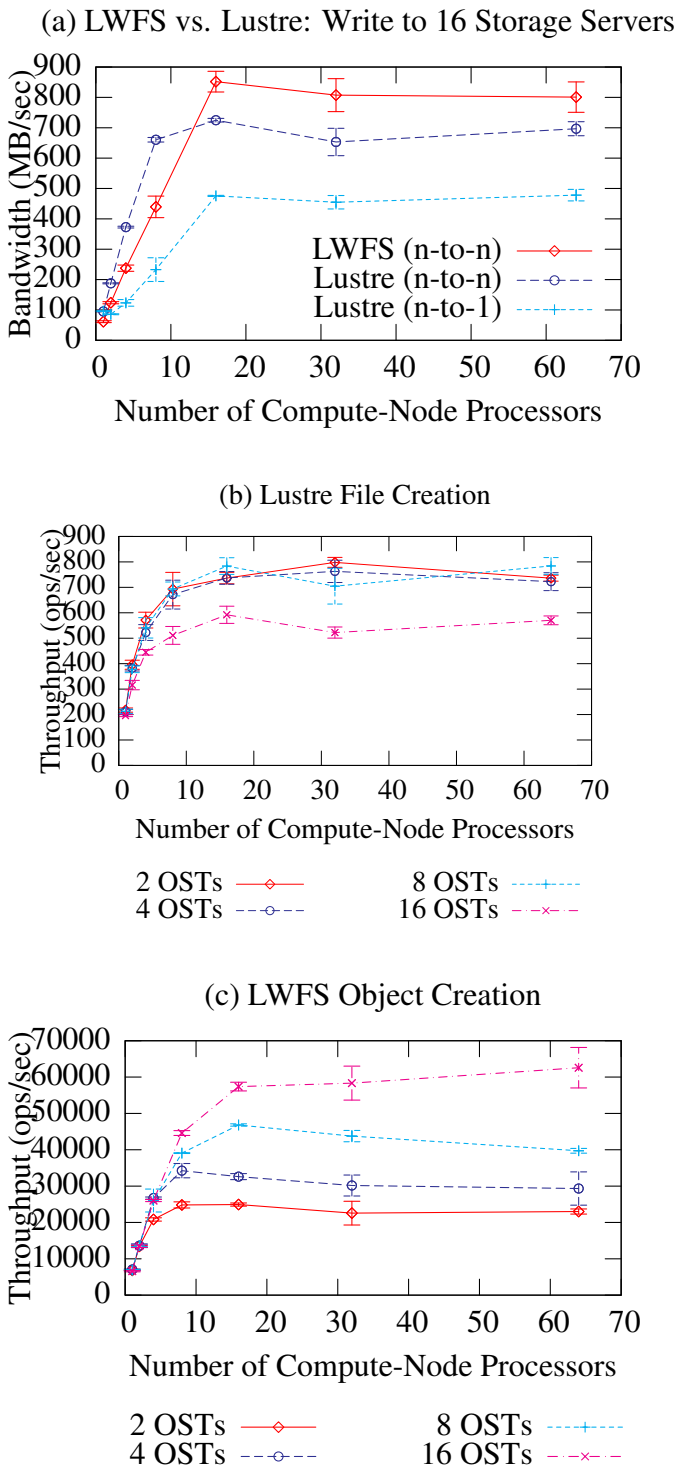

Figure 4.2: The first figure (a) shows *n*-to-1 (shared-file) and *n*-to-*n* (file-per-process) write performance of Lustre compared to the *n*-to-*n* write performance of LWFS. The second and third figures (b and c) show throughput (ops/sec) of creating files using Lustre and creating objects (in parallel) using LWFS. All experiments were performed on SNL's Darkstar cluster.

For the purpose of improving the I/O performance of checkpoint operations, a simple use for overlay networks is to buffer checkpoint data for applications that have compute-node memory constraints. This approach allows the application to checkpoint some, if not all, of its state at network bandwidths rather than storage bandwidths. On all of the systems studied in this paper, bisection bandwidth is at least an order of magnitude greater than the peak storage system bandwidth. An important benefit of this approach is that it does not require additional memory resources on the compute nodes, a limitation that makes checkpoint-to-memory, asynchronous checkpoint, and incremental checkpoint approaches impractical for many large-scale applications [34]. Note that for applications that are not memory constrained, using overlay nodes purely as buffers is not practical. These applications will benefit more from using an asynchronous checkpoint approach that exploits buffers in the compute nodes to overlap I/O and computation.

To model the use of overlay networks for checkpoint operations, we compute the checkpoint overhead as follows:

$$\delta = \alpha_c + \begin{cases} \frac{dn}{\beta_n} & d \cdot n \leq k \\ \frac{k}{\beta_n} + \frac{(d \cdot n - k)}{\beta_s} & d \cdot n > k \end{cases}, \tag{4.1}$$

where k is the effective memory that can be used at the network bandwidth, i.e., the amount of checkpoint data transferred over the network before the transfer becomes bound by the storage system bandwidth, and $\beta_N = \min(n\beta_L, \beta_n)$ is the minimum of the aggregate link bandwidth and the bisection bandwidth of the network. The variable k is the sum of μ, the combined memory in the overlay network, and the amount of data transferred to storage while μ is being transferred to the overlay network. Thus,

$$\begin{aligned} k &= \mu + \mu\left(\frac{\beta_s}{\beta_n}\right) + \mu\left(\frac{\beta_s}{\beta_n}\right)^2 + \cdots \\ &= \sum_{i=0}^{\infty}\left(\frac{\beta_s}{\beta_n}\right)^i \\ &= \mu\left(\frac{1}{1-\frac{\beta_s}{\beta_n}}\right) \end{aligned}$$

As shown in Equation 4.1, when $(d \cdot n > k)$, the terms $\frac{k}{\beta_N}$ and $\frac{(d \cdot n - k)}{\beta_s}$ represent the spent bound by the network bandwidth (β_N) and the time spent bound by the storage system bandwidth (β_s), respectively.

The memory in overlay nodes can be used to provide buffers for bursts of I/O. The size of the checkpoint data and the size of this memory determine how much the system can hide the

checkpoint overhead. However, this introduces a lower bound on the checkpoint interval that can be employed in the system, τ_{lb}, given by

$$\tau_{lb} = \frac{\mu}{\beta_s} \min\left(1, \frac{nd}{k}\right). \tag{4.2}$$

Recall that k is size of the effective memory that can be written at the network bandwidth rate and $k > \mu$, where μ is the size of the memory in the overlay network.

For a system, as described in this section, that uses the memory in an overlay network as a buffer for checkpoint data, we define the optimal checkpoint interval, τ_{opt} as

$$\tau_{opt} = \max\left(\tau_{opt-D}, \tau_{lb}\right),$$

where τ_{opt-D} is Daly's optimal checkpoint interval defined by Equation 3.2 and τ_{lb} is defined by Equation 4.2.

Note that systems like Red Storm and BlueGene/L already pass data through intermediate "I/O nodes", which run Linux. On BG/L, there are 1,024 I/O nodes, one for every 64 compute nodes, that act as filesystem clients, which simply forward calls to the back-end filesystem. On Red Storm there are 256 nodes (128 on each side), each attached to storage devices. One goal of our future work is to investigate how to use these nodes for more application-specific purposes (for example, buffers), rather than just interfaces to the I/O system. We also want to explore using application-dedicated nodes for this purpose, and eventually investigate opportunities to use these nodes to manage state in a way that allows recovery from individual node failure without restarting the entire application.

We reiterate a point made in the introduction: We do not explore how failures in the overlay network or the underlying storage system affect our model. However, we believe that it is possible to design libraries that use overlay network memory to buffer checkpoint data and transfer it to the storage system in such a way that it has minimal impact on the probability of application failure (i.e., employing the memory in overlay networks will not influence the calculation for M_{app}). This is another topic for future work.

4.3 Model Results

For the Red Storm system at SNL, we use the models derived in Sections 3.2, 4.1, and 4.2 to estimate the potential benefit of LWFS+overlay on checkpoint performance. The results for Red Storm are presented in Figures 4.3 and 4.4, as well as in 4.5 and 4.6, which present results for the other three systems. In all of our models, we assume that there are 1,024 intermediate-node processors in the overlay network (less than 1% of the total compute-node processors), each with double the memory of a normal compute-node processor.

