ABSTRACT

WANG, CHAO. Transparent Fault Tolerance for Job Healing in HPC Environments.
(Under the direction of Associate Professor Frank Mueller).

As the number of nodes in high-performance computing environments keeps increasing, faults are becoming common place causing losses in intermediate results of HPC jobs. Furthermore, storage systems providing job input data have been shown to consistently rank as the primary source of system failures leading to data unavailability and job resubmissions.

This dissertation presents a combination of multiple fault tolerance techniques that realize significant advances in fault resilience of HPC jobs. The efforts encompass two broad areas.

First, at the job level, novel, scalable mechanisms are built in support of proactive FT and to significantly enhance reactive FT. The contributions of this dissertation in this area are (1) a transparent job pause mechanism, which allows a job to pause when a process fails and prevents it from having to re-enter the job queue; (2) a proactive fault-tolerant approach that combines process-level live migration with health monitoring to complement reactive with proactive FT and to reduce the number of checkpoints when a majority of the faults can be handled proactively; (3) a novel back migration approach to eliminate load imbalance or bottlenecks caused by migrated tasks; and (4) an incremental checkpointing mechanism, which is combined with full checkpoints to explore the potential of reducing the overhead of checkpointing by performing fewer full checkpoints interspersed with multiple smaller incremental checkpoints.

Second, for the job input data, transparent techniques are provided to improve the reliability, availability and performance of HPC I/O systems. In this area, the dissertation contributes (1) a mechanism for offline job input data reconstruction to ensure availability of job input data and to improve center-wide performance at no cost to job owners; (2) an approach to automatic recover job input data at run-time during failures by recovering staged data from an original source; and (3) "just in time" replication of job input data so as to maximize the use of supercomputer cycles.

Experimental results demonstrate the value of these advanced fault tolerance techniques to increase fault resilience in HPC environments.

Transparent Fault Tolerance for Job Healing in HPC Environments

by
Chao Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Vincent W. Freeh substituted by Dr. Nagiza Samatova	Dr. Yan Solihin
Dr. Frank Mueller Chair of Advisory Committee	Dr. Xiaosong Ma

DEDICATION

To my parents,
my wife, Ying, and our child, William.

BIOGRAPHY

Chao Wang grew up in Jiangsu, China. He received his Bachelor of Science degree in Computer Science from Fudan University at Shanghai, China in 1998, and his Master of Science degree in Computer Science from Tsinghua University at Beijing, China in 2001. He then worked at Lucent Bell-labs as a researcher for two years, after which he worked at Cluster File System Inc. as a senior system engineer for another two years.

Chao came to the North Carolina State University in Fall 2005. With the defense of this dissertation, he will receive a PhD degree in Computer Science from North Carolina State University in Summer 2009.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Frank Mueller, my advisor, for his guidance, support and patience both personally and professionally. I am also indebted to Dr. Xiaosong Ma, Dr. Vincent Freeh, Dr. Yan Solihin and Dr. Nagiza Samatova for serving on my advisory committee.

I would also like to thank Dr. Xiaosong Ma, Dr. Sudharshan Vazhkudai, Dr. Stephen L. Scott and Dr. Christian Engelmann for their guidance in the process of working with the collaborative research projects. Especially, I am obliged to Zhe Zhang for his contributions on the initial work of the offline recovery of job input data, his assistance in the experiments for recovery of job input data and his work on the simulations of fault tolerance techniques for job input data. I also would like to thank Paul H. Hargrove from Lawrence Berkeley National Laboratory for his valuable help on some of the technical details of BLCR.

Finally, I want to thank my wife, Ying Zheng, for her endless support to me through my doctoral studies.

TABLE OF CONTENTS

LIST OF TABLES ..... **ix**

LIST OF FIGURES ..... **x**

1 Introduction **1**
1.1 Background 2
1.1.1 Foundations of HPC and the MPI Standard 2
1.1.2 Multi-level Storage and I/O in HPC Environments 3
1.1.3 Fault Resilience of HPC Jobs 5
1.2 Motivation 5
1.3 Contributions of this Dissertation 7
1.3.1 Reactive and Proactive FT for MPI Jobs 7
1.3.2 FT for Job Input Data 10
1.4 Dissertation Outline 13

2 A Job Pause Service under LAM/MPI+BLCR **14**
2.1 Introduction 14
2.2 Design 16
2.2.1 Group Communication Framework and Fault Detector 17
2.2.2 Internal Schedule Mechanism in LAM/MPI 18
2.2.3 Job-pause 19
2.2.4 Process Migration 20
2.3 Implementation Details 21
2.3.1 Group Communication Framework and Fault Detector 21
2.3.2 Internal Schedule Mechanism in LAM/MPI 21
2.3.3 Job-pause 23
2.3.4 Process Migration 24
2.4 Experimental Framework 25
2.5 Experimental Results 26
2.5.1 Checkpointing Overhead 26
2.5.2 Overhead of Job Pause 30
2.5.3 Membership/Scheduler Performance 31
2.5.4 Job Migration Overhead 31
2.6 Conclusion 33

3 Proactive Process-level Live Migration **35**
3.1 Introduction 35
3.2 Design 37
3.2.1 Live Migration at the Job Level 37

