論文 / 著書情報
Article / Book Information

題目(和文)	最適なチェックポ　イント・リスタートのための設計及び　実装
Title(English)	Design and Implementation for Optimal Checkpoint/Restart
著者(和文)	佐藤賢斗
Author(English)	Kento Sato
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第9422号, 授与年月日:2014年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,増原 英彦,三好 直人,脇田 建
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第9422号, Conferred date:2014/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO INSTITUTE OF TECHNOLOGY

Doctoral Thesis

Design and Implementation for Optimal Checkpoint/Restart

Author:
Kento Sato

Supervisor:
Prof. Satoshi Matsuoka

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Mathematical and Computing Sciences
Graduated School of Information Science and Engineering

March 2014

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

Department of Mathematical and Computing Sciences
Graduated School of Information Science and Engineering

Doctor of Philosophy

Design and Implementation for Optimal Checkpoint and Restart

by Kento SATO

The computational power of High Performance Computing (HPC) systems is growing exponentially, which enables finer grained scientific simulations. However, as the capability and component count of the systems increase, the overall failure rate increases accordingly. To make progress despite of system failures, applications periodically write checkpoints to a reliable parallel file system (PFS) so that the applications can restart from the last checkpoint even on a failure. While simple, this straightforward can impose huge overhead on application run times for both checkpoint and restart operations. Although several techniques have been proposed to reduce the overhead, these techniques are not sufficient, had several limitation for extreme scale computing.

The first approach, *multi-level asynchronous checkpointing*, solves the checkpointing overhead to PFS. The multi-level asynchronous checkpointing writes checkpionts through agents running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers application checkpoint overhead by overlapping computation and writing checkpoints to the PFS. Also, it reduces PFS load by using fewer concurrent writers and moderating the rate of PFS I/O operations. Our experiments show that the asynchronous checkpointing system can improve efficiency by 1.1 to 2.0 times on future machines. Additionally, applications using the asynchronous checkpointing system can achieve high efficiency even when using a PFS with lower bandwidth.

The second approach, *a user-level infiniband-based file system and checkpoint strategy for burst buffers*, consider using burst buffers to improve system resiliency. Burst buffers are a new tier in the storage hierarchy to fill the performance gap between node-local storage and the PFS. They absorb the bursty I/O requests from applications and thus reduce the effective load on the PFS With burst buffers, an application can quickly store checkpoints with increased reliability. We explore how burst buffers can improve efficiency compared to using only compute-node storage, and we develop and apply models to investigate the best checkpoint strategy with burst buffers for a wide range of checkpoint/restart strategies.

The third approach, a *fault tolerant messaging interface (FMI) for fast and transparent recovery*, is a survivable messaging interface that uses fast, transparent in-memory checkpoint/restart and dynamic node allocation. With FMI, a developer writes an application using semantics similar to Message Passing Interface (MPI). The FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance, and achieve quick recovery and restart. Our experiments show that FMI runs with similar failure-free performance as MPI, but FMI incurs only a 28% overhead with a very high mean time to failure of 1 minute.

As a whole, these approaches enable fast, scalable and transparent checkpoint and restart for extreme scale systems.

Acknowledgements

The writing of this dissertation has been one of the most significant academic challenges I have ever had to face. This dissertation would not have been possible without the support of my advisors, colleagues, my friends and my family. I'd like to acknowledge all of people that have contributed, and extend my special thanks to a few.

Firstly, I would like to express my special appreciation and thanks to my advisor Professor Dr. Satoshi Matsuoka, you have been a tremendous mentor for me. I would like to thank you for giving me a number of advices and opportunities for participating in international conferences as well as conducting my research outside Japan. Your advice on both research as well as on my career have been priceless.

I would also like to thank my committee members, Associate Professor Dr. Toshio Endo, Professor Dr. Hidehiko Masuhara, Professor Dr. Naoto Miyoshi and Associate Professor Dr. Ken Wakita, for serving as my committee members. I also want to thank you for your brilliant comments and productive suggestions to my dissertation.

I would also like to thank Assistant Professor Dr. Hitoshi Sato for his assistance and guidance in getting my first graduate career on the right foot, and providing me with the foundation for becoming a computer scientist.

I would also like to express my gratitude to Dr. Naoya Maruyama. He has continuously given me right instruction for my research. I always admire his large variety of knowledge and ideas. I also owe a very important debt to him. He gave me a great opportunity to work at Lawrence Livermore National Laboratory.

I would like to show my greatest appreciation to staff in Lawrence Livermore National Laboratory, Adam Moody, Kathryn Mohror, Todd Gamblin, and Bronis R de. Supinski. They spent great deal of time for over fifty times of meetings. Their feedback enabled me to think about my research from various different perspectives, and to improve the overall quality of our papers we submitted to the past several international conferences.

I would like to express the deepest appreciation to talented interns in the Team SCR, Tanzima Islam, Raghunath Raja Chandrasekar and Sagar Thapaliya. In particular, I have truly enjoyed collaborating, and having several meetings with them. I would also like to thank all of them that their deep understanding of system software incented me to strive towards my goal.

I would like to offer my special thanks to my wonderful friends who accommodated me during my stay in Livermore, Misty Marin, Paul Simpson and James Tanton. My life in the States would have not be fruitful without them.

Finally there are people who I must not forget to express my appreciation. They are my parents and my sisters. Without their support and understanding, this thesis could never be completed. I greatly appreciate it so much.

Kento Sato, March 2014

Paper List

Conferences

[1] Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "A User-level Infiniband-based File System and Checkpoint Strategy for Burst Buffers", In Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid2014), Chicago, USA, May, 2014.

[2] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery", In Proceedings of the International Conference on Parallel and Distributed Processing Symposium 2014 (IPDPS2014), Phoenix, USA, May, 2014.

[3] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System", In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis 2012 (SC12), Salt Lake, USA, Nov, 2012.

Workshops

[4] Kento Sato, Satoshi Matsuoka, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski and Naoya Maruyama, "Burst SSD Buffer: Checkpoint Strategy at Extreme Scale", IPSJ SIG Technical Reports 2013-HPC-141, Okinawa, Sep, 2013.

[5] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Design and Modeling of an Asynchronous Checkpointing System", IPSJ SIG Technical Reports 2012-HPC-135 (SWoPP 2012), Tottori, Aug, 2012.

[6] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Towards an Asynchronous Checkpointing

System", IPSJ SIG Technical Reports 2011-ARC-197 2011-HPC-132 (HOKKE-19), Hokkaido, Nov, 2011.

Posters

[7] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Towards a Light-weight Non-blocking Checkpointing System", In HPC in Asia Workshop in conjunction with the International Supercomputing Conference (ISC'12), Hamburg, Germany, June, 2012.

[8] Kento Sato,Adam Moody,Kathryn Mohror,Todd Gamblin,Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Design and Modeling of a Non-Blocking Checkpoint System", In ATIP - A*CRC Workshop on Accelerator Technologies in High Performance Computing, Singapore, March, 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

The growing computational power of high performance computing (HPC) systems enables increasingly larger scientific simulations. However, as the number of system components increase, the overall failure rate of systems increases. Further, the mean time between failures (MTBF) of future systems is projected to be on the order of tens of minutes or hours [1, 2, 3]. In fact, an earlier failure analysis on Hera, Atlas and Coastal clusters at Lawrence Livermore National Laboratory (LLNL) [4] showed that a production application, the `pF3D` laser-plasma interaction code [5], experienced 191 failures out of 5-million node-hours. If we simply scale out the system while keeping the failure rate constant, the estimated MTBF is about 1.2 days for a 1,000-node cluster, 2.9 hours for a 10,000-node cluster, and 17 minutes for a 100,000-node cluster. Without fault tolerant techniques and more reliable hardware, applications will be unable to run continuously for even one day on such a larger system. Therefore, as we look towards extreme scale systems, fault tolerance is becoming more important [6].

Checkpointing is one of indispensable fault tolerance techniques, commonly used by HPC applications that run continuously for hours or days at a time. A *checkpoint* is a snapshot of application state that can be used to restart execution if a failure occurs. However, when checkpointing large-scale systems, tens of thousands of compute nodes write checkpoints to a parallel file system (PFS) concurrently, and the low I/O throughput becomes a bottleneck. Although simple, this straightforward checkpointing scheme can impose huge overheads on application run times.

Multilevel checkpointing [4, 7] is a promising approach for addressing these problems. This approach uses multiple storage levels, such as RAM, local disk, and the PFS, according to the different degrees of resiliency and the checkpoint/restart time in those storage levels. Multilevel checkpointing systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures or failures of a few nodes (level-1/L1 failure), and the PFS for more catastrophic failures (level-2/L2 failure). By taking frequent, inexpensive node-local checkpoints (level-1/L1 checkpoint), and less frequent, high-cost checkpoints to the PFS (level-2/L2 or PFS checkpoint), applications can reduce PFS checkpointing counts, and achieve both high resilience and better efficiency.

However, if we apply the multi-level checkpoint/restart technique to extreme scale systems, where failure rate is expected to increase much higher than one of current existing systems, there are several difficulties:

≤ **Checkpointing overhead/workload to PFS:** Computational capabilities are increasing faster than PFS bandwidths. This imbalance in performance means applications can be blocked for order of hours for a single PFS checkpoint [4]. Thus, the overhead of checkpointing to the PFS can dominate overall application run time even with infrequent PFS checkpoint by multilevel checkpointing. Further, the huge numbers of concurrent I/O operations from large-scale jobs burden the PFS and are themselves a major source of failures. Therefore, we must reduce the PFS load in order to achieve high reliability and efficiency.

≤ **Unreliable storage architecture:** In multi-level checkpoint/restart, applications generally cache checkpoints in in-system storage such as RAM or other node-local storage on compute nodes because aggregate bandwidth of in-system storage scales with the increasing number of compute nodes. However the most common failures affect the compute nodes. Storing checkpoints in the in-system storage on compute nodes is not reliable solution because an application can not restart its execution if *one* of checkpoints stored across compute nodes is lost due to a failed compute node.

≤ **Inefficient recovery:** As mentioned above, most failures only affect a small portion of the compute nodes so the vast majority of processes and connections are still valid after a failure [4]. It is inefficient for the runtime to tear all of these down only to immediately relaunch and reconnect it all. Launching large sets of processes, loading executables and libraries from shared file systems, and bootstrapping connections between those processes takes non-trivial amounts of time.

At extreme scale, where the failure rate is expected to be much higher, solving the above problems is critical for future extreme scale systems.

1.2 Approaches and Contributions

To solve the problems, we develop *asynchronous checkpointing system* for light-weight checkpointing to PFS, explore *multi-tier storage design with burst buffer* for reliable checkpointing, and propose *fault tolerant messaging interface (FMI)* for fast and transparent recovery.

1.2.1 Asynchronous Checkpointing System

First, we develop an asynchronous checkpointing system to minimize level-2 checkpointing overhead/workload to PFS. The asynchronous checkpointing system solves the problem through agents running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers application checkpoint overhead by overlapping computation and writing checkpoints to the PFS. Also, it reduces PFS load by using fewer concurrent writers and moderating the rate of PFS I/O operations. In particular, our asynchronous checkpointing system coupled with a multi-level checkpoint/restart technique maintains a given application efficiency with significantly lower PFS requirements than simple synchronous checkpointing, which write checkpoints such that all processes write their own checkpoints concurrently, and are blocked until the checkpoint operation completes.

With the multi-level asynchronous checkpointing, we make the following major contributions:

- ≤ the design of an asynchronous checkpointing system minimizing checkpointing overhead down to only a few percent;
- ≤ modeling of asynchronous checkpointing giving optimal checkpoint intervals, required file system throughput and the degree of the efficiency for execution of real-applications; and
- ≤ a comprehensive exploration of asynchronous checkpointing on current and future systems.

Especially, our results show that combining asynchronous and multi-level checkpointing results in highly efficient application runs with low PFS bandwidth requirements.

1.2.2 Exploration of Multi-tier Storage Design with Burst Buffer

Second, we explore multi-tier storage design for resilient architecture using burst buffers. Burst buffers have been proposed to alleviate the problems of I/O operations to a shared PFS [8, 9]. A Burst buffer is a new tier in the storage hierarchy to fill the performance gap between node-local storage and the PFS, and is shared by a subset of compute nodes. The new tire can absorb the bursty I/O requests from applications and thus can reduce the effective load on the PFS. We consider using burst buffers from different viewpoint, and try to improve system resiliency with burst buffer storage design. With burst buffers, an application can store checkpoints on a smaller number of dedicated burst buffer nodes, so the probability of lost checkpoints is decreased. We explore how burst buffers can improve efficiency and resiliency compared to using node-local storage instead of burst buffers based on a performance model combined with our multi-level asynchronous checkpoint/restart model.

The key contributions include:

≤ an Infiniband-based file system (IBIO) that exploits the bandwidth of burst buffers;
≤ a model to evaluate system resiliency given checkpoint/restart and storage configurations;
≤ simulation results showing how system resiliency improves from the use of burst buffers;
≤ an analysis of which tiers in the storage hierarchy impact system reliability and efficiency; and
≤ an exploration of burst buffer configurations to discover the best for future large-scale systems.

Especially, these contributions can benefit system designers in making the trade-offs in performance of components so that they can create efficient and cost-effective machines for future extreme scale systems.

1.2.3 FMI: Fault Tolerant Messaging Interface

Third, we propose FMI for fast and transparent recovery. FMI is a survivable messaging interface that uses fast, transparent in-memory checkpoint/restart and dynamic node allocation. With FMI, a developer writes an application using semantics similar to Message Passing Interface (MPI). The FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance, such as checkpointing, failure detections, and recoveries. All of this motivates the need for a survivable messaging runtime system. Such a system should be able to maintain processes and connections that are unaffected by the failure while starting and integrating replacement processes as needed.

The key contributions are:

≤ a simplified programming model to enable fast, transparent fault tolerance based on checkpoint/restart;

≤ implementation of a runtime that withstands process failures and allocates spare resources;

≤ a new overlay network structure called *log-ring* for scalable failure detection and notification; and

≤ and demonstration of the fault tolerance and scalability of FMI even with a MTBF of 1 minute.

Especially, our implementation of FMI has failure-free performance that is comparable with MPI, and our experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high mean time to failure of *1 minute*.

1.3 Outline

The dissertation is divided into seven chapters, and is organized as follows:

Chapter 2: Fault Tolerance on HPC Systems and the Challenges To clear our target failure models, we first introduce types of faults, errors, and failures. Then we present characteristics of failures on current supercomputers, such as the failure rate, and the localities. We also review prior fault tolerant techniques on supercomputers mainly in checkpoint/restart techniques. Then we list the existing challenges of the current approaches, *minimizing checkpoint overhead*; *efficient recovery*; *reliable storage architecture*.

Chapter 3: Asynchronous Checkpointing System We present the first approach, asynchronous checkpointing system, for *minimizing checkpoint overhead*. This chapter explains the asynchronous checkpointing system by using RDMA (Remote Direct Memory Access), and shows experimental results on current and simulated future supercomputers.

Chapter 4: A User-level Infiniband-based File System and Checkpoint Strategy for Burst Buffer We deliver the second approach, a burst buffer system with an infiniband-based file system, to explore *reliable storage architectures*. This chapter introduces burst buffer systems and infiniband-based file system for burst buffers, and explore the optimal checkpoint/restart strategies on burst buffers for future extreme scale supercomputers.

Chapter 5: FMI: Fault Tolerant Messaging Interface We propose the third approach, fault tolerant messaging interface (FMI), to achieve *efficient recovery*. This chapter explains the fast, scalable and transparent recovery algorithm on in FMI and delivers experimental results showing the scalability of FMI.

Chapter 6: Conclusions and Future Work Finally, we summarize the contributions made by our works and discuss possible directions for future research.

Chapter 2

Fault Tolerance on HPC Systems and the Challenges

To clear our target failure models, we first introduce types of faults, errors, and failures. Then we present characteristics of failures on current supercomputers, such as the failure rate, and the localities. We also review prior fault tolerant techniques on supercomputers mainly in checkpoint/restart techniques. Then we list the existing challenges of the current approaches, *asynchronous checkpointing*, *reliable storage design*, *fault tolerant messaging interface*

2.1 Failure Models

We first clarify what fault tolerance actually means for distributed HPC systems in order to under stand the role of fault tolerance. All the participants of the HPC domain involved in fault tolerance come with their own culture and perception of faults, errors, failures(fail), their origins and consequences [10]. In this dissertation, we explain based on Andrew's definition [11].

A malfunction event of a system can be categorized into a failure(fail), a error and a fault in terms of its degree of impact to system components. A system is said to *fail* when the system cannot perform its required functions within specified performance requirements. In particular, if a HPC system is designed to provide its users with execution environment where applications run properly, and output results at the end, the system has failed when an application does not complete the execution. For example, if a system crashes due to a component failure, such as CPU, GPU, DRAM, M/B, and local storage, the system is said to fail.

Table 2.1: System malfunction categories: Failure, Error, and Fault

Failure	Inability of a system or a component to perform required function according to its specification (e.g. machine crash)
Error	A part of a system state that may lead to a failure (e.g. bit flip error)
Fault	Cause of an error (e.g. aging hardware)

Table 2.2: System malfunction categories: transient, intermittent, and permanent

Transient	Occur once and then disappear
Intermittent	Periodically occur
Permanent	Continuously occur (until its faulty devices are repaired or replaced)

An *error* is a part of a system's state that may lead to a failure. For example, when transmitting messages across networks, it is to be expected that some packets of the messages have been damaged or dropped off before they arrive at the receiver. Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived. Another but more important example is *Silent Data Corruption (SDC)*. SDC occurs when incorrect data is delivered by a computing system to the user without any error being logged (Cristian Constantinescu, AMD). Errors may or may not cause failures. For example, if ECC (Error Correction Code) of a system is disabled and a bit flip occurs in a memory region, which is not being used, then nothing happens. But if the memory region is a pointer of a program and the bit flip direct the pointer to bad address, then the process crashes.

The cause of an error is called a *fault*. Clearly, finding out what caused an error is important. For example, a wrong or bad transmission device may easily cause packets to be damaged or dropped off. In this case, it is relatively easy to remove the fault by replacing the failed devices.

As mentioned above, there are three levels of malfunctions of a system. As we will show the details in Section 2.2, 2.3, the most common malfunction is *failure*, and the most of the existing studies target failures. Thus, in this dissertation, we target *failures*, which terminate processes, and prevent applications from continuously running.

In term of occurrence of system defects, the malfunction events of systems are also classified as *transient*, *intermittent*, or *permanent*. *Transient* failures, errors and faults are a brief malfunction that often occurs at irregular and unpredictable times, and usually occur once and then disappear. *Intermittent* failures, errors and faults occurs,

TABLE 2.3: TSUBAME2.0 failure analysis (of Thin nodes)

Failure level	approximate # of nodes affected	Failure points	Failure rate (failures/sec)	MTBF
5	1408	PFS, Core switch	$1.778 \otimes 10\ ^{8}$ (0.9%)	65.10 days
4	30	Rack	$1.332 \otimes 10\ ^{8}$ (0.7%)	86.90 days
3	15	Edge switch	$6.665 \otimes 10\ ^{8}$ (3.5%)	17.37 days
2	4	PSU	$3.999 \otimes 10\ ^{8}$ (2.1%)	28.94 days
1	1	Compute nodes	$1.757 \otimes 10\ ^{6}$ (92.7%)	15.8 hours

then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. *Permanent* failures, errors and faults are ones that continues to exist until the faulty component is replaced. Burnt-out CPUs, GPUs, and disk crashes are the examples of permanent failures, errors and faults. In this dissertation, we target any types of malfunctions of systems in Table 2.2.

2.2 Failure Analysis on Supercomputers

In this section, we show detailed failure analysis on several supercomputers and HPC systems. Especially, we analyze the failure rate and the locality.

2.2.1 The TSUBAME Supercomputer Case Study

First, we analyzed the failure history of TSUBAME2.0 to obtain its failure rate during the period of November 1, 2010 to April 6, 2012 based on a TSUBAME2.0 failure history [12]. TSUBAME2.0, ranking 5th in the Top500 list (November 2011) [13], experienced 962 failures ranging from memory errors to whole rack failure. In Table 2.3, we show the levels of failures that occurred during the period. The failures are ranked highest to lowest in terms of how many compute nodes are affected by a failure in a particular level. For example, failures of the PFS are at level 5, because the PFS is shared by all compute nodes; if it fails, all running processes that access the PFS fail.

Level 5 failures include failures of the PFS and core switches. *Core switches* are upper level switches, which connect lower level *edge switches*. A core switch is redundantly connected to edge switches in order to ensure quality of service. On a core switch failure, the running job using that switch fails. However, routing is updated and later jobs can communicate across processes without the failed core switches unless multiple core switches fails, then each compute node is physically disconnected. We categorize rack failures at level 4. Because one rack of TSUBAME2.0 contains 30 compute nodes, these

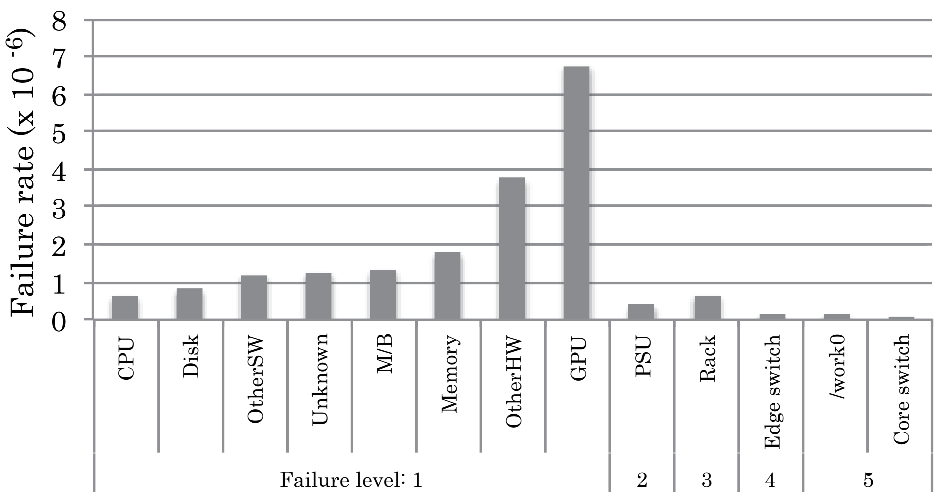


FIGURE 2.1: Break-down of system failure rates (failures/second)

failures affect 30 compute nodes at a time. Rack failures are usually caused by a faulty water-cooling unit sensor. Because the water-cooling unit sensors are independent from computing components, running jobs do not fail immediately. However, a job running on the rack will fail at some point, assuming the faulty sensor is not fixed. Failure levels 3, 2, and 1 include edge switch, PSU (power supply unit) and compute node failures. Because 15 compute nodes are connected to an edge switch, 15 compute nodes are affected on an edge switch failure, i.e., level 3 failure. Likewise, a PSU is shared by 4 compute nodes. Thus, on a failure of PSU, 4 compute nodes are affected.

As in Table 2.3, most failures are in level 1, affecting only a single node. The failure rate (failures/second) is approximately two orders of magnitude larger than the failure rates of the other categories. Because compute nodes are becoming more complex with increasing computational power, compute node failures are becoming more frequent. Figure 2.1 shows break-down of failure rates of each component on a compute node; approximately half of the failures on a compute node arise from GPU failures. On TSUBAME2.0, the GPU usage is becoming high, as more applications use the GPUs through CUDA, OpenACC or OpenCL. This burdens GPUs and increases the overall temperature, which results in high failure rate of GPUs. In the Table, we excluded fan failures of compute nodes because compute nodes usually have redundant fans, and the failures do not immediately affect the compute nodes. As a whole, we can see that the most common failure is a compute node failure, and fast recovery from the failure is important.

TABLE 2.4: `pf3D` failures on three different LLNL clusters; Coastal, Hera and Atlas [14]

	Coastal	Hera	Atlas	Total
Time span	Oct 09 - Mar 10	Nov 08 - Nov 09	May 08 - Oct 09	
Number of jobs	135	455	281	871
Node hours	2,830,803	1,428,547	1,370,583	5,629,933
`LOCAL required`	2 (08%)	36 (41%)	21 (26%)	59 (31%)
failure rate	$2.136 \otimes 10^{7}$	$5.714 \otimes 10^{6}$	$4.564 \otimes 10^{6}$	$3.497 \otimes 10^{6}$
MTBF	54.1 days	2.0 days	2.5 days	3.3 days
`PARTNER/XOR required`	18 (75%)	32 (37%)	54 (68%)	104 (54%)
failure rate	$1.922 \otimes 10^{6}$	$5.079 \otimes 10^{6}$	$1.174 \otimes 10^{5}$	$6.247 \otimes 10^{6}$
MTBF	6.0 days	2.3 days	1.0 days	1.9 days
`PFS required`	4 (17%)	19 (22%)	5 (06%)	28 (15%)
failure rate	$4.279 \otimes 10^{7}$	$3.016 \otimes 10^{6}$	$1.087 \otimes 10^{6}$	$1.510 \otimes 10^{6}$
MTBF	27.1 days	3.8 days	10.6 days	7.7 days
Total failures	24	87	80	191

2.2.2 Other Supercomputers' Case Study

In this section, we show failure rate on other systems. Table 2.4 shows categorization of each failure according to the checkpoint level that was required for recovery, which are collected by Moody et al.[4]. The checkpoint is based on a scalable checkpoint/restart library (SCR) [14].

SCR is a checkpoint/restart library that production LLNL (Lawrence Livermore National Laboratory) applications use. It supports globally-coordinated checkpoints that the application writes. SCR uses hierarchically distributed storage to implement a multi-level checkpointing system. For instance, SCR can cache the most recent checkpoints in RAM disks or SSDs that are local to the compute nodes (`LOCAL` checkpoint). SCR also applies redundancy schemes using simple replications or XOR encoding of checkpoint across the distributed storage devices (*Redundancy group* or *XOR group*) according to configurations such that the application can recover from most failures using the cached checkpoints. To recover from more catastrophic failures that cannot be recovered by `LOCAL`, `XOR`, and `PARTNER`, SCR can also store checkpoints on the PFS (`PFS`).

They analyzed the job logs of 871 runs of the `pF3D` laser-plasma interaction code [5], and aggregated to over 5.6 million node-hours on three dierent clusters. `LOCAL`, `PARTNER/XOR` and `PFS` required means that a failure requiring a recovery from `LOCAL`, `PARTNER/XOR` and `PFS` checkpoint occurred. In `LOCAL` checkpointing, each processes only cache a checkpoint in in-system storage of its own compute node, and recover from a failure by using the same set of compute nodes as before the failure occurs. The failures requiring

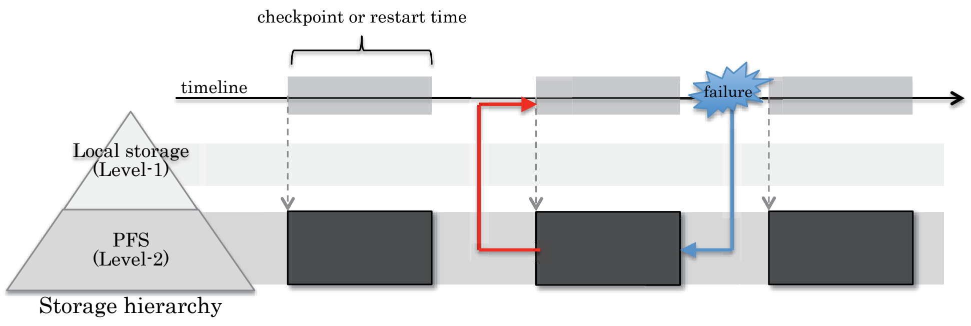


FIGURE 2.2: Simple Checkpoint/Restart

`LOCAL` is expected to be transient failures. `PARTNER/XOR` creates parity code across in-system storage on compute nodes within a redundancy group as in RAID-5 [15, 16, 17]. `PARTNER/XOR` checkpointing can recover from a single compute node failure within a redundancy group. From Table 2.4, we can find that the most common failure is a compute nodes failures that can be recovered from `LOCAL` or `PARTNER/XOR` checkpoint.

2.3 Fault Tolerant Techniques

As described in the previous section, fault tolerance is becoming more important as the system size increase for higher performance computing. Nowadays, fault tolerance is essential techniques for applications, which need to continuously run for days or weeks. A lot of fault tolerance techniques have been proposed over the pas years to address the problems. In this section, we introduce these fault tolerance techniques mainly in checkpoint/restart techniques as well as other state-of-the-art fault tolerant techniques. First, we introduce several checkpoint/restart techniques in Section 2.3.1, which are our target fault tolerance techniques. Second, we review other fault tolerant techniques in Section 2.3.2.

2.3.1 Checkpoint/Restart

Checkpointing is one of indispensable fault tolerance techniques, commonly used by HPC applications that run continuously for hours or days at a time. A checkpoint is a snapshot of application state that can be used to restart execution if a failure occurs (Figure 2.2). However, when checkpointing extreme scale systems, tens of thousands of compute nodes write checkpoints to a PFS concurrently, and the low I/O throughput becomes a bottleneck. Figure 2.3 presents expected checkpoint/restart time using PFS in order to dump whole data on distributed memories across TSUBAME2.0/2.5 thin nodes. TSUBAME2.0/2.5 has 1,408 thin nodes, which account for over 95% of TSUBAME

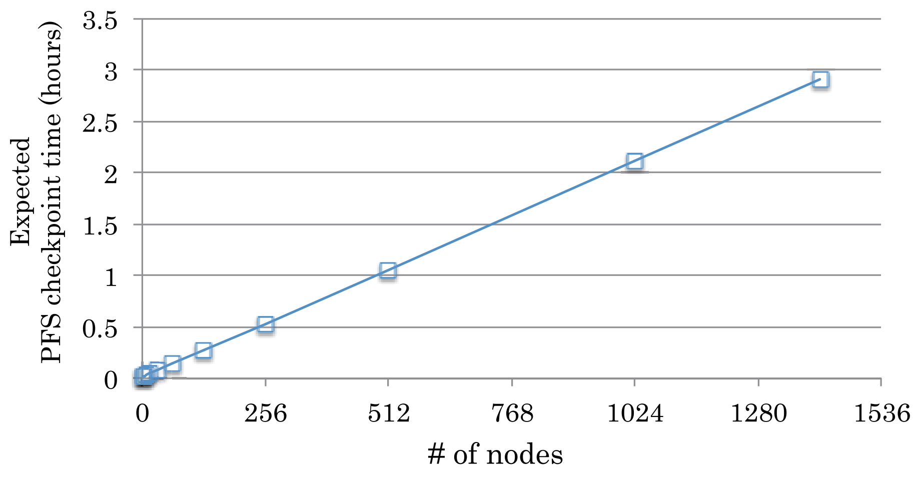


FIGURE 2.3: Expected checkpointing time to PFS

TABLE 2.5: Potential System Architecture

Systems	2011 (K computer)	2019	Difference (2011 & 2019)
System peak	10.5 PFlops	1 EFlops	⌊ (100)
Power	12.7 MW	20 MW	⌊ (1)
System memory	1.6 PB	32 - 64 PB	⌊ (10)
Node performance	128 GFlops	1,2 or 15 TFlops	⌊ (10) ⌊ (100)
Node memory BW	64 GB/s	2 - 4 TB/s	⌊ (100)
Node concurrency	8	1K or 10K	⌊ (100) ⌊ (1000)
Node Interconnect BW	20 GB/s	200 - 400 GB/s	⌊ (10)
System size (nodes)	88,124	⌊ (100K) or ⌊ (1M)	⌊ (10) ⌊ (100)
Total concurrency	705,024	⌊ (1 *billion*)	⌊ (1K)
MTTI	days	⌊ (1*day*)	⌊ (1)

compute nodes. As shown, it approximately takes over three hours to write checkpoints of all the data on thin nodes.

Thus, although simple, this straightforward checkpointing scheme can impose huge overheads on application run times. In addition, future extreme scale systems are expected to have more memory modules by one order of magnitude. Reducing checkpoint time is critical for future extreme scale computing.

In this section, we explain the latest checkpoint/restart techniques to address the above problems. These checkpoint/restart techniques include *diskless checkpoint/restart*, *multi-level checkpoint/restart*, *uncoordinated checkpoint/restart*, *checkpoint compression* and *incremental checkpoint* techniques.

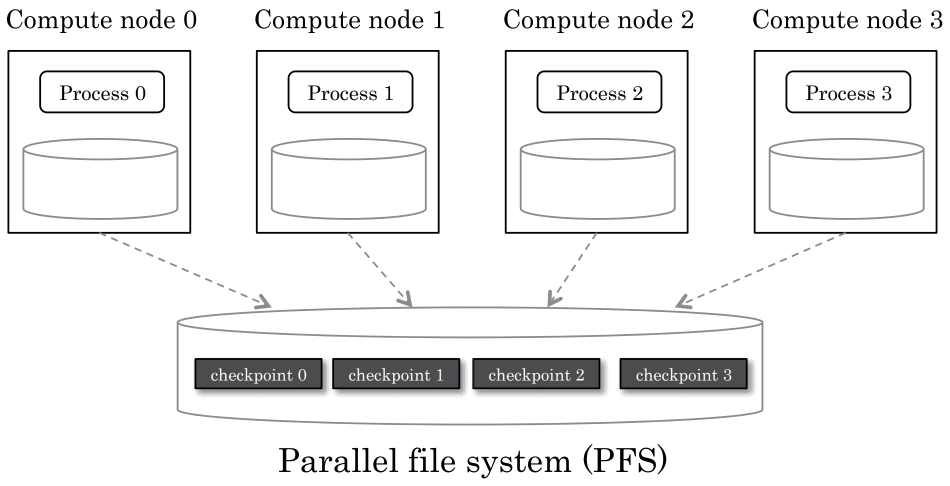


FIGURE 2.4: Low I/O throughput of PFS becomes a bottleneck for checkpoint/restart

Diskless checkpointing is a technique to store checkpoints without relying on PFS. Instead, diskless checkpointing utilize local storage by using encoding techniques. Because diskless checkpoint use distributed in-system storage, the checkpointing technique is scalable. However, diskless checkpointing is not a silver bullet. Diskless checkpointing distributes encoded redundant data across distributed storage on compute nodes. If we lose all of the redundant data due to a catastrophic failure by concurrent multiple compute nodes, we cannot restart the execution on such a failure. As we seen in Section 2.2, the common failure is a single compute node failure. Although a failure by multiple nodes does not occur frequently, we need write checkpoints to PFS for more catastrophic failures. Multi-level checkpoint/restart is a technique to solve the problem. Multi-level checkpoint/restart uses multiple storage levels, such as RAM, node-local storage, and the PFS, according to the different degrees of resiliency and the checkpoint/restart time in those storage levels. Multi-level checkpoint/restart systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures, and the PFS for more catastrophic failures. By taking frequent, inexpensive node-local checkpoints, and less frequent, high-cost checkpoints to the PFS, applications can achieve both high resilience and better efficiency.