Figure 4.3 shows Daly's optimal checkpoint interval for our representative parallel application executed on Red Storm with the normal parallel file system (PFS), LWFS, and LWFS with an

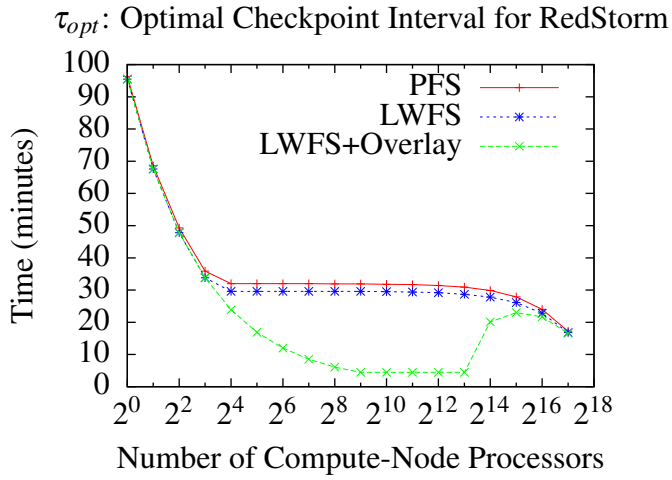

Figure 4.3: Optimal checkpoint interval for PFS, LWFS, and LWFS+overlay on Red Storm.

overlay network. An interesting side-effect of Daly's equation is that a reduction in checkpoint overhead also reduces the checkpoint interval. Initially a decrease in application execution time due to a reduction in checkpoint frequency seemed counter-intuitive. However, when you consider rework time (the time required to recompute data lost due to failures), which is included in Daly's model, it makes sense that as checkpoint operations become cheaper, the application performs checkpoints more often in order to reduce the amount of work lost due to failures. A detailed exploration of this phenomenon is presented in [3].

The effective throughput of an LWFS+overlay checkpoint operation on Red Storm, illustrated in Figure 4.4, behaves as we expected. The additional memory provided by the intermediate nodes in the overlay network causes the throughput to be bounded by the aggregate link bandwidth, then by the bisection bandwidth, and finally, when the size of a checkpoint exhausts the memory on the intermediate nodes, by the storage bandwidth. As shown in Figure 4.5, for all the MPP systems studied, this has a dramatic effect on the percentage of execution time spent checkpointing. Figure 4.6 is another bar graph that shows, for all the MPP systems studied, the relative difference between the standard filesystem approach and the LWFS+overlay approach. We calculate the relative difference as

$$\frac{P_{fs} - P_{overlay}}{P_{fs}},$$

where P_{fs} is the percentage of execution time for the standard filesystem approach, and $P_{overlay}$ is the percentage of execution time for the LWFS+overlay approach. The relative difference plot gives insight into how much better one approach is than the other. Below 8K processors, LWFS+overlay reduces the checkpoint overhead to less than 1% of the total application execution time. At 16K processors, checkpoint data exhausts the memory in the overlay network and the operations again become bound by the storage system bandwidth.

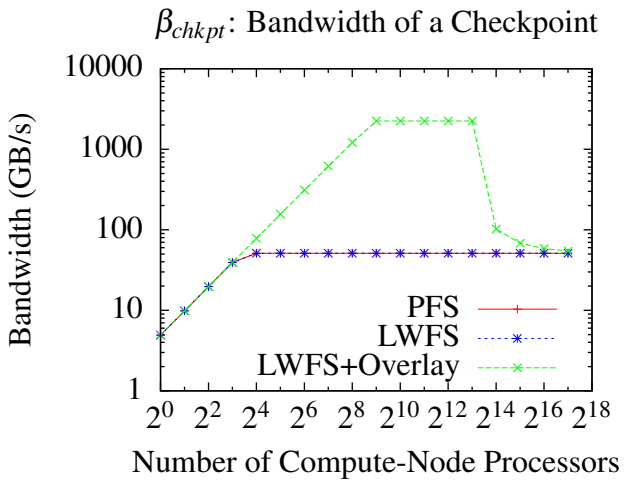

Figure 4.4: Throughput of LWFS+overlay checkpoint operation on Red Storm.

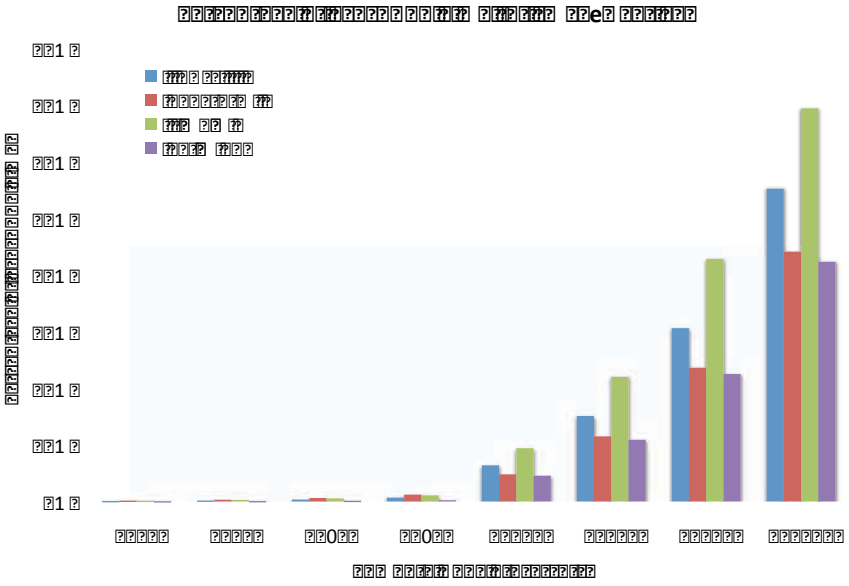

Figure 4.5: LWFS+overlay checkpoint overhead as a percentage of total application execution time for the MPP systems studied.

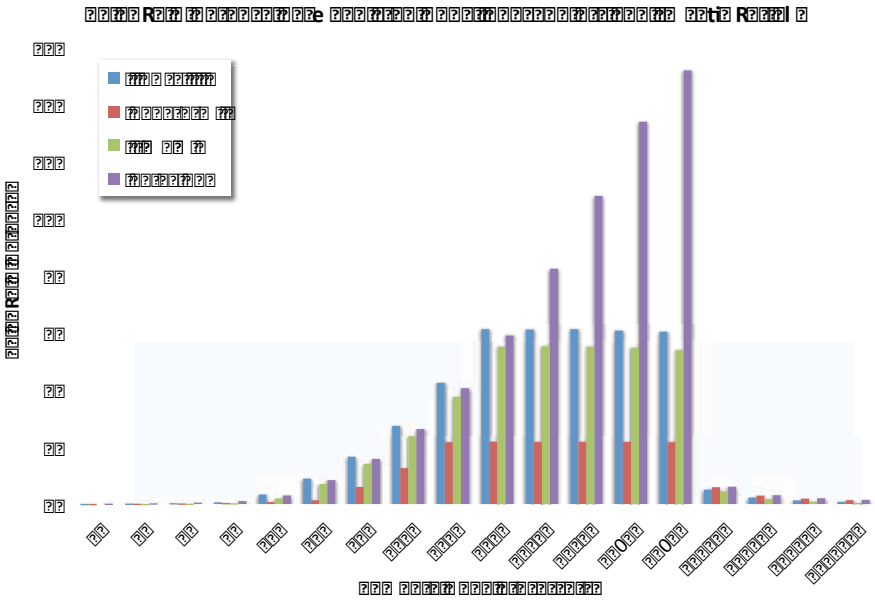

Figure 4.6: Relative difference between checkpoint overheads for standard parallel filesystems and LWFS+overlay for the MPP systems studied.