3.2.2 Live Migration at the Process Level . . . 40
3.2.3 Memory Management . . . 42
3.3 Implementation . . . 43
3.3.1 Failure Prediction and Decentralized Scheduler . . . 43
3.3.2 Process-Level Support for Live Migration . . . 44
3.3.3 Job Communication and Coordination Mechanism for Live Migration 47
3.4 Experimental Framework . . . 48
3.5 Experimental Results . . . 49
3.5.1 Migration Overhead . . . 49
3.5.2 Migration Duration . . . 51
3.5.3 Effect of Problem Scaling . . . 53
3.5.4 Effect of Task Scaling . . . 53
3.5.5 Page Access Pattern & Iterative Live Migration . . . 54
3.5.6 Process-Level Live Migration vs. Xen Virtualization Live Migration 55
3.5.7 Proactive FT Complements Reactive FT . . . 56
3.6 Conclusion . . . 57

4 Process-level Back Migration . . . 58
4.1 Introduction . . . 58
4.2 Design . . . 58
4.3 Implementation . . . 59
4.4 Experimental Results . . . 60
4.5 Conclusion . . . 61

5 Hybrid Full/Incremental Checkpoint/Restart . . . 64
5.1 Introduction . . . 64
5.2 Design . . . 65
5.2.1 Scheduler . . . 66
5.2.2 Incremental Checkpointing at the Job Level . . . 66
5.2.3 Incremental Checkpointing at the Process Level . . . 67
5.2.4 Modified Memory Page Management . . . 70
5.2.5 MPI Job Restart from Full+Incremental Checkpoints . . . 70
5.3 Implementation Issues . . . 71
5.3.1 Full/Incremental Checkpointing at the Job Level . . . 71
5.3.2 Full/Incremental Checkpointing at the Process Level . . . 71
5.3.3 Restart from Full+Incremental Checkpoints at Job and Process Levels 72
5.4 Experimental Framework . . . 72
5.5 Experimental Results . . . 73
5.5.1 Checkpointing Overhead . . . 73
5.5.2 Checkpointing File Size . . . 75
5.5.3 Restart Overhead . . . 78
5.5.4 Benefit of Hybrid Full/Incremental C/R Mechanism . . . 79
5.6 Conclusion . . . 81

6 Offline Reconstruction of Job Input Data . . . 83
6.1 Introduction . . . 83
6.2 Architecture . . . 84
6.2.1 Metadata: Recovery Hints . . . 84
6.2.2 Data Reconstruction Architecture . . . 85
6.3 Experimental Setup . . . 87
6.4 Performance of Transparent Input Data Reconstruction . . . 88
6.5 Conclusion . . . 93

7 On-the-fly Recovery of Job Input Data . . . 94
7.1 Introduction . . . 94
7.2 On-the-fly Recovery . . . 96
7.2.1 Architectural Design . . . 97
7.2.2 Automatic Capture of Recovery Metadata . . . 97
7.2.3 Impact on Center and User . . . 98
7.3 Implementation . . . 99
7.3.1 Phase 1: PFS Configuration and Metadata Setup . . . 100
7.3.2 Phase 2: Storage Failure Detection at Compute Nodes . . . 101
7.3.3 Phase 3: Synchronization between Compute and Head Nodes . . . 101
7.3.4 Phase 4: Data Reconstruction . . . 104
7.4 Experimental Framework . . . 105
7.5 Experimental Results . . . 105
7.5.1 Performance of Matrix Multiplication . . . 105
7.5.2 Performance of mpiBLAST . . . 107
7.6 Conclusion . . . 108

8 Temporal Replication of Job Input Data . . . 109
8.1 Introduction . . . 109
8.2 Temporal Replication Design . . . 111
8.2.1 Justification and Design Rationale . . . 112
8.2.2 Delayed Replica Creation . . . 114
8.2.3 Eager Replica Removal . . . 116
8.3 Implementation Issues . . . 116
8.4 Experimental Results . . . 118
8.4.1 Failure Detection and Offline Recovery . . . 118
8.4.2 Online Recovery . . . 119
8.5 Conclusion . . . 121

9 Related Work . . . 122
9.1 Fault Tolerance Techniques for MPI Jobs . . . 122
9.2 Fault Tolerance Techniques for Job Input Data . . . 127

10 Conclusion 130
10.1 Contributions 130
10.2 Future Work 132

Bibliography 135

LIST OF TABLES

Table 1.1 Reliability of HPC Clusters 1

Table 2.1 Size of Checkpoint Files $[MB]$ 28

Table 4.1 Minimal Time Steps (Percentage) Remained to Benefit from Back Migration 61