Diskless and Multi-level checkpointing reduce checkpointing time by making use of a storage hierarchy. Other methods are minimization of checkpointing time by reducing checkpoint size. For the methods, we introduce two famous techniques, checkpoint compression and incremental checkpointing. The other methods we review in this section is uncoordinated checkpoint/restart. When we take a checkpoint, all distributed processes usually need to synchronize in order to write *consistent* checkpoint, which introduce the checkpoint overhead. With uncoordinated checkpointing, each process can write its

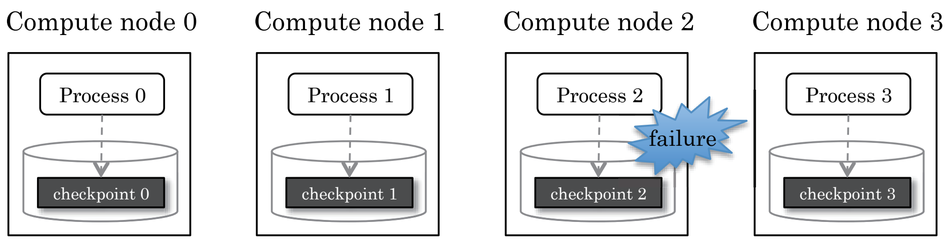


FIGURE 2.5: Checkpointing to node-local storage: If compute node 2 fails, the application lose access to checkpoint 2. The application can not recover form the failure

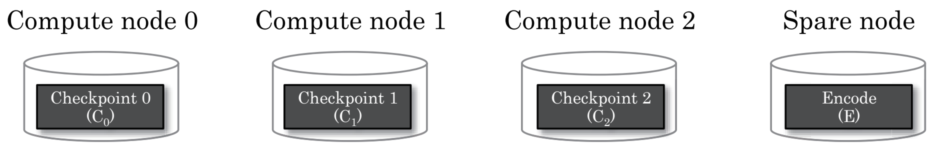


FIGURE 2.6: Generate redundant data by erasure encoding techniques, and then store the original (C_i) and encoded (E) checkpoint to main memory of compute nodes and spare nodes respectively

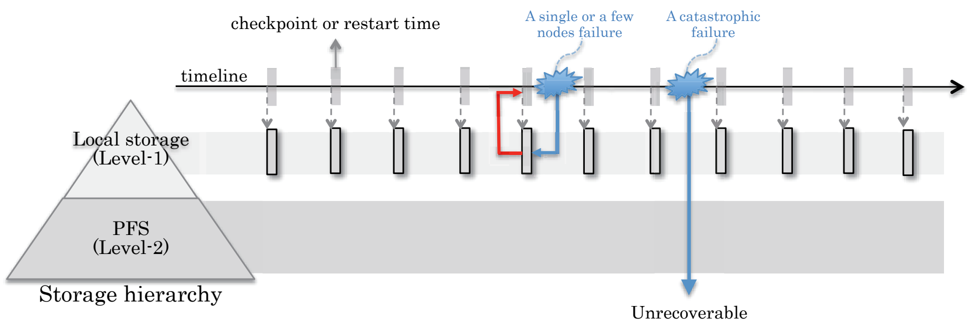


FIGURE 2.7: Instead, diskless checkpointing stores checkpoints on distributed in-system storage, and apply a redundancy scheme

checkpoint without synchronizations, and reduce the number of processes to restart. Finally, we also explain about application-level and system-level checkpointing, which have trade-off between usability and performance. For the rest of the sections, we explain the details of these checkpoint/restart techniques.

Diskless Checkpoint/Restart

First methods we review are diskless checkpointing techniques [1]. Diskless checkpoint was proposed by Plank et al. [18, 19]. Diskless checkpointing stores checkpoints without relying on a slow PFS. Instead, diskless checkpointing stores checkpoints on distributed in-system storage such as main memory and node-local storage (e.g. SSD) as in Figure

[1]The term may lead to misunderstanding because *diskless* does not mean not using disks. This approach does not rely on PFS, but may rely on disks if the local-storage is a disk drive insted of a SSD

2.7. However, if a compute node failed and an application loses access to the failed in-system storage, the application can not recover form the checkpoint (Figure 2.5). To address the problem, on checkpointing, diskless checkpointing generates redundant data by erasure encoding techniques, and then store the original (C_i) and encoded (E) checkpoint to main memory of compute nodes and spare nodes respectively (Figure 2.6).

The most common encoding is *XOR encoding*. XOR encoding is an encryption algorithm that uses an exclusive disjunction operation ($\bullet$). If we assume that an application does diskless checkpoint with n compute nodes, the XOR encoding is computed as:

$$E = C_0 \bullet C_1 \bullet \times\times\times\bullet C_{n\ \ 1} \tag{2.1}$$

If checkpoint, C_i, is lost due to a failure of the compute nodes, the lost checkpoint can be restored as:

$$C_i = C_0 \bullet C_1 \bullet \times\times\times\bullet C_{i\ \ 1} \bullet C_{i+1} \bullet \times\times\times\bullet C_{n\ \ 1} \bullet E \tag{2.2}$$

Thus, in Figure 2.6, an application can restore the lost checkpoint, C_2, by computing $C_2 = C_0 \bullet C_1 \bullet E$ on a failure of compute node 2. This method is simple, and the encoding overhead is relatively small. However, If more than one compute node fail, we cannot restore the lost checkpoints.

Another encoding technique, which is widely used for encoding checkpoints of HPC applications, is a *Reed-Solomon encoding* [20]. A Reed-Solomon encoding is particularly useful for burst failures by a failure of concurrent multiple compute nodes. With a Reed-Solomon encoding, an application can recover from any k failures of compute nodes. Essentially this technique is based on an XOR encoding technique described above, but the encoding uses a distribution matrix (size: $n + k \otimes n$), $A = \begin{bmatrix} I \\ M \end{bmatrix}$, where the upper matrix of the distribution matrix (size: $n \otimes n$) is an identity matrix, I, and the lower matrix is a matrix (size: $k \otimes n$), M. The matrix, M, has to satisfy a condition that any sub-square matrix including minor is non-singular. It is known that any Vandermnde matrix and any Cauchy matrix satisfy this condition. If we assume set of checkpoints of each compute nodes is $C = \}C_0, C_1, \ldots, C_{n\ \ 1}\backslash$, encoded checkpoints are calculated as $A \otimes C^T = \begin{bmatrix} C^T \\ E \end{bmatrix}$. Although a Reed-Solomon encoding is more fault tolerant than XOR encoding, a Reed-Solomon encoding costs larger overhead, and produce a larger amount of encoding data to store.

Plank et al. [21] also presented first exploration of diskless checkpointing for distributed systems. They proposed high-performance implementations of several fault tolerant algorithms using diskless checkpointing. These algorithms include Cholesky factorization,

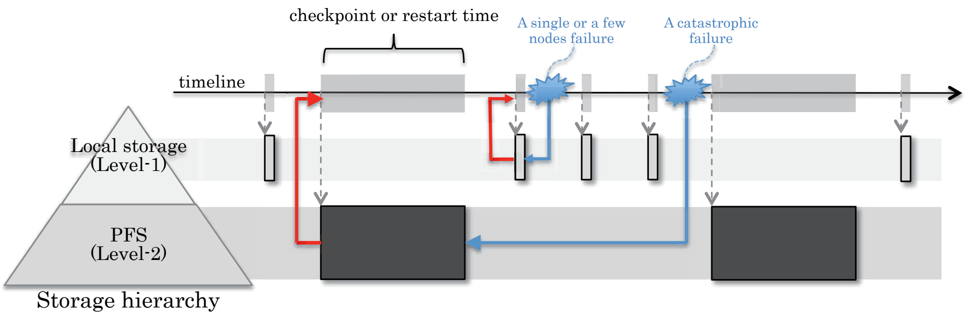


FIGURE 2.8: Multi-level checkpoint/restart

LU factorization, QR factorization, and Preconditioned Conjugate Gradient. Chen et al. [22] proposed a chain-piplined Reed-Solomon encoding algorithm for diskless checkpointing. The technique organizes all computational processors and the checkpoint processor as a chain, and segment the data on each processor into smalll pieces. With the algorithm, they reduced the encoding overhead to survive k failures in p processes from $2\rceil \log p\{ \ast k \left((\beta + 2\gamma)\, m + \alpha \right)$ to $\left(1 + \lfloor \left(\frac{1}{m}\right)\right) \ast k \left((\beta + 2\gamma)\, m \right)$, where α is the communication latency, $1/\beta$ is the network bandwidth between processes, $1/\gamma$ is the rate to perform calculations, and m is the size of local checkpoint per process.

However, the above studies have several issues. First, the techniques assume there are spares nodes for storing checkpoints, which decreas available compute nodes for applications. Seconds, main memory is used to store encoded checkpoints, which decreases available memory capacity for applications. To address the problem, Bautista-Gomez et al. [23] proposed distributed diskless checkpointing using weighted checksum encoding and a fault tolerant model. The distributed diskless checkpointing does not rely on spare nodes for encoded checkpoints with a partitioning and mapping technique. The distributed diskless checkpointing runtime utilize local SSDs to solve the memory overhead introduced by the classic diskless checkpoint techniques above.

Diskless checkpointing can recovery from the most common failure, a single or a few node failure, and also exhibit high scalability as the number of compute nodes increases towards future extreme scale systems.

Multi-level Checkpoint/Restart

As described in Section 2.3.1, diskless checkpointing is a scalable algorithm while recovering the most common failure. However, diskless checkpointing is not a silver bullet. Diskless checkpointing distributes encoded redundant data across distributed storage on compute nodes. If we lose all of the redundant data due to a catastrophic failure by concurrent multiple compute nodes, we can not restart the execution on such a failure.

For example, if more than one compute node fail, applications can not restart form the failure with XOR encoding. Even with Reed-Solomon encoding, applications may not be able to recover form a failure if the failure is catastrophic to restore the checkpoints (Figure 2.7). As we seen in Section 2.2, the common failure is a single compute node failure and a failure by multiple nodes does not occur frequently, but we need write checkpoints to PFS for more catastrophic failures.

Multi-level checkpoint/restart is a technique to solve the problem. Multi-level checkpoint/restart uses multiple storage levels, such as RAM, node-local storage, and the PFS, according to the different degrees of resiliency and the checkpoint/restart time in those storage levels. Multi-level checkpoint/restart systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures, and the PFS for more catastrophic failures. By taking frequent, inexpensive node-local checkpoints and apply cross-node redundancy schemes (e.g. XOR encoding, Reed-Solomon encoding), and also less frequent, high-cost checkpoints to the PFS, which can withstand a failure of an entire machine, applications can achieve both high resilience and better efficiency (Figure 2.8).

The Scalable Checkpoint/Restart library [4] is a first production multi-level checkpoint/restart library for MPI applications. Checkpoints are cached in in-system storage on compute nodes, and then a redundancy scheme is applied. SCR provides two redundancy schemes, `XOR` encoding, `PARTNER`. `PARTNER` is a simple replication creating the replicated checkpoint on a partner node. SCR also flushes the checkpoints in node-local storage to a PFS with specified frequency. The Fault Tolerance Interface (FTI) [7] is also production multi-level checkpoint/restart library for MPI applications. The main difference between FTI and SCR is the redundancy scheme used for node-local checkpointing. FTI use Reed-Solomon encoding for the redundancy scheme, and generate encoded checkpoints in the background of application's computation to minimize the overhead. Although the time of Reed-Solomon encoding is much longer than XOR encoding, FTI provides higher resiliency than SCR.

Checkpoint Size Reduction Techniques

Checkpointing methods in Section 2.3.1, 2.3.1 reduce checkpoint/restart time by storing checkpoint to node-local storage, and then applying redundancy schemes. Another method minimizes the checkpoint/restart time by reducing checkpoint size. The most common methods are *Checkpoint compression* and *incremental checkpointing*. Checkpoint compression reduces checkpoint size by applying a compression technique or combinations of several compression techniques. Incremental checkpointing is a checkpoint

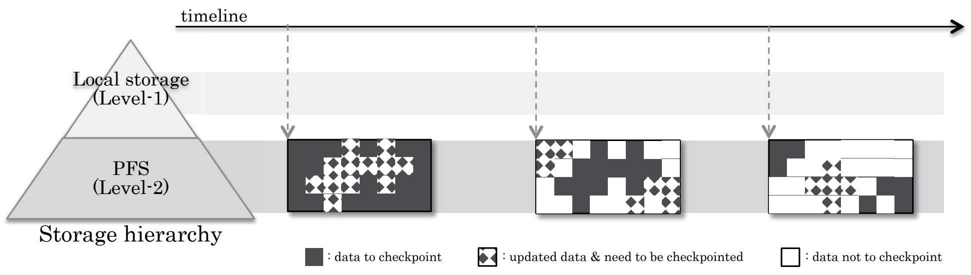


FIGURE 2.9: Incremental checkpoint writes only data which is update from the last checkpoint

reduction technique that only writes variables or data which is updated from the last checkpoint (Figure 2.9).

For checkpoint compression, a simple method is to use existing fast compression libraries such as Parallel-Gzip [24], fpzip [25], FPC [26] and QuickLZ [27]. However, compression is relatively costly operation, and compression time (T_c) and write time for the compressed checkpoint (T_{wc}) may exceed write time for the none non-compressed checkpoint (T_w), i.e., $T_c + T_{wc} > T_w$. The same is true in decompression. In such a case, compression is not effective for reducing checkpoint/restart time. Ibtesham et al. [28] explored the feasibility of checkpoint compression techniques for efficiently reducing checkpoint sizes with checkpoint compression viability model to determine when checkpoint compression is a sensible choice. The preliminary experiments showed that checkpoint compression is a useful solution if the product of checkpoint throughput is less than the product of compression factor and compression throughput. Islam et al. [29] developed MCRENGINE. MCRENGINEaggregates checkpoints form multiple processes and compresses them through widely-used I/O libraries, e.g., HDF5 [30] and netCDF[31]. Further, MCRENGINEextracts application specific data semantics by analyzing the metadata contained in the checkpoint files, and apply compression to distributed variables or data across multiple processes instead of their own local variables or data. They showed that the data-aware checkpoint compression significantly increases the compressibility of checkpoints.

Another method to reduce checkpoint is incremental checkpointing. There are several incremental checkpoint/restart techniques. The simple technique is to use *binary diff* such as `bsdiff`, `bspatch` and `xdelta`. As described for checkpoint compression, if time of updated data detection (T_d), and write time of incremental checkpoint (T_{wd}) exceed checkpoint time without incremental checkpointing (T_w), i.e., $T_d + T_{wd} > T_w$, incremental checkpointing is not effective solution for reducing checkpoint/restart time, and the straightforward approach may not work.

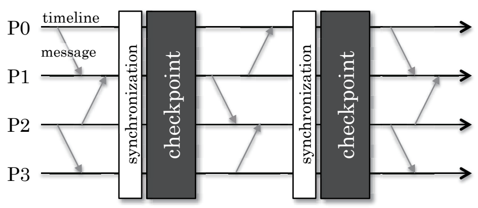


Figure 2.10: Coordinated checkpointing: synchronization across all processes is required to ensure the checkpoints are consistent

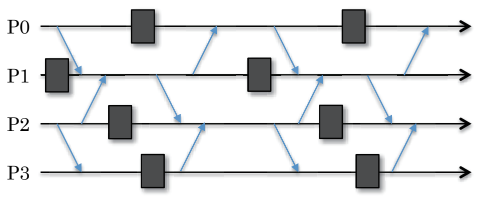


Figure 2.11: Uncoordinated checkpointing: message logging is required to ensure the checkpoints are consistent

Agarwal et al. [32] presented an adaptive incremental checkpoint technique. The incremental checkpointing dynamically identify near-exact changed blocks in memory to minimize size of the data to store in the file system. They initially splits checkpoint data into blocks of equal sizes. Then the block size is refined based on the access pattern history of the application at every checkpoint. This approach finds near-optimal sizes of the changed blocks over a period of time. Naksinehaboon et al. [33] proposed incremental checkpointing model, and explore the improvement of incremental checkpointing compared to a conventional checkpointing storing entire application's data.

When keeping track of updated data, incremental checkpointing uses a hash-based table by page or block-wise granularity. But computation of hash values is costly because checkpoint size is usually large. In addition, if only one bit in a page is updated, the whole page is marked as a dirty page, and is written in the next incremental checkpointing. The problem increases incremental size. Ferreira et al. [34], proposed hash-based incremental checkpointing using GPUs. This approach detects updated memory data with byte-wise granularity to minimize incremental size. They developed `libhashckpt` that provide the page protection scheme, and compute hashes using a GPU to reduce the overhead of the hash computation.

Uncoordinated Checkpoint/Restart

Generally, checkpoint/restart techniques can be categorized into two approaches, *coordinated* and *uncoordinated checkpoint/restart.* With coordinated checkpoint/restart, all processes globally synchronize before taking checkpoints to ensure the checkpoints are consistent and that no messages are in flight (Figure 2.10). Coordinated checkpoint/restart is applicable to a wide range of applications. However, the global synchronization can cause overhead due to propagation of system noise at extreme scale [35]. In addition, when checkpointing to or restarting from a PFS, tens of thousands of compute nodes concurrently write or read checkpoints, which can cause large overhead due to contention. Meanwhile, uncoordinated checkpointing does not require global synchronization, and allows processes to write/read checkpoints at different times, which lowers checkpoint overhead (Figure 2.10). However, with uncoordinated checkpointing there may be messages in flight from one process to another when a checkpoint is taken, which can cause the so-called *domino effect* [36] preventing an application from rolling-back to the last checkpoint at restart. To handle this, uncoordinated checkpointing libraries log messages, i.e. *message logging* [37, 38, 39]. But message logging has its own overhead problem. Thus, reducing message logging overhead is critical for uncoordinated checkpointing while avoiding domino effects.

To reduce the message logging overhead, several techniques are proposed. Bouteiller [40] proposed a hybrid approach combining coordinated and uncoordinated checkpointing. The technique divides processes into correlated process groups (*cluster*). When checkpointing, each process synchronizes within its cluster, and writes a consistent checkpoint (coordinated checkpointing). Meanwhile, each cluster asynchronously write checkpoints without synchronization with the other clusters. The approach can reduce message logging overhead because only messages, which are transferred between clusters, need to be logged.

Guermouche et al. [41] proposed a new uncoordinated checkpointing for *send-deterministic* MPI applications. Send-deterministic applications send messages in the same order as long as it's correct execution [42]. By taking advantage of the send-determinacy, only a small subset of application messages are required to be logged, which can reduce the message overhead. Bautista-Gomez et al. [43] proposed a hierarchical and hybrid approach combining with multi-level checkpoint/restart and uncoordinated checkpoint/restart. The approach create clusters giving a good trade-off between the message logging overhead and restart time based on a graph partitioning technique [44], then creates subclusters with in the cluster in order to maximize the reliability with Reed-Solomon encoding.

Checkpoint/Restart Models

Another technique to reduce checkpoint/restart time is optimization of checkpointing interval. Optimization of checkpoint interval is critical. For example, if we write checkpoint frequently, it reduces the wasted computation on a roll-back, but applications end up spending the most of time for checkpointing. Meanwhile, if we write checkpoint infrequently, applications can spend much more time for its computation, but much more computations are wasted on a failure. Thus, checkpoint interval optimization, which gives a good trade-off between reliability and checkpoint overhead, is important.

Several optimization techniques have been studied. Young [45] proposed a method to determine the optimal checkpoint interval for single-level, synchronous checkpoints. Vaidya extended the model to support asynchronous checkpoints, called *forked checkpoints* [46], and two-level checkpoints [47]. Vaidya also combined both methods to support a two-level asynchronous checkpoint system to achieve higher efficiency [48]. Vaidya's model assumes that at most one fast-level checkpoint is taken between each slow-level checkpoint. However, in current multi-level checkpointing systems, the fastest checkpoints, which are often saved to node-local storage, are orders of magnitude faster than the slowest checkpoints saved to the PFS. To account for this, we extend prior work to model an arbitrary number of node-local checkpoints between consecutive PFS checkpoints.

2.3.2 Other Techniques Related to Fault Tolerance

Optimization of I/O operations itself is also critical approach to reduce checkpoint/restart time because checkpoint/restart is an I/O intensive operation. Adding a new tier of storage is one solution. Rajachandrasekar et al. [49] presented a staging server which drains checkpoints on compute nodes using RDMA (Remote Direct Memory Access), and asynchronously transfers them to the PFS via FUSE (Filesystem in Userspace). Other solutions are I/O overhead hiding techniques. Asynchronous I/O has long been used to hide the I/O bottlenecks by checkpoint/restart. These techniques enable applications to parallelize I/O with computation to increase CPU utilization and to enhance I/O performance. Patrick et al. [50] presented a comprehensive study of different techniques of overlapping I/O, communication, and computation, and showed the performance benefits of asynchronous I/O. Nawab et al. [51] asynchronously transfer multiple striped TCP data streams to increase I/O performance in Grid environments. An asynchronous staging service using RDMA proposed by Hasan et al. [52] is the closest research to our asynchronous checkpointing system. The authors achieved high I/O throughput by using additional nodes. As we will observe, optimizing performance requires determination of the proper number of staging nodes for a given number of compute nodes. However, the comprehensive study on the problem is not shown nor do they present their solution. To deal with bursty I/O requests, Liu et al. [53] simulated a system design that integrates SSD storage on I/O nodes as burst buffers. They found that such a system could deliver high I/O bandwidth to an application while reducing the demands on the PFS. In this dissertation, we effectively implement a system with burst buffers on the compute nodes, and through experiment and modeling, we arrive at the same conclusions regarding the need for and benefits of asynchronous I/O to the PFS.

The state-of-the-art I/O techniques themselves are limited in their ability to improve I/O efficiency at extreme scale because I/O time depends on underlying I/O device performance. Wickberg et al. [54] introduced an aggregated DRAM buffer on top of the PFS called RAMDISK Storage Accelerator (RSA). RSA constructs a low latency and high bandwidth buffer on the fly, and asynchronously stages in files ahead of execution coupled with I/O scheduler. Kannan et al. [55] also presented a data staging approach using active NVRAM (Non-volatile RAM) [56] technology.

Most of all applications exchanges messages during their runtime using messaging interfaces such as MPI. Fault tolerant capabilities in messaging interfaces are important.

Fault tolerant messaging runtimes will be critical for applications to be able to recover from failures on future systems. FT-MPI[57] and ULFM (User Level Fault Mitigation)[58] implement fault tolerant capabilities on top of MPI. In these libraries, failures are visible to applications, so users need to write fault tolerant code, such as communicator recovery and checkpoint/restart. In other efforts, [59, 60], the libraries write checkpoints transparently on top of MPI, but recovery is not transparent to applications. Adaptive MPI[61], developed on top of Charm++[62], provides transparent recovery using checkpoint/restart. However, failed processes are relaunched on existing nodes instead of spare nodes, which can cause unbalanced process affinity, and performance degradation after recovery.

2.4 Towards Optimal Checkpoint/Restart

To achieve more efficient checkpoint/restart, existing techniques are not sufficient. This clarifies the differences between existing techniques and our approaches, *asynchronous checkpointing* (in Chapter 3), *reliable storage design* (in Chapter 4), and *fault tolerant messaging interface* (in Chapter 5).

2.4.1 Light-weight Asynchronous Checkpointing and the Model

Table 2.6: Comparison to existing multi-level asynchronous checkpoint/restart

	SCR [4]	FTI [7]	Our approach (in Chapter 3)
Multi-level Checkpoint/Restart	Yes (XOR)	Yes (Reed-Solomon)	Yes (XOR)
Asynchronous checkpointing	Yes (w/o stating)	Yes (w/o staging)	Yes (w/ staging)

Multi-level checkpointing is one of promising approaches for extreme scale computing. Several implementations exist [4, 7].

SCR [4] is the first implementation of multi-level checkpoint/restart using XOR encoding for diskless checkpointing. SCR also supports asynchronous checkpointing to PFS. FTI also implements multi-level checkpoint/restart with Reed-Solomon encoding for diskless checkpointing, and asynchronous checkpointing to PFS. On asynchronous checkpointing, however, both checkpoint/restart runtimes directly write checkpoints from compute nodes to PFS. As will be described in Section 3.2.2, if we directly write checkpoints from compute nodes to PFS, the net cost to the application increase. A lighter weight asynchronous checkpoint method to minimize the impact on application is important. Our asynchronous checkpointing system solves the problem through staging nodes running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS (in Chapter). In Chapter 3, we describe more details.

Table 2.7: Comparison to existing checkpoint/restart model

	Young [45]	Vaidya [48]	Moody et al. [4]	Our approach (in Chapter 3)
Multi-level Checkpoint/Restart	No (Single level)	Yes (Two levels)	Yes (Arbitrary levels)	Yes (Arbitrary levels)
Asynchronous checkpointing	No	Yes (to only computation)	No	Yes

Although reducing checkpointing overhead is important, the other factors, such as checkpoint interval and frequency of each level of checkpoint, are also critical to performance. However, as we described in Section 2.3, existing checkpointing models are not sufficient to the asynchronous checkpointing. We highlights the existing models in Table 2.7. Although the latest model supports arbitrary levels of checkpoint/restart model [4], the model does not support asynchronous checkpointing. Meanwhile, Vaiday's model [48] supports asynchronous checkpointing. However, the model only support two-level case, and the model assumes that the asynchronous checkpointing to PFS does not overlap with other levels of faster checkpointing. In current multi-level checkpointing, the fastest checkpoints, which are often saved to node-local storage, are orders of magnitude faster than the slowest checkpoints saved to the PFS. Asynchronous checkpointing can overlaps other levels of checkpointings as well as computation. We must account for this matter. We describe how we models the asynchronous checkpointing in Chapter 3.

2.4.2 Multi-tier Storage Design with Bust Buffer

Burst buffer itself is not new technology. As described in Section 2.3.2, Liu et al. [53] already simulated a system design that integrates SSD storage on I/O nodes as burst buffers. However, they did not investigate the reliability when we use burst buffer for a checkpointing target. We describe how we models burst buffer architectures, and exploit them in Chapter 4.

2.4.3 Fault Tolerant Messaging Interface

TABLE 2.8: Existing approaches

	FT-MPI [57] (ULFM [58])	SCR [4]	Charm++ [62]	Our approach (in Chapter 5)
Survivable	Yes	No	Yes	Yes
Failure detection	Yes (w/ explicit call)	No	Yes (Not scalable)	Yes (Scalable)
Auto-recovery	No	No	Yes	Yes
MPI Compatibility	Yes	Yes	Yes (w/o FT feature)	Yes (w/ additonal function calls)
Checkpoint	No	Yes	Yes (Only partner)	Yes (XOR)

As described in Section 2.3.2, a survivable messaging interface itself is not new techniques, and there are a few existing survivable messaging interfaces. FT-MPI [57] and ULFM [58] are survivable messaging interfaces. However only processes, which issues communication request to failed processes, can detect the failure. To globally notify the failure to all running processes, users need to explicitly call a *revoke* function. In addition, for recovery, the interfaces do not provide any automated recovery schemes. The users need to handle the failures, and write codes to store checkpoints, and restore communication and application states to restart its computation. SCR [4] provide scalable checkpoint/restart capability for MPI applications, but the runtime is not a messaging interface. Charm++ [62] is also survivable messaging interface, which detects process failures, recovery all contexts to restart its computation, and write checkpoints with scalable methods (Copying checkpoints to another local-disk or in-memory space). However, global failure detection is not scalable because a process, which noticed a failure, sends the notification to the other processes. At extreme scale, the scalable global notification is critical. In addition, Charm++ applications are written by different programming model to MPI applications. To support MPI applications, Charm++ provide MPI-based APIs, but several fault tolerant capabilities, such as auto-recovery, becomes unavailable on the MPI-based APIs. At extreme scale, the rack of the fault tolerance capabilities of existing survivable messaging interfaces will make fault tolerant programming complicated. Thus, a survivable messaging interface, which provides scalable failure detection, auto-recovery, and fast checkpoint/restart for MPI applications, is critical. We describe our fault tolerant messaging interface, which solve the above problems in Chapter 5.

Chapter 3

Asynchronous Checkpointing System

The first approach, *multi-level asynchronous checkpointing*, solves the checkpointing overhead to PFS. The multi-level asynchronous checkpointing writes checkpoints through agents running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers application checkpoint overhead by overlapping computation and writing checkpoints to the PFS. Also, it reduces PFS load by using fewer concurrent writers and moderating the rate of PFS I/O operations. Our experiments show that the asynchronous checkpointing system can improve efficiency by 1.1 to 2.0 times on future machines. Additionally, applications using the asynchronous checkpointing system can achieve high efficiency even when using a PFS with lower bandwidth.

3.1 Introduction to Multi-level Asynchronous Checkpointing System

As described in Chapter 2, Checkpointing is an indispensable fault tolerance technique, commonly used by HPC applications that run continuously for hours or days at a time. A *checkpoint* is a snapshot of application state that can be used to restart execution if a failure occurs. However, when checkpointing large-scale systems, tens of thousands of compute nodes write checkpoints to a parallel file system (PFS) concurrently, and the low I/O throughput becomes a bottleneck. Although simple, this straightforward checkpointing scheme can impose huge overheads on application run times.

Multi-level checkpointing [4, 7] is a promising approach for addressing these problems. This approach uses multiple storage levels, such as RAM, local disk, and the PFS, according to the different degrees of resiliency and the checkpoint/restart time in those storage levels. Multi-level checkpointing systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures, and the PFS for more catastrophic failures. By taking frequent, inexpensive node-local checkpoints, and less frequent, high-cost checkpoints to the PFS, applications can achieve both high resilience and better efficiency.

However, computational capabilities are increasing faster than PFS bandwidths. This imbalance in performance means applications can be blocked for tens of minutes for a single checkpoint [4]. Thus, the overhead of checkpointing to the PFS can dominate overall application run time even with infrequent PFS checkpoints. Further, the huge numbers of concurrent I/O operations from large-scale jobs burden the PFS and are themselves a major source of failures. Thus, we must reduce the PFS load in order to achieve high reliability and efficiency.

Our asynchronous checkpointing system solves this problem through agents running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers application checkpoint overhead by overlapping computation and writing checkpoints to the PFS. Also, it reduces PFS load by using fewer concurrent writers and moderating the rate of PFS I/O operations. In particular, our asynchronous checkpointing system maintains a given application efficiency with significantly lower PFS requirements than synchronous checkpointing.

We make the following major contributions:

≤ The design of an asynchronous checkpointing system;

≤ A detailed failure analysis of a petascale supercomputer;

≤ A Markov model of asynchronous checkpointing on top of an existing multi-level checkpointing system;

≤ A comprehensive exploration of asynchronous checkpointing on current and future systems.

Our results show that combining asynchronous and multi-level checkpointing results in highly efficient application runs with low PFS bandwidth requirements.

3.2 Scalable Checkpoint/Restart

Our asynchronous checkpointing system is developed as an extension to the SCR library [63]. Our system asynchronously transfers node-local checkpoints written by SCR to the PFS.

3.2.1 Checkpoint/Restart Scheme

SCR is a checkpoint/restart library that production LLNL applications use. It supports globally-coordinated checkpoints that the application writes, primarily as a file per MPI process – a common checkpointing technique in large-scale codes. SCR uses hierarchically distributed storage to implement a multi-level checkpointing system. For instance, SCR can cache the most recent checkpoints in RAM disks or SSDs that are local to the compute nodes on which the application is running, and it can also store checkpoints on the PFS.

SCR applies redundancy schemes to the distributed storage devices such that the application can recover from most failures using the cached checkpoints. When these storage devices are local to and scale with the number of compute nodes, the cost to cache a checkpoint is low, so the application can checkpoint frequently. However, this approach cannot handle failures involving many nodes so SCR periodically copies (flushes) a cached checkpoint to the PFS in order to recover from those more severe, but less frequent failures. After a failure, SCR tries to restart from the most recent checkpoint in cache. If it cannot, it fetches and restarts from the most recent checkpoint on the PFS.

3.2.2 SCR Flush

As mentioned, to withstand catastrophic failures, such as a rack-level failure or a data center-wide power outage, SCR periodically flushes a checkpoint from node-local storage to the PFS. SCR supports two types of flush operations: synchronous and asynchronous. When SCR copies a checkpoint to the PFS synchronously, it blocks the application until the copy has completed. In large-scale computations on tens of thousands or more compute nodes, the total checkpoint size can reach multiple terabytes, which may take tens of minutes to complete. Thus, synchronous flushes can dominate application run time. SCR also supports an asynchronous flush in which it starts the flush and immediately returns control to the application. An external and independent process that runs in the background then copies the checkpoint to the PFS, so the application is not blocked during this transfer.

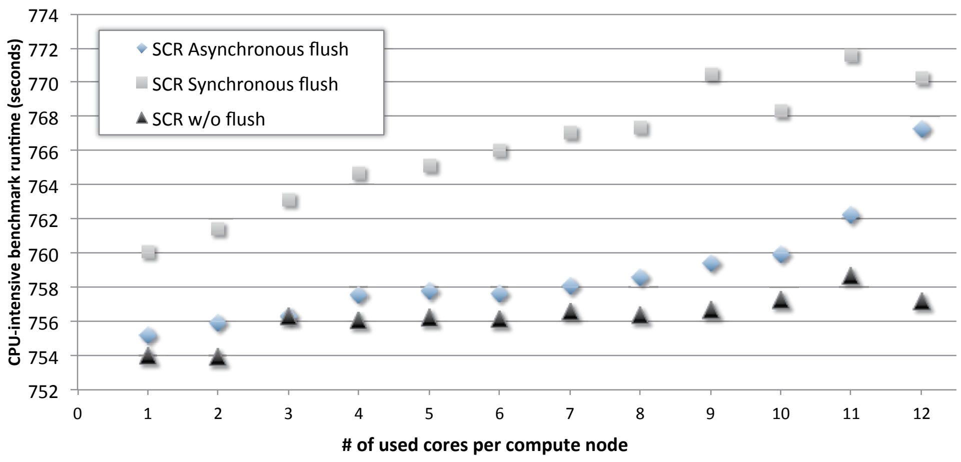


FIGURE 3.1: IOR run time on a 12-core machine with SCR

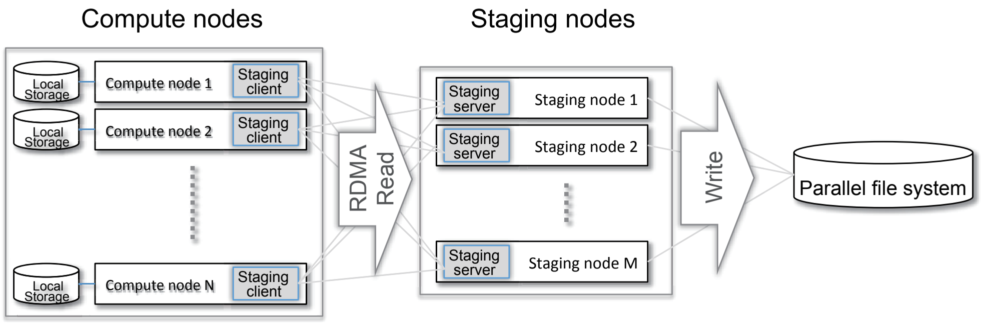


FIGURE 3.2: The asynchronous checkpointing system

In the existing asynchronous flush implementation, an additional process runs on each compute node to read data from node-local storage and to write data to the PFS. Even though this process self-throttles its run time, it uses CPU time and other resources that can impact the application. Figure 3.1 shows the run time of the IOR benchmark [64] using SCR with synchronous flush, asynchronous flush, and no flush on the LLNL Sierra system, a 1944-node Linux cluster with 12 cores per node. The result shows, if we consume a core to conduct asynchronous checkpointing, then we contend with and slow down the application. The net cost to the application when it uses all 12 cores is similar to the cost of simply halting the application until it finishes writing a synchronous checkpoint to the PFS. Motivated by this result, in this work we develop a lighter weight asynchronous checkpoint method to minimize the impact on application run time.