Chapter 5

Parallel NetCDF Caching Service

Chapter 4 describes a method to reduce checkpoint overhead by routing application state through an overlay network of compute nodes between the application and the destination storage devices. In this chapter, we describe the design, implementation, and performance of an application-level *caching service* that provides this functionality for data written using the Parallel NetCDF (PnetCDF) I/O library [27].

The PnetCDF caching service is a parallel application launched concurrently with parallel "client" application that acts as a *proxy* for PnetCDF procedures. The client-side library translates unmodified PnetCDF calls into remote requests for the caching service. When the caching service receives a request, it pulls the data from the client into a local buffer on the service, then it executes the actual PnetCDF call writing the data to persistent storage. The idea is to provide a large enough buffer in the network to absorb large bursts of I/O, then filter the I/O to the storage system during the compute-intensive phase of the application. With sufficient memory in the network, the application should observe "effective" I/O rates at the speed of the network instead of the storage system.

We demonstrate that for sufficiently large applications, our approach improves the performance of unchanged applications which use parallel netCDF, and that the use of compute nodes for caching is justified when applications scale beyond a certain threshold.

5.1 Implementation

5.1.1 Parallel netCDF Semantics

The parallel netCDF library's API provides applications some guarantees that are not ideal for I/O performance. To obtain good I/O performance, we relaxed guarantees which we felt were unlikely to be relied upon in any real application.

Primarily, we relaxed the implicit assumption that when a file is closed, it has been written to non-volatile storage. We, however, provide a similar guarantee that, as long as staging nodes and storage remain up for the time it takes to write the file, it will be written to non-volatile storage. Even if the staging nodes are unreliable, at worst, the application being an additional checkpoint

behind. Since shorter checkpoint times (somewhat non-intuitively) decrease the optimal checkpoint interval, we would expect to often be recovering from a more recent version in spite of this problem.

Many collective parallel netCDF operations act as barriers. Even an application that does not use collective reads or write could take advantage of these implicit barriers since opening, closing, and creating variable definitions are collective in parallel netCDF. Coordinating most operations is not useful for the staging nodes; it is more important that the staging nodes evacuate the application nodes quickly, and filesystem-related coordination can be delayed until the application's compute phase. So, while we can easily make these operations a barrier in our wrapper, doing so results in substantial performance degradation because the staging nodes cannot keep their pipeline of requests full.

We do not, however, entirely de-synchronize collective operations. Writing out a file still acts as a barrier for all processes involved, all we believe that real applications are likely to assume. As an implementation artifact, open operations are barriers since we do not transmit enough information from each application node for the staging servers to coordinate what staging servers will participate in the collective open or create operation. Also, so we may avoid keeping track of extra per-client state, mode-changing operations like switching between collective and independent operations and ending the definitions segment act as barriers.

Commonly used parallel netCDF write functions assume that memory regions passed to them cannot be accessed until after the call returns. Although there are many cases in practice when they can as the application is passing arrays that will not be modified until the next compute cycle begins, we cannot distinguish this situation from when a temporary buffer is passed. We avoided allocating buffer space on application nodes, assuming that applications would not leave us with a substantial amount of memory. Parallel netCDF does not have any explicit contract about how much memory it uses internally, but besides header information, parallel netCDF buffer space does not remain allocated after a synchronous write call returns. If we could relax these assumptions, we could provide implicit asynchronous I/O, removing the overhead of waiting for staging nodes to finish fetching each datum before preparing or sending new data.

Parallel netCDF's API does provide asynchronous writing functions. We do not believe that these are in widespread use and thus did not implement them or test some application which used them. On Red Storm, these asynchronous calls are, in fact, synchronous because the underlying MPI-IO library does not support asynchronous writes. It would, however, be easy for our proxy library to implement this API, providing asynchronous I/O from the clients to the staging servers.

5.1.2 RPC Library

For communication between application nodes and staging nodes, we used the Network-Scalable Service Interface (Nessie) [30], a framework for data services developed as part of the the Lightweight File Systems Project [29]. Nessie is a remote procedure call (RPC) based library that is designed to be scalable and handle large I/O. It uses XDR [22] to encode requests and remote direct-memory

access (RDMA) to transfer data to and from the client application. The library makes separate RDMA requests for large data segments, which are not XDR encoded. Thus, most copying of request and response data is avoided. The server coordinates the sending and receiving of these large data segments, so it is not overwhelmed by client requests. Nessie currently has implementations for Portals-based networks [7], such as those for the Cray XT series, and InfiniBand networks.

5.2 Evaluation

5.2.1 Platform Descriptions

We tested our Pnetcdf caching service on the Red Storm and Thunderbird systems at Sandia.

Red Storm

Red Storm is a Cray XT3 located at Sandia. At the time of testing, Red Storm had 12960 dual-core compute nodes. The compute nodes are arranged in a regular three-dimensional grid, connected with a hypertorus topology. Each node has an interconnect with a custom Cray SeaStar networking chip and an dedicated PowerPC chip. The interconnect is coupled to the processor using a HyperTransport link, which has a theoretical (excluding wire protocol overhead) bandwidth of 2.8GB/s [8]. Each of the six links from each node can support 2.5GB/s, after protocol overheads . Low-level software access to the interconnects is provided through the Portals library [7], which provides a connectionless RDMA-based interface.

Although the maximum node-to-node unidirectional bandwidth through Portals is around 2.1GB/s [8], in microbenchmarks, we observed that total bandwidth when receiving from multiple nodes simultaneously can exceed 2.3GB/s. Because all communication between compute nodes and service nodes takes place over these interconnects, our results could include some interference from other jobs.

All experiments detailed in this report that wrote to disk used `/scratch1` on Red Storm, a Lustre [10] version 1.4 filesystem which had 160 OSTs (Object Storage Targets) distributed over 80 OSSs (Object Storage Servers). The staging server's writes were performed through the real parallel netCDF library, which uses MPI-IO internally. The performance of writing through parallel netCDF and writing directly with POSIX or MPI-IO in tests with IOR were nearly identical on Red Storm. Based on past observed performance, the maximum I/O bandwidth to this filesystem should be around $176\text{MiB/s} \cdot 160 \approx 28\text{GiB/s}$ [26], but values observed in practice (except for carefully coordinated tests) are much lower.

We did not use MPI collective I/O for writes because our tests indicated that collective I/O consistently performed worse or no better than independent I/O on this filesystem. Except where otherwise noted, for all tests we set the stripe count of the directory where our file was written to

the maximum and wrote one shared file. The stripe size was kept at the default of 2 MiB.

All I/O was done synchronously, as the Red Storm's MPI implementation does not support asynchronous I/O[1]. For staging I/O, as long as staging nodes had sufficient free memory, we prevented the latency of synchronous I/O from interfering with transfers from the application by delaying disk I/O until the application closed the file. Except when otherwise noted, the staging nodes did, in fact, have enough memory to buffer the data.

Although some nodes on Red Storm have 3GiB of memory, we tested as if all staging nodes had only 2GiB available. When running IOR [1] to gauge I/O performance to the staging nodes, one instance of IOR ran on each of the two cores of each application node. The staging nodes, however, always ran in single-core mode so that more memory was available to them.

Our staging servers ran entirely on normal compute nodes on Sandia's Cray XT3, Red Storm, which use the lightweight operating system Catamount [23]. This imposed some restrictions on our staging application: Catamount does not have support for threading or multiprocessing and the only available internode communication mechanism is through Portals [7], which provides an RDMA interface, and through a Portals-based MPI implementation.

We also limited ourselves to performing I/O using parallel netCDF API, rather than handling the file format ourselves. On Catamount, this meant we had no asynchronous write support, so we had no way of overlapping writes with communication. The netCDF format itself has some limitations; for example, datatypes large than a byte require byte swapping, which uses extra buffer space and slows down the staging node's I/O phase. (Since we performed our tests with IOR [1], which only writes arrays of bytes, this was not necessary in our measurements.)

Thunderbird

The Thunderbird system is Sandia National Laboratories largest capacity cluster. It is composed of 4,480 compute nodes, each with dual 3.6 GHz Intel EM64T processors with 6 GB of memory. Thunderbird uses an InfiniBand network with a two level CLOS topology with eight top-level core switches and 280 leaf switches (24 ports per leaf switch). Each leaf switch has 16 downlinks (16 compute nodes per leaf switch) and 8 uplinks. Thus, the network is 2-to-1 oversubscribed in terms of raw number of links.