Table 5.1 Savings by Incremental Checkpoint vs. Overhead on Restart 81

Table 8.1 Configurations of top five supercomputers as of 06/2008 112

LIST OF FIGURES

Figure 1.1 Event timeline of a typical MPI job's data life-cycle 4

Figure 2.1 Pause & Migrate vs. Full Restart 15

Figure 2.2 Job Pause and Migrate Mechanism 16

Figure 2.3 Group Membership and Scheduler 22

Figure 2.4 BLCR with Pause in Bold Frame 23

Figure 2.5 Single Checkpoint on 4 Nodes 27

Figure 2.6 Single Checkpoint on 8 Nodes 27

Figure 2.7 Single Checkpoint on 16 Nodes 28

Figure 2.8 Single Checkpoint Overhead 29

Figure 2.9 Single Checkpoint Time 30

Figure 2.10 Pause and Migrate on 4 Nodes 32

Figure 2.11 Pause and Migrate on 8 Nodes 32

Figure 2.12 Pause and Migrate on 16 Nodes 33

Figure 3.1 Job Live Migration 38

Figure 3.2 Process Migration with Precopy (Kernel Mode in Dotted Frame) 41

Figure 3.3 Process Migration without Precopy (Kernel Mode in Dotted Frame) 42

Figure 3.4 Evaluation with NPB (C-16: Class C on 16 Nodes) 50

Figure 3.5 Page Access Pattern vs. Iterative Live Migration 52

Figure 4.1 NPB Results for Back Migration (C-16: Class C on 16 Nodes) 63

Figure 5.1 Hybrid Full/Incremental C/R Mechanism vs. Full C/R 65

Figure 5.2 Incremental Checkpoint at LAM/MPI 67

Figure 5.3 BLCR with Incremental Checkpoint in Bold Frame 68

Figure 5.4 Structure of Checkpoint Files 69

Figure 5.5 Fast Restart from Full/Incremental Checkpoints 70

Figure 5.6 Full Checkpoint Overhead of NPB Class D and mpiBLAST 74

Figure 5.7 Evaluation with NPB Class C on 4, 8/9, and 16 Nodes 75

Figure 5.8 Evaluation with NPB Class D 76

Figure 5.9 Evaluation with NPB EP Class C/D/E on 4, 8 and 16 nodes 77

Figure 5.10 Evaluation with mpiBLAST 78

Figure 5.11 Savings of Hybrid Full/Incremental C/R Mechanism for NPB and mpiBlast on 16 Nodes 80

Figure 6.1 Cost of finding failed OSTs 89

Figure 6.2 Round-robin striping over 4 OSTs 90

Figure 6.3 Patching costs for single OST failure from a stripe count 32. Filesize/32 is stored on each OST and that is the amount of data patched. Also shown is the cost of staging the entire file again, instead of just patching a single OST worth of data. 90

Figure 6.4 Patching from local NFS. Stripe count increases with file size. One OST fails and its data is patched. 92

Figure 6.5 Patching costs with stripe size 4MB 92

Figure 7.1 Architecture of on-the-fly recovery 98

Figure 7.2 Steps for on-the-fly recovery 100

Figure 7.3 File reconstruction 103

Figure 7.4 Matrix multiplication recovery overhead 104

Figure 7.5 mpiBLAST performance 105

Figure 8.1 Event timeline with ideal and implemented replication intervals 110

Figure 8.2 Per-node memory usage from 300 uniformly sampled time points over a 30-day period based on job logs from the ORNL Jaguar system. For each time point, the total memory usage is the sum of peak memory used by all jobs in question. 114

Figure 8.3 Objects of an original job input file and its replica. A failure occurred to OST1, which caused accesses to the affected object to be redirected to their replicas on OST5, with replica regeneration on OST8. 115

Figure 8.4 Offline replica reconstruction cost with varied file size 119

Figure 8.5 MM recovery overhead vs. replica reconstruction cost 119

Figure 8.6 Recovery overhead of mpiBLAST 121

Chapter 1

Introduction

Recent progress in high-performance computing (HPC) has resulted in remarkable Terascale systems with 10,000s or even 100,000s of processors. At such large counts of compute nodes, faults are becoming common place. Reliability data of contemporary systems illustrates that the mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours depending on the maturity / age of the installation [1]. Table 1.1 presents an excerpt from a recent study by Department of Energy (DOE) researchers that summarizes the reliability of several state-of-the-art supercomputers and distributed computing systems [2, 3]. The most common causes of failure are processor, memory and

Table 1.1: Reliability of HPC Clusters

System	# Cores	MTBF/I	Outage source
ASCI Q	8,192	6.5 hrs	Storage, CPU
ASCI White	8,192	40 hrs	Storage, CPU
PSC Lemieux	3,016	6.5 hrs	
Google	15,000	20 reboots/day	Storage, memory
Jaguar@ORNL	23,416	37.5 hrs	Storage, memory

storage errors / failures. When extrapolating for current systems in such a context, the MTBF for peta-scale systems is predicted to be as short as 1.25 hours [4].

Furthermore, data and input/output (I/O) availability are integral to providing non-stop, continuous computing capabilities to MPI (Message Passing Interface) applications. Table 1.1 indicates that the storage subsystem is consistently one of the primary sources of failure on large supercomputers. This situation is only likely to worsen in the

near future due to the growing relative size of the storage system forced by two trends: (1) disk performance increases slower than that of CPUs and (2) users' data needs grow faster than does the available compute power [5]. A survey of DOE applications suggests that most applications require a sustained 1 GB/sec I/O throughput for every TeraFlop of peak computing performance. Thus, a PetaFlop computer will require 1 TB/sec of I/O bandwidth. The Roadrunner system at the Los Alamos National Laboratory, which was the world's fastest supercomputer on the top500 list in June of 2008 and the first to break the PetaFlop barrier, has 216 I/O nodes attaching 2048 TB of storage space [6]. An I/O subsystem of this size makes a large parallel machine itself and is subject to very frequent failures.