3.3 Asynchronous Checkpointing System

Overlapping I/O with computation by delegating operations to dedicated I/O nodes improves application performance and mitigates PFS workload [52, 65]. Further, multi-level checkpointing reduces the frequency of PFS checkpoints. By combining long intervals between consecutive PFS checkpoints with asynchronous flushes (i.e., *asynchronous checkpoints*), we can copy data to the PFS at a slow rate to reduce the impact on the application as well as the PFS. This section details the design of our asynchronous checkpointing system.

3.3.1 Architecture

As Figure 3.2 shows, our asynchronous checkpointing system has two types of nodes: *compute nodes* and *staging nodes*. The compute nodes are the nodes on which the application executes. The staging nodes are a group of nodes that we use to transfer checkpoints from the compute nodes to the PFS. The staging nodes asynchronously read checkpoint data from the compute nodes and write data to the PFS while the application continues to execute and to write node-local checkpoints. Generally, each staging node handles multiple compute nodes; we determine the exact ratio through modeling and experimental testing (See Sections 3.4 and 3.5).

A *staging client* process runs on each compute node, and a *staging server* process runs on each staging node. When SCR finishes caching a checkpoint (node-local checkpoint) that is to be flushed to the PFS, it signals the staging client process via a library function call. The staging client then sends a request to the staging server and the two processes execute a protocol to transfer the checkpoint; details appear in Section 3.3.2. The staging server reads checkpoints from the compute nodes using Remote Direct Memory Access (RDMA) to minimize CPU usage on the compute nodes.

Using additional nodes to transfer checkpoint data with RDMA provides our asynchronous checkpointing system with two advantages. First, it minimizes compute node CPU usage. On a flush operation, even asynchronous checkpointing can impact application runtime as Figure 3.1 shows. With RDMA, the asynchronous checkpointing system drains a checkpoint from compute nodes to the PFS while minimizing the impact on application runtime. Because staging nodes are independent from compute nodes, we can coordinate between the staging nodes to throttle the RDMA read rate without impacting the performance of the running application.

Second, staging nodes can support balancing overall data center I/O. A PFS is often a shared resource. Ill-timed I/O patterns between two applications accessing the PFS can

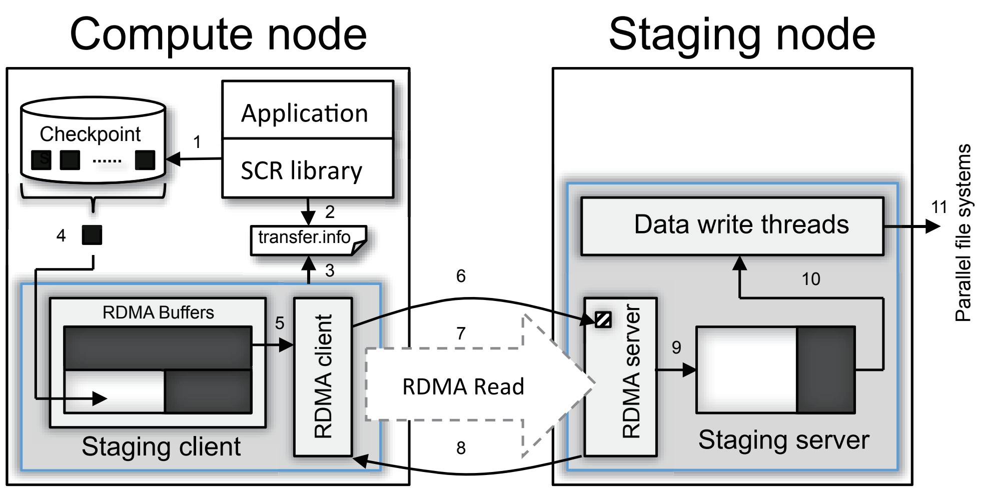


FIGURE 3.3: Asynchronous checkpointing client/server using RDMA

reduce the performance of both applications, which is particularly likely with checkpointing since it is one of the most I/O intensive operations. Staging nodes write checkpoints to the PFS independently of compute node activities, which allows us to throttle I/O to the PFS without directly throttling the application I/O rate on the compute nodes. This paper focuses on minimizing CPU usage; we leave I/O throttling techniques for optimizing overall data center I/O as future work.

3.3.2 RDMA Checkpoint Transfers

We implement an RDMA transfer system for asynchronous checkpointing based on the SCR library [63]. The existing SCR asynchronous flush implementation executes an extra process on each compute node, which reads a checkpoint from local storage and directly writes that checkpoint to the PFS. This extra process does substantial work on the compute node, and so it contends with and slows down the application. In contrast, our staging client process that runs on the compute node does minimal work, and most of the effort is delegated to the staging server process on the staging node. Other checkpoint management, such as versioning, checkpoint location, and redundancy scheme, relies on the original SCR library.

Figure 3.3 illustrates the architecture of our design with an example. First, assume that SCR has cached a checkpoint in local storage (Step 1). After applying its redundancy scheme to this checkpoint, SCR writes information into a file called *transfer.info* requesting that the checkpoint be copied to the PFS (Step 2). Among other information, the transfer.info file includes the source path of the checkpoint files in local storage as well as the destination paths to which the files should be written on the PFS.

The staging client periodically checks the transfer.info file for requests, and sleeps for the rest of the time. Thus, our design does minimal work on the compute node (Step 3). If the staging client detects a new request, it reads the checkpoint file from the source path and copies the data to a local RDMA buffer (Step 4). Once the staging client fills the buffer, it calls an RDMA client function to initiate the data transfer (Step 5). Since the RDMA client function returns control immediately, the staging client can read the next chunk to one of the buffer entries in the buffer pool (Figure 3.3 shows the double-buffering case) while the RDMA client transfers checkpoint chunks to the staging server. The RDMA client sends a short message containing details about the checkpoint and the RDMA buffer to the RDMA server (Step 6). When the RDMA server receives this message, it issues an RDMA read request to read a chunk of the remote buffer space into a local buffer (Step 7). Then it sends an acknowledgment message to the RDMA client (Step 8), and it copies the data to the staging server buffer (Step 9). This protocol continues until all checkpoint data has been copied from the staging client to the staging server. Finally, data writer threads write the checkpoint to the PFS in parallel with RDMA reads by the RDMA server (Steps 10 and 11).

An RDMA operation can only read or write remote memory regions of a few MB of data. Thus, we divide a checkpoint into smaller chunks, which the RDMA server remotely reads one by one. To reduce the number of staging nodes, a transfer server can concurrently handle RDMA requests from multiple transfer clients. However, a large amount of incoming checkpoint data can cause buffer overflow on a staging node. Thus, our staging client, which runs on the compute node, reads a small chunk of data from the compute node storage to a registered memory region for RDMA. The staging server then pulls the chunk (*incoming*) region to its own space and writes to the PFS (*outgoing*). If buffered checkpoint data on a staging node exceeds a specified buffer size limit, the staging server throttles the RDMA read rate (*incoming*) to balance incoming and outgoing checkpoint data. To avoid imbalance in incoming and outgoing data, we determine the number of staging nodes according to outgoing PFS throughput.

3.3.3 Synchronous and Asynchronous Checkpointing

Figure 3.4 shows the difference between synchronous and asynchronous checkpointing. To clarify the differences, we characterize both schemes with two metrics, *checkpoint overhead* and *checkpoint latency*. Checkpoint overhead (C) is the increased execution time of an application because of checkpointing. Checkpoint latency (L) is the time to complete a checkpoint.

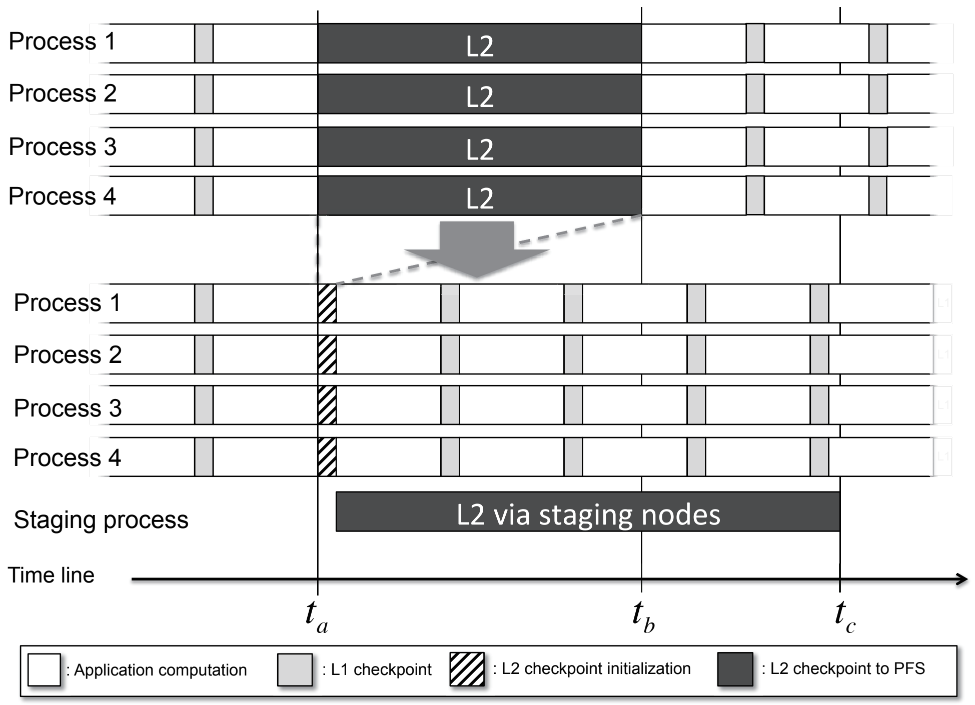


FIGURE 3.4: Asynchronous checkpointing can hide L2 checkpoint overhead

During synchronous checkpointing, each process writes its checkpoint data to the PFS, and blocks until the checkpoint operation completes. Thus, the checkpoint overhead is identical to the checkpoint latency, i.e., $C_{blk} = L_{blk} = t_b \quad t_a$. N iterations of synchronous checkpointing increase application run time by $N \otimes C_{blk}$.

With asynchronous checkpointing, each process continues computation during the PFS checkpoint so checkpoint overhead $({}_{nblk})$ is generally smaller than checkpoint latency $(L_{nblk} = t_c \quad t_a)$. Application characteristics determine C_{nblk} and L_{nblk}. If an application is computation or network bound, C_{nblk} and L_{nblk} increase due to resource contention, and L_{nblk} can become larger than L_{blk}. To initiate an asynchronous checkpoint, the application must write its checkpoint data to local storage. During the write operations, the application is blocked so we add this overhead to C_{nblk}.

Asynchronous checkpointing has advantages over synchronous checkpointing. We can minimize C_{nblk} by slowly writing checkpoint data to the PFS, thereby alleviating resource contention. Because lower-level checkpoints can continue to be cached on the compute nodes during an asynchronous checkpoint, the application can take more frequent checkpoints and increase resiliency with low checkpoint overhead. In contrast, when an application takes a synchronous PFS checkpoint, the application loses C_{blk} potential computation time and it is significantly more vulnerable to failure, as heavy PFS load increases the likelihood of PFS failures.

Thus, we intuitively expect asynchronous checkpointing to be more efficient than synchronous checkpointing. However, asynchronous checkpointing has a disadvantage. In Figure 3.4, the synchronous checkpoint completes at t_b while the asynchronous checkpoint finishes at t_c. If a failure that requires a PFS checkpoint occurs in the period between t_b and t_c, an asynchronous checkpointing system incurs a catastrophic rollback to an older checkpoint. Alternatively, synchronous checkpointing only rollbacks to t_b. Therefore, with asynchronous checkpointing, the checkpoint interval, C_{nblk}, L_{nblk}, and the frequency of each level of checkpoint must be optimized to lower the risk of the catastrophic rollback.

3.4 Asynchronous Checkpointing Model

As mentioned previously, with asynchronous checkpointing, several factors are critical to performance: checkpoint interval, C_{nblk}, L_{nblk}, and frequency of each level of checkpoint. To determine the optimal values, we extend an existing model of a multi-level checkpointing system [4] to support our asynchronous checkpointing system.

3.4.1 Assumptions

For simplicity, we make several assumptions in our asynchronous checkpointing model. Because we build on an existing model, we include that model's assumptions [4]. We highlight the important assumptions here.

We assume that failures are independent across components and occur following a Poisson distribution. Thus, a failure within a job does not increase the probability of successive failures. In reality, some failures can be correlated. For example, failure of a PSU can take out multiple nodes. However, topology-aware techniques can provide very low probability of those failures affecting processes in the same XOR set, which eliminates the need to restart from the PFS. SCR also excludes problematic nodes from restarted runs. Thus, the assumption implies that the average failure rates do not change and dynamic checkpoint interval adjustment is not required during application execution.

We also assume that the times to write and to read checkpoints are constant throughout job execution. In reality, I/O performance can fluctuate because of contention for shared PFS resources. However, staging nodes serve as a buffer between the compute nodes and the PFS. Thus, our system mitigates PFS performance variability.

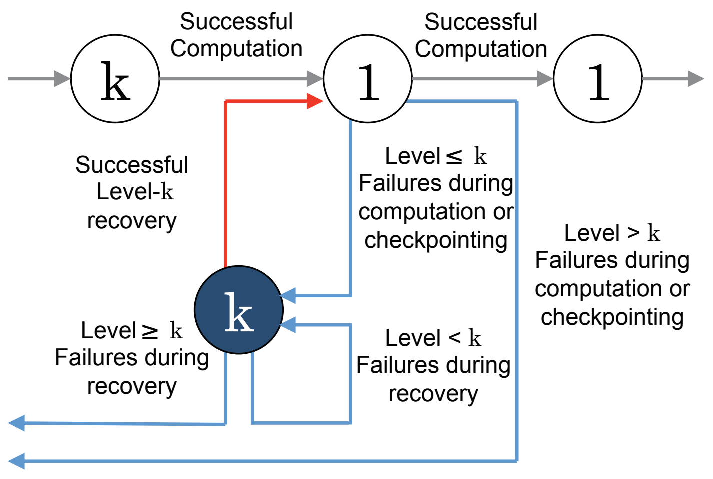

Successful Computation
Successful Computation
k
1
1
Successful Level-k recovery
Level ≤ k Failures during computation or checkpointing
Level > k Failures during computation or checkpointing
k
Level ≥ k Failures during recovery
Level < k Failures during recovery

FIGURE 3.5: The basic structure of the asynchronous checkpointing model

If a failure occurs during asynchronous checkpointing, we assume that checkpoints cached on failed nodes have not been written to the PFS. Thus, we must recover the lost checkpoint data from redundant stores on the compute nodes, if possible, and if not, locate an older checkpoint to restart the application. We can use either an older checkpoint cached on compute nodes, assuming multiple checkpoints are cached, or a checkpoint on the PFS.

3.4.2 Basic model structure

As in the existing model [4], we use a Markov model to describe run time states of an application. We construct the model by combining the basic structures that Figure 3.5 shows. The basic structure has *computation* (white circle) and *recovery* (blue circle) states labeled by a checkpoint level. The computation states represent periods of application computation followed by a checkpoint at the labeled level. The recovery state represents the period of restoring from a checkpoint at the labeled level.

If no failures occur during a compute state, the application transitions to the next right compute state (gray arrow). We denote the checkpoint interval between checkpoints as t, the time of a level c checkpoint as c_c, and rate of failure requiring level k checkpoint as λ_k. The probability of transitioning to the next right compute state and the expected time before transition are $p_0(t + c_c)$ and $t_0(t + c_c)$ where:

$$\begin{aligned} p_0(T) &= e^{\;\lambda T} \\ t_0(T) &= T \end{aligned}$$

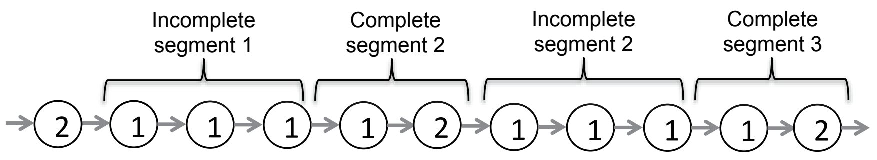
We denote λ as the summation of all levels of failure rates, i.e., $\lambda = \sum_{i=1}^{L} \lambda_i$ where L represents the highest checkpoint level. If a failure occurs during a compute state, the application transitions to the most recent recovery state that can handle the failure (blue arrow). If the failure requires a level i checkpoint or less to recover and the most recent recover state is at level k where $i \geq k$, the application transitions to the level k recovery state. The probability of a level i failure in an interval $t + c_c$, and the expected run time before the transition from the compute state c to the recovery state k are $p_i(t + c_c)$ and $t_i(t + c_c)$ where:

$$
\begin{aligned}
p_i(T) &= \frac{\lambda_i}{\lambda}(1 \quad e^{\quad \lambda T}) \\
t_i(T) &= \frac{1 \quad (\lambda T + 1) \times e^{\quad \lambda T}}{\lambda \times (1 \quad e^{\quad \lambda T})}
\end{aligned}
$$

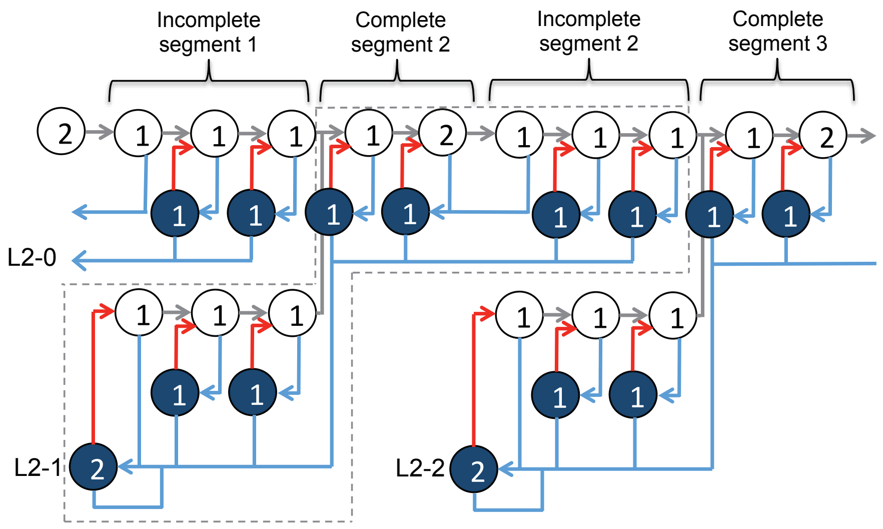
During recovery, if no failures occur, the application transitions to the compute state at the restored checkpoint. If the recovery time from a level k checkpoint is r_k, the probability of the transition and the expected run time are $p_0(r_k)$ and $t_0(r_k)$. If a failure requiring a level i checkpoint occurs while recovering, and $i < k$, we assume the current recovery state can retry the recovery. However, if $i \approx k$, we assume the application must transition to a higher-level recovery state. This assumption in recovery states supposes a pessimistic case. For example, if a compute node fails (level $\hat{k}$ failure) during level $\hat{k}$ recovery, and the failed nodes are distributed across different XOR groups, then the application can still recover from the multiple level $\hat{k}$ failures via the level $\hat{k}$ recovery state. However if multiple failed nodes are in the same XOR group, the application can not recover from the failures. Thus, the assumption supposes the latter pessimistic case. The probabilities and expected times of failure during recovery are $p_i(r_k)$ and $t_i(r_k)$. We also assume that the highest level recovery state (level L), which uses checkpoints on the PFS, can be restarted in the event of any failure $i \geq L$. The detailed derivation processes of $p_i(T)$ and $t_i(T)$ are described in [14].

3.4.3 Asynchronous checkpoint model

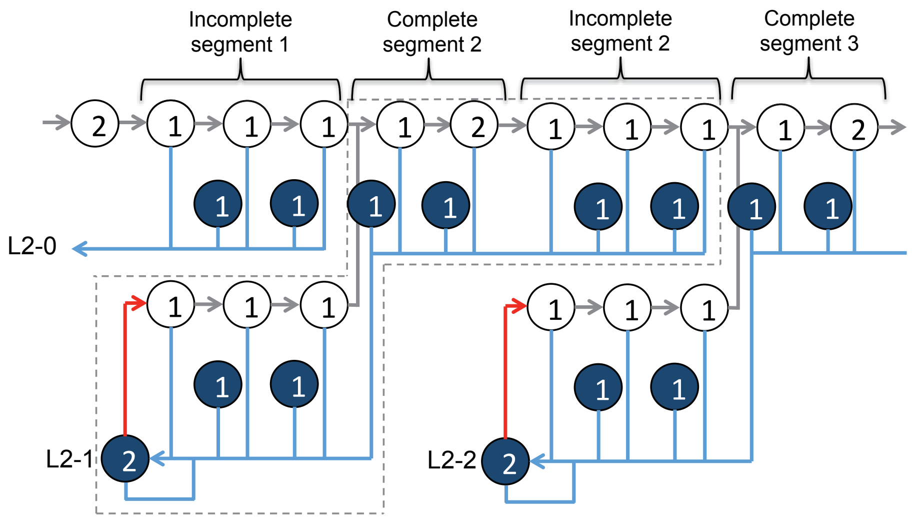
Our model of asynchronous checkpointing combines the basic structures from Figure 3.5. Figure 3.6 shows a two level example. If no failures occur during execution, the application transitions across the compute states in sequence (Figure 3.6(a)). In this example, level 1 (L1) checkpoints (e.g., XOR checkpoints) are synchronous and level 2 (L2) checkpoints (e.g., PFS checkpoints) are asynchronous. With synchronous checkpointing, the checkpoint is available when the corresponding compute state completes. Thus, if an L1 failure occurs, the application transitions to the most recent L1 recovery state (Figure 3.6(b)). Alternatively, with asynchronous checkpointing, the L2 checkpoint is not



(a) No failures and no recoveries



(b) Level-1 (L1) failures and recoveries



(c) Level-2 (L2) failures and recoveries

FIGURE 3.6: Transition diagram of the asynchronous checkpointing

finished when the L2 compute state completes. Thus, we divide compute states into two segment types: *incomplete segments* and *complete segments*. A computation state in an incomplete segment represents a period when the L2 checkpoint has started but is not yet completed. For example, if an L2 failure occurs in incomplete segment 1, the application transitions to the recovery state for the last completed L2 checkpoint (L2-0

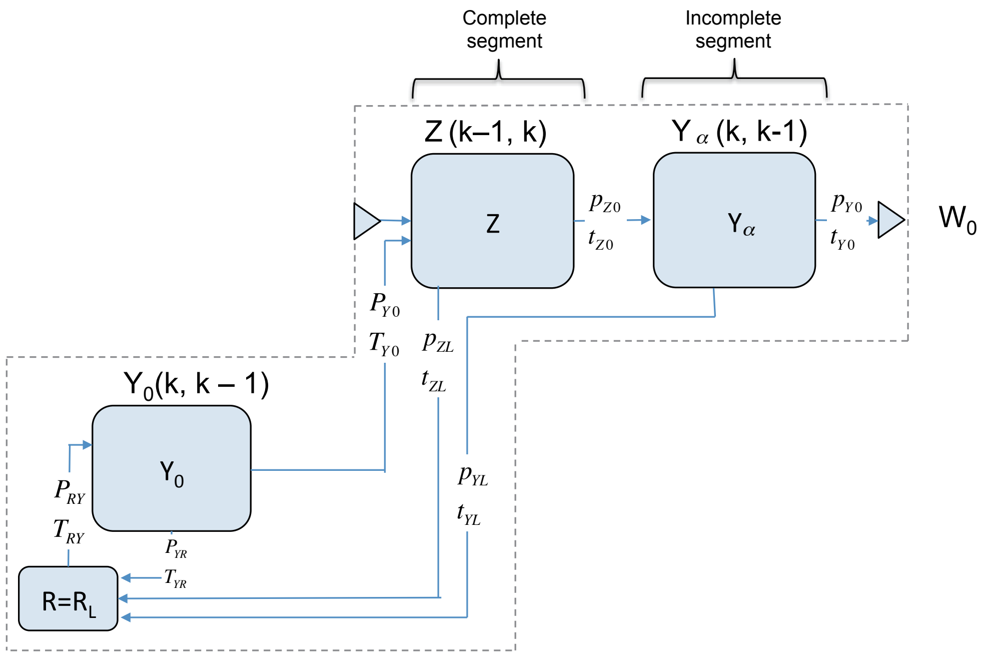


FIGURE 3.7: General asynchronous model structure: $W(k)$ state

in Figure 3.6(c)). When an L2 failure occurs in a complete segment 2, the application transitions to the recovery state for the completed L2 checkpoint (L2-1 in Figure 3.6(c)).

Interference from other asynchronous operations can inflate the time in compute states before transitions. An overhead factor, α, quantifies the overhead on compute states in incomplete segments. We define the time spent in compute states in an incomplete segment as $(1 + \alpha)t$, where t is the sum of the computation time and the time to complete a checkpoint. Thus, the probability and expected time for compute states in an incomplete segment, become $p_0(T)$, $t_0(T)$, $p_i(T)$ and $t_i(T)$ where $T = (1 + \alpha)\,t + c_c$.

Throughout our evaluation (Section 3.5), we employ two-level checkpointing, but our model can be extended to an arbitrary N-level checkpointing model. Figure 3.7 shows the general structure of our asynchronous checkpointing model. Our model is composed of hierarchical states, with the outermost state denoted as a $W(k)$ *state*. The definition of $Z(k, c)$ and its probabilities and expected times for transition to other states were defined in the existing model [14]. To distinguish compute states in complete and incomplete segments, we extend the definition of $Y(k, c)$ to reflect the interference as $Y_x(k, c)$, where x denotes the overhead factor. Using the model, we can compute the *expected time* to complete a given number of compute states with an arbitrary number of checkpointing levels. By merging the states, we can calculate the expected time of $W(k)$ *state* as in Appendix A.

Table 3.1: Sierra Cluster Specification

Nodes	1,856 compute nodes (1,944 nodes in total)
CPU	2.8 GHz Intel Xeon EP X5660 ⊗ 2 (12 cores in total)
Memory	24 GB (Peak CPU memory bandwidth: 32 GB/s)
Interconnect	QLogic InfiniBand QDR

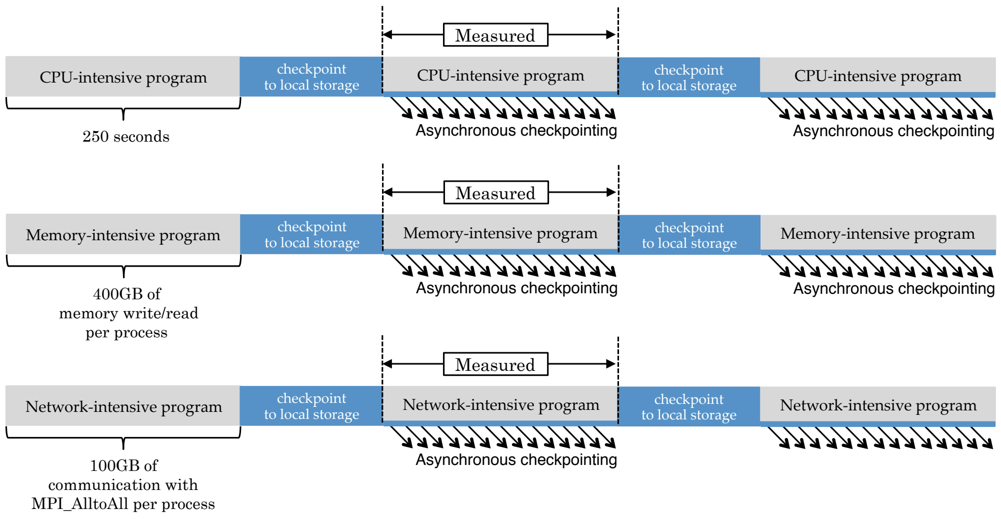

Measured
CPU-intensive program
checkpoint to local storage
CPU-intensive program
checkpoint to local storage
CPU-intensive program
250 seconds
Asynchronous checkpointing
Asynchronous checkpointing
Measured
Memory-intensive program
checkpoint to local storage
Memory-intensive program
checkpoint to local storage
Memory-intensive program
400GB of memory write/read per process
Asynchronous checkpointing
Asynchronous checkpointing
Measured
Network-intensive program
checkpoint to local storage
Network-intensive program
checkpoint to local storage
Network-intensive program
100GB of communication with MPI_AlltoAll per process
Asynchronous checkpointing

Figure 3.8: Experimental setups for (Top) CPU-intensive benchmark, (Middle) Memory-intensive benchmark, and (Bottom) Network-intensive benchmark

3.5 Evaluation

This section compares our asynchronous checkpointing system to the existing SCR implementation [4]. For illustration, we model a two-level system in which the first level uses SSD on the compute nodes with a RAID-5-like redundancy scheme and the second level is a PFS.

3.5.1 Asynchronous Checkpointing Overhead

First, we evaluate the asynchronous checkpointing system on the Sierra cluster at LLNL using a PFS designed to deliver a peak performance of 30GB/s. Each Sierra node has two 2.8 GHz 6-core Intel Xeon 5660 processors (12 cores in total) and 24GB of memory, and nodes share a QLogic IBA7322 QDR InfiniBand HCA (4X) interconnect.

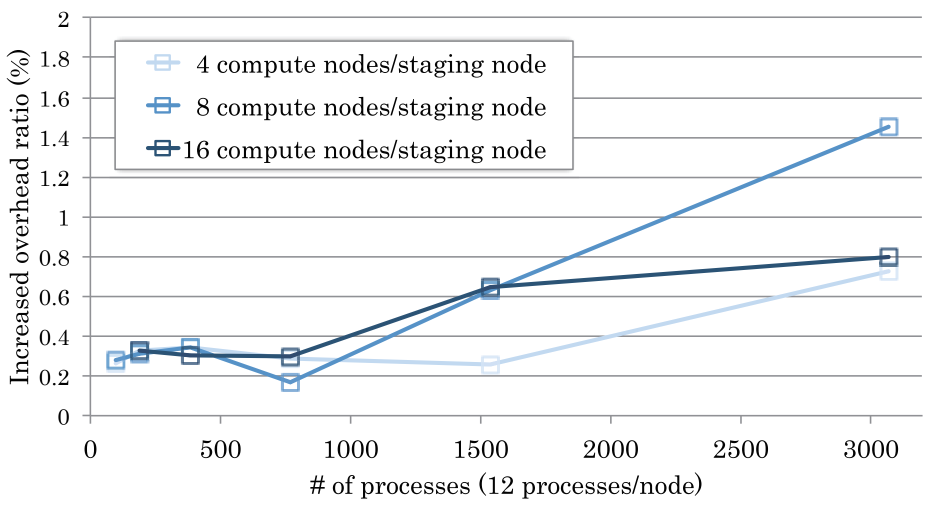


FIGURE 3.9: IOR Runtime with ATS varying number of compute nodes

CPU-intensive benchmark

We use the IOR benchmark [64] developed by LLNL, which measures I/O performance using POSIX, MPI I/O, or HDF5 with several access patterns. IOR has a main loop that executes read/write operations followed by a *sleep-delay* interval. The number of loop iterations is specified by the user along with the number of seconds for the delay interval. We ran IOR to write a 200 MB per-process checkpoint files to local storage through the HDF5 API. IOR repeats this write operation three times with a 250 second sleep interval between writes, for 750 seconds of delay in total. We write node-local checkpoints to RAM disk (/tmp) and flush them to a Lustre PFS [66]. To emulate a compute-bound scientific application writing checkpoints periodically, we replace the *sleep-delay* with a CPU-intensive loop based on the *do_work*() function in the ATS (APART Test Suite) [67]. The *do_work*() function repeatedly generates a random index using the C *rand*() function, copies a value of one array at the index to another array at the same index a specified number of times. We calibrate this computation time to take 250 seconds.

Figure 3.9 shows increased IOR runtime ratio to 250 seconds of ATS run with an increasing number of MPI processes, and different number of staging nodes. In the evaluation, the MPI processes occupy all cores on computes node, i.e., 12 MPI processes per compute node (Figure 3.8).

The staging clients uses a 200MB of internal double buffers to flush each checkpoint to the PFS after IOR writes it. Meanwhile, staging servers use a 20GB buffer. We vary the number of compute nodes, which a staging node handles, from 4 to 16 in increasing powers of 2. The first asynchronous checkpointing starts at the end of the first checkpoint to local storage. The first call of *do_work()* is not affected by asynchronous checkpointing.

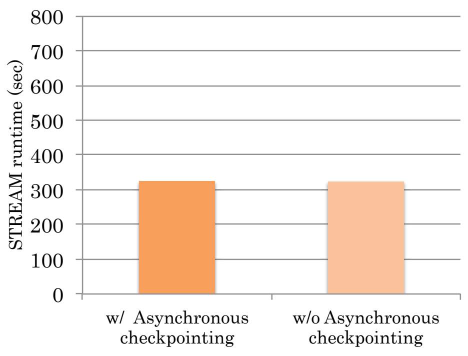


FIGURE 3.10: IOR runtime with STREAM w/ and w/o asynchronous checkpointing

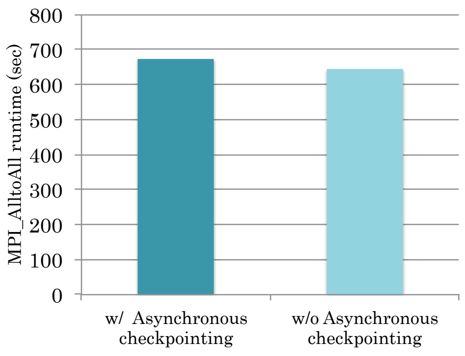


FIGURE 3.11: IOR runtime with MPI_AlltoAll w/ and w/o asynchronous checkpointing

Thus, we measure runtime of second *do_work()* execution. As presented in Figure 3.9, our asynchronous checkpointing can flush checkpoints to the PFS with less runtime overhead. Since the asynchronous checkpointing uses RDMA to drain checkpoint data from compute nodes to the PFS, our system uses less CPU, which can reduce the impact on the application. Even with over 3,000 processes, the overhead ration is less than 2%.