Although Thunderbird is primarily a capacity cluster, designed for large numbers of small jobs, it is still useful as a system to evaluate performance of the InfiniBand port of Nessie and the use of staging nodes for the caching service. It is also generally more accessible than Red Storm for testing.

[1] MPI-IO's asynchronous I/O functions are provided, but tests showed that they were apparently synchronous. There is theoretically asynchronous I/O support in Catamount on Red Storm, for example using the Cray `iwrite` function, but we did not test this.

```
struct data_t {
    int int_val;          /* 4 bytes */
    float float_val;      /* 4 bytes */
    double double_val;    /* 8 bytes */
};
```

Figure 5.1: The 16-byte data structure used for the data-movement experiments.

5.2.2 Results

Application to Caching Service

To evaluate the performance of transfers from the application to the caching service, we ran experiments that measure the rate at which we can move data from a parallel application to a set of servers. In these experiments, each client sends an array of 16-byte data structures to the server. Each 16-byte data structure (defined in Figure 5.1) consists of a 4-byte integer, a 4-byte float, and a 8-byte double.

Figure 5.2 shows the scaling performance of the NSSI interface for Red Storm and Thunderbird. The limiting factor in transferring data from the application to the caching service is the network bandwidth of the individual servers. On Red Storm with no network contention, RDMA fetches proceed at an average speed of 2200MiB/s, the maximum speed we have seen for such transfers even in small-scale tests. This speed is only achieved by overlapping at least two RDMA transfers. Under normal conditions, the RMA transfer speed is measured at 1800 or 1900 MiB/s. Other overheads can further reduce the effective bandwidth to 1600 to 1700 MiB/s, but this is as low as we have consistently observed with reasonable node allocations. We believe that this reduction is primarily caused by the increased likelihood of contention and the overall effective bandwidth being determined by the slowest staging server.

End-To-End Performance

We measured the end-to-end I/O performance by evaluating two benchmark codes: IOR and FLASH I/O. IOR (Interleave-or-random) [1], is a highly configurable benchmark code from LLNL that emulates the I/O phase of a typical HPC application. IOR is often used to find the peak measurable throughput of an I/O system.

Figure 5.3 shows measured performance results of three different experiments: writing a single shared file using PnetCDF (no cache), writing a file per process using standard netCDF3, and writing a single shared file using PnetCDF through the caching service. Each experiment wrote 25% of each compute-node's memory to the file(s). The caching service required 1/4 additional compute nodes to make sure the data remained in-core for the output dump.

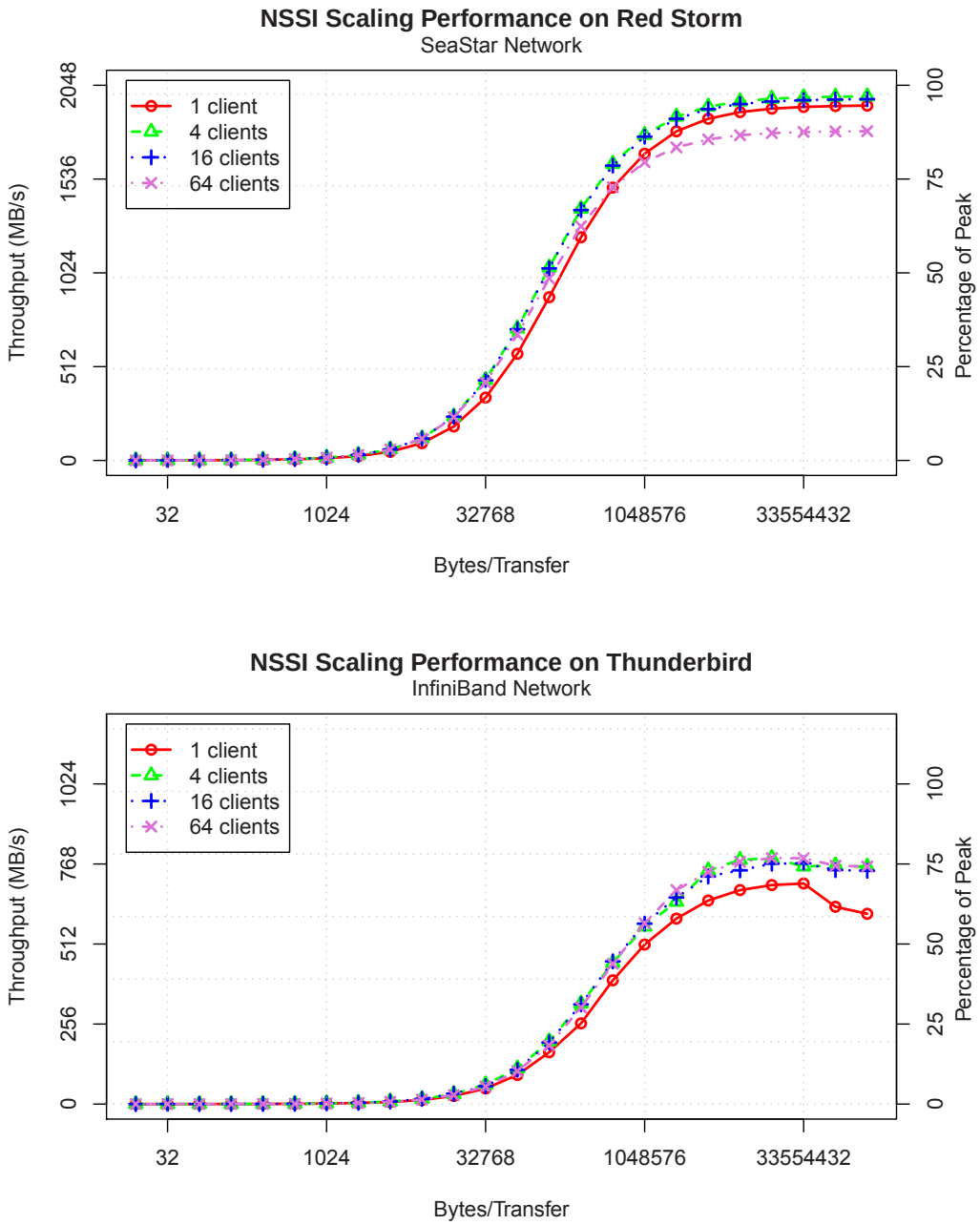

Figure 5.2: Measured throughput from a set of compute-node clients to a single caching server on Red Storm (upper) and Thunderbird (lower). The time includes including bandwidth excluding open time and the raw RMA get bandwidth.

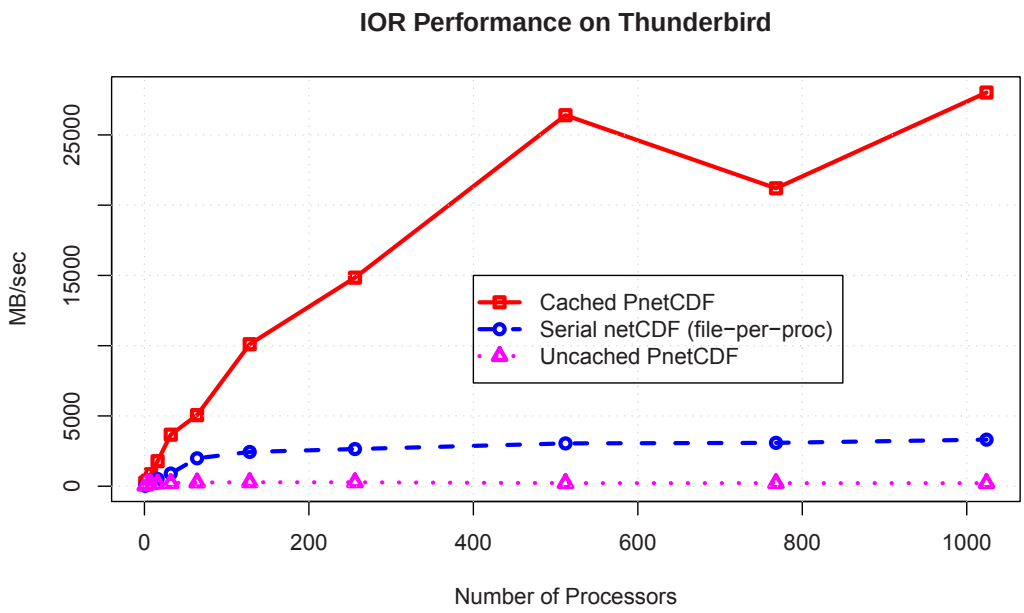

Figure 5.3: Measured throughput of the IOR benchmark code using a non-cached PnetCDF to a single file, non-cached netCDF using a file per process, and PnetCDF through our caching service.