To prevent valuable computation to be lost due to failures, fault tolerance (FT) techniques must be provided to ensure that HPC jobs can make progress in the presence of failures.

1.1 Background

This section provides background information required for the topic of this dissertation. Section 1.1.1 briefly describes the foundations of HPC, as well as the MPI standard that is widely used for applications running in HPC environment. Section 2.2 introduces the storage hierarchies and I/O in most modern HPC systems. Section 2.3 describes fault resilience of HPC jobs and various solutions.

1.1.1 Foundations of HPC and the MPI Standard

The term "high-performance computing (HPC)" is closely related to the term "supercomputing". Though they are sometimes used interchangeably, supercomputing is a more powerful subset of HPC. HPC also includes computer clusters, a group of linked computers, working together as a single compute facility. HPC is widely used to solve advanced computational problems, such as problems involving weather forecasting, climate research, molecular modeling, physical simulations, and so on. Today, computer systems approaching the teraflops-region are counted as HPC-computers.

The TOP500 project [6] ranks the world's 500 most powerful HPC systems. The TOP500 list is updated twice a year, once in June at the International Supercomputer

Conference and again at the IEEE Super Computer Conference in the USA in November. As of June 2009, the fastest heterogeneous machine is the Cell/AMD Opteron-based IBM Roadrunner at the Los Alamos National Laboratory (LANL), which consists of a cluster of 3240 computers, each with 40 processing cores, and includes both AMD and Cell processors. It was announced as the fastest operational supercomputer, with a rate of 1.105 PFLOPS in June 2008.

Special programming techniques are required to exploit the parallel architectures of HPC computers. MPI is one such techniques and widely used in HPC environments. MPI targets high performance, scalability, and portability, which made it the de facto standard for jobs running on HPC systems, and it remains the dominant model used in HPC today. Currently, there are two popular versions of the standard: MPI-1 [7] (which emphasizes message passing and has a static runtime environment), and MPI-2 [8] (which includes new features, such as parallel I/O, dynamic process management and remote memory operations). Furthermore, the MPI forum is currently working on MPI-3.

There are numerous implementations of MPI, such as MPICH [9] from Argonne National Laboratory and Mississippi State University and LAM/MPI [10] from the Ohio Supercomputing Center. LAM/MPI and a number of other MPI efforts recently merged to form a combined project, Open MPI [11]. Furthermore, OpenMP [12] is another model for parallel programming and used for tightly coordinated shared memory machines while MPI is commonly used for loosely connected clusters. OpenMP divides a task and parallelizes it among different threads, which run concurrently on different processors/cores. MPI and OpenMP can be used in conjunction with each other to write hybrid code.

1.1.2 Multi-level Storage and I/O in HPC Environments

Figure 1.1 gives an overview of an event timeline describing a typical supercomputing job's data life-cycle. Users stage their job input data from elsewhere to the scratch space, submit their jobs using a batch script, and offload the output files to archival systems or local clusters.

Here, we can divide the storage system into four levels:

1. **Primary storage** is the memory of the compute nodes. Main memory is directly or indirectly connected to the CPU of compute node via a memory bus. The I/O device

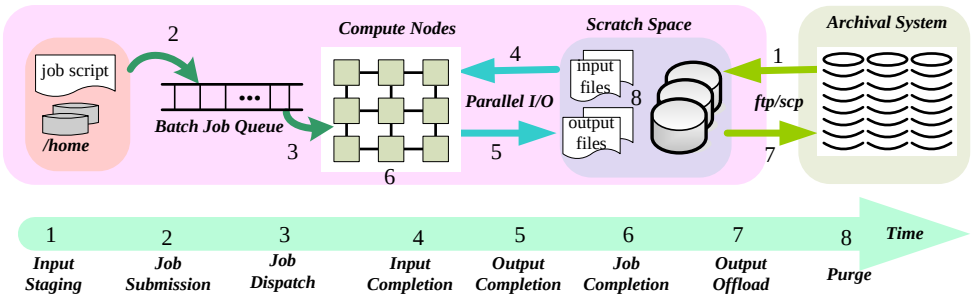


Figure 1.1: Event timeline of a typical MPI job's data life-cycle

at this level is actually comprised of two memory buses: an address bus and a data bus.

2. **Secondary storage** is the local disk of compute node. It is not directly accessible by the CPU. I/O channels are used by the compute node to access secondary storage. However, some HPC systems, such as BG/L, have no local disk.

3. **The scratch space** of an HPC system is a storage subsystem. It is managed by a parallel file system, such as Lustre, and is connected to the compute node (clients of the PFS) through I/O nodes and network connections.

4. **The archival system**, also known as disconnected storage, is used to backup data. The archival system increases general information security, since it is unaffected by the failure of compute nodes and cannot be affected by computer-based security attacks. Also, archival systems are less expensive than scratch space storage. The I/O between the compute systems and the archival systems is typically through network transfer protocols, such as ftp and scp, intervened by human beings or system schedulers through job scripts.