Therefore, we found that performance analysis and modeling of our asynchronous checkpointing system are required to determine the appropriate number of staging nodes given the number of compute nodes, the checkpoint size, network and effective I/O throughput. In addition, we use the *do_work*() function of ATS, which is a CPU-bound function. However since our asynchronous checkpointing system shares a network bandwidth, it is expected to affect network-bound applications. We consider these problems to be future work.

Memory-intensive and Network-intensive benchmarks

To investigate the impact to memory-intensive and network-intensive applications, we conduct the same experiment with a memory-intensive and network-intensive benchmark. In the experiment, we replace the *sleepy-delay* with STREAM [68], which write and read 400GB of data to simulate a memory-intensive application. For a network-intensive benchmark, we employ `MPI_AlltoAll`, which communicates 100GB of data per process (Figure 3.8). Figure 3.10 and 3.11 show the runtime of the benchmarks with and without asynchronous checkpointing. As presented in the figures, overhead is negligibly small in both benchmarks. Because throughput to PFS is low, required bandwidth by asynchronous checkpointing is much smaller than memory and network bandwidth. For example, when we asynchronously write checkpoints of 1,000 nodes to PFS whose throughput is 10GB/s. 10MB/s of bandwidth is required per nodes on average. The

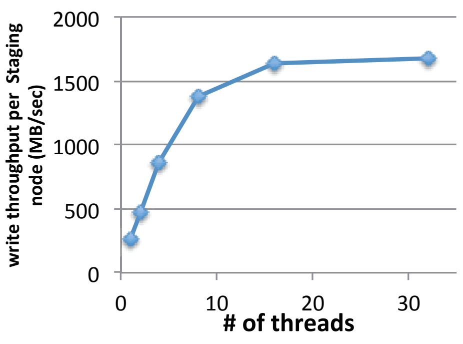


FIGURE 3.12: Impact of varying data writer count

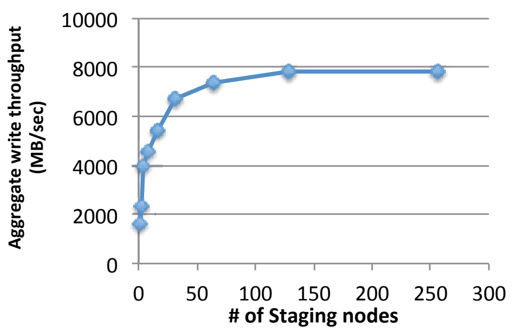


FIGURE 3.13: Aggregate write throughputs

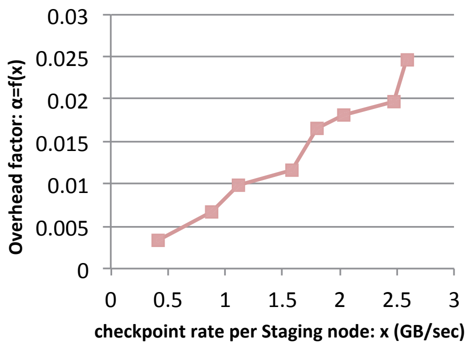


FIGURE 3.14: Empirical overhead factor

required bandwidth is much smaller than bandwidths of Sierra nodes (Theoretical peak memory bandwidth:32GB/s, theoretical peak network bandwidth: 4GB/s).

3.5.2 Tuning of Asynchronous Checkpointing

Checkpoint efficiency highly depends on I/O throughput so we must tune I/O operations such that staging nodes fully exploit the available I/O performance. Generally, we can increase I/O throughput to a PFS by writing with multiple threads, so we multi-thread our staging server process. Our goal is to find the optimal numbers of threads and staging nodes such that we can obtain near peak PFS performance. As a target PFS, we use the Lustre file system [66] on TSUBAME2.0. A TSUBAME2.0 node has two sockets of Intel Xeon X5670, 58GB of DDR3 1333MHz memory, 120GB of local SSD and three Tesla M2050 GPUs. The nodes are connected through Dual-Rail QDR Infiniband(x4).

Figure 3.12 shows the write throughput of one staging node with different numbers of *data writer threads*. We achieve the highest performance (1.6 GB/s) on a single staging node with 16 threads. We then explore how many staging nodes can exploit the Lustre file system. Figure 3.13 presents aggregate write throughput with different staging node counts, each using 16 data writers. Aggregate write throughput rapidly increases from 1 to 32 staging nodes but then quickly saturates around 8 GB/s beyond 32 staging nodes. Thus, we use 32 staging nodes and set the staging server to run with 16 data writer threads. Under this set up, checkpoint data can be transferred to the PFS at a rate of 6.4 GB/s via 32 staging nodes, which is only 2.3% of the 1408 TSUBAME2.0 thin nodes.

Whenever the staging client and server processes read checkpoint data from compute nodes in the background, significant overhead is added to the application runtime due to resource contention. The amount of overhead depends on the read rate. To estimate this overhead, we transferred checkpoints while running the Himeno benchmark [69] as a target application. This benchmark solves Poisson's equation using the Jacobi iteration method. The Himeno benchmark is a stencil application in which each grid point is repeatedly updated using only neighbor points in a domain. This computational pattern frequently appears in numerical simulation codes for solving partial differential equations. Many fluid dynamics phenomena can be described by partial differential equations over multi-dimensional Cartesian grids, including weather, seismic waves, heat flow, and electric charge and magnetic field distribution in a domain.

Figure 3.14 shows the overhead factor imposed on the Himeno benchmark while varying the checkpoint read rate of a staging node. We find that the overhead factor roughly increases linearly with the read rate. Based on the result, we model the overhead factor (α) of the Himeno benchmark as $\alpha = cx$ where c is 0.008768, and x represents the checkpoint read rate of a staging node in GB/s. The parameter c is derived from the slope of fitting a line to the data in Figure 3.14. With 32 staging nodes, we calculate the read rate per staging node to be 209.5 MB/s, which is derived by dividing the aggregate write throughput when using 32 nodes in Figure 3.13 by 32, the number of staging nodes. Thus, the overhead factor model gives us the overhead factor of 0.00184 ($= 0.008768 \otimes 0.20905$). We add this overhead to the time of computation in our model when we compute efficiency of asynchronous checkpointing.

SCR can adjust the degree of resiliency by changing the number of processes in each XOR group used to compute redundancy data. Figure 3.19 shows encoding throughput for different XOR group sizes. Resiliency improves with smaller XOR groups, but XOR encoding throughput decreases. For an XOR checkpoint, SCR computes the parity of each block as in RAID-5 [16, 17], which creates $S = B + \frac{B}{N\ 1}$ bytes of encoded checkpoint data from B bytes of original checkpoint data within N members of an XOR group. Since the encoding time increases linearly with the encoded checkpoint size, the large XOR group size, NS, saturates the XOR encoding rate. As Section 2.2 showed,

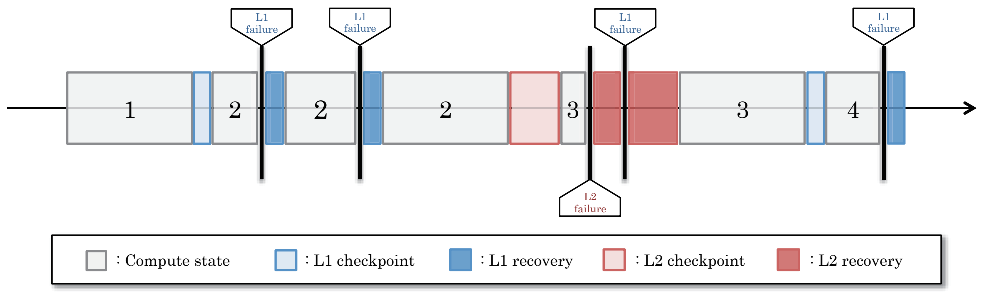


FIGURE 3.15: The multi-level checkpoint/restart simulator transition to compute/checkpoint/recovery states according to failure intervals generated by random numbers for a exponential distribution

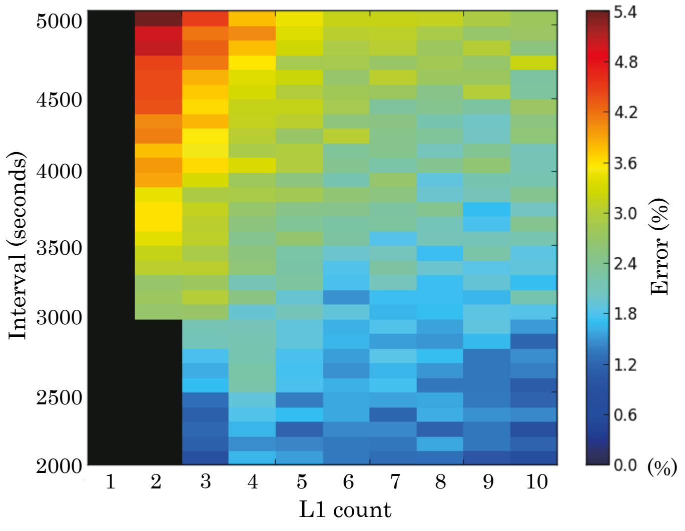


FIGURE 3.16: Model errors in different interval and L1 checkpointing count

most failures affect just one node. Thus, we use XOR checkpoints only to handle failure category 1 in Table 2.3, and we handle the other failure categories $k = 2, 3, 4 \ldots 5$ by a PFS checkpoint. Thus, we set the XOR encoding rate as the saturated maximal rate, 400MB/s.

3.5.3 Model Accuracy

First we verify our model accuracy. We develop a multi-level checkpoint/restart simulator, and compares efficiency computed by our model with efficiency by the simulator. The simulator keeps track of application states, i.e, compute/checkpoint/restart states, according to failure intervals generated by random numbers for a exponential distribution. The transition of the application follows a Poisson process (Figure 3.15). The checkpoint overhead, checkpoint latency and recovery time is based on parameters in

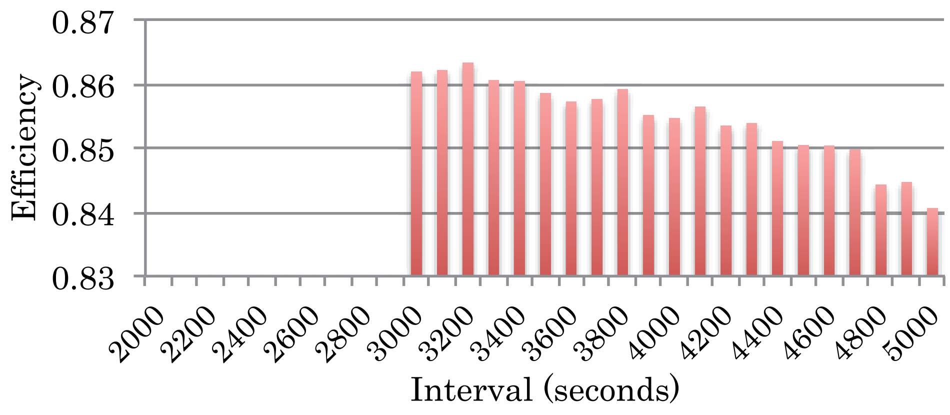


FIGURE 3.17: Efficiency computed by the multi-level checkpoint/restart simulator: L1 count = 2

Section 3.5.2. For failure rates, we use the ones of TSUBAME2.0 in Table 2.3. In this evaluation, we assume an application writes two level checkpoints to recovery from *Failure level 1* by level-1 checkpoints, and *Failure level 2 to 5* by level-2 checkpoints. We run the simulator until the application transition to 300,000 states, which give us a stable number of efficiency.

Figure 3.16 shows error ratios of efficiency of our model to our multi-level checkpoint/restart simulator. Because the checkpoint intervals (level-2 and 1) are determined by an interval of level-1 checkpoint, and the number of level-1 checkpoints per level-2 checkpoint. Thus, the figure becomes a two-dimensional figure. Black-colored panels means the model and the simulator can not compute the efficiency because level-2 checkpoint intervals are too short to complete. As shown in the figure, our model is fairly accurate, and the error rations becomes smaller as the checkpoint interval and L1 count decrease. As system size grows and the failure rate increase at extreme scale, the checkpoint interval will be expected to shrink. Thus our model is expected to give us more accurate estimation, and becomes more necessary model to estimate efficiency at extreme scale.

Based on our model, we can find its optimal checkpoint interval and level-1 checkpoint count as 4100 seconds of the interval and 2 level-1 checkpoints per level-2 checkpoint for TSUBAME2.0. Meanwhile the simulator give us 3000 seconds and 4 of level-1 checkpoint counts for the optimal numbers. To validate our model, we also explore the impact of the different checkpoint interval and level-1 checkpoint count. Figure 3.17 and 3.18 show efficiency computed by the simulator with two and four of level-1 checkpoint count in increasing level-1 checkpoint intervals. As shown in the figures, the best efficiency is 0.864 by 3000 seconds interval and four level-1 checkpoint counts. Meanwhile, with our model, applications can still achieve high efficiency as 0.856, which means our model can give us the nearly best checkpoint interval and level-1 checkpoint count.

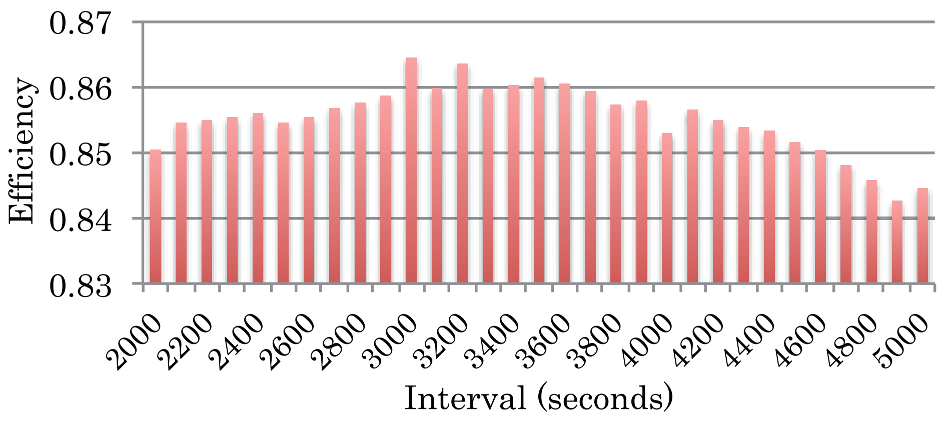


FIGURE 3.18: Efficiency computed by the multi-level checkpoint/restart simulator: L1 count = 4

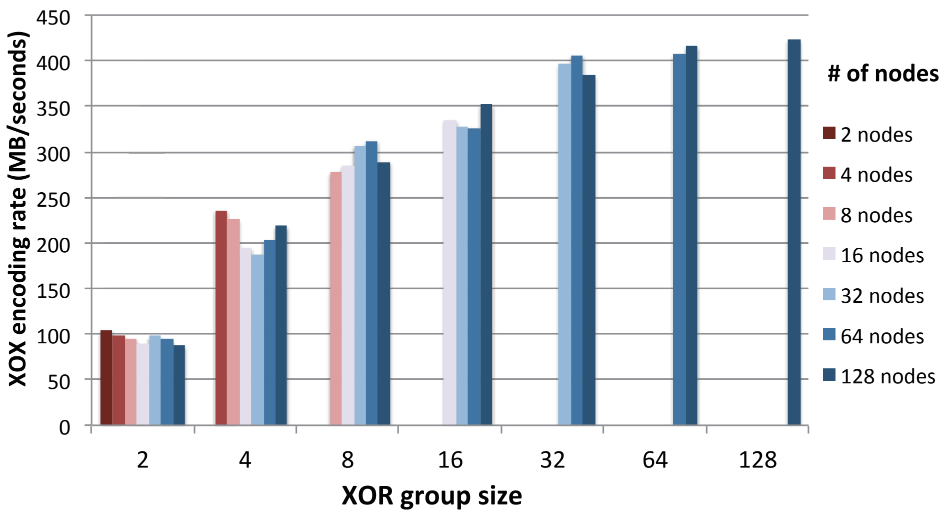


FIGURE 3.19: XOR encoding performance per node

3.5.4 Efficiency Comparison

As future systems will be larger and will have larger memory sizes, failure rates and checkpoint size are expected to increase. To explore the effects, we increase failure rates and checkpoint/restart times by factors of 1, 2, and 10, and compare efficiency between synchronous and asynchronous checkpointing. We use 29 GB for the checkpoint size per compute node, which is half of the memory of a TSUBAME2.0 thin node. As Figure 3.19 shows, an XOR encoding rate is constant regardless of the number of compute nodes, which means XOR encoding scales with system size. Thus, when we increase checkpoint/restart times, we increase only PFS checkpoint/restart time (PFS cost).

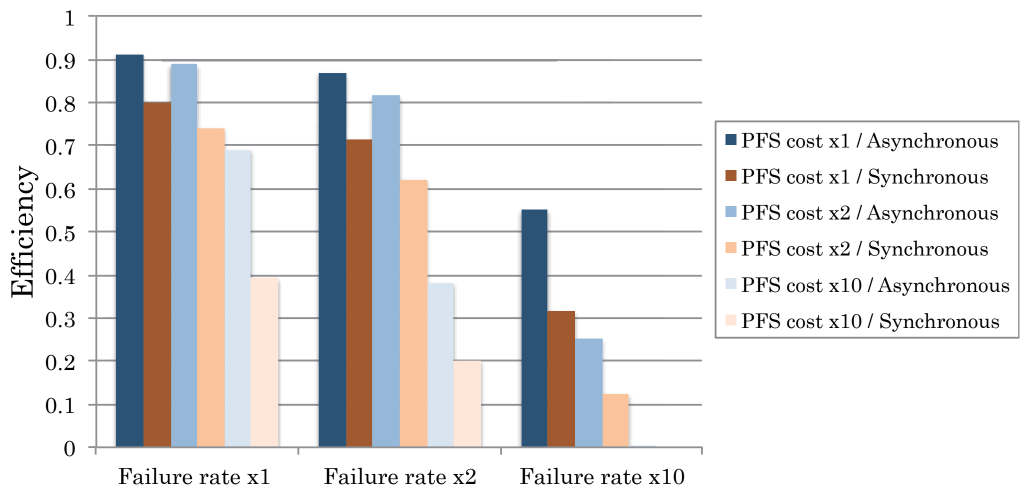


FIGURE 3.20: Efficiency of synchronous and asynchronous checkpointing

Figure 3.20 shows the *efficiency* of both checkpointing methods under different failure rates and PFS costs. We define the *efficiency* as $\frac{ideal\_time}{expected\_time}$. The *ideal_time* is the run time if the application encounters no failures and takes no checkpoints, while *expected_time* is the expected run time computed from our model for asynchronous checkpointing and the original model [4] for synchronous checkpointing. When we compute efficiency, we optimize Level 1 counts between Level 2 checkpoints, and the interval between checkpoints, given failure rates and PFS costs, which yields the *maximal* efficiency. The asynchronous method always achieves higher efficiency than the synchronous method. The efficiency gap becomes more apparent with higher failure rates and higher PFS costs. This is because the long time to take a PFS checkpoint during synchronous checkpointing increases the likelihood of a lower level failure occurring during the PFS checkpoint, so the application must rollback to the beginning. However, with asynchronous checkpointing, the application can rollback to the most recent XOR checkpoint. Further, since overhead of a synchronous checkpoint is identical to checkpoint latency, which is directly added to application run time, the efficiency decreases more quickly than with asynchronous checkpointing.

Since staging nodes write checkpoints to the PFS independently of compute node activities, they can throttle their write rate without reducing application performance. We found that we could read checkpoints from compute nodes at half of the maximum bandwidth (i.e., PFS cost $\otimes 2$), but still maintain 90% efficiency with current failure rates.

Because an asynchronous checkpoint overlaps with application computation, asynchronous checkpointing can impact the application run time depending on the overhead factor,

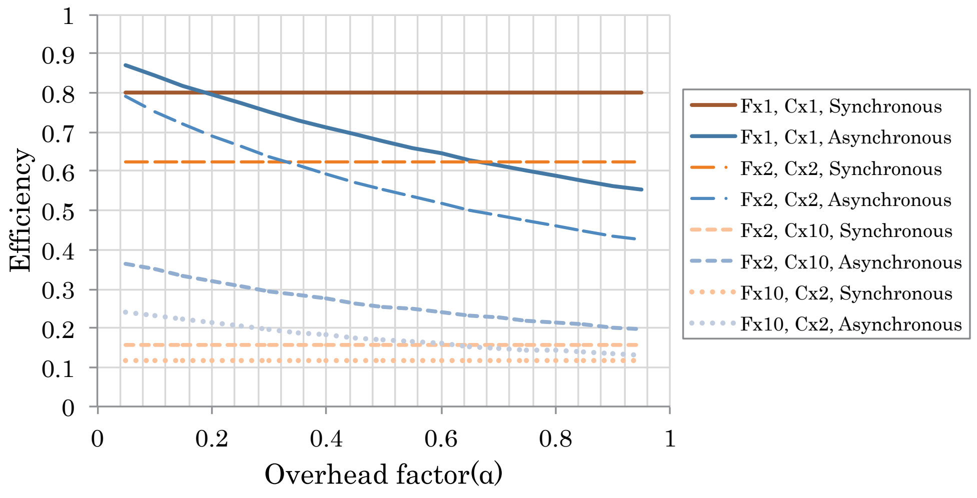


FIGURE 3.21: Efficiency under varying the overhead factor: α

α, in different applications. If the overhead factor increases enough, our asynchronous checkpointing could be less efficient than synchronous checkpointing. Figure 3.21 shows efficiency with increasing overhead factor and different failure rates and PFS costs. F and C denote the base failure rate and PFS cost. Synchronous checkpointing can become more efficient than asynchronous with a larger overhead factor at current failure rates and PFS costs. However, in future systems where the failure rates and PFS costs increase, asynchronous checkpointing can be effective even with a large overhead factor. With large failure rates and PFS costs, the checkpointing interval decreases so that checkpointing overhead dominates the overall run time. Since an application is blocked with synchronous checkpointing, the checkpoint latency impacts application run time more than with asynchronous checkpointing in future systems. As for different systems, we observed that three clusters at LLNL show similar failure rates [4], where system failures can be recovered from level 1 checkpoints and frequent level 2 checkpoints are not required. Thus, asynchronous checkpointing would benefit other systems.

3.5.5 Building an Efficient and Resilient System

When building a reliable data center or supercomputer, two major concerns are financial cost of the PFS and the PFS throughput required to maintain high efficiency. Generally, we want to minimize financial cost, but not sacrifice performance. Using our model, we can predict the required PFS bandwidth to achieve high system efficiency when using our checkpointing system.

Figure 3.22 shows the PFS bandwidth required to maintain 90%, 80%, and 70% efficiency under increasing failure rates, scaled from $1\otimes$ up to $16\otimes$ today's rates. Overall,

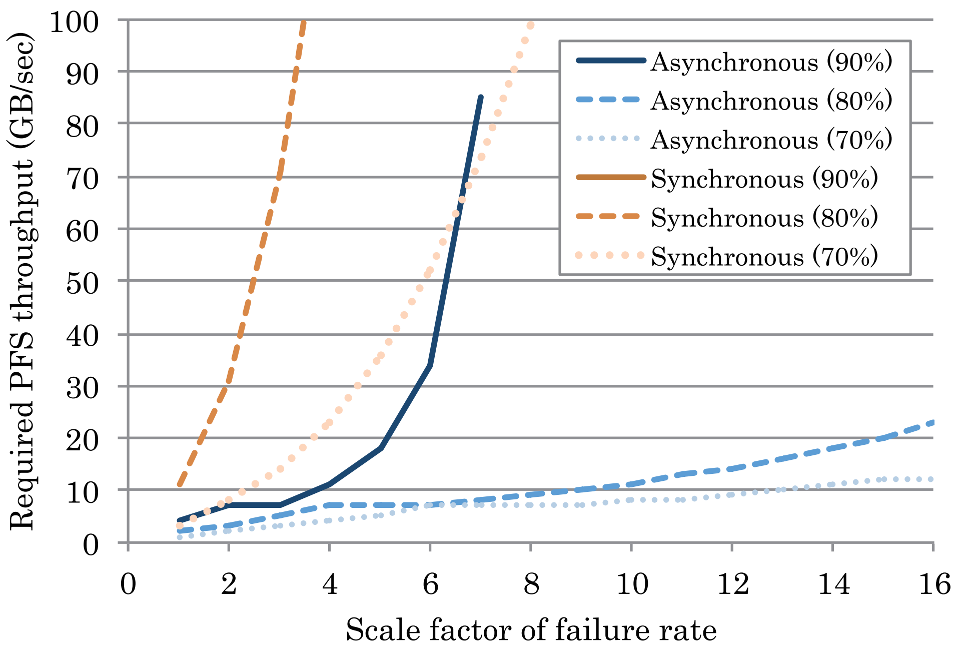


Figure 3.22: Required PFS throughput at different failure rates

our checkpointing system outperforms synchronous checkpointing. Failure rates can increase by 16⊗ and still require modest PFS throughputs for 80% application efficiency. However, at 90% efficiency, the bandwidth requirement rises sharply with failure rates larger than 5⊗. Here, as failure rates increase, the optimal checkpoint interval decreases, increasing the overhead contribution of the L1 checkpoints. As the overhead of L1 checkpoints approaches 10%, the allowed overhead for writing PFS checkpoints approaches 0%. Therefore, achieving 90% efficiency at higher failure rates requires throughput values approaching infinity. The curve for synchronous checkpointing at 90% efficiency follows the same trend, but is much more severe, so much so that it does not appear in the figure. These trends imply that reducing the overhead of L1 checkpoints will be necessary on machines with higher failure rates.

With synchronous checkpointing, systems require higher PFS throughput to minimize the risk of failure during a PFS checkpoint. Further, synchronous checkpointing uses the PFS only when taking a PFS checkpointing, which means the PFS underutilized during most of the application run. With our checkpointing system, we not only hide PFS checkpoint overhead, but use PFS throughout application execution.

3.6 Summary

We have designed and modeled an asynchronous checkpointing system that extends an existing multi-level checkpointing system. Our asynchronous checkpointing system enables applications to save checkpoints to fast, scalable storage located on the compute nodes and then continue with their execution while dedicated staging nodes copy the checkpoint to the PFS in the background. This capability simultaneously increases system efficiency and decreases required PFS bandwidth. Since applications spend less time in defensive I/O, we find that our asynchronous checkpointing system can improve machine efficiency by 1.1 to 2.0 times on future systems. Further, our model predicts that asynchronous checkpointing significantly reduces the PFS bandwidth required to maintain application efficiency. For example, to maintain 80% efficiency at 4$\otimes$ today's failure rates, the PFS bandwidth required for asynchronous checkpointing is an order of magnitude less than that required by synchronous checkpointing. Additionally, our asynchronous checkpointing system can maintain 80% efficiency with only modest PFS bandwidth requirements even when failure rates are 16$\otimes$ higher than on current petascale systems.

Chapter 4

A User-level Infiniband-based File System and Checkpoint Strategy for Burst Buffer

The third approach, *a user-level infiniband-based file system and checkpoint strategy for burst buffers*, consider using burst buffers to improve system resiliency. Burst buffers are a new tier in the storage hierarchy to fill the performance gap between node-local storage and the PFS. They absorb the bursty I/O requests from applications and thus reduce the effective load on the PFS With burst buffers, an application can quickly store checkpoints with increased reliability. We explore how burst buffers can improve efficiency compared to using only compute-node storage, and we develop and apply models to investigate the best checkpoint strategy with burst buffers for a wide range of checkpoint/restart strategies.

4.1 Introduction to Checkpointing Strategies for Burst Buffer

As described in Chapter 2, one indispensable technique for fault tolerance is checkpoint/restart, in which the application periodically writes a snapshot of its state to reliable storage, like a parallel file system (PFS). Then when a failure occurs, the application is restored to its previous state recorded in the snapshot. Although storing checkpoints in the PFS is highly reliable, this method imposes high overhead on application run time at large scales. Multilevel checkpoint/restart improves on this approach by storing most checkpoints in fast, scalable storage on the compute nodes and only copying a select few to the more reliable PFS [4, 7]. This reduces the overhead of writing checkpoints in the common case, which greatly increases application efficiency.

Further efficiency gains can be achieved by combining multilevel checkpoint/restart with asynchronous I/O [50, 52, 70] or uncoordinated checkpointing [40, 43, 44].

However, even with these state-of-the-art checkpoint/restart techniques, high failure rates at large scale will significantly limit application efficiency. Our earlier failure analysis study showed that most failures affect a single compute node [4, 70]. To tolerate node failures, multilevel checkpointing libraries apply redundancy schemes to the cached checkpoints across compute-node storage. For example, each checkpoint may be copied to a partner node, or the library may utilize a RAID algorithm and spread redundancy data across multiple compute nodes. This allows the application to recover from node failures assuming the number of nodes lost is less than what is tolerated by the redundancy scheme. However, with higher failure rates, the likelihood of multiple simultaneous node failures increases. If the simultaneous failures affect nodes in a shared redundancy set, the cached checkpoints will be lost and the application will need to restart from the PFS. This could mean the application would spend the majority of its time in checkpoint/restart activities [70]. Thus, writing checkpoints to in-system storage is a scalable, but not reliable, solution, and application efficiency may suffer at extreme scales.

Burst buffers have been proposed as in-system storage to alleviate the problems of writing to a shared PFS [8, 9]. Burst buffers are a new tier in the storage hierarchy to fill the performance gap between node-local storage and the PFS. They absorb the bursty I/O requests from applications and thus reduce the effective load on the PFS. System software can manage moving data between the burst buffers and the PFS asynchronously. The storage design is expected to be a promising architecture of future supercomputers.

In this paper, we consider using burst buffers to improve system resiliency. With burst buffers, an application can quickly store checkpoints with increased reliability. We explore how burst buffers can improve efficiency compared to using only compute-node storage, and we develop and apply models to investigate the best checkpoint strategy with burst buffers for a wide range of checkpoint/restart strategies. Our key contributions include:

≤ An Infiniband-based file system (IBIO) that exploits the bandwidth of burst buffers;

≤ A model to evaluate system resiliency given checkpoint/restart and storage configurations;

≤ Simulation results showing how system resiliency improves from the use of burst buffers;

≤ A quantitative examination of the trade-offs between coordinated and uncoordinated multilevel checkpoint/restart;

≤ An analysis of which tiers in the storage hierarchy impact system reliability and efficiency; and

≤ An exploration of burst buffer configurations to discover the best for future large-scale systems.

In the next section, we categorize checkpoint/restart strategies and describe the targets for our study. In Section 4.3, we introduce our SSD burst buffer machine. In Section 4.4, we detail IBIO. We model multi-tiered storage and checkpoint/restart strategies in Section 4.5. In Section 4.6, we show IBIO performance, and in Section 4.7 we simulate system efficiency given checkpoint/restart strategies and storage configurations. We summarize this chapter in Section 4.8.

4.2 checkpoint/restart Strategies

Over the years, many checkpoint/restart strategies have been studied. These techniques can be roughly categorized into single or multilevel, synchronous or asynchronous, and coordinated or uncoordinated checkpoint/restart. We explain each checkpoint/restart strategy and their advantages and disadvantages. We describe our target checkpointing strategies.

4.2.1 Single vs. Multilevel Checkpointing

The simplest approach for checkpoint/restart is to write all checkpoints to a single location, such as the PFS [45, 46]. However, when a large number of compute nodes write their checkpoints to a PFS, contention for shared PFS resources leads to low I/O throughput. Multilevel checkpointing is an approach for alleviating this bottleneck [4]. Earlier failure analysis showed most failures on current supercomputers affect a single compute node, which does not necessarily require writing checkpoints to the reliable, but slow PFS [4]. For example, only 15% of failures on the Hera, Atlas and Coastal systems at LLNL required checkpoints on the PFS for restarts. The study also showed multilevel checkpoint/restart can benefit application efficiency in the face of higher failure rates and increased relative overhead of checkpointing to the PFS that may occur on future systems. Therefore, in this study we only target multilevel checkpoint/restart.

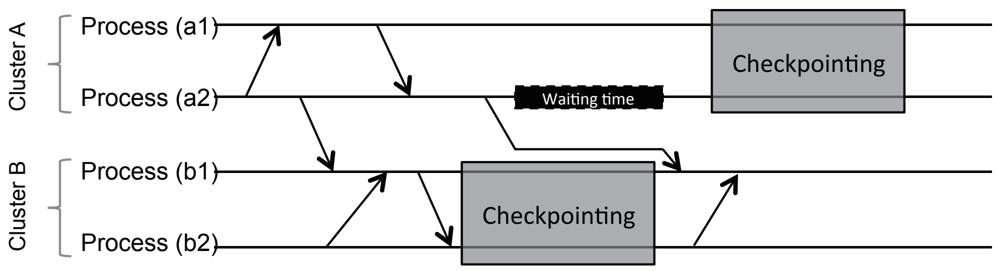


FIGURE 4.1: Indirect global synchronization on uncoordinated checkpoint/restart

4.2.2 Synchronous vs. Asynchronous Checkpointing

Checkpointing libraries can write checkpoints either synchronously or asynchronously. Synchronous checkpointing methods write checkpoints such that all processes write their own checkpoints concurrently, and are blocked until the checkpoint operation completes [4, 45]. In asynchronous checkpointing [7, 46, 70], the methods write checkpoints to the PFS in the background of application computation, which can reduce checkpointing overhead experienced by applications. With asynchronous checkpointing, after each process writes its checkpoint data to RAM or node-local storage, it can continue its computation. Another process or thread reads the checkpoint from the storage location, and writes it to the PFS. Although asynchronous checkpointing can increase an application's runtime due to resource contention from the background checkpoint transfer process, it resolves the blocking problem of synchronous checkpointing.

Intuitively, one would expect asynchronous checkpointing to be more efficient than synchronous checkpointing. However, our earlier study showed that simple asynchronous checkpointing, can inflate an application's runtime and can be worse than synchronous checkpointing [70]. But with our asynchronous checkpointing system using RDMA, we minimized the inflated overhead, and showed that the asynchronous checkpointing is more efficient given current and future failure rates, and expected checkpointing overhead. Thus, in this paper, we explore only asynchronous checkpointing.