Results on Thunderbird[2] show that uncached PnetCDF to a single shared file has horrible performance, maxing out at 217 MiB/s. The file-per-process experiment had much better performance, achieving a peak throughput of 3.3GiB/s. The cached version, however, was able to achieve an "effective" 28 GiB/sec to a single shared file. This is the rate observed by the application as the time to transfer the data from the application to the set of caching service nodes. In cases where the caching service has sufficient memory and a sufficiently long compute phase of the application, our caching service results in an order of magnitude improvement in I/O performance with only 25% additional compute resources.

We also evaluated the caching service performance on the FLASH I/O benchmark code. FLASH I/O is an extracted I/O kernel from the FLASH2 code [51], a parallel hydrodynamics code used to simulate astrophysical phenomenon. Figure 5.4 shows timings on Thunderbird as the size of the application scales to 768 processors.

While the FLASH I/O benchmark is useful because it represents a "real" application I/O workload, it does not generate a tremendous amount of data. For our experiments, a single service node had enough memory to absorb the checkpoint dumps of the FLASH code for runs as large as 256 processors; however the advantage of offloading the PnetCDF writes to the caching service, even though the service only consisted of a few nodes, was substantial. Figure 5.4 shows the execution time of FLASH I/O for one version that writes direct to storage and one version that uses the caching service. In this case, a less than 1% increase in compute nodes for the caching service resulted in a 60% reduction in execution time.

[2]Results for Red Storm and the Cray XT4/5 systems at ORNL are in progress.

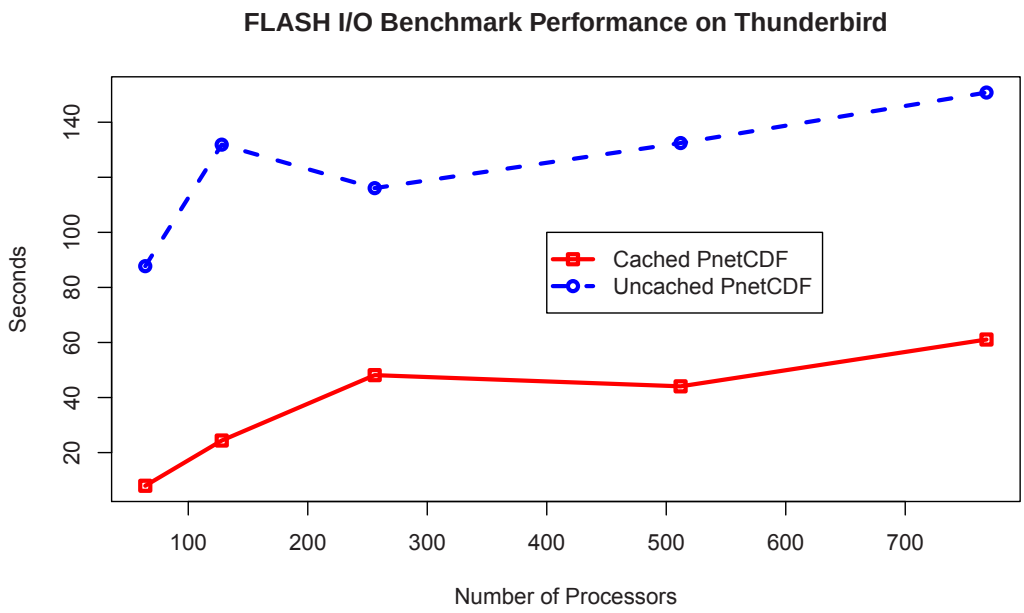

Figure 5.4: Measured timings of the FLASH I/O benchmark code using a non-cached PnetCDF to a single file, and PnetCDF through our caching service.

5.2.3 Placement for Compute-Node Servivces

When using more than one caching server the the placement of the service nodes is important. The minimum bisection bandwidth of Red Storm is on the order of terabytes per second, larger than the interconnect-to-memory bandwidth of hundreds of nodes, and even the bandwidth of the physical links is larger than the interconnect-to-memory bandwidth. Thus, one might expect that the interconnect-to-memory bandwidth would determine the overall bandwidth we could provide, and the interconnect bandwidth itself would be largely irrelevant. Unfortunately, although there is plenty of overall interconnect bandwidth, the routing scheme is fairly static and so cannot take into account changing communication patterns. When the routes for several pairs of nodes that are transferring data intersect, these shared interconnects thus limit the total bandwidth.

Figure 5.5 illustrates this problem. With the ratio between staging and application nodes increased, bandwidth can decrease substantially as the number of staging nodes is increased. We believe this results from network contention.

Naive allocations are much more susceptible: if one allocates the staging nodes and the application nodes as separate job invocations, the job launcher, by default, tends to allocate contiguous blocks of nodes as would be expected when intrajob communication speed is the primary concern. But with such an allocation, it is inevitable that there will be many shared links on paths between the application nodes and their corresponding staging nodes. Thus, as seen in the figure, performance decreases to half the maximum bandwidth with only 32 staging nodes (288 nodes in total).

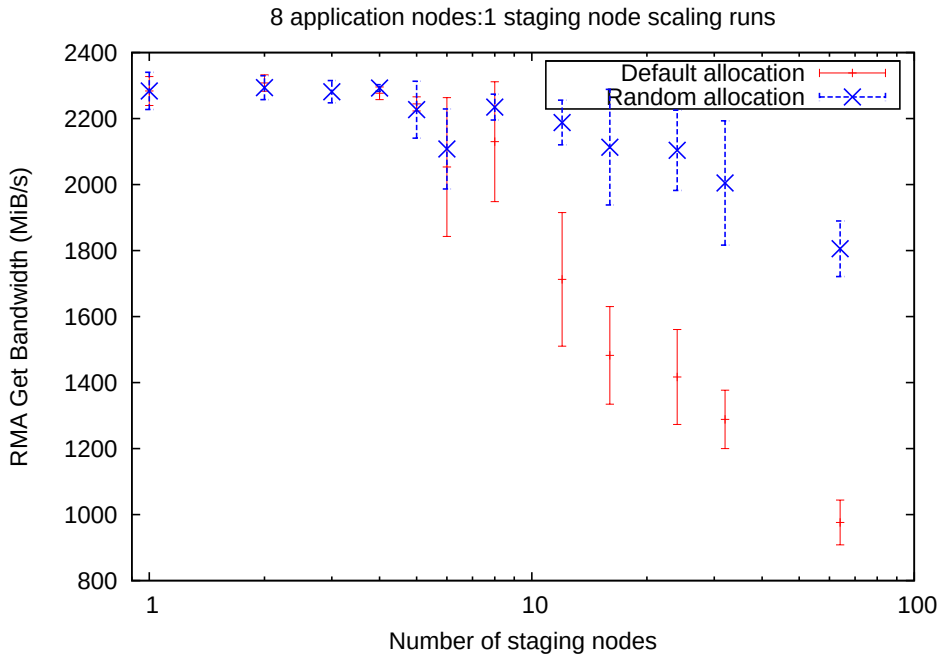

Figure 5.5: Staging node input performance with default allocation versus random allocation of staging nodes.