HPC systems consume and produce massive amounts of data in a short time, which turns the compute-bound problems into an I/O-bound problem. However, HPC storage systems are becoming severe bottlenecks in current extreme-scale supercomputers in both reliability and performance. First of all, storage systems consistently rank as the primary source of system failures, as depicted in Table 1.1. Second, I/O performance is lagging

behind. Much work on the multi-level storage and I/O in HPC environments is needed to ensure that the data can be transferred, stored and retrieved quickly and correctly.

1.1.3 Fault Resilience of HPC Jobs

The size of the world's fastest HPC systems has increased from current tera-scale to next-generation peta-scale. Failures are likely to be more frequent as the system reliability decreases with the substantial growth in system scale. This poses a challenge for HPC jobs with respect to fault tolerance and resilience. However, even though it is widely used for parallel applications running on HPC system, MPI does not explicitly support fault tolerance.

Resilience mechanisms include fault prevention, detection and recovery, both reactively and proactively. At the system level, many approaches have been developed for fault resilience, such as RAID [13], node/disk failover, and so on. At the job/application level, some techniques, such as C/R, are deployed. The MPI forum is also working on FT support in the MPI-3 standard. In this dissertation, we develop FT techniques at the job level, including both process-level self-healing and job input data recovery.

1.2 Motivation

In today's large-scale HPC environment, checkpoint/restart (C/R) mechanisms are frequently deployed that periodically checkpoint the entire process image of all MPI tasks. The wall-clock time of a 100-hour job could well increase to 251 hours due to the C/R overhead of contemporary fault tolerant techniques implying that 60% of cycles are spent on C/R alone [4]. However, only a subset of the process image changes between checkpoints. In particular, large matrices that are only read but never written, which are common in HPC codes, do not have to be checkpointed repeatedly. Coordinated checkpointing for MPI jobs, a commonly deployed technique, requires all the MPI tasks to save their checkpoint files at the same time, which leads to an extremely high I/O bandwidth demand. This C/R mechanism also requires a complete restart upon a single node failure even if all the other nodes are still alive. These problems are projected to increase as HPC has entered the Petascale era. They require to be addressed through (1) reducing size of the checkpoint file and overhead of the checkpoint operations and (2) avoiding a complete restart and retaining

execution of MPI jobs as nodes fail.

Meanwhile, these frequently deployed techniques to tolerate faults focus on reactive schemes for recovery and generally rely on a simple C/R mechanism. Yet, they often do not scale due to massive I/O bandwidth requirements of checkpointing, uninformed checkpoint placements, and selection of sub-optimal checkpoint intervals. They also require that a new job be submitted after a failure. Yet, some node failures can be anticipated by detecting a deteriorating health status in todays systems. This knowledge can be exploited by proactive FT to complement reactive FT such that checkpoint requirements are relaxed. Thus, to develop novel, scalable mechanisms in support of proactive FT and to significantly enhance reactive FT is not only possible but also necessary.

Furthermore, jobs generally need to read input data. Jobs are interrupted or rerun if job input data is unavailable or lost. However, storage systems providing job input data have been shown to consistently rank as the primary source of system failures, according to logs from large-scale parallel computers and commercial data centers [2]. This trend is only expected to continue as individual disk bandwidth grows much slower than the overall supercomputer capacity. Therefore, the number of disk drives used in a supercomputer will need to increase faster than the overall system size. It is predicted that by 2018, a system at the top of the top500.org chart will have more than 800,000 disk drives with around 25,000 disk failures per year [14].

Thus, coping with failures for job input data is a key issue as we scale to Peta- and Exa-flop supercomputers.

Hypothesis: The hypothesis of this dissertation is as follows:

> *By employing a combination of multiple fault tolerance techniques, significant advances in fault resilience of HPC jobs can be realized through (1) graceful recovery from faults through dynamic reassignment of work to spare nodes and (2) input data recovery while jobs are queued or even executing.*

In this dissertation, we demonstrate FT techniques in two broad areas. First, at the job level, we build novel, scalable mechanisms in support of proactive FT and to significantly enhance reactive FT.

Second, for the job input data, we provide offline recovery, online recovery, and temporal replication to improve the reliability, availability and performance of HPC I/O

systems.

1.3 Contributions of this Dissertation

In the following, we present contributions in detail for both of the directions.

1.3.1 Reactive and Proactive FT for MPI Jobs

The first objective of this work is to alleviate limitations of current reactive FT schemes through the following contributions:

1. We provide a transparent job pause mechanism within LAM (Local Area Multicomputer)/MPI+BLCR (Berkeley Labs C/R). This mechanism allows jobs to pause when a process fails, with the failing process being restarted on another processor. This prevents the job from having to re-enter the queue, i.e., the job simply continues its execution from the last checkpoint.

2. An approach that combines process-level live migration with healthy monitoring was provided for proactive fault-tolerant in HPC environments. It is imperative to have such a proactive fault-tolerant middleware in that as the number nodes in a HPC system increases the failure probability of the system increases and commonly deployed reactive fault-tolerant techniques do not scale well due to their massive I/O requirement. However, the intent is not to replace reactive fault-tolerant mechanisms, such as checkpointing, but to complement them (using checkpointing as the last resort). By complementing reactive with proactive FT at the process level, the approach also reduces the number of checkpoints when a majority of the faults can be handled proactively.