4.2.3 Coordinated vs. Uncoordinated Checkpointing

Last, we consider whether checkpoint/restart is coordinated or uncoordinated. With coordinated checkpoint/restart, all processes globally synchronize before taking checkpoints to ensure the checkpoints are consistent and that no messages are in flight. Coordinated checkpoint/restart is applicable to a wide range of applications. However, at large scales the global synchronization can cause overhead due to propagation of system noise at extreme scale [35]. In addition, when checkpointing to or restarting from a PFS, tens of thousands of compute nodes concurrently write or read checkpoints, which can cause large overhead due to contention. Meanwhile, uncoordinated checkpointing [40] does not require global synchronization, and allows processes to write/read checkpoints at different times, which lowers checkpoint overhead. However, with uncoordinated checkpointing there may be messages in flight from one process to another when a checkpoint is taken. To handle this, uncoordinated checkpointing libraries log messages, which has its own overhead problem. This protocol can cause the so-called *domino effect* preventing an application from rolling-back to the last checkpoint at restart [39] without message logging.

Earlier uncoordinated checkpoint techniques [43, 44] reduce the message logging overhead by partitioning processes into clusters, and only logging the inter-cluster communications. Although the clustering approach can reduce message logging overhead while minimizing the number of processes that need to restart on failure, application runtime is still inflated by the logging overhead. In addition, if we apply uncoordinated checkpointing to MPI applications, *indirect global synchronization* can occur. In Figure 4.1, for example, process (a2) in cluster (A) wants to send a message to process (b1) in cluster (B), which is writing its checkpoint at that time. Process (a2) waits for process (b1) because process (b1) is doing I/O and can not receive or reply to any messages, which keeps process (a1) waiting to checkpoint with process (a2). If such a dependency propagates across all processes, it results in indirect global synchronization. Many MPI applications exchange messages between processes in a shorter period of time than is required for checkpoints. In uncoordinated checkpointing, we assume applications restart with uncoordinated manner (partial restart), but globally synchronize before checkpointing like coordinated checkpointing. Thus, as in the model in Section 4.5, we model uncoordinated checkpointing time is same as coordinated checkpointing.

4.2.4 Target checkpoint/restart Strategies

As discussed previously, multilevel and asynchronous software-level approaches are more efficient than single and synchronous checkpoint/restart respectively. However, there is a trade-off between coordinated and uncoordinated checkpointing given an application and the configuration. In this work, we compare the efficiency of multilevel asynchronous coordinated and uncoordinated checkpoint/restart. However, because we have already found that these approaches may be limited in increasing application efficiencies at extreme scale [70], we also consider storage architecture approaches.

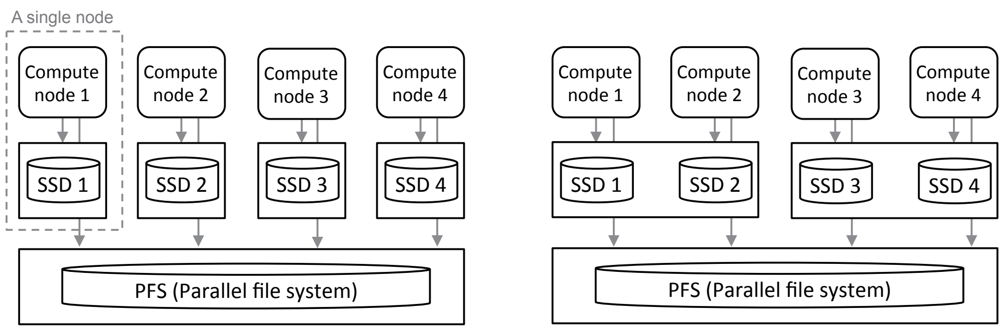


FIGURE 4.2: (a) Left: Flat buffer system (b) Right: Burst buffer system

4.3 Storage designs

Our goal is to achieve a more reliable system with more efficient application executions. Thus, we consider not only a software approach via checkpoint/restart techniques, but also consider different storage architectures. In this section, we introduce an mSATA-based SSD burst buffer system (*Burst buffer system*), and explore the advantages by comparing to a representative current storage system (*Flat buffer system*).

4.3.1 Current Flat Buffer System

In a flat buffer system (Figure 4.2 (a)), each compute node has its dedicated node-local storage, such as an SSD, so this design is scalable with increasing number of compute nodes. Several supercomputers employ this flat buffer system [14, 71, 72]. However this design has drawbacks: unreliable checkpoint storage and inefficient utilization of storage resources for partial restart. Storing checkpoints in node-local storage is not reliable because an application cannot restart its execution if a checkpoint is lost due to a failed compute node. For example, if compute node 1 in Figure 4.2 (a) fails, a checkpoint on SSD 1 will be lost because SSD 1 is connected to the failed compute node 1. Storage devices can be underutilized with partial restart by uncoordinated checkpoint/restart. While the system can limit the number of processes to restart, i.e., perform a partial restart, in a flat buffer system, local storage is not utilized by processes which are not involved in the partial restart. For example, if compute node 1 and 3 are in a same cluster for uncoordinated checkpoint/restart, and restart from a failure, the bandwidth of SSD 2 and 4 will not be utilized. Compute node 1 can write its checkpoints on the SSD of compute node 2 as well as its own SSD in order to utilize both of the SSDs on restart, but as argued earlier distributing checkpoints across multiple compute nodes is not a reliable solution.

4.3.2 Burst Buffer System

To solve the problems in a flat buffer system, we consider a burst buffer system [53]. A burst buffer is a storage space to bridge the gap in latency and bandwidth between node-local storage and the PFS, and is shared by a subset of compute nodes. Additional nodes are required for burst buffers, and the increasing number of nodes may inflate the overall failure rate. However, a burst buffer can offer a system many advantages including higher reliability and efficiency over a flat buffer system. A burst buffer system is more reliable for checkpointing because burst buffers are located on a smaller number of dedicated I/O nodes, so the probability of lost checkpoints is decreased. In addition, even if a large number of compute nodes fail concurrently, an application can still access the checkpoints from the burst buffers. A burst buffer system provides more efficient utilization of storage resources for partial restart of uncoordinated checkpoint/restart because processes involving restart can exploit higher storage bandwidth. For example, if compute node 1 and 3 are in the same cluster, and both restart from a failure, the processes can utilize all SSD bandwidth (Bandwidth of SSD 1 and 2 for node 1, SSD 3 and 4 for node 3) unlike a flat buffer system. This capability accelerates the partial restart of uncoordinated checkpoint/restart.

Table 4.1: Node specification

CPU	Intel Core i7-3770K CPU (3.50 GHz x 4 cores)
Memory	Cetus DDR3-1600 (16 GB)
M/B	GIGABYTE GA-Z77X-UD5H
SSD	Crucial m4 msata 256 GB CT256M4SSD3 (Peak read: 500 MB/s, Peak write: 260 MB/s)
SATA converter	KOUTECH IO-ASS110 mSATA to 2.5' SATA Device Converter with Metal Fram
RAID Card	Adaptec ASR-7805Q Single

To explore the bandwidth we can achieve with only commodity devices, we developed an mSATA-based SSD test system. The detailed specification is shown in Table 5.1. The theoretical peak throughputs of sequential read and write operation of the mSATA-based SSD is 500 MB/sec and 260 MB/sec, respectively. We aggregate the eight SSDs into a RAID card, and connect the RAID cards via PCIe 3.0 (x8). The theoretical peak performance of this configuration is 4 GB/sec for read and 2.08 GB/sec for write in total. To use the storage systems as burst buffers, the mSTA-based SSDs must be accessed via a high-speed network (e.g., Infiniband) with a network-based file system. However, simple methods cannot exploit the bandwidth. For example, if we use NFS with IPoIB for the network-based file system, the useful bandwidth is only 1 GB/s for

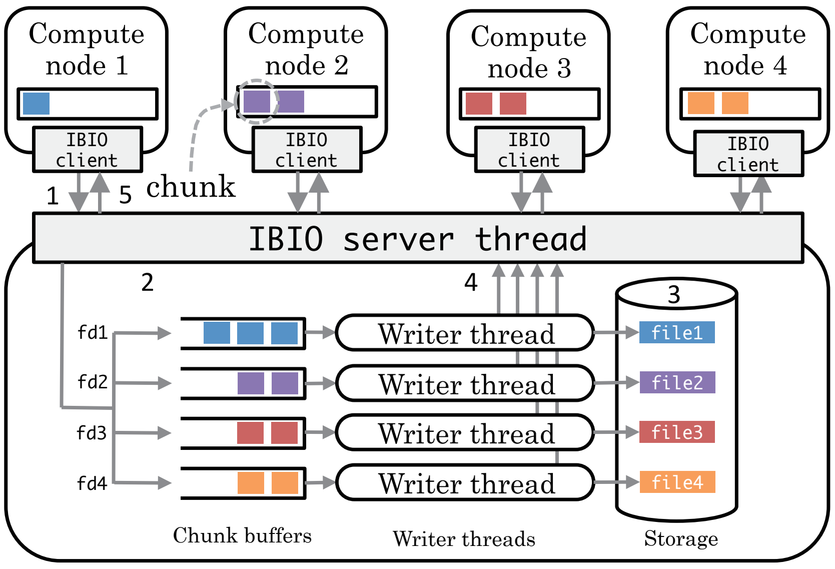


FIGURE 4.3: IBIO Write: four IBIO clients and one IBIO server

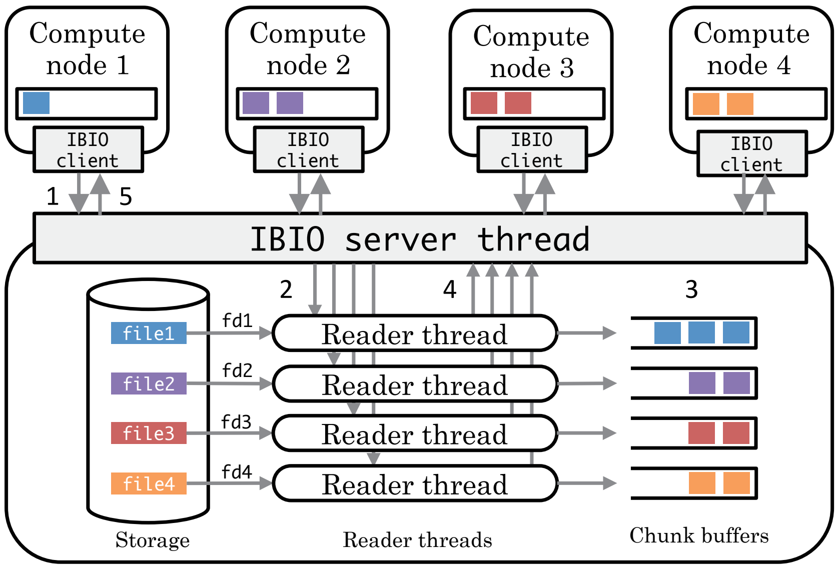


FIGURE 4.4: IBIO Read: four IBIO clients and one IBIO server

both read and write (details in Section 4.6). A new network-based file system is required to exploit the PCIe-attached high bandwidth storage.

4.4 User-level Infiniband-based File System for Burst Buffers

To exploit the bandwidth of burst buffers, we developed a user-level Infiniband-based file system and I/O API called IBIO. Our earlier work showed that I/O operations with concurrent multiple threads can effectively exploit the high bandwidth of PCIe-attached storage [73] as well as the PFS [70]. Thus, we parallelize the operations of IBIO with multiple reader and writer threads. The current API of IBIO includes `open`, `write`, `read` and `close`. The interfaces are identical to POSIX except that of `open`. The IBIO `open` requires a *hostname* as well as a *pathname* so that IBIO clients can access any files on any IBIO server. IBIO `open` sends a query to the IBIO server to open the file, and it returns the file descriptor (`fd`).

Figure 4.3 and Figure 4.4 show the design of IBIO `write` and `read`. When a client on a compute node writes its file to remote storage via the IBIO `write` call (Figure 4.3), the IBIO client divides the data into smaller chunks, and it transfers the chunks using an RDMA API we developed in prior work [70] (Step 1). The RDMA transfer API enables one-sided communications from the client to the server using a low-level, user-space Infiniband API called *ibverbs*. Once the IBIO server thread receives a write request from a client, the IBIO server reads the chunk into the selected buffer according to the `fd` using an RDMA read (Step 2). Then, the IBIO server creates a *writer thread* to asynchronously write the chunk to the file (Step 3) so that the IBIO server thread can receive subsequent chunks from IBIO clients. When all chunks for the write call are written, the writer threads inform the IBIO server (Step 4), and the IBIO server informs the IBIO client that the write is complete (Step 5).

Since communications from server to client are required for IBIO read operations, we extended the RDMA transfer API from prior work to support bi-directional communications. When an IBIO client reads a file from remote storage via IBIO `read` (Figure 4.4), the IBIO client sends the read request containing the `fd` to the IBIO server (Step 1). Once the IBIO server receives the read request, the IBIO server thread identifies the file according to the `fd`, and it creates a *reader thread* to handle the read request (Step 2). Then the IBIO server thread waits for the next request. The reader thread reads a chunk of the file with file descriptor `fd` into its buffer (Step 3). It requests that the IBIO server thread send the chunk to the IBIO client (Step 4), and then it reads the next chunk. When all chunks for the read call have been read, the reader thread informs the IBIO server, and the IBIO server informs the IBIO client that the read is complete (Step 5).

4.5 Modeling

As described in Sections 4.2 and 4.3, each checkpoint strategy and storage architecture has advantages and disadvantages. Here we discuss the model we developed to identify the best checkpoint strategy for a given configuration.

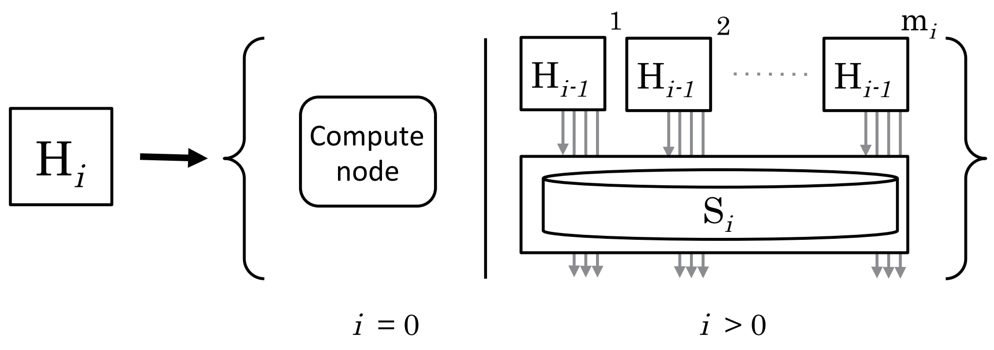


FIGURE 4.5: Recursive structured storage model

4.5.1 Recursive Structured Storage Model

We introduce a recursive structured storage model to generalize storage architectures to describe both flat and burst buffer systems with a single model. Figure 4.5 shows the recursive structured storage model based on a *restricted context-free grammar*. A tier i hierarchical entity, H_i, has storage S_i shared by m_i upper hierarchical entities, $H_i\ \ 1$. We denote $H_{i=0}$ as a compute node. If each tier of hierarchical storage is shared as $\}m_1, m_2, \ldots, m_N\backslash$ within N-tired hierarchical storage, we denote the storage architecture as $H_N\ \}m_1, m_2, \ldots, m_N\backslash$. For example, the flat buffer system in Figure 4.2 (a) can be represented as $H_2\ \}1, 4\backslash$. It has 2 levels of storage: the node-local storage is not shared, so $m_1 = 1$; however, the PFS is shared across all compute nodes, so $m_2 = 4$. In the same manner, the burst buffer system in Figure 4.2 (b) can be represented as $H_2\ \}2, 2\backslash$. The total number of compute nodes can be calculated as $\prod_{i=1}^{2} m_i = 4$ nodes.

In this model, we do not distinguish between node-local storage and network-attached storage. Instead, we differentiate the storage levels using performance parameters. We consider only sequential read/write bandwidth because typically the I/O pattern of checkpoint/restart is sequential. Note that the read and write bandwidth values are not the peak performance of the storage but the throughput between compute nodes and the storage location. For example, if tier i storage has a read bandwidth of r, but the network-based file system delivers a bandwidth of $\hat{r} < r$, then the model parameter is set as $r_i = \hat{r}$. Using these performance parameters, we estimate checkpoint/restart time. However, as we show in Section 4.6, IBIO can minimize the performance gap between local and remote read/write accesses.

4.5.2 Modeling of checkpoint/restart Strategies

Given the storage performance parameters of each tier, we model level i *checkpoint overhead* (O_i), *checkpoint latency* (L_i), and *restart overhead* (R_i) in a multilevel checkpointing library [70]. For simplicity, if multiple compute nodes concurrently access a single storage location, we assume the read/write throughput scales down linearly with the number of concurrent accesses.

Checkpoint overhead O_i and restart overhead R_i are the increased execution time of an application because of checkpointing and restarting, respectively. Checkpoint latency L_i is the time to complete a checkpoint. If a checkpoint strategy conducts erasure encoding, such as XOR [4] and Reed-Solomon encoding [7], the checkpoint overhead and latency also include the encoding overhead and latency. We differentiate between checkpoint overhead and latency to show the differences between synchronous and asynchronous checkpointing. During synchronous checkpointing, checkpoint overhead and latency are equal, i.e., $O_i = L_i$, because each process is blocked until the checkpoint is completed. Asynchronous checkpointing, meanwhile, incurs only initialization overhead, so checkpoint overhead is equal or smaller than checkpoint latency, i.e., $O_i < L_i$.

We model level i checkpoint overhead and latency as

$$O_i = \begin{cases} \lceil C_i + E_i & \text{(synchronous checkpointing)} \\ \lfloor I_i & \text{(asynchronous checkpointing)} \end{cases}$$

$$L_i = C_i + E_i$$

where C_i denotes actual checkpointing time, E_i denotes encoding time, and I_i denotes initialization time for asynchronous checkpointing. If the level i checkpointing does not encode checkpoints, E_i becomes 0; otherwise we model the encoding time as $E_i = D/e_i$ where D is the checkpoint size per compute node and e_i is encoding throughput. The actual checkpointing time C_i, i.e., sequential write time, is calculated as

$$C_i = \begin{cases} D \otimes M/w_i & (i = N) \\ \left\lceil D \otimes \left\lceil \frac{M}{\prod_{k=i+1}^{N} m_k} \right\rceil /w_i \right. & (otherwise) \end{cases}$$

where M denotes the total number of checkpointing compute nodes, i.e., $M = \prod_{i=1}^{N} m_i$. With uncoordinated checkpointing, we assume the checkpointing time is identical to coordinated checkpointing time because of indirect global synchronization as described in Section 4.2.3. Because $\prod_{k=i+1}^{N} m_k$ is the number of storage locations S_i, the quantity $\left\lceil \frac{M}{\prod_{k=i+1}^{N} m_k} \right\rceil$ represents the max number of compute nodes per storage location S_i.

When restarting with uncoordinated checkpointing, the restart overhead is different from coordinated checkpointing. We model the restart overhead R_i, i.e., sequential read time, as:

$$R_i = \begin{cases} D \otimes K/r_i & (i = N) \\ \lceil \\ D \otimes \dfrac{K}{\prod_{k=i+1}^{N} m_k} /r_i & (otherwise) \end{cases}$$

where K is the number of restarting compute nodes. With coordinated restart, all compute nodes concurrently read their checkpoints, so K is identical to the total number of compute nodes M. With uncoordinated restart, only the cluster, which includes failed compute nodes will read its checkpoints and restart. Here, K is the cluster size, and we assume that each compute node in a cluster is distributed across $S_{i>N}$ storage locations with a topology-aware process mapping technique.

4.5.3 Multilevel Asynchronous checkpoint/restart Model

Our multilevel asynchronous checkpoint/restart model [70] computes the expected runtime $\hat{T}$ given the checkpoint overheads at each storage level $O = \}O_1, O_2, \ldots\backslash$, the checkpoint latencies $L = \}L_1, L_2, \ldots\backslash$, the restart overheads $R = \}R_1, R_2, \ldots\backslash$, the failure rates $F = \}F_1, F_2, \ldots\backslash$, the checkpoint frequencies $V = \}v_1, v_2, \ldots\backslash$, and the checkpoint interval T. Here, v_i is the number of level i checkpoints within each level $i+1$ period. For example, if an application writes fifteen level 1 checkpoints for every level 2 checkpoint, and five level 2 checkpoints for every level 3 checkpoint, V is $\}15, 5, 1\backslash$. Given these parameters, we can compute the optimal checkpoint frequency an interval by minimizing $\hat{T}$, where $f(O, L, R, F, V, T) \mathcal{O} \hat{T}$.

To evaluate the checkpoint strategies given a storage configuration, we use *efficiency* defined as

$$\textit{efficiency} = \frac{\textit{ideal time}}{\textit{expected time}} = \frac{I}{\hat{T}}.$$

I is the minimum run time assuming the application spends no time in checkpointing activities and encounters no failures. So I is simply computed as:

$$\begin{aligned} I &= T \otimes (v_1 + 1) \otimes \times\!\!\times\!\!\times\!\otimes (v_{N\ 1} + 1) \\ &= T \times \prod_{i=1}^{N\ 1} (v_i + 1). \end{aligned}$$

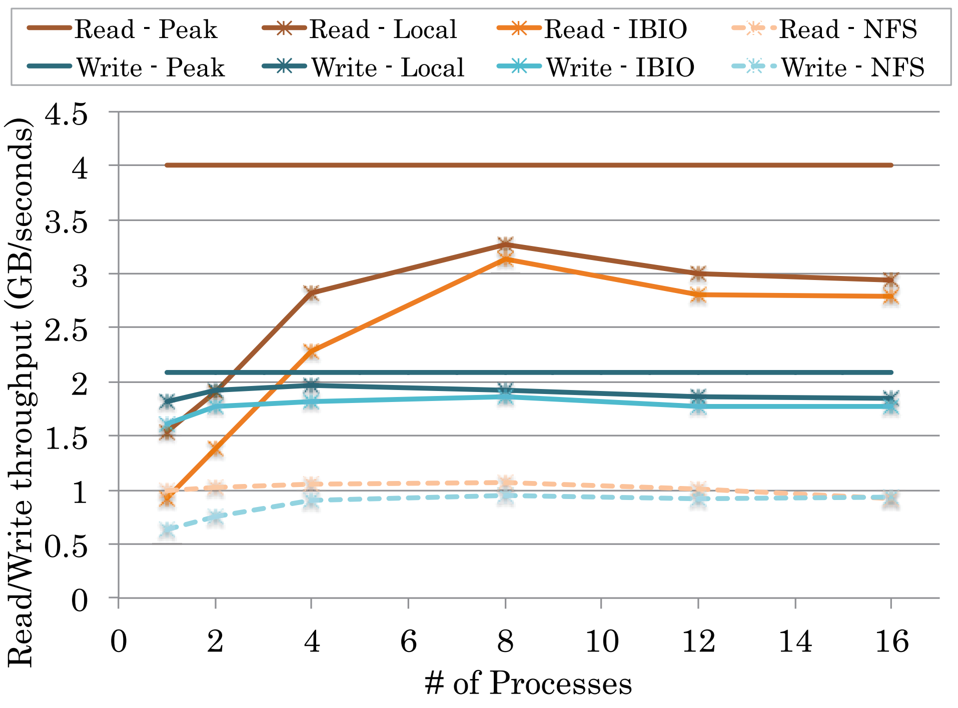


FIGURE 4.6: Sequential read and write throughput of local I/O, and I/O with IBIO and NFS via FDR Infiniband networks

The efficiency metric indicates the fraction of time an application spends only in computation. We use this metric to compare the checkpoint strategies. Our earlier study provides more details of the model [70].

4.6 Performance of IBIO

For a burst buffer system, it is important to exploit the high bandwidth of the burst buffers through network access. To evaluate this property of IBIO we conducted sequential read and write tests using the mSATA-based SSD system described in Section 4.3.2. Figure 4.6 shows the local sequential read and write performance of IBIO, NFS with increasing numbers of client processes from a single compute node. We connected the mSATA-based SSD system to an Infiniband network with a Mellanox FDR HCA (Model No.: MCX354A-FCBT) for remote access evaluations of IBIO and NFS. We use 64 MB for the maximal chunk size to read and write files to and from the NFS and IBIO servers to maximize the sequential read and write throughputs. In NFS, the parameters are configured by `rsize` and `wsize` options.

We find that the read and write throughputs of IBIO are almost identical to the local throughputs with 8 processes and more. When the number of read processes is 4 or less,

Table 4.2: Simulation configuration

	Flat buffer system	Burst buffer system
$H_2\}m_1, m_2\backslash$	$H_2\}1, 1088\backslash$	$H_2\}32, 34\backslash$
$\}r_1, r_2\backslash$	}500 MB/s, 10 GB/s\	}16 GB/s, 10 GB/s\
$\}w_1, w_2\backslash$	}260 MB/s, 10 GB/s\	}8.32 GB/s, 10 GB/s\
$\}e_1, e_2\backslash$	}400 MB/s, N/A\	
D	5 GB	
$\}F_1, F_2\backslash$	$\}2.14\otimes 10^{\ 6},\ 4.28\otimes 10^{\ 7}\backslash$	$\}2.63\otimes 10^{\ 6},\ 1.33\otimes 10^{\ 8}\backslash$

we see performance degradation of read operations on IBIO (Read-IBIO) compared to the local read access (Read-Local). In IBIO read, the client first sends a message to request the first 64 MB of chunk of the file, and the constant request latency is added to the read time. Because of the constant latency, the read throughput decreases with a smaller number of clients. However, when applications write and read checkpoints, multiple processes concurrently access the storage, so applications can exploit the bandwidth. In contrast, when IBIO clients write data to a file, the clients send the first chunk with the first request message. Thus, the write throughput does not decrease with the fewer processes.

We also evaluate the sequential read and write performance of NFS. Because NFS runs on IPoIB, NFS incurs as much as 49.7% and 35.3% performance degradation compared to local read and write, respectively, while IBIO incurs only 4.3 % over local read and 2.7 % over local write. NFS can not exploit the FDR network bandwidth, because IPoIB becomes a bottleneck. In contrast, IBIO uses the low-level infiniband API (*ibverbs*). Thus IBIO can minimize data transfer overhead even over networks.

4.7 Resiliency Exploration

In this section, we evaluate the trade-offs of different checkpointing and storage configurations.

4.7.1 Experimental Setup

We describe our experimental setup including configuration details for checkpoint/restart and storage configurations, and how we determine the failure rates to use in our model.

Table 4.3: Checkpoint Levels and Failure Rates

		Flat Buffer System	Burst Buffer System
Level 1	C/R method	`XOR` on local store	`XOR` on burst buffer
	Failure rate	$2.14 \otimes 10^{\ 7} + 1.92 \otimes 10^{\ 6}$ $= 2.14 \otimes 10^{\ 6}$	$4.28 \otimes 10^{\ 7}$
Level 2	C/R method	PFS	PFS
	Failure rate	$2.14 \times 10^{-7} + 1.92 \times 10^{-6} + 4.28 \times 10^{-7} + 6.67 \times 10^{-8}$ $= 2.63 \otimes 10^{\ 5}$	$3.93 \otimes 10^{\ 10} \otimes 34$ $= 1.33 \otimes 10^{\ 8}$

checkpoint/restart and Storage Configuration

In this study, we evaluate multilevel checkpoint/restart on a 2-tiered storage system. Table 4.2 shows the base configuration. The system sizes (H_i) are based on the Coastal cluster at LLNL, which is an 88.5 TFLOP theoretical peak system consisting of 1,088 batch nodes. We use 500 MB/s for local read throughput, 260 MB/s for local write throughput for the Flat buffer system. The burst buffer system has 34 burst buffer nodes, each of which is shared by 32 compute nodes. In this exploration, we simulate that we aggregate the 32 storage into the single burst buffer node, and connect the burst buffer nodes via a high speed interconnect which does not create a bottleneck in bandwidth to simulate future extreme scale system. With IBIO, applications can remotely access to the burst buffers with the almost same throughput as local access throughput. Thus, we use 16 GB/s for read throughput, and 8.32 GB/s for write throughput in the burst buffer system.

For uncoordinated checkpointing, we use 16 nodes for the cluster size (K). Earlier studies showed that the optimal cluster size is from 32 to 128 processes, i.e., 4 to 16 nodes for a 8-core Coastal compute node, to provide a good trade-off between the size of the clusters and the amount of messages to log for most applications [44, 74]. Because the cluster size is small enough to assign a compute node to a single burst buffer node, $\left\lceil \frac{K}{\prod_{k=2}^{2} m_k} \right\rceil$ is computed as 1 compute node for uncoordinated restart.

We use asynchronous checkpointing for `PFS`, and synchronous checkpointing for `XOR`. For the encoding rate, we only provide an encoding rate (e_1) for level 1 (`XOR`) because `PFS` does not need encoding.

Failure Rate Estimation

Failure rates (F) are based on a failure analysis study using a multilevel checkpoint/restart library called the Scalable checkpoint/restart (SCR) Library [4]. SCR provides several checkpoint options: `LOCAL`, `XOR`, and `PFS`. With `LOCAL`, SCR simply writes the checkpoint data to node-local storage. In this case, if one of the checkpoints is lost due to a failure, an application would not be able to restart its execution. So, SCR provides `XOR`, which is a RAID-5 strategy that computes XOR parity across subgroups of processes so that SCR can restore the lost checkpoint data. SCR also provides `PFS` to keep checkpoint data on the most reliable storage level, the PFS. The failure analysis study shows that the average failure rate (failure/seconds) per a single compute node requiring `LOCAL` is $1.96 \otimes 10^{10}$, `XOR` is $1.77 \otimes 10^{9}$, and `PFS` is $3.93 \otimes 10^{10}$.

In a flat buffer system, each failure rate is calculated by simply multiplying the failure rate by the number of compute nodes, 1088 nodes. This leads to failure rates of $2.14 \otimes 10^{7}$ $(= 1.96 \otimes 10^{10} \otimes 1,088)$ for `LOCAL`, $1.92 \otimes 10^{6}$ $(= 1.77 \otimes 10^{9} \otimes 1,088)$ for `XOR`, and $4.28 \otimes 10^{7}$ $(= 3.93 \otimes 10^{10} \otimes 1,088)$ for `PFS`. The failure rates are identical to the measured ones of the LLNL Coastal cluster. Actually, if the level-i failure rate is lower than the level-$i+1$ rate, the optimal level i checkpoint count is zero because a level i failure can be recovered with a level $i+1$ checkpoint, which is written more frequently than level i [14] . Thus, we do not consider `LOCAL` checkpointing for the simulation. We evaluate the two level checkpoint/restart case where level 1 is `XOR`, and level 2 is `PFS`, with failure rates of $\}F_1,\ F_2\backslash = \}2.14 \otimes 10^{6}, 4.28 \otimes 10^{7}\backslash$ (See Table 4.3).

In a burst buffer system, we use 34 burst buffer nodes, and assume the failure rate of a burst buffer node is identical to a compute node. On a compute node failure, an application does not lose checkpoint data because the checkpoint data is not in compute nodes. However, if a burst buffer node fails, checkpoint data on the failed burst buffer nodes is lost. Thus, we also use two level checkpoint/restart where level 1 is `XOR`, and level 2 is `PFS`. Because the total number of nodes increases, failure rate requiring level 1 checkpoint increases according to the number of burst buffer nodes. For 34 burst buffer nodes, the level 1 failure rate is calculated as $6.67 \otimes 10^{8}$ $(= (1.96 \otimes 10^{10} + 1.77 \otimes 10^{9}) \otimes 34)$. Meanwhile, checkpoint data is stored on fewer nodes, which decreases the failure rate requiring `PFS` for recovery. The level 2 failure is $1.33 \otimes 10^{8}$ (See Table 4.3). On compute node failures, application can restart from level 1 checkpoint regardless of the number of failed compute nodes in a burst buffer system. Thus, the failure rate of each level is $\}F_1,\ F_2\backslash = \}2.63 \otimes 10^{6}, 1.33 \otimes 10^{8}\backslash$ for the burst buffer system. Because the burst buffer system uses more nodes for burst buffers, the overall failure rate of the burst buffer system is higher than one of the flat buffer system.

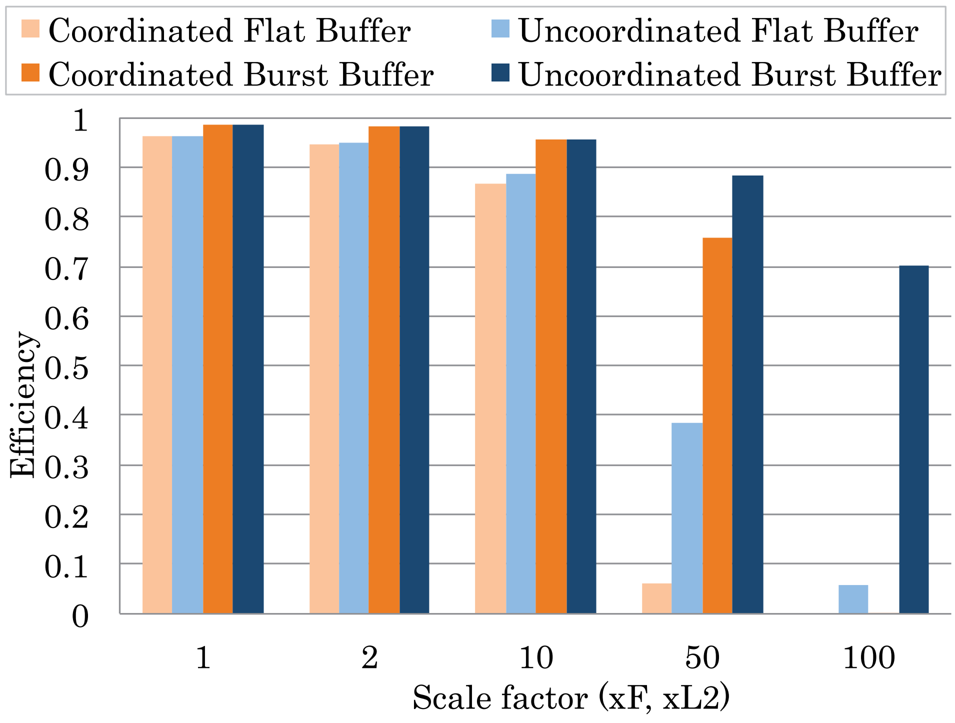


FIGURE 4.7: Efficiency of multilevel coordinated and uncoordinated checkpoint/restart on a flat buffer system and a burst buffer system

4.7.2 Efficiency with Increasing Failure Rates and Level 2 checkpoint/restart Costs

We expect the failure rates and aggregate checkpoint sizes to increase on future extreme scale systems. To explore the effects, we increase failure rates and level 2 (`PFS`) checkpoint/restart costs by factors of 1, 2, 10, 50 and 100 from the base configuration in Table 4.2, and compare the efficiencies of multilevel coordinated and uncoordinated checkpoint/restart on a flat buffer system and on a burst buffer system. We do not change the level 1 (`XOR`) checkpoint cost; because the total performance of node-local storage and burst buffer will scale with increasing system size.