Fortunately, deliberately spreading the staging nodes out across our allocation avoids this contention. Random allocation of staging nodes should evenly spread out the load on the average. While random allocations perform much better than the system's default allocations as figure 5.5 illustrates, there is a substantial amount of variance in its performance. Figure 5.6 illustrates this performance variation based on approximately 200 tests performed with 50 application nodes feeding into 3 staging nodes. Ten percent of the randomly chosen staging node locations provided less than five-sixths the median bandwidth. It is possible that some of these cases could be eliminated by more intelligently assigning application nodes to staging nodes: currently, we allocate groups of contiguous ranks to a staging node, rather than trying to examine how the placements relate on the network topology. But, since we use the system-provided rank orders to make this assignment, there should be some locality between application nodes and their corresponding staging nodes. Without doing any detailed analysis of the topology, we might also avoid poor-performing random allocations by briefly testing the I/O performance before deciding on an allocation.

We, however, achieved more consistent performance by using a deterministic allocation strategy. Using the MPI ranks of each node over the entire allocation, we assigned staging nodes to every kth rank (with appropriate rounding, setting k to the number of staging nodes divided by the total number of staging and application nodes). Since application nodes are assigned to staging nodes in groups of neighboring rank, this should ensure that the distance between application and their corresponding staging nodes is reasonable. Figure 5.6 shows the results of tests of this allocation strategy with 50 application nodes feeding into 3 staging nodes. These allocations performed substantially better than random allocations while achieving approximately the same median bandwidth. As with random allocations, there are some outliers with substantially less bandwidth that

smarter allocation strategies might be able avoid.

Even with this allocation policy, node placement was still responsible for most of the variance in bandwidth between application and staging nodes. We tested this by allocating segments of 600 compute nodes and choosing random 534 node suballocations from it, and observed the performance of the every *k*th rank selection on these suballocations. We had 22 staging nodes, the number required to buffer 32 MiB each from 1024 cores. Although performance of the each suballocation was consistent over time, we observed a great deal of variance between performance of different suballocations. Figure 5.7 illustrates the results of this test over 52 trials.

Running the same test with a single 534 node allocation yields very consistent performance (less than 5% difference in observed bandwidths) even over hours. Thus, the most variance observed in our tests was probably not caused by inconsistent interconnect performance or by interference from other applications on the machine.

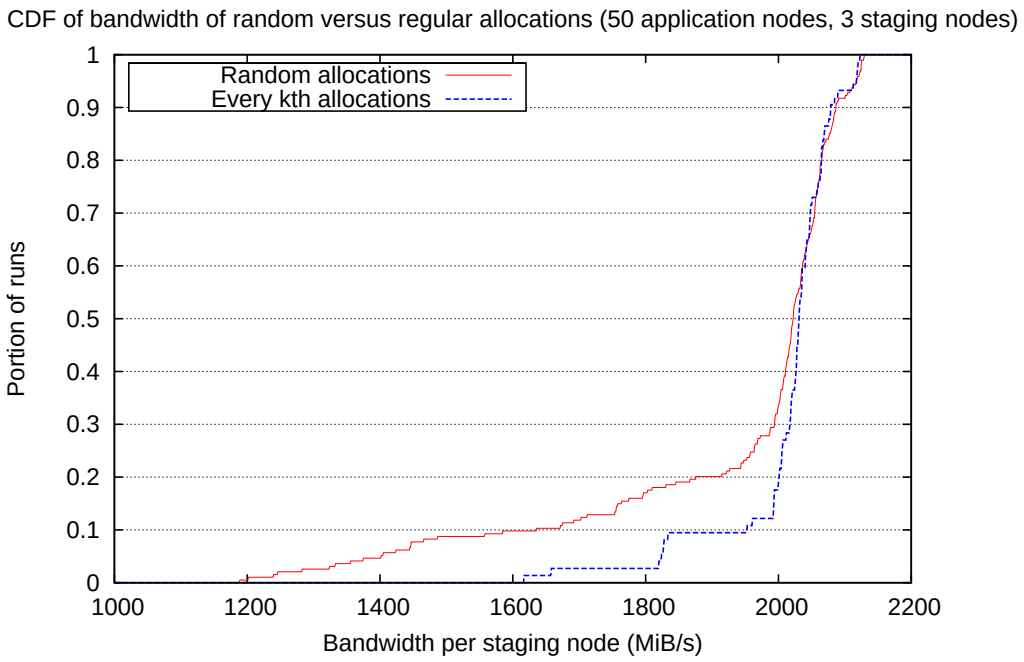

Figure 5.6: Cumulative distribution function (CDF) of performance with random allocations and rank-based allocations. These runs were performed with 50 application nodes and 3 staging nodes.

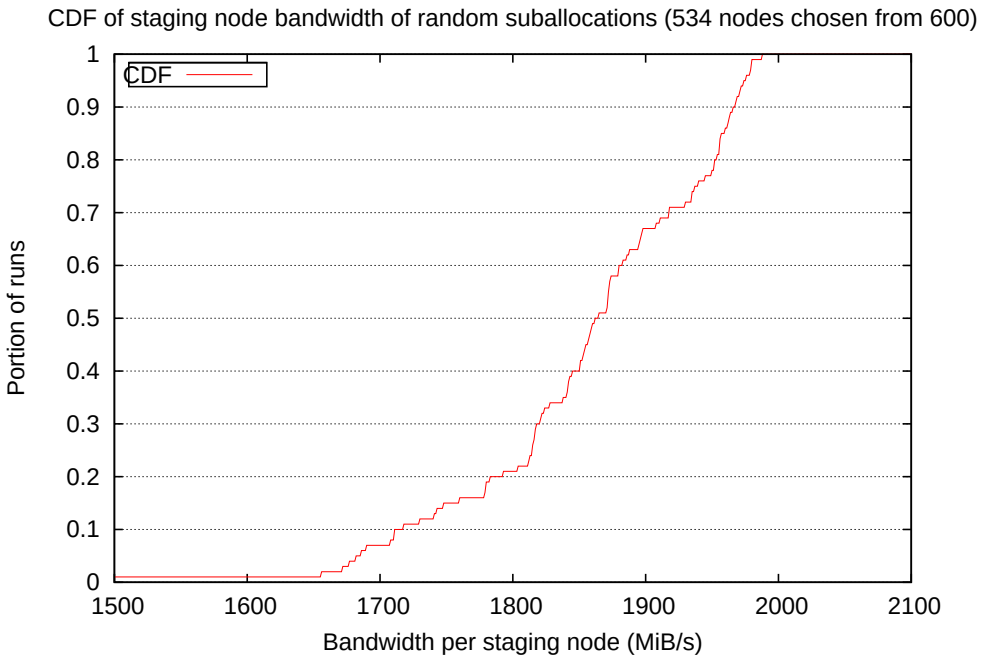

Figure 5.7: CDFs of performance of random suballocations of larger allocations.

```
[Cabinet 2]
0:   cccccccc--------c-------
1:   cccccccc--------c-------
2:   cc*ccccc--------c-------
3:   cccccccc----------------
4:   ccccccc-----------------
5:   ccccccc-----------------
6:   *ccccc*c----------------
7:   cccccccc----------------
[Cabinet 3]
12:  --------c---------------
13:  --------c---------------
14:  -------------------------
15:  ---------c--------------
[Cabinet 4]
16:  cccccccc--------cccccccc
17:  cc*ccccc--------cccccccc
18:  cccccccc--------ccc*cccc
19:  cccccccc--------ccccccc-
[Cabinet 5]
0:  cc*ccc--cccccccccccc*cccc
1:  cccccc--*ccccc*ccccccccc
2:  ccccccc-cccccccccccccccc
3:  ccccccc-cccccccccccccccc
4:  ---ccccc----------cccccc
5:  ---cc*cc---------ccc*ccc
6:  ---ccccc---------ccccccc
7:  ----cccc---------ccccccc
```

(a) Example of our every *k*th node allocation.

```
[Cabinet 2]
0:   *cccccccc--------c-------
1:   cccccccc--------c-------
2:   cccccccc--------c-------
3:   cccccccc----------------
4:   ccccccc-----------------
5:   ccccccc-----------------
6:   cc*ccccc----------------
7:   cccccccc----------------
[Cabinet 3]
12:  --------c---------------
13:  --------c---------------
14:  -------------------------
15:  ---------c--------------
[Cabinet 4]
16:  cccccccc--------cccccccc
17:  cc*ccccc--------cccccccc
18:  cc*ccccc--------cccccccc
19:  cccccccc--------ccccccc-
[Cabinet 5]
0:  ccc*cc--cccccccccccccccc*
1:  cccccc--cccccccccccccccc
2:  ccccccc-cccccccccccccccc
3:  ccccccc-ccccccccccc**c*cc
4:  ---ccccc----------cccccc
5:  ---ccccc---------ccccccc
6:  ---ccccc---------c*ccccc
7:  ----cc*c---------ccccccc
```

(b) Poor performing random allocation.