3. Our work also provides a novel back migration approach to eliminate load imbalance or bottlenecks caused by migrated tasks. Experiments indicate that the larger the amount of outstanding execution, the higher the benefit due to back migration will be.

4. We develop an incremental checkpointing mechanism and combine it with full checkpoint. We explore the possibility of reducing the overhead of checkpointing by periodically performing full checkpoints, separated by several incremental checkpoints. The

results show that the performance of the hybrid full/incremental C/R mechanism is significantly lower than that of the original full mechanism.

Job Pause Service

C/R has become a requirement for long-running jobs in large-scale clusters due to a mean-time-to-failure (MTTF) in the order of hours. After a failure, C/R mechanisms generally require a complete restart of an MPI job from the last checkpoint. A complete restart, however, is unnecessary since all nodes but one are typically still alive. Furthermore, a restart may result in lengthy job requeuing even though the original job had not exceeded its time quantum.

We overcome these shortcomings. Instead of job restart, we have developed a transparent mechanism for job pause within LAM/MPI+BLCR. This mechanism allows live nodes to remain active and roll back to the last checkpoint while failed nodes are dynamically replaced by spares before resuming from the last checkpoint. Our methodology includes LAM/MPI enhancements in support of scalable group communication with fluctuating number of nodes, reuse of network connections, transparent coordinated checkpoint scheduling and a BLCR enhancement for job pause. Experiments in a cluster with the NAS (NASA Advanced Supercomputing) Parallel Benchmark suite show that our overhead for job pause is comparable to that of a complete job restart. Yet, our approach alleviates the need to reboot the LAM run-time environment, which accounts for considerable overhead resulting in net savings of our scheme in the experiments. Our solution further provides full transparency and automation with the additional benefit of reusing existing resources. Executing continues after failures within the scheduled job, *i.e.*, the application staging overhead is not incurred again in contrast to a restart. Our scheme offers additional potential for savings through incremental checkpointing and proactive diskless live migration.

Proactive Live Migration

As the number of nodes in HPC environments keeps increasing, faults are becoming common place. Reactive FT often does not scale due to massive I/O requirements and relies on manual job resubmission.

This work complements reactive with proactive FT at the process level. Through health monitoring, a subset of node failures can be anticipated when one's health deteriorates. A novel process-level live migration mechanism supports continued execution of applications during much of processes migration. This scheme is integrated into an MPI execution environment to transparently sustain health-inflicted node failures, which eradicates the need to restart and requeue MPI jobs.

The objective is to reduce the aggregated downtimes over all nodes. It is accomplished by avoiding roll-back (restarting the entire application from the last checkpoint) by engaging in preventive migration. Such migration moves a process from one node to an other while maintaining the consistency of the global state of the entire MPI job. Health monitoring techniques, such as supported by the Baseboard Management Controller (BMC) and the Intelligent Platform Management Interface (IPMI), are used to anticipate node failures. A deteriorating health of a node triggers live process-level migration, which allows execution to continue while a process image is incrementally and asynchronously transferred to a spare node as an enhancement to the Linux BLCR.

Back Migration

After an MPI task is migrated, the performance of this and other MPI tasks may be affected due to dependence on the interconnect topology, even in a homogeneous cluster. It has been shown that task placement in large-scale interconnects can have a significant impact on performance for certain communication patterns. Communication with neighbors one to three hops away may deteriorate after migration if the distance increases to k hops since a spare node is often at the boundary of an interconnect volume. The additional cost in latency may result in imbalance at collectives, where the spare node incurs the most overhead. In such an environment, migrating computation back to the failed node once it has recovered can improve overall performance.

We design a strategy to assess the benefit of back-migration at the process level that dynamically decides if it is beneficial to re-locate a migrated task running on a spare node back on the original node once the node has recovered if such relocation has the potential to increase performance. We implement a back migration mechanism within LAM/MPI and BLCR based on process-level live migration in reverse direction.

Hybrid Full/Incremental Checkpoint/Restart

Checkpointing addresses faults in HPC environments but captures full process images even though only a subset of the process image changes between checkpoints.

We design a high-performance hybrid disk-based full/incremental checkpointing technique for MPI tasks to capture only data changed since the last checkpoint. Our implementation integrates new BLCR and LAM/MPI features that complement traditional full checkpoints. This results in significantly reduced checkpoint sizes and overheads with only moderate increases in restart overhead. After accounting for cost and savings, benefits due to incremental checkpoints significantly outweigh the loss on restart operations.

The savings due to replacing full checkpoints with incremental ones increase with the frequency of incremental checkpoints. Overall, our novel hybrid full/incremental checkpointing is superior to prior non-hybrid techniques.

1.3.2 FT for Job Input Data

The second objective of this work is to improve the reliability, availability and performance of the job input data through the following contributions:

1. We explore a mechanism of offline job input data reconstruction to ensure availability of job input data and to improve center-wide performance. With data source information transparently extracted from the staging request, our framework allows jobs to be scheduled even when parts of their prestaged input data are unavailable due to storage system failures. This is accomplished through transparent data reconstruction at no cost to job owners.