Figure 4.7 shows application efficiency under increasing failure rates (xF) and level 2 checkpoint/restart costs (xL2). When we compute efficiency, we optimize the level-1 and 2 checkpoint frequencies (v_1 and v_2), and the interval between checkpoints (T) to discover the maximal efficiency. The burst buffer system achieves a higher efficiency than the flat buffer system in most cases. The efficiency gap becomes more apparent with higher failure rates and higher checkpoint costs because the burst buffer system stores checkpoints on fewer burst buffer nodes. By using uncoordinated checkpoint/restart and leveraging burst buffers, we achieve 70% efficiency even on systems that are two orders of magnitude larger. This is because partial restart with uncoordinated checkpointing can exploit the bandwidth of both burst buffers and the PFS, and accelerate restart time.

4.7.3 Allowable Message Logging Overhead

The efficiencies shown in Figure 4.7 do not include message logging overhead. We consider this factor in Table 4.4 which shows how much ratio the message logging overhead of uncoordinated checkpointing is allowed to occupy the given computation time to achieve a higher efficiency than coordinated checkpointing. As in Figure 4.7, we increase both the failure rates and level 2 checkpoint/restart cost by the scale factor shown on each row. We find that the logging overhead must be relatively small, less than a few percent, for scale factors up to 10. However, at scale factors of 50 and 100, very high message logging overheads are tolerated. This shows that uncoordinated checkpointing can be more efficient on future systems even with high logging overheads.

4.7.4 Effect of Improving Storage Performance

When building a reliable data center or supercomputer, significant efforts are made to maximize system performance given a fixed budget. It can be challenging to decide which system resources will most affect overall system performance. To explore how the performance of different tiers of the storage hierarchy impact system efficiency, we increase performance of each tier of storage by factors of 1, 2, 10, and 20. Figures 4.8 and 4.9 show efficiency with increasing performance of level 1 and 2 checkpoint/restart, i.e., decreasing level 1 and 2 checkpoint/restart time, using failures rates at 100 $\otimes$ current rates. We see that improvement of level 1 checkpoint/restart does not impact efficiency for either flat buffer or burst buffer systems. However, as shown in Figure 4.9, increasing the performance of the PFS does impact system efficiency. We can achieve over 80% efficiency with both coordinated and uncoordinated checkpoint/restart on the

Table 4.4: Allowable message logging overhead

Flat buffer		Burst buffer	
scale factor	Allowable message logging overhead	scale factor	Allowable message logging overhead
1	0.0232%	1	0.00435%
2	0.0929%	2	0.0175%
10	2.45%	10	0.468%
50	84.5%	50	42.0%
100	$\Rightarrow$100%	100	99.9%

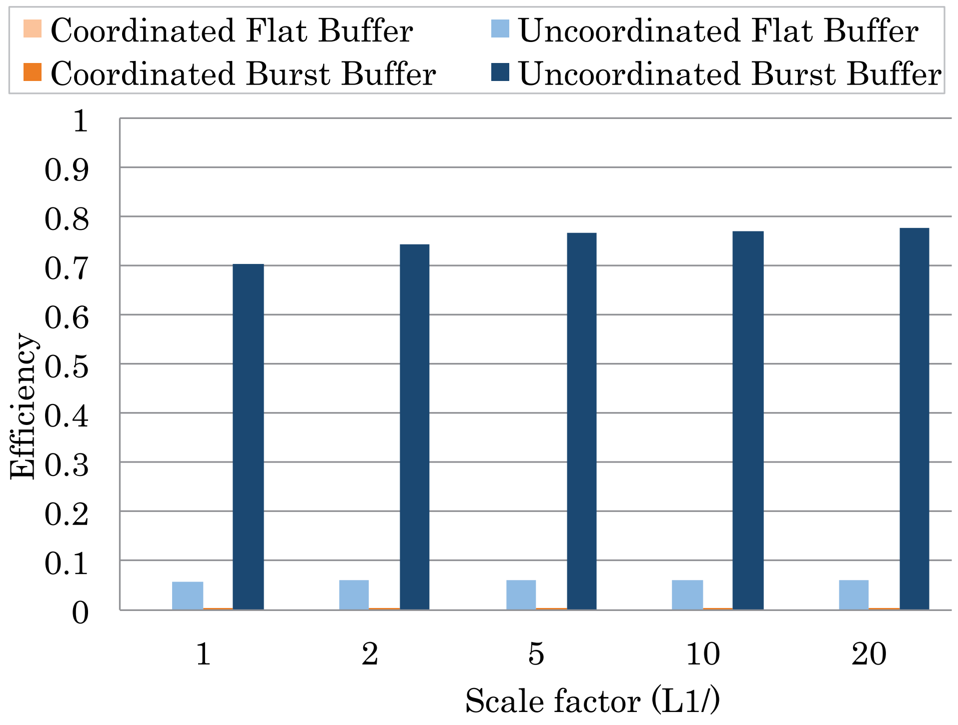


FIGURE 4.8: Efficiency in increasing level-1 checkpoint/restart performance in x100 failure rate: L1 checkpoint/restart time/scale factor

burst buffer system with improved PFS performance of 10 and 20 ⊗. These results tell us that level 2 checkpoint/restart overhead is a major cause of degrading efficiency, and its performance affects the system efficiency much more than that of level 1. We also find that improvement of system reliability for failures requiring level 2 checkpoint is important for future extreme scale systems.

4.7.5 Optimal Ratio of Compute Nodes to Burst Buffer Nodes

Another thing to consider when building a burst buffer system is the ratio of compute nodes to burst buffer nodes. A large number of burst buffer nodes can increase the total bandwidth, but the large node counts increase the overall failure rate of the system. To explore the effect of the ratio of compute node and burst buffer node counts, we evaluate efficiency under different failure rates and level 2 checkpoint/restart costs while keeping I/O throughput of a single burst buffer node constant. Figures 4.10 and 4.11 show the results with coordinated and uncoordinated checkpoint/restart. We see that the ratio is not significant up to scale factors of 10 ⊗. However, at a scale factor of 50 ⊗, a larger number of burst buffer nodes decreases efficiency. Adding additional burst buffer nodes increases the failure rate which degrades system efficiency more than the efficiency gained by the increased bandwidth. Thus, increasing the number of compute

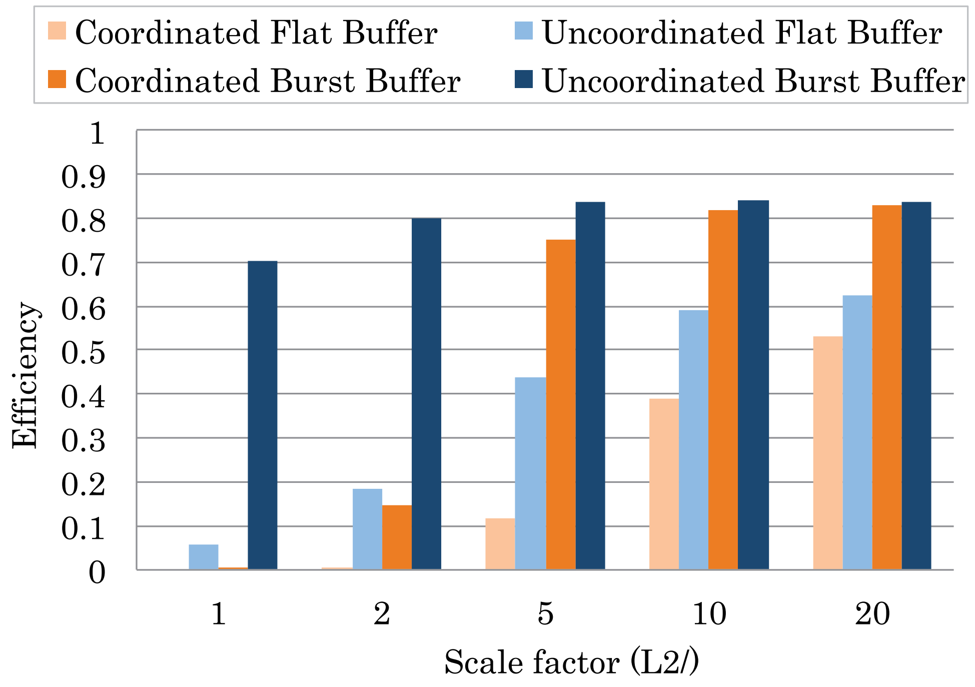


FIGURE 4.9: Efficiency in increasing level-2 checkpoint/restart performance in x100 failure rate: L2 checkpoint/restart time/scale factor

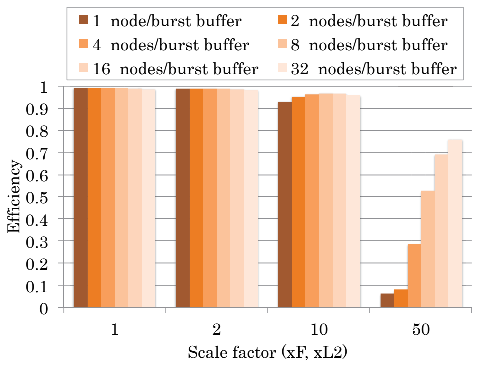


FIGURE 4.10: Coordinated: Efficiency in different ratios of compute nodes to a single burst buffer nodes with coordinated checkpoint/restart

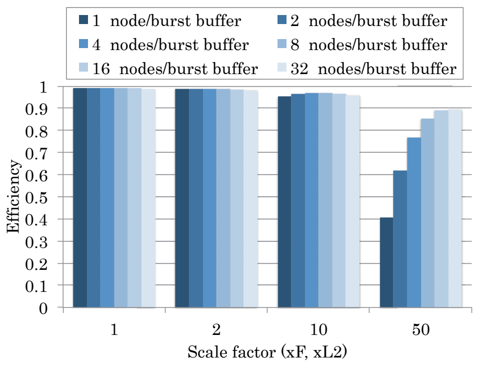


FIGURE 4.11: Uncoordinated: Efficiency in different ratios of compute nodes to a single burst buffer nodes with uncoordinated checkpoint/restart

nodes sharing a burst buffer node is optimal as long as the burst buffer throughput can scale to the number of sharing compute nodes.

4.8 Summary

In this work, we explored the use of burst buffer storage for scalable checkpoint/restart, and developed IBIO to exploit the high bandwidth of the burst buffers for future extreme scale systems. We also developed a model to explore the performance difference of checkpointing strategies, specifically coordinated and uncoordinated checkpointing. We used the model to evaluate multilevel checkpointing on flat buffer storage systems that are currently available on today's machines and hierarchical storage systems using burst buffers.

From our exploration, we found that burst buffers are indeed beneficial for checkpoint/restart on future systems, increasing reliability and efficiency. We also found that the performance of the parallel file system has high impact on the efficiency of a machine, while increased bandwidth to burst buffers did not affect overall machine efficiency. However, the reliability of burst buffers does impact efficiency, because unreliable buffers mean more I/O traffic to a PFS when multiple burst buffer nodes fail, and checkpoints on the failed burst buffer nodes are lost. Overall, uncoordinated checkpointing was more efficient than coordinated checkpointing, even with high message logging overhead. These findings can benefit system designers in making the trade-offs in performance of components so that they can create efficient and cost-effective machines.

Chapter 5

FMI: Fault Tolerant Messaging Interface

The second approach, a *fault tolerant messaging interface (FMI) for fast and transparent recovery*, is a survivable messaging interface that uses fast, transparent in-memory checkpoint/restart and dynamic node allocation. With FMI, a developer writes an application using semantics similar to Message Passing Interface (MPI). The FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance, and achieve quick recovery and restart. Our experiments show that FMI runs with similar failure-free performance as MPI, but FMI incurs only a 28% overhead with a very high mean time to failure of 1 minute.

5.1 Introduction to Fault Tolerant Messaging Interface

The Message Passing Interface (MPI) [75] is the de-facto HPC programming paradigm, but it employs a fail-stop model. On failure, all processes in the MPI job are terminated. MPI applications generally cope with failures using checkpoint/restart schemes. They periodically write their state to files on a reliable store, such as a parallel file system (PFS). When a failure occurs, the current job is terminated, and the application is relaunched as a new job that restarts from the last checkpoint. The approach is simple, but it incurs significant overheads due to high checkpoint and restart costs [76, 77].

In environments with high-frequency failures, it is critical that applications restart quickly, so that useful work can be done before the next failure occurs. There are four capabilities needed for resilience in such an environment: a messaging interface that can run through faults, fast in-memory or node-local checkpoint storage, fast failure detection, and a mechanism to dynamically allocate additional compute resources in the event of hardware failures.

Our approach to satisfy these requirements is the Fault Tolerant Messaging Interface (FMI), a survivable messaging interface that uses fast, transparent in-memory checkpoint/restart and dynamic node allocation. With FMI, a developer writes an application using semantics similar to MPI. The FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance. Our implementation of FMI has failure-free performance that is comparable with MPI. Experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high mean time to failure of *1 minute*.

Our key contributions are:

≤ a simplified programming model to enable fast, transparent checkpoint/restart;

≤ implementation of a runtime that withstands process failures and allocates spare resources;

≤ a new overlay network structure called *log-ring* for scalable failure detection and notification;

≤ and demonstration of the fault tolerance and scalability of FMI even with a MTBF of 1 minute.

Our paper is organized as follows. We present characteristics of failures on large systems, and we identify critical resilience capabilities in Section 5.2. In Section 5.3, we introduce FMI, and we detail its implementation in Section 5.4. We describe our in-memory checkpoint/restart strategy in Section 5.5. In Section 5.6, we present our experimental results. In Section 5.7, we summarize this chapter.

5.2 Requirement for Survivable Messaging Interfaces

5.2.1 Fast Checkpoint/Restart

A non-fail-stop communication interface itself is not sufficient for fault tolerant computing at extreme scales, because on node failure needed application state can be lost. Today, applications use simple checkpoint/restart with checkpoints written on a reliable PFS to mitigate node failures. While simple checkpoint/restart is sufficient for small

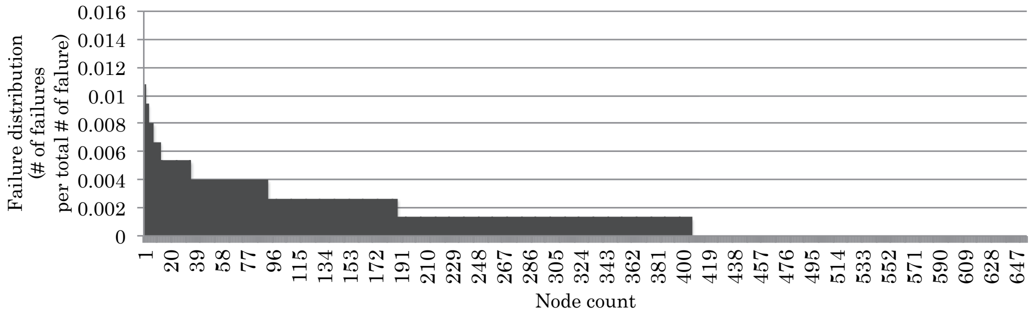


FIGURE 5.1: TSUBAME1: Number of failures normalized by total failures (Period: 3.5 years of Oct 26^{th} 2006 - March 21^{th} 2010, Total # of failures: 744)

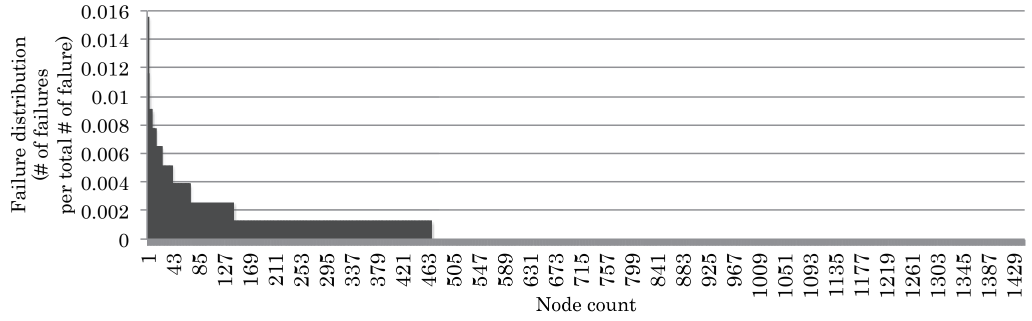


FIGURE 5.2: TSUBAME2.0: Number of failures normalized by total failures (Period: 1.5 years of Nov 1^{st} 2010 - April 6^{th} 2012, Total # of failures: 962)

systems, it can incur huge overheads, and is not practical as the sole method of fault tolerance at large scales.

Multilevel checkpoint/restart is an approach for lowering the overheads associated with traditional checkpoint/restart. Multilevel checkpoint/restart [4] caches checkpoints in in-system storage such as RAM or other node-local storage. It also uses encoding techniques to preserve application data on node failures, and copies a select few to the PFS to survive more catastrophic failures. Because we expect node failure to be more common on future systems, we need the fast checkpoint/restart provided by multilevel checkpointing to make these failures recoverable in a reasonable amount of time.

5.2.2 Failure locality

In a non-fail-stop model, non-failed processes are not terminated and keep running. Newly allocated spare nodes dynamically join the execution without stopping the running processes. One solution for acquiring additional nodes is to request additional nodes in the allocation, and use a subset of the allocated nodes for the job, reserving the additional nodes as spares. If we know the failure rate of the target system, we can estimate the appropriate number of spare nodes for a given job. However, we expect this estimation to be more difficult at extreme scales because there may be a larger number of nodes that fail concurrently. Another alternative is to request compute nodes from the resource manager as needed. This has the disadvantage of possibly being a very high overhead operation. However, this could be mitigated if the resource manager kept a reserve of spare nodes for just this use.

Another complication is that we observe that the occurrence of failures has both spatial and temporal locality. For example, four TSUBAME2.0 nodes and two LLNL Coastal nodes respectively share a single power supply. When a power supply fails, it takes several compute nodes with it. If system has a fat-tree network topology, a switch failure can causes more catastrophic failures.

Figure 5.1 and 5.2 show the number of failures normalized by the total number of failures on TSUBAME1 and TSUBAME2.0 supercomputers respectively. The failure analysis shows that the most of nodes are highly reliable, and the overall failure rate is inflated by a small number of faulty nodes. By replacing the failed components, the component may decrease. But a small number of nodes, which stochastically several faulty components, fails several times, which result in the spatial locality on failures. Moreover, another failure analysis [78] by Los Alamos National Laboratory (LANL) show about 19% of root causes is unknown. In that case, replacement of the components is obviously impossible. If the failure is transient, the node may fail again after the short period of time.

The LANL failure analysis also showed that there is temporal locality, and failure rate fluctuate one order of magnitude during different time period. Due to the fact, failure rate can severely vary depending on application's resource usage, allocated nodes, and time period. Thus, determining of the number of spare nodes is impossible on a execution, and dynamic node allocation are important.

5.2.3 Existing Survivable Messaging Interfaces

As described in Section 2.4.3, a survivable messaging interface itself is not new techniques, and there are several existing survivable messaging interfaces [57, 58, 62]. However, the capabilities are limited as in Section 2.4.3. At extreme scale, the rack of the fault tolerance capabilities of existing survivable messaging interfaces will make fault tolerant programming complicated. Thus, a survivable messaging interface, which provides scalable failure detection, auto-recovery, and fast checkpoint/restart for MPI applications, is critical.

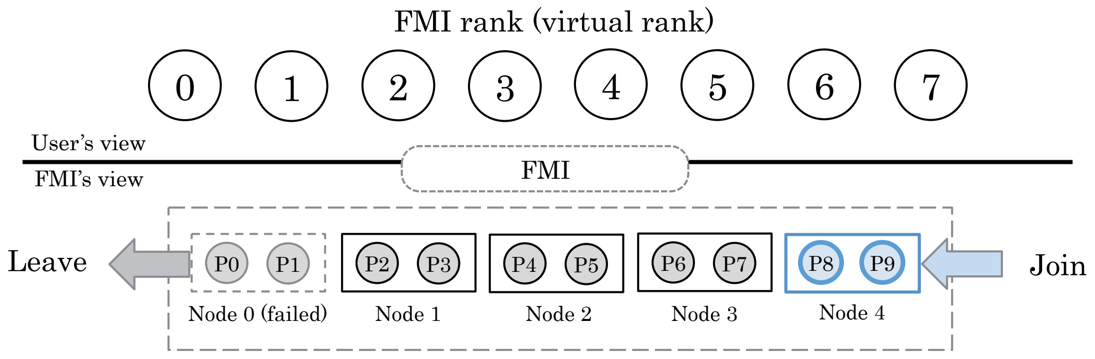


FIGURE 5.3: Overview of FMI

5.3 FMI Programming Model

In this section, we describe the FMI programming model with an overview and then with an example.

5.3.1 Overview

With FMI, an application developer writes an application with MPI semantics, and FMI ensures that the application is agnostic to failure. The FMI runtime software handles the fault tolerance activities, including fast checkpointing of application state, restarting failed processes on failure, restoring application state, and allocating additional nodes when needed. FMI does not belong to any existing MPI implementations, and is developed from full scratch for fault tolerance while maintaining MPI programing model and API as much as possible.

In Figure 5.3, we give an overview of FMI. Each process in an FMI application has an *FMI rank* as in MPI. But unlike MPI, an FMI rank is virtual and not bound to a particular process (Px) on a physical node. FMI may change the mapping of FMI ranks to processes to hide underlying hardware failures, transparently to the application. FMI also provides a capability for compute nodes to join or leave the job dynamically, primarily to replace failed nodes with spare nodes. Although our current prototype of FMI has limitations to support general applications, FMI transparently intercepts MPI calls, so that existing MPI applications can run on top of FMI without any code changes or with minimal code changes.

```
int main(int *argc, char *argv[]) {
  FMI_Init(&argc, &argv);
  while ((n = FMI_Loop(...)) < numloop) {
     /*user program*/
  }
  FMI_Finalize();
}
```

FIGURE 5.4: FMI example code

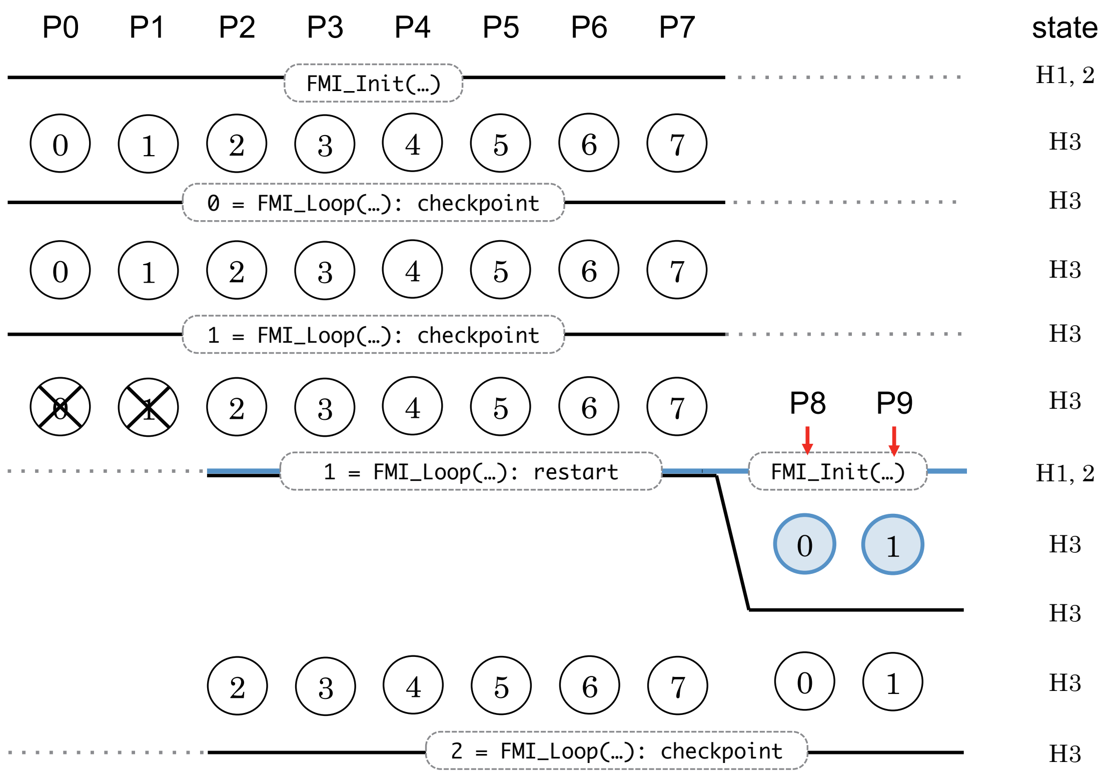


FIGURE 5.5: Example: P0(rank=0) and P1(rank=1) fail after loop_id=1. P8 and P9 start from loop_id=1 as rank 0 and 1 each, and the other processes retry loop_id=1

5.3.2 Writing an FMI Application

Writing a fault tolerant application is usually a complex ordeal, especially if there are a large number of dependencies across processes. With FMI, application developers simply write their code with MPI semantics and fault tolerance is provided by FMI. Figure 5.4 shows an example main loop for a code using FMI. The primary difference between the FMI and MPI programming models is that FMI provides the function `FMI_Loop` that synchronizes the application, writes checkpoints, or rolls back and restarts as needed. This single function call makes an application fault tolerant:

`int FMI_Loop(void`$^{\infty\infty}$`ckpts, int`$^{\infty}$`sizes, int len)`.

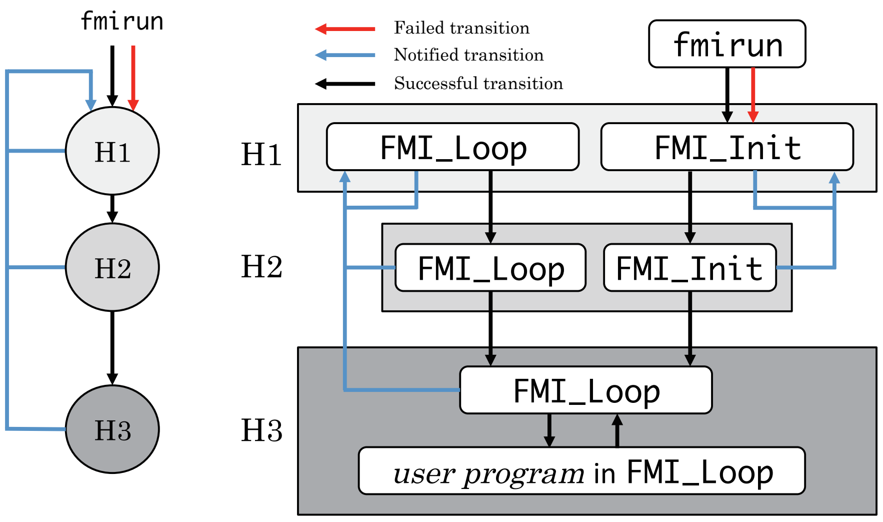
The parameter `ckpts` is an array of pointers to variables that contain data that needs to be checkpointed. If a failure occurred during the previous loop iteration, the last good values of the variables replace the values in `ckpts` to roll back to the last checkpoint. The parameter `sizes` is an array of sizes of each checkpointed variable, and `len` is the length of the arrays. `FMI_Loop` returns the loop iteration count (`loop_id`) incremented from 0 regardless of whether a checkpoint was written during this loop or not. However, if `FMI_Loop` rolls back and restores the last checkpoint, it returns the `loop_id` during which the last checkpoint was written.

When `FMI_Loop` is called the first time at the beginning of the execution, it writes checkpoints in memory using `memcpy` to minimize checkpoint time, and applies erasure encoding to the checkpoints using XOR encoding (See Section 4.2). When completed, `FMI_Loop` guarantees that an application can continue to run even on a failure within the loop as long as any failures that occur are recoverable. After the first call, `FMI_Loop` writes checkpoints at an interval specified by an *interval* environmental variable. Alternatively, if a user specifies an *MTBF* environmental variable, FMI dynamically auto-tunes the checkpoint interval to maximize efficiency according to the MTBF based on Vaidya's model [46].

Figure 5.5 shows an example where `FMI_Loop` writes checkpoints every loop, i.e. *interval*=1. If a failure occurs (e.g., after `loop_id` = `1`), all FMI ranks are notified of the failure by FMI (See Section 5.4.3), and all FMI communication calls return an error until recovery is performed in `FMI_Loop`. Then, the FMI process management program (`fmirun`) transparently allocates another node and spawns new processes (P8, P9 in the example) on the node to keep the number of FMI ranks constant. Then, `FMI_Loop` restores the values of the checkpointed variables from `loop_id` = `1` and returns `loop_id` = `1`. All recovery operations are transparent to the application, and all processes are simply rolled back to the last good state.

5.4 FMI System Design

In this section, we detail our implementation of FMI. We describe our methods for keeping track of process states, managing dynamic node allocation, and joining new processes into the running application; our new overlay network design called *log-ring* for scalable failure detection and notification; and our method for transparently recovering communicators.



(a) High level view (b) Low level view

FIGURE 5.6: Process states for FMI

5.4.1 Process State Management

FMI manages the states of all processes, tracking whether or not processes are running successfully, and synchronizing for recovery when a failure occurs. Figure 5.6 shows a high level and low level view of transitions of process states. There are three process states in FMI: *Bootstrapping* (H1), *Connecting* (H2), and *Running* (H3).

The H1 and H2 states involve launching processes and establishing internal FMI communication networks, while the H3 state represents the running state of the application. In the H1 state, `fmirun` launches the FMI ranks (See Section 5.4.2), which then gather connection information (*endpoints*) to establish a dedicated low-latency communication network, similar to Open MPI's Matching Transfer Layer [79]. FMI uses *PMGR* [80], which provides a scalable communication interface for bootstrapping MPI jobs and exchanging messages via TCP/IP. On success, the processes transition to the H2 state, where the FMI ranks create a *log-ring* overlay network for scalable failure detection (See Section 5.4.3). If both H1 and H2 states succeed, the processes transition to H3. In the H3 state, the processes execute the application code, with the addition of `FMI_Loop`, that performs fault tolerance activities as described in Section 5.3.2.

If any processes terminate because of a failure, `fmirun` launches new processes to replace them; they begin in the H1 state. The non-failed processes transition from their current state back to H1 on the *Notified transition* path. All processes then update endpoints to transparently recover communicators (See Section 5.4.4). On success, all processes transition to H2 and then H3 on *Successful transition* paths.

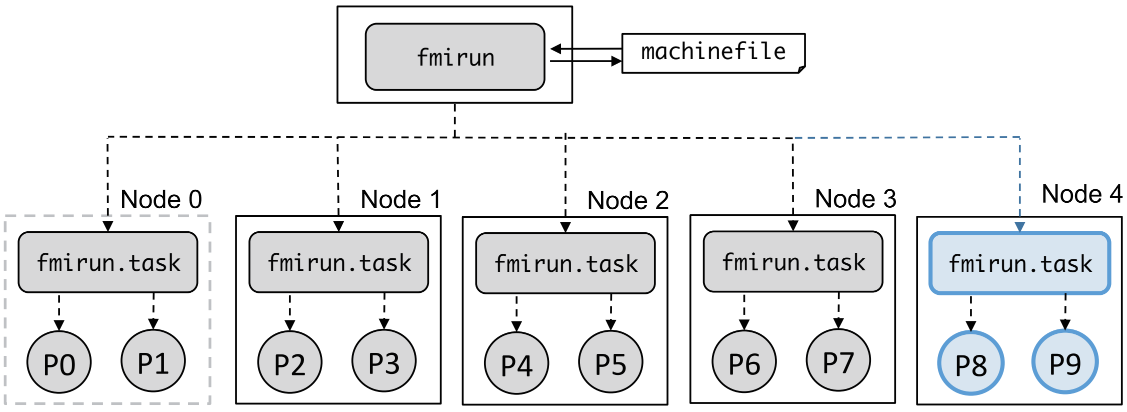


FIGURE 5.7: FMI process management

Figure 5.6(b) shows details of the states and transitions. H1 and H2 are synchronizing states because they involve collective communications. Newly launched processes in H1 execute `FMI_Init`. Non-failed processes block in `FMI_Loop` until the new processes are bootstrapped and endpoints in the internal communication network are established. Then all processes transition to H2 to rebuild the *log-ring* network. Following this, the application computation begins from the previous iteration or the iteration with the last good checkpoint.

5.4.2 Hierarchical Process Management

Figure 5.7 shows an overview of the hierarchical structure of FMI process management. The master process `fmirun` is at the top level. `fmirun` has similar functionality to `mpirun` in MPI, but also manages processes during recovery in the event of failure. `fmirun` spawns `fmirun.task` processes on each node, which are at the second level of the hierarchy. Each `fmirun.task` calls `fork/exec` to launch a user program (`Px`) and manages the processes on its own local node.

If any `fmirun.task` receives an unsuccessful exit signal from a child process, `fmirun.task` kills any other running child processes, and exits with `EXIT_FAILURE`. When `fmirun` receives an exit signal from an `fmirun.task`, `fmirun` attempts to find spare nodes to replace those that failed in the `machinelist` file; if no spare nodes are found, or if there are not enough to replace all that failed, it requests new nodes from the resource manager. It then spawns any lost `fmirun.task` processes onto the spare nodes.

In our design, the master process (`fmirun`) becomes a single point of failure. However, because the MTBF of a single node in HPC systems is an order of years[4, 81], the failure rate for `fmirun` is negligibly small. That said, we plan to explore distributed management designs in future work.

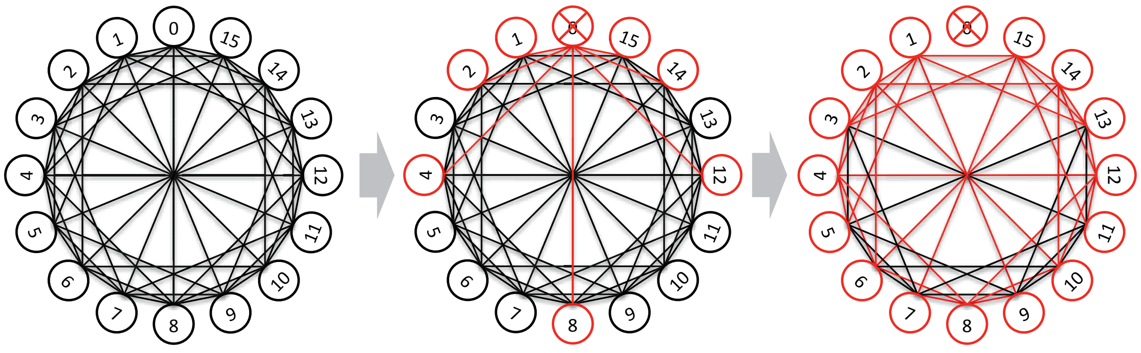


FIGURE 5.8: Left: Structure of the *log-ring* overlay network ($n = 16$). Middle: If process 0 fails, processes 1, 2, 4, 8, 12, 14, and 15 are notified by ibverbs. Right: All processes are notified with 2 hops.