Figure 5.8: Example of node allocations. ‘c’ represents a compute node, ‘*’ represents a staging node, and ‘-’ represents a node not allocated to our job.

Chapter 6

Summary and Conclusions

The massive scale of petascale and beyond systems present significant challenges related to fault tolerance. This project focused on improvements to application-directed checkpoints, the most commonly used method for application fault tolerance used on current MPP machines. Over the three years of work supported by this LDRD, we accomplished a great deal of work. In our preliminary investigations, we developed general analytic models for checkpoint overhead that accounts for network and file system performance. Then we adapted those models to investigate an alternative method to that used memory in additional compute nodes as a buffer for checkpoints. Finally, we implemented a caching service to evaluate this approach for real applications.

This chapter provides details on our accomplishments and contributions, identifies areas for future work, and discusses conclusions that we can draw from our results.

6.1 Research Contributions

The high-level goal of this project was to develop a new scheme to improve performance of application-directed checkpoints for large-scale HPC applications. The primary contributions include detailed analytic models to approximate the impact of checkpointing on future architectures and a PnetCDF caching service that uses memory in available compute nodes to buffer checkpoint data.

6.1.1 Modeling the Impact of Checkpoints

The mathematical models approximate the performance impact of application-directed checkpointing for a representative scientific application running on three existing MPP systems and one theoretical petaflop system. To establish a lower bound on the performance impact of checkpointing, our models assume perfect scalability of the filesystem, no overheads or contention in the network, and periodic checkpoints at the interval defined by Daly's function for the optimal checkpoint interval.

Using our analytic models, we investigated three different application-directed, checkpoint approaches: a typical approach that dumps direct to a parallel file system, an approach that dumps

to a lightweight file system, and an approach that exploits available memory in an overlay network as a buffer between the application and the storage system.

Our analysis of the traditional approach illustrates two important points. First, for applications that use fewer than 16K processors, checkpoint overhead accounts for less than 10% of the overall application execution time. Since today's massive-scale systems are just now reaching this scale, the impact of the choice of fault-tolerance approach on application execution time has not yet been realized. This explains the general lack of interest we have experienced when discussing checkpoint optimizations with computational scientists at SNL.

The results also show that the compounding effect of larger checkpoint data files and increased checkpoint frequency to account for increased probability of failure, contribute to significant overheads when applications employ more than 16K processors. On some systems this overhead accounts for more than 50% of the total execution time when the application scales beyond 64K processors.

Our analysis of application performance when using overlay networks to buffer checkpoints is encouraging. The results show that when the overlay network contains sufficient memory, the approach is quite effective at reducing the checkpoint overhead. However, as the size of a checkpoint file increases beyond the size of the overlay network memory, the performance attained by using overlay networks approaches that of traditional filesystems. In our experiments, using an overlay network with 1,024 processors (1% of the total number of system processors), each with a memory capacity of twice the normal compute-node processor, when the application uses 16K processors or fewer, all of the data in a checkpoint file can be transferred at network rates rather than storage system rates.

6.1.2 The PnetCDF Caching Service

The second major contribution of our work was the design and development of a "caching service" that allowed unmodified parallel applications using Parallel NetCDF to buffer checkpoint data in the available memory of compute nodes between the application and the storage devices. Performance results of this service demonstrated a 10× improvement in I/O performance over traditional approaches, justifying our approach, but also identified a number of issues critical to achieve performance for this type of architecture.

As pointed out in Section 5.2.3, the placement of staging nodes is critical to avoid network contention. Even more concerning is the fact that the default allocation scheme shows the worst performance of all our experiments and recent changes to our scheduling software now prevent the ability to control how the staging nodes are placed. Hopefully, these results, and our continued interest in developing application-level services will motivate the need to support application-control of node placement within an assigned allocation of nodes.

6.2 Future Work

One metric of success for research projects is the number of new research projects inspired by the results. This project opened a number of interesting research questions that went beyond the scope of our work and, in some cases, led to new funding. This section describes some of the ways we can extend and continue the progress made in this project.

6.2.1 Validation of Models

Although our experimental results provide validation of our approach for I/O benchmarks on large-scale systems, we have not yet validated the performance models on "real" applications at a large scale. We will complete this work as part of the scalable I/O services project funded by the ASC CSSE program, and expect to publish these results as part of journal submission to the Journal of Parallel and Distributed Computing.

6.2.2 Enhanced I/O Models

Our models take an overly optimistic view of application performance to establish a lower bound on the performance impact of application-directed checkpoints. On most systems, however, measured performance is often much worse than hardware rates. A logical next step is to refine our models to include realistic, expected overheads (for production systems) for operations associated with the storage system, network, and other system components.

We would also like to classify applications based on, for example, memory use and examine the impact of checkpointing on a more broad set of applications. Our current analysis makes assumptions about available memory that may not be an issue on some types of applications. It would be interesting to develop analytic models and simulations to explore checkpoint schemes for other types of applications.

6.2.3 Alternative Methods to Reduce Checkpoint Overhead

The LWFS and overlay network approach, LWFS+overlay, presented in Chapter **??**, is just one of a number of interesting approaches to reduce checkpoint overhead. Our results show that when the overlay network contains sufficient memory, the approach is quite effective at reducing checkpoint overhead; however, when the size of the checkpoint data is larger than the memory in the overlay network, LWFS+overlay is only slightly better than a traditional filesystem approach. We would also like to investigate hybrid approaches that combine the use of overlay networks and client-side buffering. Such an approach would be effective, for example, if compute nodes have a small portion of available memory that, when combined with the memory in the overlay network, is sufficient to buffer all checkpoint data. Our modeling approach gives us a tool that can be used

to evaluate and theorize on the value and applicability of some of these approaches on the next generation of systems.

6.2.4 Extensions to Support Continuous Computing

Although our analytical models and experimental results provide some evidence that we can improve I/O performance for checkpoint operations, it is clear that I/O improvements alone are not enough. As part of a "spin-off" LDRD project to investigate resiliency challenges for exascale systems, we plan to investigate algorithms that use the intermediate nodes in an overlay network, along with lightweight storage architectures, for continuous computing, even when application nodes fail. For example, one interesting idea is to use nodes in an overlay network in the same way diskless checkpointing approaches use compute nodes. While the overlay nodes provide the memory lacking on the compute nodes, the parity computation can be "offloaded" to the overlay network, resolving some of the issues discussed in [37].

In addition to algorithmic work for diskless approaches, we still lack mechanisms that allocate and inject a new compute node into an existing application (allowing the new node to assume the identity of the failed node). Partial application recovery not only involves the integration of LWFS with overlay networks, it also requires integration and cooperation with other systems services that do not yet exist. For example, if a compute node dies, we need to allocate a new compute node, install the previous state (either stored in memory or on disk) onto the new node, possibly rolling-back all other nodes or updating the new node to the current state, and then we need to have the new node resume the identity of the failed node. None of these services exist in MPP systems software.

6.2.5 Other Application-Level Services

The caching-service concept led to a whole range of questions about the use of compute nodes for other application-specific purposes. Our ASC/CSSE project on scalable I/O services will investigate these types of services. As part of that project, we are currently designing an application-level service for identifying and tracking particles generated by the shock physics code CTH. This idea was also generated interest at ORNL and led to a new collaboration on data staging for

6.3 Conclusions

Overall, this was a successful project. We accomplished most of our goals and developed software that we expect to have a lasting impact on internal and external application performance. Our work also led to a number of new projects and collaborations that will extend this work, adding to the importance of this LDRD.