2. We present an approach to automatic recover job input data during failures by recovering staged data from an original source. The proposed mechanism captures I/O failures (e.g., due to storage node failures) while a running job is accessing its input file(s) and retrieves portions of the input file that are stored on the failed nodes. We further deploy this technique in current high performance computing systems to recover the input data of applications at run-time. The implementation of the mechanism is realized within the Lustre parallel file system. An evaluation of the system using real machines is also presented.

3. We propose "just in time" replication of job input data so as to maximize the use of supercomputer cycles. The mechanism adds selective redundancy to job input data by synergistically combining the parallel file system with the batch job scheduler to perform temporal replication transparently. These replicas will be used for fast data recovery during a job execution and are subsequently removed when the job completes or finishes consuming the input data.

   We show that the overall space and I/O bandwidth overhead of temporal replication is a reasonable small fraction of the total scratch space on modern machines, which can be further improved by optimizations to shorten the replication duration.

Offline Reconstruction of Job Input Data

Our target environment is that of shared, large, supercomputing centers. In this setting, vast amounts of *transient* job data routinely passes through the scratch space, hosted on parallel file systems. Supercomputing jobs, mostly parallel time-step numerical simulations, typically initialize their computation with a significant amount of staged input data.

We propose *offline data reconstruction* that transparently verifies the availability of the staged job input data and fetches unavailable pieces from external data sources in case of storage failures. Our approach takes advantage of the existence of an external data source/sink and the immutable nature of job input data. Collectively, from a center standpoint, these techniques globally optimize resource usage and increase data and service availability. From a user job standpoint, they reduce job turnaround time and optimize the usage of allocated time.

Periodic data availability checks and transparent data reconstruction is performed to protect the staged data against storage system failures. Compared to existing approaches, this technique incurs minimum user operational cost, low storage usage cost, and no computation time cost. It also reduces the average job waiting time of the entire system and increases overall center throughput and service.

We implement offline recovery schemes into parallel file systems so that applications can seamlessly utilize available storage elements.

On-the-fly Recovery of Job Input Data

Storage system failure is a serious concern as we approach Petascale computing. Even at today's sub-Petascale levels, I/O failure is the leading cause of downtimes and job failures.

We contribute a novel, on-the-fly recovery framework for job input data into supercomputer parallel file systems. The framework exploits key traits of the HPC I/O workload to reconstruct lost input data during job execution from remote, immutable copies. Each reconstructed data stripe is made immediately accessible in the client request order due to the delayed metadata update and fine-granular locking while unrelated accesses to the same file remain unaffected.

We implement the recovery component within the Lustre parallel file system, thus building a novel application-transparent online recovery solution. Our solution is integrated into Lustre's two-level locking scheme using a two-phase blocking protocol.

Temporal Replication of Job Input Data

Storage systems in supercomputers are a major reason for service interruptions. RAID (Redundant Arrays of Inexpensive Disks) solutions alone cannot provide sufficient protection as 1) growing average disk recovery times make RAID groups increasingly vulnerable to disk failures during reconstruction, and 2) RAID does not help with higher-level faults such as failed I/O nodes.

We present a complementary approach based on the observation that files in the supercomputer scratch space are typically accessed by batch jobs whose execution can be anticipated. Therefore, we propose to transparently, selectively, and temporarily replicate "active" job input data by coordinating parallel file system activities with those of the batch job scheduler. We implement a temporal replication scheme in the popular Lustre parallel file system and evaluate it with real-cluster experiments. Our results show that the scheme allows for fast online data reconstruction with a reasonably low overall space and I/O bandwidth overhead.

1.4 Dissertation Outline

The remaining chapters are structured as follows. Chapters 2, 3, 4 and 5 present the reactive FT and proactive FT approaches for MPI jobs, i.e., the job pause service, proactive live migration, back migration and hybrid full/incremental C/R, respectively. We describe in detail the offline reconstruction in Chapter 6, the on-the-fly recovery in Chapter 7 and the temporal replication in Chapter 8. Chapter 9 discusses the related work and Chapter 10 summarizes the contributions of this research and discussed possibilities for future work.

Chapter 2

A Job Pause Service under LAM/MPI+BLCR

2.1 Introduction

To prevent valuable computation to be lost due to failures, C/R has become a requirement for long-running jobs. Current C/R mechanisms commonly allow checkpoints to be written to a global file system so that in case of failure the entire MPI job can be restarted from the last checkpoint. One example of such a solution is LAM/MPI's C/R support [10] through BLCR [15]. A complete restart, however, is unnecessary since all but one node are typically still alive.

The contribution of this chapter is to avoid a complete restart and retain execution of MPI jobs as nodes fail, as depicted in Figure 2.1. As such, we pause the MPI processes on live nodes and migrate the MPI processes of failed nodes onto spare nodes. We have developed a transparent mechanism as an extension to LAM/MPI+BLCR that reuses operational nodes within an MPI job. Functional nodes are rolled back to the last checkpoint, retaining internal communication links within LAM/MPI+BLCR, before the corresponding processes are paused. Meanwhile, a failed node is replaced with a spare node where the corresponding MPI task is recovered from the last checkpoint. Hence, live nodes remain active while failed nodes are dynamically and transparently replaced. This solution removes any requeuing overhead by reuses existing resources in a seamless and transparent manner.