5.4.3 Scalable Failure Detection

On failure, all surviving processes need to be notified so that the recovery process can begin. However, not all low-level communication libraries include a failure detection capability. For example, the Performance Scaled Messaging (PSM) library, a low-latency communication library for QLogic Infiniband, returns an error if there is a failure during connection establishement. However, once the connection is established, successive communication calls (e.g., sends or receives), do not return any errors even in the event of a peer failure.

One approach for detecting failures is that when `fmirun` receives a `EXIT_FAILURE` signal from an `fmirun.task`, `fmirun` could send notification signals to all other `fmirun.task` processes. However, the time complexity of this approach is $O(N)$ in the number of compute nodes, which is not scalable.

Our approach to failure detection is to use a *log-ring* overlay network across the FMI ranks which can propagate failure notification in $\rceil\rfloor\log_2(n)\lceil/2\{$ messages. We use the Infiniband Verbs API, *ibverbs*, a low-level communication library for Infiniband. The ibverbs library includes an event driven error notification capability, such that, if a process fails, all processes connected to the failed process can detect it by catching the error event.

The structure of the overlay network is critical for scalable failure detection. One option is a completely connected graph, where each process connects to all the other processes. In this option, notification of failure to all n processes occurs in $O(1)$ steps. However, establishing the complete graph overlay network is $O(n)$. In contrast, if we establish a ring overlay network, the connection cost is $O(1)$, but propagation of failure notification to all processes is $O(n)$.

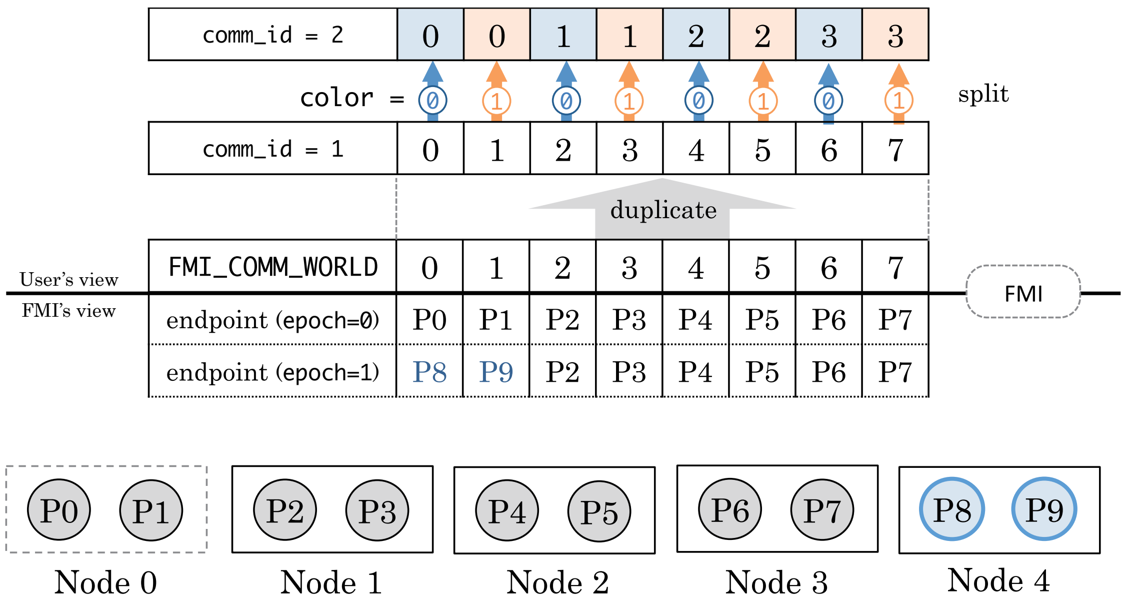


FIGURE 5.9: Transparent communicator recovery

To achieve a good balance between the overlay establishment cost and the global detection cost, we propose a *log-ring* overlay network. In a *log-ring* overlay network, each process makes $\log_2(n)$ connections with neighbors that are 2^k hops away in the FMI rank space ($2^k < n$). Figure 5.8 shows an example *log-ring* overlay network with $n = 16$ processes. In the example overlay, process 0 connects to 1, 2, 4, and 8 ($\log_2(16)$=4 connections), and receives connections from processes 8, 12, 14 and 15 (left in Figure 5.8). If process 0 fails, processes 1, 2, 4, 8, 12, 14 and 15 receive a disconnection event from ibverbs (middle in Figure 5.8). Once these processes receive the disconnection event, they explicitly close their other existing connections to propagate the failure notification. In this way, the failure notification reaches all processes with 2 hops in the example overlay network (right in Figure 5.8). In general, the *log-ring* overlay network can propagate failure notification across all processes with $\rceil\rfloor\log_2(n)\lceil/2\{$ hops. Thus, the overlay network establishment and global failure notification are of order $O(\log(n))$ and scalable.

The value of k in $\log_k(n)$ is a tunable parameter in FMI $\log_k(n)$. Changing its value can change the overlay establishment and the global detection costs. As we show in Section 5.6.1, the establishment cost is negligible even with $k = 2$, which has maximal connections in the log-ring. Thus, we use $k2$ as the default value, but we leave the optimization of k for future work.

5.4.4 Transparent Communicator Recovery

One of obstacles of fault tolerant parallel computing is recovering communicators used in message passing. In FMI, communicators are a mapping between FMI ranks and physical processes, so recovering communicators is necessary for communication with the newly launched processes after recovery. Because FMI virtualizes the rank-to-process mapping, it can transparently recover communicators. Figure 5.9 shows an example where the application duplicates `FMI_COMM_WORLD`, and splits the communicator into two communicators. If a failure occurs on Node 0 and P0 and P1 terminate, FMI changes the process mapping by updating connection information (endpoint). In this example, FMI transparently updates the endpoints of FMI ranks 0 and 1 to P8, P9 respectively during the H1 bootstrapping state (See Figure 5.6(b)).

Another problem to address is that stale messages could be received if they are sent before a failure and not yet received by the target rank. For example, if process A sends a message before a failure, but process B does not receive it before the failure, it may receive the stale message when it executes the receive operation after recovery. To address the problem, FMI increments an `epoch` variable after each recovery, and discards all messages that arrive with an older `epoch` value.

5.5 Fast and Scalable In-Memory checkpoint/restart

With FMI, an application can continue to run even if a failure occurs. However, if a node fails, we may lose needed simulation data from processes on the failed node, so checkpoint/restart is critical and must be scalable to be effective at large scales. Because the most common failures affect only a single or a few nodes [4, 70], multilevel checkpoint/restart is effective and scalable.

5.5.1 Implementation

FMI employs a multilevel checkpoint/restart strategy, using the same checkpoint and encoding algorithm as the Scalable Checkpoint/Restart library (SCR) [4]. However, while SCR requires a file system interface for storing checkpoints, FMI writes checkpoints directly to memory without involving a file system for faster checkpoint/restart throughput. Unlike with MPI, FMI does not terminate non-failed processes on a failure, and in-memory checkpoint data is not flushed.

FMI employs the same encoding algorithm as SCR. At initialization, FMI splits ranks into XOR encoding groups (XOR group) with ranks in each group distributed across

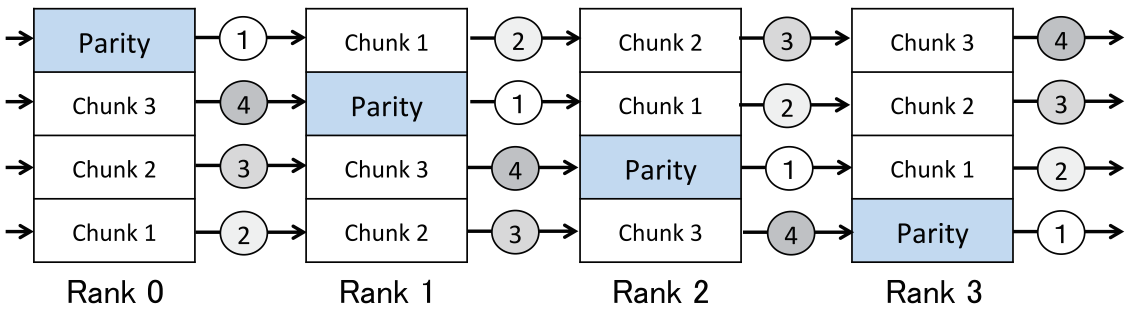


FIGURE 5.10: XOR encoding algorithm: The circled numbers are the steps of sending/receiving parity

nodes. Because the common failure affects a single node, FMI ensures that each rank in the same node belongs to a different XOR group. When processes are launched as in Figure 5.7, FMI splits the ranks as shown in Figure 5.9. Figure 5.10 shows the encoding algorithm for one XOR group. First, for an XOR group size of n, FMI divides a checkpoint into $n \quad 1$ chunks, and allocates an additional parity chunk initialized with zeros. Each rank sends the parity chunk to its "right-hand" neighbor, and receives from its "left-hand" neighbor. and calculates "`parity` $\surd=$ `chunk 1`". In general, each rank sends and receives a parity chunk, and computes "`parity` $\surd=$ `chunk k`" at step k (circled number in Figure 5.10). Thus, each rank receives back the encoded parity chunk after n steps. When FMI restores a checkpoint, FMI decodes it with the same algorithm as the encoding, and then a newly launched rank collects the decoded checkpoint chunks from the other ranks.

5.5.2 Performance Model

When FMI writes s bytes of checkpoint data and encodes it, s bytes are copied by `memcpy`, and $s + \frac{s}{n \quad 1}$ bytes are transferred, with s bytes are encoded in total. Therefore, the time for checkpoint/restart can be modeled as:

$$\frac{s}{mem\_bw} + \frac{s + s/(n \quad 1)}{net\_bw} + \frac{s}{mem\_bw}$$

where mem_bw is memory bandwidth, and net_bw is network bandwidth. Because the XOR operation is memory-bound, the time becomes $\frac{s}{mem\_bw}$. When restoring a checkpoint, a newly launched rank collects the decoded checkpoint chunks from the other ranks at the end, so $\frac{s}{net\_bw}$ is added for restart.

The model tells us that the checkpoint/restart time is constant regardless of the total number of processes. Thus, the in-memory XOR checkpoint/restart is scalable as well as fast.

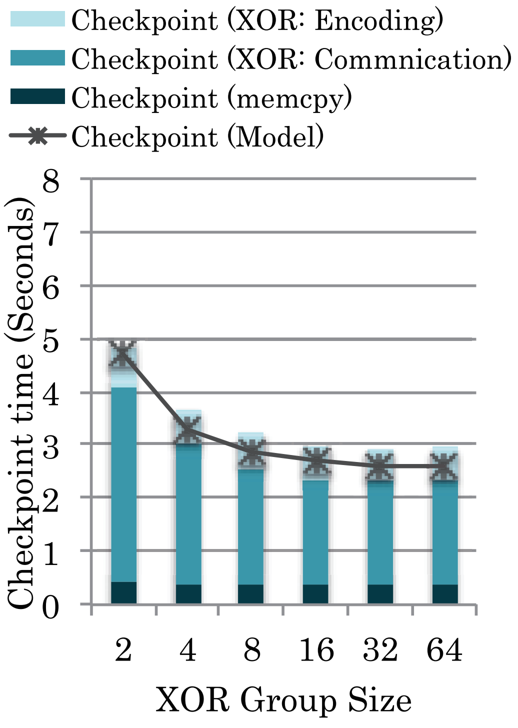


FIGURE 5.11: XOR checkpoint time

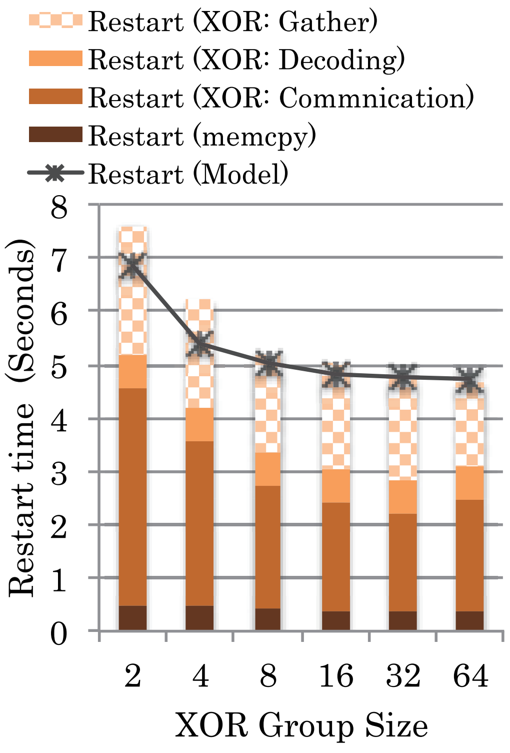


FIGURE 5.12: XOR restart time

5.5.3 XOR Group Size Tuning

FMI uses memory to store checkpoints. Thus, reducing memory consumption while maintaining resiliency is important. If an XOR group size is small, memory consumption and checkpoint/restart time become large. For large XOR group sizes, resiliency decreases because the XOR checkpoint/restart encoding is tolerant to only a single rank failure in a XOR group. We performed experiments to evaluate the trade-offs of checkpoint/restart time and XOR group size.

Figures 5.11 and 5.12 show the checkpoint and restart times where the checkpoint size is 6GB per node. For the memory and network bandwidths in the model, we use the peak bandwidth of the Sierra cluster at LLNL in Table 5.1. We find that the checkpoint/restart time starts to saturate at an XOR group size of 16 nodes. For this XOR group size, the parity chunk size is only 6.6 % of the full checkpoint size. Thus, we use 16 nodes for the XOR group size in the rest of our experiments.

TABLE 5.1: Sierra Cluster Specification

Nodes	1,856 compute nodes (1,944 nodes in total)
CPU	2.8 GHz Intel Xeon EP X5660 ⊗ 2 (12 cores in total)
Memory	24GB (Peak CPU memory bandwidth: 32 GB/s)
Interconnect	QLogic InfiniBand QDR

5.6 Experimental Results

To evaluate the performance and resiliency of FMI, we measured several benchmarks with FMI, and predict the performance of an FMI application run at extreme scale. We ran our experiments on the Sierra cluster at LLNL. The details of Sierra are in Table 5.1. Because FMI follows the messaging semantics of MPI, we want to compare the performance of FMI with an MPI implementation. For those experiments, we used MVAPICH2 version 1.2 running on top of SLURM [82].

5.6.1 FMI Performance

TABLE 5.2: Ping-Poing Performance of MPI and FMI

	1-byte Latency	Bandwidth (8MB)
MPI	3.555 usec	3.227 GB/s
FMI	3.573 usec	3.211 GB/s

We measured the point-to-point communication performance on Sierra, and compare FMI to MVAPICH2. Table 5.2 shows the ping-pong communication latency for 1-byte messages, and bandwidth for a message size of 8 MB. Because FMI can intercept MPI calls, we compiled the same ping-pong source for both MPI and FMI. The results show that FMI has very similar performance compared to MPI for both the latency and the bandwidth. The overhead for providing fault tolerance in FMI is negligibly small for messaging.

Because failure rates are expected to increase at extreme scale, checkpoint/restart for failure recovery must be fast and scalable. To evaluate the scalability of checkpoint/restart in FMI, we ran a benchmark which writes checkpoints (6 GB/node), and then recovers using the checkpoints. Figure 5.13 shows the checkpoint/restart throughput including XOR encoding and decoding. The checkpoint time of FMI is fairly scalable because the checkpointing and encoding times are constant regardless of the total number of nodes. Also, because FMI writes and reads checkpoints to and from memory, the throughputs

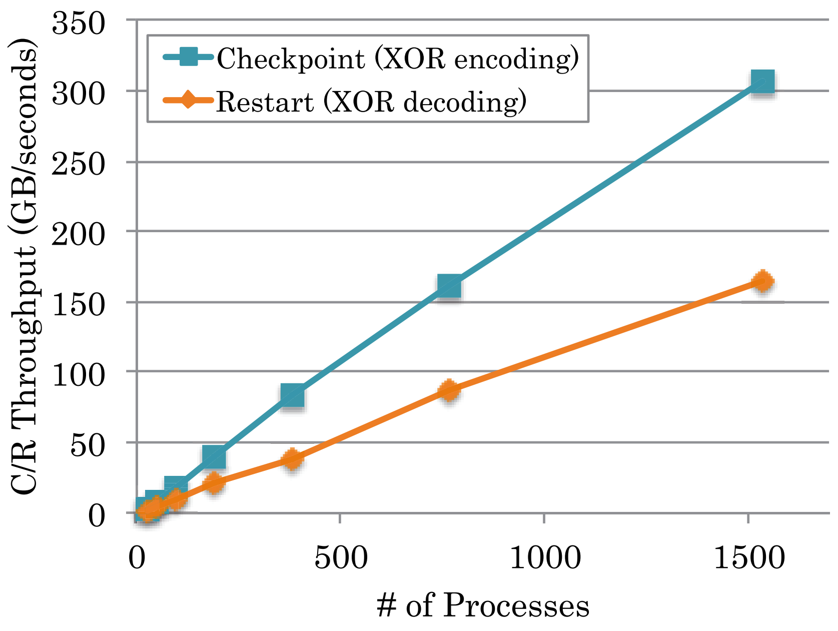


FIGURE 5.13: Checkpoint/Restart scalability with 6 GB/node checkpoints, 12 processes/node

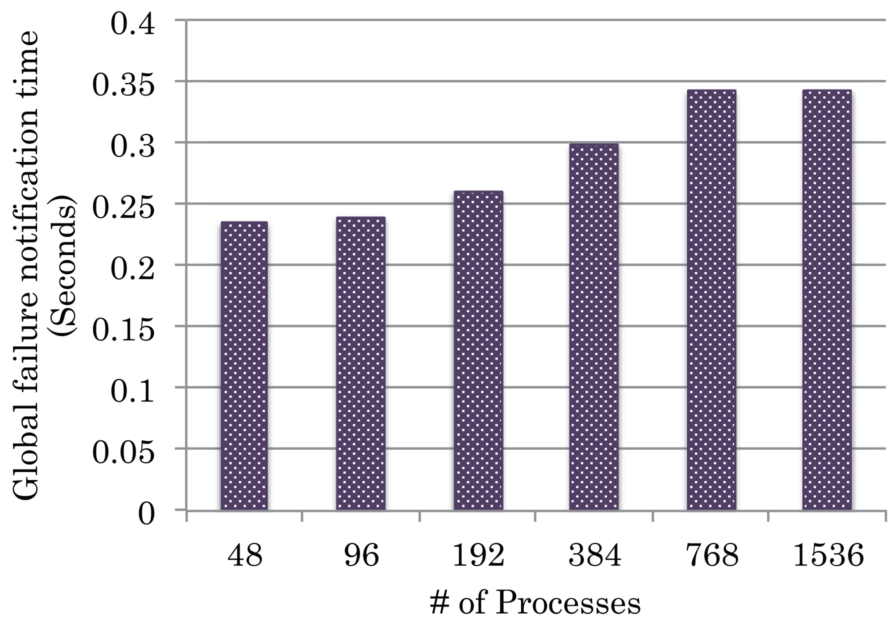


FIGURE 5.14: Failure notification time with *log-ring* overlay network

are high. FMI achieves 2.4 GB/sec checkpointing throughput per node, and 1.3 GB/sec restart throughput per node. On a restart, newly launched processes gather the restored checkpoint chunks from the other processes in the XOR group after the decoding as in Figure 5.12, so the restart throughput is lower than that of checkpointing.

Fast and scalable failure detection time and reinitialization time (H1 and H2 states)

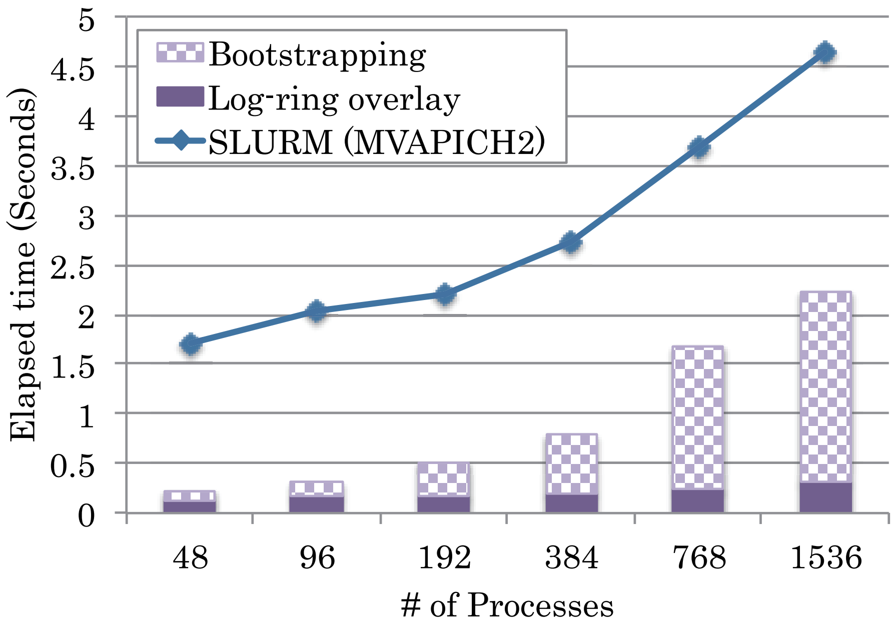


Figure 5.15: MPI_Init vs. FMI_Init

are critical in environments with high failure rates. Figure 5.14 shows the time for all processes to be notified of failure with the *log-ring* overlay In this experiment, we inject a failure by sending a signal to kill a process. As shown, the global detection is scalable because the log-ring propagates the notification in logarithmic time. When a process terminates, ibverbs waits approximately 0.2 seconds before closing the connection to the terminated process. Therefore there is a constant overhead of 0.2 seconds before the notification process begins in the *log-ring*.

FMI establishes the log-ring overlay network (H2 states) on the recovery. The initialization must be fast and scalable for fast recovery. In Figure 5.15, we show the initialization time for MVAPICH2 and FMI. For FMI, this is time spent in the H1 and H2 states. We compare the time in `FMI_Init` with that in MVAPICH2's `MPI_Init`. We see that the time to build the *log-ring* (H2 state) is small and scalable, because each process only connects to $\log_2 n$ other processes. The FMI bootsrapping time (H1 state) is about two times faster than that of MVAPICH2. The current prototype of FMI has limited capabilities compared to MPI. A smaller number of messages are exchanged in FMI initialization than in MVAPICH2, which results in faster bootstrapping. However, we expect that if FMI evolves to support more capabilities, it will also exchange more messages and its initialization time will approach that of MVAPICH2.

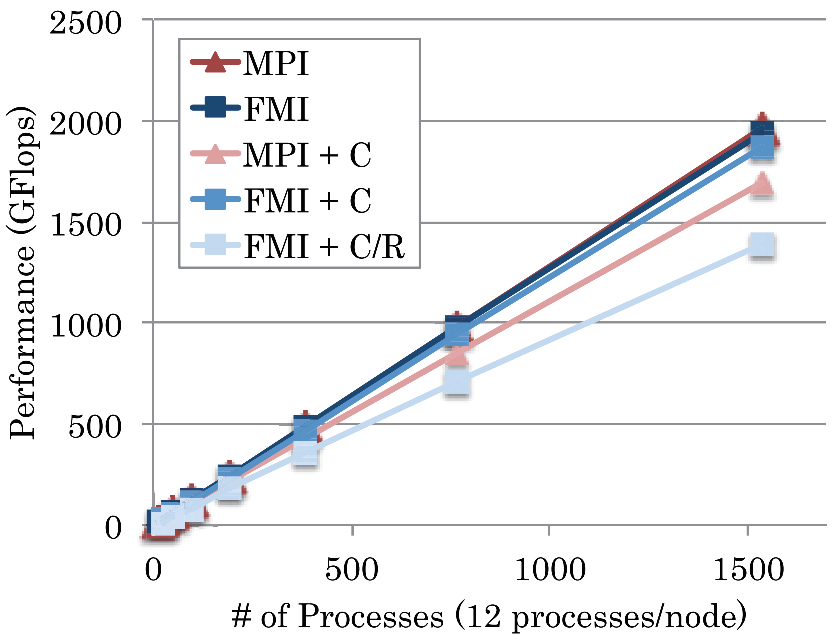


FIGURE 5.16: Himeno benchmark (Checkpoint size: 821 MB/node, MTBF: 1 minute)

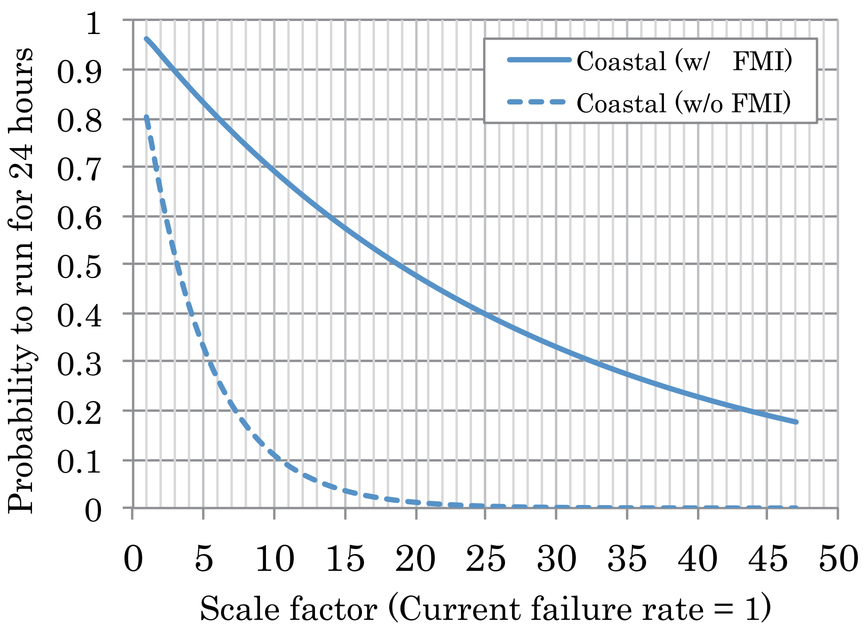


FIGURE 5.17: Probability to continuously run for 24 hours

5.6.2 Application Performance with FMI

To investigate the impact of FMI on the performance of an actual application run, we used a Poisson equation solver, the Himeno benchmark [69]. Himeno is a stencil application in which each grid point is iteratively updated using only neighbor points.

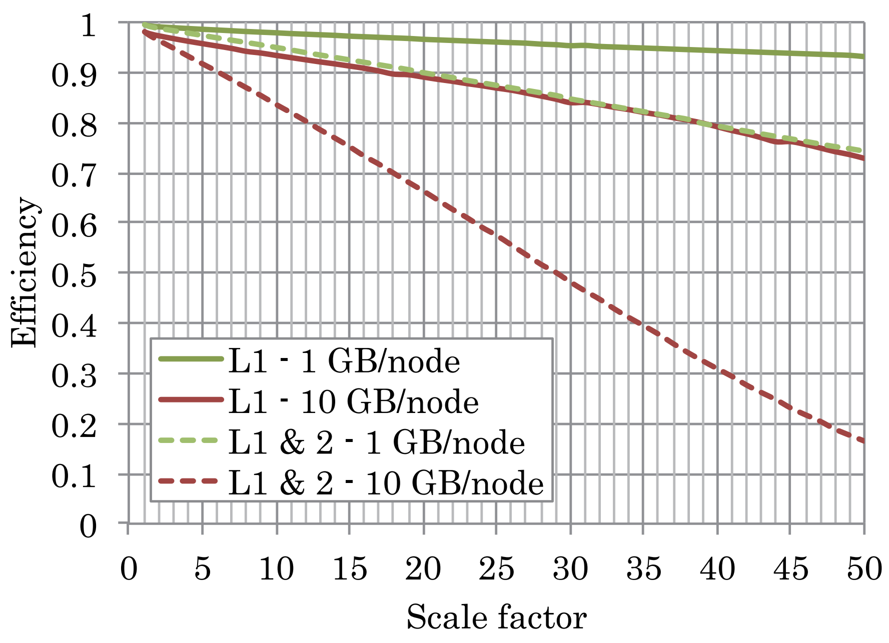


Figure 5.18: Efficiency of multilevel checkpoint/restart under increasing failure rate and L2 checkpoint/restart time

The computational pattern frequently appears in numerical simulation codes for solving partial differential equations. Himeno uses point-to-point communications and one Allreduce at the end of each iteration.

Figure 5.16 shows the performance of Himeno compared with MPI using SCR [4]. The FLOPS metric is computed based on time spent in application code making useful progress. For example, if an application fails at time t_1, and rolls back to time t_0, the FLOPS metric does not include the lost computation done to restore the application back to the state at t_1. We configured SCR to write checkpoints to `tmpfs` and optimize the checkpoint interval of both SCR and FMI with Vaidya's model [46] based on configured MTBF of 1 minute, and measured checkpointing time.

Because the point-to-point communication performance of FMI and MVAPICH2 are nearly the same (Table 5.2), the performance of Himeno is nearly the same for FMI and MPI if we do not write any checkpoints during the execution (FMI & MPI in Figure 5.16). For checkpointing, SCR writes to memory via a file system (MPI + C), while FMI writes checkpoints directly to memory using `memcpy` (FMI + C). Thus, FMI exhibits higher performance by 10.3 % with the same memory consumption as MPI when checkpointing is enabled. We also injected failures into Himeno to see the impact of killing a process with a MTBF of 1 minute during the execution. Even with the very high failure rate, we found that Himeno incurred only a 28% overhead with FMI. Because the FMI checkpoint/restart time is constant regardless of the total number of nodes according to performance model in Section 5.5.2, we expect FMI to scale to a much larger number of nodes.

5.6.3 Resiliency with FMI

FMI applications can continue to run as long as all failures are recoverable. To investigate how long an application can run continuously with or without FMI, we simulated an application running at extreme scale. If we assume failures occur according to a homogeneous Poisson process, the probability that an application runs for time T continuously is $e^{\ \lambda * T}$ where λ is the unrecoverable failure rate.

Figure 5.17 shows the probability to run continuously for 24 hours using failure rates from the LLNL failure analysis of the Coastal cluster [4], with a level-1 failure rate of 2.13 ^{6} (MTBF = 130 hours) (recoverable by XOR encoding), and level-2 failure rate of 4.27 ^{7} (MTBF = 650 hours) (unrecoverable failures). We increase the failure rates from the observed level-1 and 2 values by scale factors of 1 (observed values) to 50 to evaluate FMI's performance at larger scales. With FMI, 80% of executions can run for 24 hours with even 6⊗ higher failure rates. At failure rates of 10⊗ higher than today's, 70% of FMI executions can run continuously for 24 hours, while only 10% of non-FMI executions can do the same. Executions without FMI are terminated by any failures, while FMI executions are terminated by only level-2 failure. Thus, using FMI effectively decreases λ, and thus the probability of long continuous runs is higher with FMI, even at very high failure rates. At scale factor 50, level-1 MTBF becomes 2.6 hours (= 130 hours /50), which is quite long MTBF for FMI. As in Figure 15, FMI achieves an only 28% overhead (72% of efficiency) even with MTBF of 1 minute. If an unrecoverable failure does not happens, an application can run with negligibly small overhead.

If FMI uses a multilevel checkpoint/restart strategy and writes some checkpoints to the PFS (level-2 checkpoint/restart) in addition to XOR checkpoint/restart (level-1 checkpoint/restart), level-2 failures can also be recoverable. Here, we predict the efficiency of using multilevel checkpoint/restart with FMI, where efficiency is the ratio of time spent in useful computation only versus computation, checkpoint/restart activities, and recomputation after recovery. As future systems become larger, we expect higher failure rates and total aggregate checkpoint sizes. Thus, to predict application efficiency at larger scales, we increase failure rates and checkpoint costs up to 50⊗, using the Coastal system as a base line. Because level-1 checkpoint/restart time is constant regardless of the total number of nodes, we only increase level-2 checkpoint/restart time. For level-2 checkpoint/restart, we assume we write checkpoints asynchronously to the PFS using the framework and model developed in our prior work[81].

Figure 5.18 shows the efficiency of multilevel checkpoint/restart in FMI. Because we are uncertain as to whether level-2 failure rates will increase at extreme scale, in our evaluation we increase only the level-1 failure rate (L1) or both the level-1 and 2 failure rates (L1,2) with different checkpoint sizes per node (1 or 10 GB/node). We estimate level-1 checkpoint/restart time using the performance model in Section 5.5.2. For level-2 checkpoint/restart time, we use a PFS bandwidth of 50 GB/s, the bandwidth of the LLNL Lustre file system `/p/lscratchd`. We find that we can achieve fairly high efficiencies if future systems can keep current level-2 failure rates constant or keep the size of checkpoints small. However, if both level-1 and 2 failure rates increase and the checkpoint size is large, the efficiency drops down to under 2%. Thus, future systems must either decrease level-2 failure rates or increase PFS throughput to achieve high system efficiency.

5.7 Summary

From our analysis of failures on tera- and peta-scale systems, we have identified four critical capabilities for resilience with extreme scale computing: a survivable messaging interface that can run through process faults, fast checkpoint/restart, fast failure detection, and a mechanism to dynamically allocate spare compute resources in the event of hardware failures. To satisfy these requirements, we designed and implemented the Fault Tolerant Messaging Interface (FMI), a survivable messaging runtime that uses in-memory checkpoint/restart, scalable failure detection, and dynamic spare node allocation. With FMI, a developer writes applications using semantics similar to MPI, and the FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance. Our implementation of FMI has performance comparable to MPI. Our experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high MTBF of 1 minute. By defining a simplified programming model and custom runtime, we find that FMI significantly improves resilience overheads compared to similar existing multi-level checkpointing methods and MPI implementations.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The computational power of HPC systems is growing exponentially enabling finer grained scientific simulations. However, as the capability and component count of the systems increase, the overall failure rate increases accordingly. Without a viable fault tolerance technique, applications will be unable to continuously run for even one day on current supercomputers, and for a hour on future systems. Therefore, resilience in HPC has become more important than ever as we plan for future larger-scale systems. To make progress despite of system failures, applications periodically write checkpoints to a reliable PFS so that the applications can restart from the last checkpoint even on a failure. While simple, this straightforward checkpointing scheme can impose huge overhead on application run times for both checkpoint and restart operations. Although several techniques have been proposed to reduce the overhead, these techniques are not sufficient, had several limitation for extreme scale computing.