This project also sucessfully identified serious concerns with checkpointing approaches as we move toward exascale systems. If technology trends continue along the same path, we soon will have systems with hundreds of thousands of compute nodes, and potentially millions of processing cores. Accordingly, if the reliability of hardware and systems software do not improve substantially, the traditional approach of application-directed checkpoint-to-disk soon will become impractical for large-scale applications. It is this realization, and the evidence supported by our modeling work, that will likely have the most dramatic impact on the direction of future resilience research toward. This conclusion reinforces the need for new algorithms, approaches, and possibly programming models that explicitely support resilience.

References

[1] IOR interleaved or random HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[2] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual International Conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM Press.

[3] Sarala Arunagiri, Seetharami Seelam, Ron A. Oldfield, Maria Ruiz Varela, Patricia J. Teller, and Rolf Riesen. The impact of checkpoint latency on checkpoint interval and execution times. Technical Report TR07-55, University of Texas at El Paso, El Paso, TX, August 2007.

[4] Andrea J. Borr and Franco Putzolu. High performance SQL through low-level system integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 342–349, Chicago, IL, 1988. ACM Press.

[5] Peter J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, November 2002. http://www.lustre.org/docs/lustre.pdf.

[6] Ron Brightwell, William Camp, Benjamin Cole, Erik DeBenedictis, Robert Leland, James Tomkins, and Arthur B. Maccabe. Architectural specification for massively parallel computers: an experience and measurement-based approach. *Concurrency and Computation: Practice and Experience*, 17(10):1271–1316, March 2005.

[7] Ron Brightwell, Rolf Riesen, Bill Lawry, and Arther B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2002.

[8] Ron Brightwell, Keith D. Underwood, and Courtenay Vaughan. An evaluation of the impacts of network bandwidth and dual-core processors on scalability. In *International Supercomputing Conference*, Dresden, Germany, June 2007.

[9] William J. Camp and James L. Tomkins. The red storm computer architecture and its implementation. In *The Conference on High-Speed Computing: LANL/LLNL/SNL*, Salishan Lodge, Glenedon Beach, Oregon, April 2003.

[10] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[11] Tzi-Cker Chiueh and Peitao Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *Annual Symposium on Fault Tolerant Computing*, pages 370–379, Sendai, Japan, June 1996. IEEE Computer Society Press.

[12] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. *Lecture Notes in Computer Science*, 2660:3–12, August 2003.

[13] John Daly. A strategy for running large scale applications based on a model that optimizes the checkpoint interval for restart dumps. In *Proceedings of the 26th International Conference on Software Engineering*, pages 70–74, Edinburgh, Scotland, UK, May 2004.

[14] John Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22:303–312, 2006.

[15] Geert Deconinck, Johan Vounckx, Rudi Cuyvers, and Rudy Lauwereins. Survey of checkpointing and rollback techniques. Technical Report O3.1.8 and O3.1.12, ESPRIT Project 6731 (FTMPS), June 1993.

[16] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375 – 408, SEP 2002.

[17] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, Houston, TX, USA, October 1992. IEEE Computer Society Press.

[18] Elmootazbellah N. Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, April–June 2004.

[19] Christian Engelmann and Al Geist. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform. In *In Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments*, pages 47–52, Seattle, WA, June 2003. IEEE Computer Society Press.

[20] Ernest Franke and Michael Magee. Reducing data distribution bottlenecks by employing data visualization filters. In *Proceedings of the EighthIEEE International Symposium on High Performance Distributed Computing*, pages 255–262, Redondo Beach, CA, August 1999. IEEE Computer Society Press.

[21] Ada Gavrilovska, Karsten Schwan, Ola Nordstrom, and Hailemelekot Seifu. Network processors as building blocks in overlay networks. In *Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI?03)*, pages 83–88, August 2003.

[22] Network Working Group. XDR: External data representation standard. Technical Report RFC 1014, Sun Microsystems, Inc., June 1987.

[23] Suzanne M. Kelly and Ron Brightwell. Software architecture of the Light Weight Kernel, Catamount. In *Proceedings of the Cray User Group Meeting*, Albuquerque, NM, May 2005.

[24] David Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995IEEE Symposium on Parallel and Distributed Processing*, pages 490–495, San Antonio, TX, October 1995. IEEE Computer Society Press.

[25] Tahsin Kurc, Chialin Chang, Renato Ferreira, and Alan Sussman. Querying very large multidimensional datasets in ADR. In *Proceedings ofSC99: High Performance Networking and Computing*, Portland, OR, November 1999. ACM Press and IEEE Computer Society Press.

[26] James H. Laros III, Lee Ward, Ruth Klundt, Sue Kelly, James L. Tomkins, and Brain R. Kellogg. Red Storm IO performance analysis. In *Proceedings of the IEEE International Conference on Cluster Computing*, Austin, Texas, September 2007.

[27] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings ofSC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[28] Kai Li, Jeffrey S. Naughton, and James S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions onParallel and Distributed Systems*, 5(8):874–879, August 1994.

[29] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[30] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[31] Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998.

[32] Karthik Pattabiraman, Christopher Vick, and Alan Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821, Washington, DC, USA, 2005. IEEE Computer Society.

[33] Fabrizio Petrini and Kei Davis. Tutorial: Achieving Usability and Efficiency in Large-Scale Parallel Computing Systems, August 31, 2004. Euro-Par 2004, Pisa, Italy.

[34] Ian R. Philp. Software failures and the road to a petaflop machine. In *1st Workshop on High Performance Computing Reliability Issues (HPCRI)*. Los Alamos National Laboratory, February 2005.

[35] James S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Symposium on Reliable Distributed Systems*, pages 76–85, 1996.

[36] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43(2):125–138, June 1997.

[37] James S. Plank and Kai Li. Faster checkpointing with n+1 parity. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, Texas, June 1994.

[38] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions onParallel and Distributed Systems*, 9(10):972–986, October 1998.

[39] Rob Ross, Jose Moreira, Kim Cupps, and Wayne Pfeiffer. Parallel I/O on the IBM Blue Gene /L system. BlueGene Consortium Quarterly Newsletter, 2006.

[40] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings ofSC2003: High Performance Networking and Computing*. ACM Press, Pheonix, AZ, November 2003.

[41] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, Philadelphia, PA, June 2006. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[42] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In *24th EUROMICRO Conference*, pages 395 – 402, Vasteras, Sweden, August 1998. IEEE Computer Society Press.

[43] Rajagopal Subramaniyan, R. Scott Studham, and Eric Grobelny. Optimization of checkpointing-related i/o for high-performance parallel and distributed computing. In *PDPTA*, pages 937–943, 2006.

[44] The BlueGene/L Team. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, MD, November 2002.

[45] Jim Tomkins. A conservative path to petaflop computing: The Red Storm architecture scaled to a petaflop and beyond. 4th Annual Workshop on Linux Clusters for Supercomputing, October 2003.

[46] Nitin H. Vaidya. A case for two-level distributed recovery schemes. *SIGMETRICS Perform. Eval. Rev.*, 23(1):64–73, 1995.

[47] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions onComputers*, 46(8):942–947, 1997.

[48] Jeffrey S. Vetter, Sadaf R. Alam, Jr. Thomas H. Dunigan, Mark R. Fahey, Philip C. Roth, and Patrick H. Worley. Early evaluation of the cray xt3. In *Proceedings of the International Parallel and Distributed Processing Symposium*. Oak Ridge National Laboratory, April 2006.

[49] Lee Ward, Jim Laros, Ruth Klundt, and Brian Kellog. Red storm IO performance and scaling. Presentation given to Cray Inc., August 2006.

[50] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[51] M. Zingale. FLASH I/O benchmark. `http://flash.uchicago.edu/zingale/~flash_benchmark_io`, 2002.

DISTRIBUTION:

1	MS 0123	Donna Chavez, 1011
1	MS 1319	Rolf Riesen, 1423
1	MS 1319	Ron Oldfield, 1423
1	MS 1319	Todd Kordenbrock, 1423
1	MS 0899	Technical Library, 9536 (electronic)

Sandia National Laboratories