First, to understand detailed system failures on current and future supercomputers, we first analyzed failure history on TSUBAME supercomputers, systems in LLNL and other systems in Chapter 2. We also review existing studies mainly in checkpoint/restart, and then described three main challenges to achieve more fault tolerant computing for future extreme scale systems.

The first challenge was reducing checkpointing overhead/workload to PFS. Computational capabilities are increasing faster than PFS bandwidths. This imbalance in performance means applications can be blocked for order of hours for a single PFS checkpoint. Thus, the overhead of checkpointing to the PFS can dominate overall application run time even with infrequent PFS checkpoint by multi-level checkpointing. Further, the huge numbers of concurrent I/O operations from large-scale jobs burden the PFS and are themselves a major source of failures. Therefore, we must reduce the PFS load in order to achieve high reliability and efficiency

The second challenge was exploration of more reliable storage architectures. In multi-level checkpoint/restart, applications generally cache checkpoints in in-system storage such as RAM or other node-local storage on compute nodes because aggregated bandwidth of in-system storage scales with the increasing number of compute nodes. However, as mentioned above, the most common failures affect the compute nodes. Storing checkpoints in the in-system storage on compute nodes is not reliable solution because an application can not restart its execution if one of checkpoints stored across compute nodes is lost due to a failed compute node.

The third challenge was a demand for efficient recovery. Most failures only affect a small portion of the compute nodes so the vast majority of processes and connections are still valid after a failure. It is inefficient for the runtime to tear all of these down only to immediately relaunch and reconnect it all. Launching large sets of processes, loading executables and libraries from shared file systems, and bootstrapping connections between those processes takes non-trivial amounts of time.

To achieve the above three challenges, we proposed *asynchronous checkpointing system* for light-weight checkpointing to PFS, propose *fault tolerant messaging interface (FMI)* for fast and transparent recovery, and explore *multi-tier storage design with burst buffer* for reliable checkpointing.

The first approach, *multi-level asynchronous checkpointing*, solves the checkpointing overhead to PFS. We designed and modeled an asynchronous checkpointing system that extends an existing multi-level checkpointing system in Chapter 3. Our asynchronous checkpointing system enables applications to save checkpoints to fast, scalable storage located on the compute nodes and then continue with their execution while dedicated staging nodes copy the checkpoint to the PFS in the background. This capability simultaneously increases system efficiency and decreases required PFS bandwidth. Since applications spend less time in defensive I/O, we find that our asynchronous checkpointing system can improve machine efficiency by 1.1 to 2.0 times on future systems. Further, our model predicts that asynchronous checkpointing significantly reduces the PFS bandwidth required to maintain application efficiency. For example, to maintain 80% efficiency at 4$\otimes$ today's failure rates, the PFS bandwidth required for asynchronous checkpointing is an order of magnitude less than that required by synchronous checkpointing. Additionally, our asynchronous checkpointing system can maintain 80% efficiency with only modest PFS bandwidth requirements even

The second approach, *a user-level infiniband-based file system and checkpoint strategy for burst buffers*, consider using burst buffers to improve system resiliency. In Chapter 4, we explored the use of burst buffer storage for scalable checkpoint/restart, and developed IBIO to exploit the high bandwidth of the burst buffers for future extreme scale systems. We also developed a model to explore the performance difference of checkpointing strategies, specifically coordinated checkpointing where all processes checkpoint simultaneously, and uncoordinated checkpointing where subsets of processes coordinate a checkpoint instead of the whole job. We also developed a model to explore both checkpointing strategies and storage architectures. We used the model to evaluate multi-level checkpointing on flat buffer storage systems that are currently available on today's machines and hierarchical storage systems using burst buffers. From our exploration, we found that burst buffers are indeed beneficial for checkpoint/restart on future systems, increasing reliability and efficiency. We also found that the performance of the PFS has high impact on the efficiency of a machine, while increased bandwidth to burst buffers did not affect overall machine efficiency. However, the reliability of burst buffers does impact efficiency, because unreliable buffers mean more I/O traffic to a PFS when multiple burst buffer nodes fail, and checkpoints on the failed burst buffer nodes are lost. Overall, uncoordinated checkpointing was more efficient than coordinated checkpointing, even with high message logging overhead. These findings can benefit system designers in making the trade-offs in performance of components so that they can create efficient and cost-effective machines.

The third approach, a *fault tolerant messaging interface (FMI) for fast and transparent recovery*, is a survivable messaging interface that uses fast, transparent in-memory checkpoint/restart and dynamic node allocation. From our analysis of failures on tera- and peta-scale systems, we have identified four critical capabilities for resilience with extreme scale computing: a survivable messaging interface that can run through process faults, fast checkpoint/restart, fast failure detection, and a mechanism to dynamically allocate spare compute resources in the event of hardware failures. To satisfy these requirements, we designed and implemented the Fault Tolerant Messaging Interface (FMI) in Chapter 5, a survivable messaging runtime that uses in-memory checkpoint/restart, scalable failure detection, and dynamic spare node allocation. With FMI, a developer writes applications using semantics similar to MPI, and the FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance. Our implementation of FMI has performance comparable to MPI. Our experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high MTBF of 1 minute. By defining a simplified programming model and custom runtime, we find that FMI significantly improves resilience overheads compared to similar existing multi-level checkpointing methods and MPI implementations. In addition, FMI enables applications to survive from a system failure, and continuously run for longer time with high efficiency.

As a whole, these approaches enabled fast, scalable and transparent checkpoint and restart for extreme scale systems.

6.2 Future Work

6.2.1 FMI for general applications

Although the current FMI prototype has demonstrated promising results, it is not yet complete enough to support a broad range of applications. Here, we discuss the limitations of our prototype and how we plan to address them.

First, our prototype FMI implementation only supports a subset of MPI functions. For example, collective I/O, i.e., MPI_IO, is an important feature of MPI, because it is often used for checkpoint/restart to the PFS. Checkpointing to a PFS can be very time consuming, especially at large scales. Additionally, an application can experience higher than average failure rates of the PFS when there is high load on the PFS. Thus, a checkpoint may never complete due to frequent roll-backs. However, if we create parity data across nodes before initiating the MPI_IO operation, we can restore lost data and continue the I/O operation in the middle without starting over. Thus, support of MPI_IO in FMI is in our plans.

Second, several applications dynamically split a communicator with nested loops in order to balance the workload across processes. Such applications change not only application state but also communicator state over the iterations. To support such applications, future versions of `FMI_Loop` will support checkpoint/restart of communicators, and nested loops.

Third, our prototype does not support multilevel checkpoint/restart. FMI cannot recover from multiple nodes failure within XOR group. Future versions of FMI will support multilevel checkpoint/restart to be able to recover from any failures occurring on HPC systems.

FMI is an on-going project, and future FMI versions will remove the above limitations to support a wider range of applications, and become more resilient.

6.2.2 Energy-Efficient Checkpoint/Restart

Increased power consumption in modern large-scale supercomputers limits the design and the scale of the systems, and this power limitation is also applied to future extreme scale supercomputing systems. For example, the IESP (International Exascale Software Project) [83] reports that the upper limit of the power in an exa-scale supercomputer should be less than 20MW, while current power efficient technique requires over 60MW of the system. As described, in such large HPC systems, applications are required to conduct tera- or peta-byte scale massive I/O operations to local storages (e.g. a NAND flash memory) for scalable checkpoint and restart. However, as HPC system size grows, compute nodes perform less computation but consume huge power during checkpoint/restart, which result in undesirable energy consumption. Thus, energy efficiency of I/O operations is important at extreme scale

Recent rapid progress of flash technology introduces new memory devices in existing computer systems. That is, modern computer systems employ NAND flash memories, such as SSD or PCIe-attached flash memory, as a padding layer in order to mitigate the performance gap between DRAM and PFS. Indeed, modern supercomputers such as TSUBAME2.0 and Gordon employ flash-based storages, and these NAND flash technologies may also improve checkpoint and restart I/O performance in terms of throughput and latency for various HPC applications. These devices can be applied to storage volumes for checkpoint/restart instead of a traditional PFS; however, the validity of energy efficiency of checkpoint/restart to these devices is not clear. We partially solved the problem [73], but the study does not support multi-level checkpoint/restart.

Appendix A

Merging Edges and Vertices of W(k) state

Although *W(k) state* is described as in Figure A.1, we can simplify the markov model by merging edges and vertices.

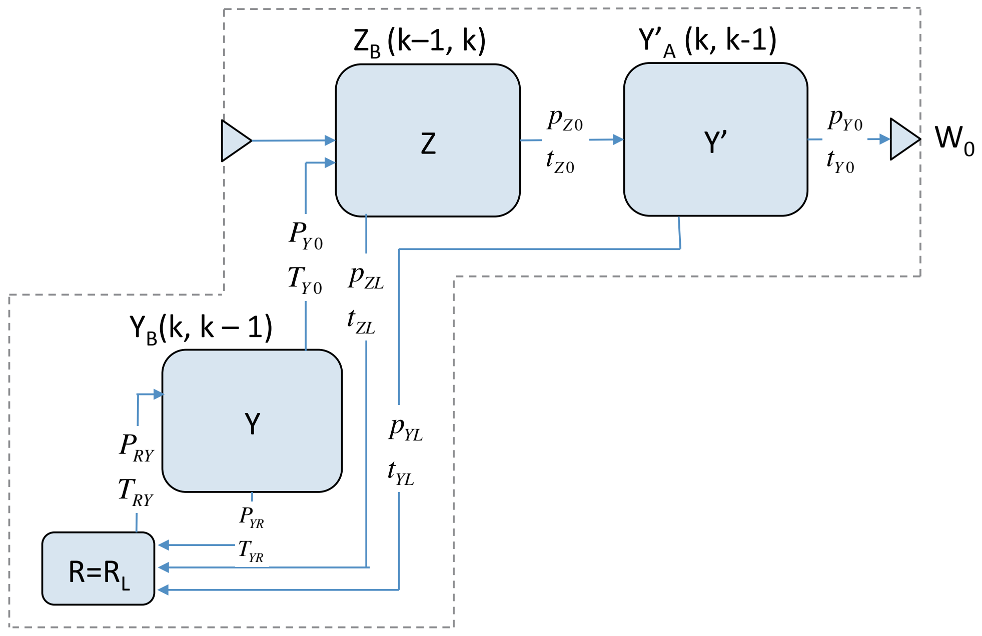


FIGURE A.1: Original $W(k)state$

First, we merge EDGE(P_{RY}, T_{RY}), EDGE(P_{Y0}, T_{Y0}) and VERTEX(Y) into EDGE(P_{RZ}, T_{RZ}) as following.

$$
\begin{aligned}
P_{RZ} &= P_{RY} \times \Big(1 + P_{YR} \times P_{RY} + (P_{YR} \times P_{RY})^2 + \dots\Big) \times P_{Y0} \\
&= \frac{P_{RY} \times P_{Y0}}{1 \quad P_{YR} \times P_{RY}} \\
&= 1 \qquad (\because P_{RY} = 1, P_{Y0} = 1 \quad P_{YR}) \\
T_{RZ} &= P_{RY} \times \Big((P_{YR} \times P_{RY})^0 \times (T_{RY} + 0 \times (T_{YR} + T_{RY}) + t_{Y0}) + \\
&\qquad (P_{YR} \times P_{RY})^1 \times (T_{RY} + 1 \times (T_{YR} + T_{RY}) + t_{Y0}) + \dots\Big) \times P_{Y0} \\
&= P_{RY} \times \left((T_{RY} + t_{Y0}) \times \sum_{i=0}^{\wedge} (P_{YR} \times P_{RY})^i + (T_{YR} + T_{RY}) \times \sum_{i=0}^{\wedge} i \times (P_{YR} \times P_{RY})^i\right) \times P_{Y0} \\
&= P_{RY} \times \left(\frac{T_{RY} + t_{Y0}}{1 \quad P_{YR} \times P_{RY}} + \frac{(T_{YR} + T_{RY}) \times P_{YR} \times P_{RY}}{(1 \quad P_{YR} \times P_{RY})^2}\right) \times P_{Y0}
\end{aligned}
$$

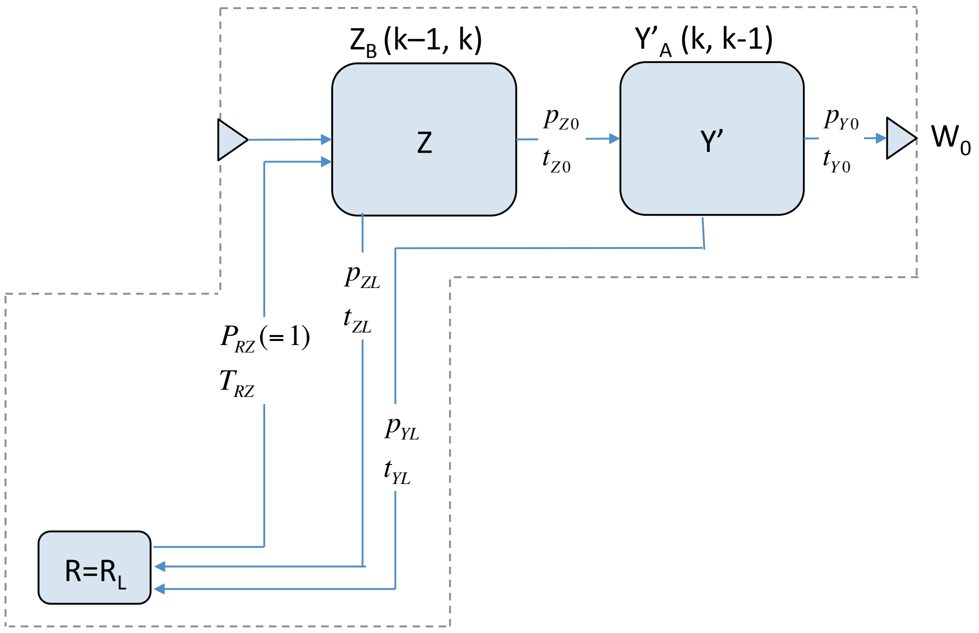


FIGURE A.2: Merging EDGE(P_{RY}, T_{RY}), EDGE(P_{Y0}, T_{Y0}) and VERTEX(Y) into EDGE(P_{RZ}, T_{RZ})

Second, we merge EDGE(p_{ZL}, t_{ZL}) and EDGE(p_{Z0}, T_{Z0}) into EDGE(P_{ZY}, T_{ZY}) as following.

$$
\begin{aligned}
P_{ZY} &= \left(1 + p_{ZL} \times P_{RZ} + (p_{ZL} \times P_{RZ})^2 + \dots\right) \times p_{Z0} \\
&= \frac{p_{Z0}}{1 \quad p_{ZL} \times P_{RZ}} \\
&= 1 \qquad (\because P_{RZ} = 1, p_{Z0} = 1 \quad p_{ZL}) \\
T_{ZY} &= p_{ZL}^0 \times p_{Z0} \times (0 \times (t_{ZL} + T_{RZ}) + t_{Z0}) \\
&\qquad + p_{ZL}^1 \times p_{Z0} \times (1 \times (t_{ZL} + T_{RZ}) + t_{Z0}) + \dots \\
&= p_{Z0} \times \left((t_{ZL} + T_{RZ}) \times \sum_{i=0}^{\wedge} i \times p_{ZL}^i + t_{Z0} \times \sum_{i=0}^{\wedge} p_{ZL}^i \right) \\
&= \frac{(t_{ZL} + T_{RZ}) \times p_{ZL}}{1 \quad p_{ZL}} + t_{Z0}
\end{aligned}
$$

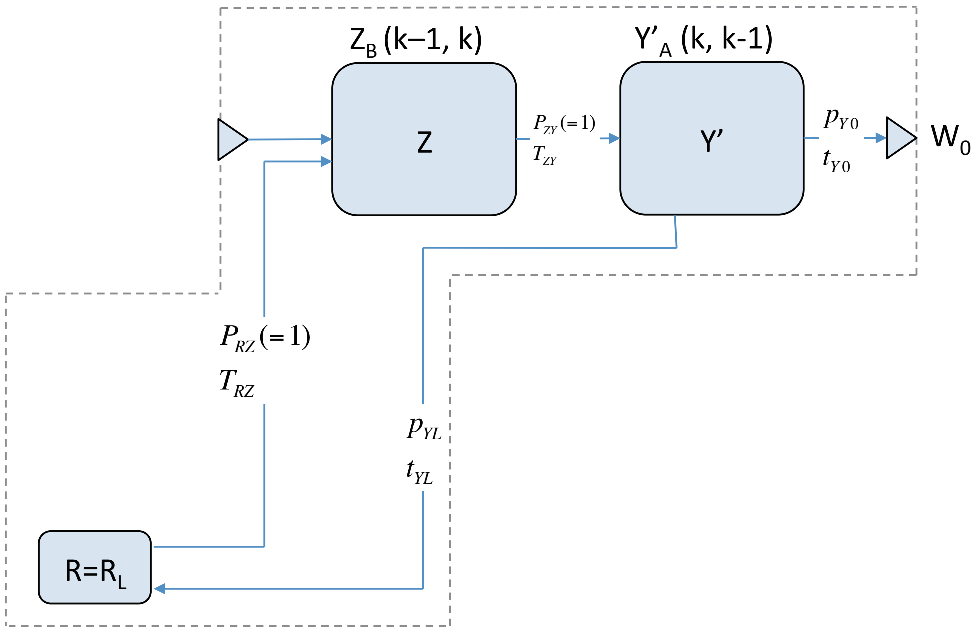


FIGURE A.3: Merging EDGE(p_{ZL}, t_{ZL}) and EDGE(p_{Z0}, T_{Z0}) into EDGE(P_{ZY}, T_{ZY})

Finally, we merge the rest of the other edges and vertices as following.

$$
\begin{aligned}
P_{W0} &= 1 \\
T_{W0} &= p_{YL}^0 \times p_{Y0} \times (T_{ZY} + 0 \times (t_{YL} + T_{RZ} + T_{ZY}) + t_{Y0}) \\
&\qquad + p_{YL}^1 \times p_{Y0} \times (T_{ZY} + 1 \times (t_{YL} + T_{RZ} + T_{ZY}) + t_{Y0}) + \dots \\
&= p_{Y0} \times \left((t_{YL} + T_{RZ} + T_{ZY}) \times \sum_{i=0}^{\wedge} i \times p_{YL}^i + (T_{ZY} + t_{Y0}) \times \sum_{i=0}^{\wedge} p_{YL}^i \right) \\
&= \frac{(t_{YL} + T_{RZ} + T_{ZY}) \times p_{YL}}{1 \quad p_{YL}} + (T_{ZY} + t_{Y0})
\end{aligned}
$$

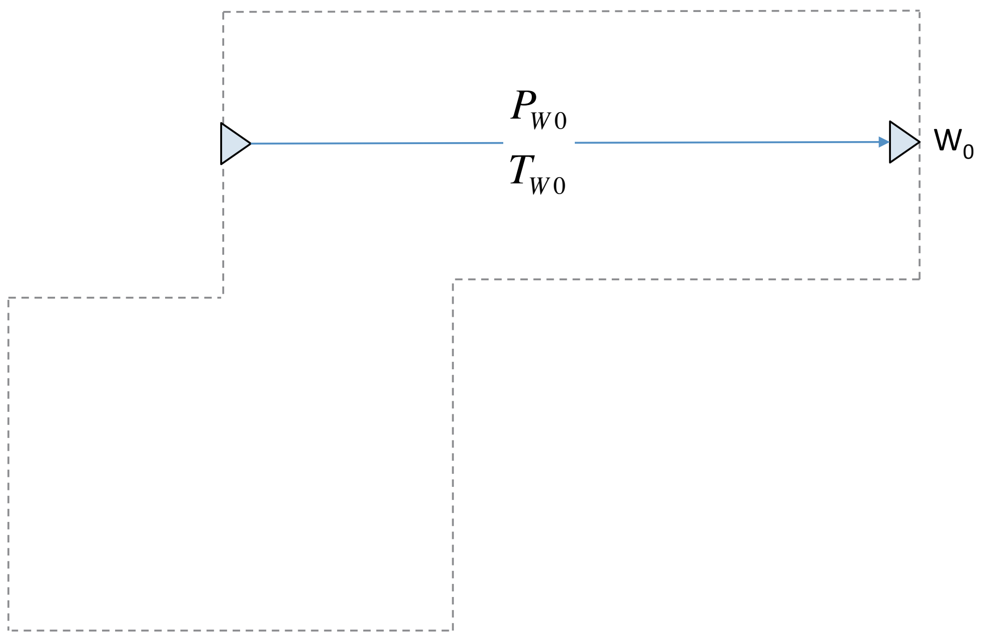


FIGURE A.4: Merging the rest of all edges and vertices

Bibliography

[1] Bianca Schroeder and Garth A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78(1):012022+, July 2007. ISSN 1742-6596. doi: 10.1088/1742-6596/78/1/012022. URL http://dx.doi.org/10.1088/1742-6596/78/1/012022.

[2] Al Geist and Christian Engelmann. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors, 2002.

[3] John Daly et al. Inter-Agency Workshop on HPC Resilience at Extreme Scale, February 2012. URL http://institutes.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf.

[4] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, November 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/sc.2010.18. URL http://dx.doi.org/10.1109/sc.2010.18.

[5] R. L. Berger, C. H. Still, E. A. Williams, and A. B. Langdon. On the Dominant and Subdominant Behavior of Stimulated Raman and Brillouin Scattering Driven by Nonuniform Laser Beams. *Physics of Plasmas*, 5:4337, 1998.

[6] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick

Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1): 3–60, February 2011. ISSN 1094-3420. doi: 10.1177/1094342010391989. URL http://dx.doi.org/10.1177/1094342010391989.

[7] Leonardo Bautista-Gomez, Dimitri Komatitsch, Naoya Maruyama, Seiji Tsuboi, Franck Cappello, and Satoshi Matsuoka. FTI: high performance Fault Tolerance Interface for hybrid systems. In *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WS, USA, 2011.

[8] Ning Liu, Jason Cope, Philip H. Carns, Christopher D. Carothers, Robert B. Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Symposium on Mass Storage Systems and Technologies, MSST 2012*, April 2012.

[9] Dries Kimpe, Kathryn Mohror, Adam Moody, Brian Van Essen, Maya Gokhale, Rob Ross, and Bronis R. de Supinski. Integrated In-System Storage Architecture for High Performance Computing. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '12, 2012.

[10] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4): 374–388, November 2009. ISSN 1094-3420. doi: 10.1177/1094342009347767. URL http://dx.doi.org/10.1177/1094342009347767.

[11] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[12] TSUBAME 2.0 - MONITORING PORTAL. URL http://mon.g.gsic.titech.ac.jp/.

[13] TOP500 Supercomputing Sites. http://www.top500.org/.

[14] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Detailed Modeling, Design, and Evaluation of a Scalable Multi-level Checkpointing System. Technical report, Lawrence Livermore National Laboratory, July 2010.

[15] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD*

international conference on Management of data, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM. ISBN 0-89791-268-3. doi: 10.1145/50202.50214. URL http://dx.doi.org/10.1145/50202.50214.

[16] D Patterson, G Gibson, and R Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, 1988.

[17] William Gropp, Robert Ross, and Neill Miller. Providing Efficient I/O Redundancy in MPI Environments. In *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting*, 2004.

[18] J.S. Plank and Kai Li. Faster checkpointing with n+1 parity. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 288–297, 1994. doi: 10.1109/FTCS.1994.315631.

[19] James S. Plank, Kai Li, and Michael A. Puening. Diskless Checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, October 1998. ISSN 1045-9219. doi: 10.1109/71.730527. URL http://dx.doi.org/10.1109/71.730527.

[20] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):pp. 300–304, 1960. ISSN 03684245. URL http://www.jstor.org/stable/2098968.

[21] J.S. Plank, Youngbae Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 351–360, 1995. doi: 10.1109/FTCS.1995.466964.

[22] Zizhong Chen and J. Dongarra. A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 71–79, 2008. doi: 10.1109/HASE.2008.13.

[23] Leonardo A. Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 63–72. IEEE, May 2010. ISBN 978-1-4244-6987-1. doi: 10.1109/ccgrid.2010.40. URL http://dx.doi.org/10.1109/ccgrid.2010.40.

[24] Heidelberg Berlin. A Parallel Implementation of gzip for modern multi-processor, multi-core machines. URL http://zlib.net/pigz/.

[25] Peter Lindstrom and Martin Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):

1245–1250, September 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.143. URL http://dx.doi.org/10.1109/TVCG.2006.143.

[26] M. Burtscher and P. Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *Computers, IEEE Transactions on*, 58(1):18–31, 2009. ISSN 0018-9340. doi: 10.1109/TC.2008.131.

[27] Lasse Reinhold. QuickLZ, 2008. URL http://www.quicklz.com/.

[28] Dewan Ibtesham, Dorian Arnold, Kurt B. Ferreira, and Patrick G. Bridges. On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 302–311, Berlin, Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-29740-3_34. URL http://dx.doi.org/10.1007/978-3-642-29740-3_34.

[29] Tanzima Z. Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. McrEngine: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. URL http://dl.acm.org/citation.cfm?id=2388996.2389020.

[30] The HDF Group. Hierarchical data format version 5), 2000-2010. URL http://www.hdfgroup.org/HDF5.

[31] R. Rew and G. Davis. NetCDF: An Interface for Scientific Data Access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990. ISSN 0272-1716. doi: 10.1109/38.56302.

[32] 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*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM. ISBN 1-58113-839-3. doi: 10.1145/1006209.1006248. URL http://doi.acm.org/10.1145/1006209.1006248.

[33] Nichamon Naksinehaboon, Yudan Liu, Chokchai (Box) Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott. Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 783–788, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. doi: 10.1109/CCGRID.2008.109. URL http://dx.doi.org/10.1109/CCGRID.2008.109.

[34] Kurt B. Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. Libhashckpt: Hash-based Incremental Checkpointing Using GPU's. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 272–281, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24448-3. URL http://dl.acm.org/citation.cfm?id=2042476.2042507.

[35] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.12. URL http://dx.doi.org/10.1109/SC.2010.12.

[36] B. Randell. System Structure for Software Fault Tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808467. URL http://doi.acm.org/10.1145/800027.808467.

[37] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992. ISSN 0018-9340. doi: 10.1109/12.142678. URL http://dx.doi.org/10.1109/12.142678.

[38] S. Rao, L. Alvisi, and H.M. Vin. Egida: an extensible toolkit for low-overhead fault-tolerance. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 48–55, 1999. doi: 10.1109/FTCS.1999.781033.

[39] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. ISSN 0360-0300. doi: 10.1145/568522.568525. URL http://doi.acm.org/10.1145/568522.568525.

[40] Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J. Dongarra. Correlated Set Coordination in Fault Tolerant Message Logging Protocols. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 51–64, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23396-8. URL http://portal.acm.org/citation.cfm?id=2033415.

[41] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *Proceedings of the 2011 IEEE International*

Parallel & Distributed Processing Symposium, IPDPS '11, pages 989–1000, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/ipdps.2011.95. URL http://dx.doi.org/10.1109/ipdps.2011.95.

[42] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8, 2010. doi: 10.1109/ICCCN.2010.5560143.

[43] Leonardo Bautista Gomez, Thomas Ropars, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Hierarchical Clustering Strategies for Fault Tolerance in Large Scale HPC Systems. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 355–363, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4807-4. doi: 10.1109/CLUSTER.2012.71. URL http://dx.doi.org/10.1109/CLUSTER.2012.71.

[44] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant V. Kalé, and Franck Cappello. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 567–578, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9. URL http://portal.acm.org/citation.cfm?id=2033406.

[45] John W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM*, 17:530–531, September 1974. ISSN 0001-0782. doi: 10.1145/361147.361115. URL http://dx.doi.org/10.1145/361147.361115.

[46] Nitin H. Vaidya. On Checkpoint Latency. Technical report, College Station, TX, USA, 1995. URL http://portal.acm.org/citation.cfm?id=892900.

[47] Nitin H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. *SIGMETRICS Perform. Eval. Rev.*, 23(1):64–73, May 1995. ISSN 0163-5999. doi: 10.1145/223586.223596. URL http://dx.doi.org/10.1145/223586.223596.

[48] Nitin H. Vaidya. Another Two-Level Failure Recovery Scheme. Technical report, College Station, TX, USA, 1994. URL http://portal.acm.org/citation.cfm?id=892923.

[49] Raghunath Rajachandrasekar, Xiangyong Ouyang, Xavier Besseron, Vilobh Meshram, and Dhabaleswar K. Panda. Can Checkpoint/Restart Mechanisms Benefit from Hierarchical Data Staging? In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 312–321, Berlin,

Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-29740-3_35. URL http://dx.doi.org/10.1007/978-3-642-29740-3_35.

[50] Christina M. Patrick, SeungWoo Son, and Mahmut Kandemir. Comparative Evaluation of Overlap Strategies with Study of I/O Overlap in MPI-IO. *SIGOPS Oper. Syst. Rev.*, 42:43–49, October 2008. ISSN 0163-5980. doi: 10.1145/1453775.1453784. URL http://dx.doi.org/10.1145/1453775.1453784.

[51] N. Ali and M. Lauria. Improving the Performance of Remote I/O Using Asynchronous Primitives. pages 218–228. ISSN 1082-8907. doi: 10.1109/hpdc.2006.1652153. URL http://dx.doi.org/10.1109/hpdc.2006.1652153.

[52] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: 10.1145/1551609.1551618. URL http://dx.doi.org/10.1145/1551609.1551618.

[53] Ning Liu, Cope Jason, Carns Philip, Carothers Christopher, Ross Robert, Grider Gary, Crume Adam, and Maltzahn Carlos. On the Role of Burst Buffers in Leadership-class Storage Systems. In *MSST/SNAPI*, April 2012.

[54] Tim Wickberg and Christopher Carothers. The ramdisk storage accelerator: a method of accelerating i/o performance on hpc systems using ramdisks. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '12, pages 5:1–5:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1460-2. doi: 10.1145/2318916.2318922. URL http://doi.acm.org/10.1145/2318916.2318922.

[55] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. Using Active NVRAM for I/O Staging. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, PDAC '11, pages 15–22, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1130-4. doi: 10.1145/2110205.2110209. URL http://doi.acm.org/10.1145/2110205.2110209.

[56] Sudarsun Kannan, Dejan Milojicic, Vanish Talwar, Ada Gavrilovska, Karsten Schwan, and Hasan Abbasi. Using Active NVRAM for Cloud I/O. *Open Cirrus Summit*, 0:32–36, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/OCS.2011.12.

[57] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-41010-4. URL http://portal.acm.org/citation.cfm?id=746632.

[58] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33517-4. doi: 10.1007/978-3-642-33518-1_24. URL http://dx.doi.org/10.1007/978-3-642-33518-1_24.

[59] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *In ICPP 06: Proceedings of the 35th International Conference on Parallel Processing*, pages 471–478. IEEE Computer Society, 2006.

[60] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[61] Chao Huang, Orion Lawlor, and L. V. Kal. Adaptive mpi. In *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[62] Gengbin Zheng, Lixia Shi, and L. V. Kale. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 93–103, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8694-9. URL http://portal.acm.org/citation.cfm?id=1111712.

[63] Scalable_Checkpoint_Restart_Library. Scalable Checkpoint / Restart Library. urlhttp://sourceforge.net/projects/scalablecr/.

[64] IOR HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[65] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362636. URL http://dx.doi.org/10.1145/1362622.1362636.

[66] Lustre. Lustre: A Scalable, High-Performance File System. http://wiki.lustre.org/index.php/Main_Page.

[67] Michael Gerndt, Bernd Mohr, and Jesper Träff. Evaluating OpenMP Performance Analysis Tools with the APART Test Suite. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, chapter 20, pages 155–162. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-22924-7. doi: 10.1007/978-3-540-27866-5_20. URL http://dx.doi.org/10.1007/978-3-540-27866-5_20.

[68] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[69] Ryutaro Himeno. Himeno Benchmark. http://accc.riken.jp/HPC_e/himenobmt_e.html.

[70] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and Modeling of a Non-Blocking Checkpointing System. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Salt Lake City, Utah, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://portal.acm.org/citation.cfm?id=2389022.

[71] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Hitoshi Sato, Shin'ichiro Takizawa, Akihiko Nomura, and Kento Sato. *TSUBAME2.0: The First Petascale Supercomputer in Japan and the Greatest Production in the World*, volume 1, chapter 20, pages 525–556. Chapman & Hall/CRC Computational Science, April 2013. URL http://www.crcnetbase.com/doi/book/10.1201/b14677.

[72] Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. DASH: A Recipe for a Flash-based Data Intensive Supercomputer. *ACM/IEEE conference on Supercomputing*, November 2010.

[73] Takafumi Saito, Kento Sato, Hitoshi Sato, and Satoshi Matsuoka. Energy-Aware I/O Optimization for Checkpoint and Restart on a NAND Flash Memory System. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, FTXS '13, pages 41–48, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1983-6. doi: 10.1145/2465813.2465822. URL http://doi.acm.org/10.1145/2465813.2465822.

[74] Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications. In *Parallel &amp; Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1216–1227. IEEE, May 2012. ISBN 978-1-4673-0975-2. doi: 10.1109/ipdps.2012.111. URL http://dx.doi.org/10.1109/ipdps.2012.111.

[75] MPI Forum. URL http://www.mpi-forum.org/.

[76] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-Forwarding Infrastructure for Petascale Architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 153–162, 2008. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345230.

[77] Rob Ross, Jose Moreira, Kim Cupps, and Wayne Pfeiffer. Parallel I/O on the IBM Blue Gene/L System. Technical report, Blue Gene/L Consortium Quarterly Newsletter, First Quarter, 2006.

[78] 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*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2607-1. doi: 10.1109/DSN.2006.5. URL http://dx.doi.org/10.1109/DSN.2006.5.

[79] Richard L. Graham, Ron Brightwell, Brian Barrett, George Bosilca, and Pjesivac-Grbović. An Evaluation of Open MPI's Matching Transport Layer on the Cray XT. Oct 2007.

[80] PMGR_COLLECTIVE. PMGR_COLLECTIVE. URL http://sourceforge.net/projects/pmgrcollective/.

[81] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama, and Satoshi Matsuoka. Design and Modeling of a Non-Blocking Checkpoint System. In *ATIP - A*CRC Workshop on Accelerator Technologies in High Performance Computing*, May 2012.

[82] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[83] Al Geist and Sudip Dosanjh. Iesp exascale challenge: Co-design of architectures and algorithms. *Int. J. High Perform. Comput. Appl.*, 23(4):401–402, November

2009. ISSN 1094-3420. doi: 10.1177/1094342009347766. URL http://dx.doi.org/10.1177/1094342009347766.