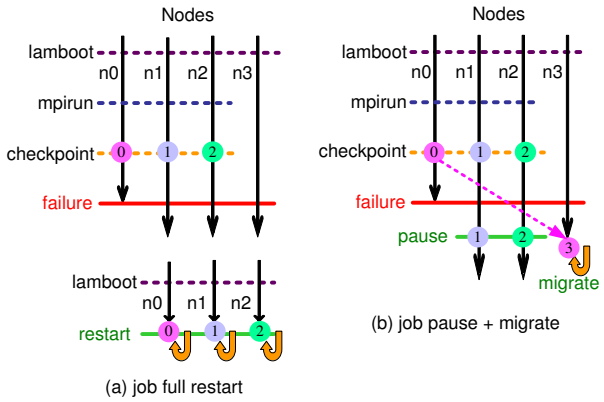


Figure 2.1: Pause & Migrate vs. Full Restart

Our solution comprises several areas of innovation within LAM/MPI and BLCR: (1) *crtcp*, one of the Request Progression Interface (RPI) options [16] of LAM/MPI, is enhanced to reuse the network connections between live nodes upon the faults. (2) Lamd (LAM daemon) is complemented with a scalable group communication framework based on our prior work [17], which notifies the Lamd of live nodes about replacement nodes. (3) Lamd is supplemented with a novel scheduler that transparently controls periodic checkpointing and triggers migration upon node failures while ensuring that live nodes are paused, which prevents LAM/MPI from prematurely terminating a job. (4) BLCR is supplemented with a job pause/restart mechanism, which supports multi-threading.

We have conducted a set of experiments on a 16-node dual-processor (each dual core) Opteron cluster. We assess the viability of our approach using the NAS Parallel Benchmark suite. Experimental results show that the overhead of our job pause mechanism is comparable to that of a complete job restart, albeit at the added benefits of full transparency and automation combined with the reuse of existing resources in our case. Hence, our approach avoids requeuing overhead by letting the scheduled job tolerate the fault so that it can continue to execute. We are furthermore investigating additional benefits of our scheme for incremental checkpointing and proactive diskless live migration.

The remainder of this chapter is structured as follows. Section 2.2 presents the design of our transparent fault-tolerance mechanism. Section 2.3 identifies and describes the implementation details. Subsequently, the experimental framework is detailed and mea-

surements for our experiments are presented in Sections 2.4 and 2.5, respectively. Finally, the work is summarized in Section 2.6.

2.2 Design

This section presents an overview of the design of transparent fault tolerance with LAM/MPI+BLCR. The approach extends the C/R framework of LAM/MPI with an integrated group communication framework and a fault detector, an internal schedule mechanism (Figure 2.3), the job pause mechanism and actual process migration (Figure 2.2). As a result, BLCR is supplemented with the new capability of *cr_pause*, the transparent job-pause functionality.

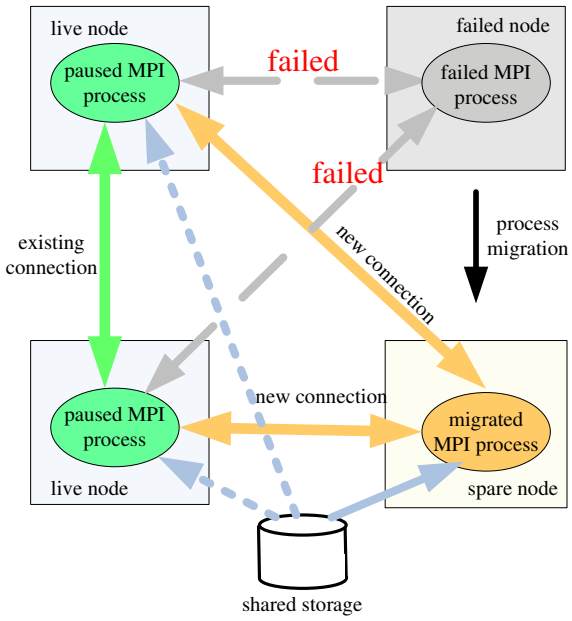


Figure 2.2: Job Pause and Migrate Mechanism

In the following, the design of the protocol is given. As depicted in Figure 2.3, the scheduler daemon acts as a coordination point between the *membership daemon* and *mpirun*, the initial LAM/MPI process at job invocation. The main logical steps can be summarized as follows. 1) The membership daemon monitors the system and notifies the scheduler daemon upon the faults. 2) The scheduler daemon coordinates the job pause for

functional nodes and process migration for failed ones. 3) The nodes perform the actual pause and migration work, as depicted in Figure 2.2. 4) All active processes (MPI tasks) continue the MPI job.

1. **membership daemon** and **scheduler daemon**: initialization through *lamboot*;
2. **membership daemon**: maintains the group membership information and monitors for faults;
3. **membership daemon**: notifies the *scheduler* upon detecting a fault;
4. **scheduler daemon**: selects replacement nodes for the failed ones and notifies the respective *mpirun* processes governed by the LAM runtime environment;
5. **each notified mpirun**: propagates the job-pause request to each MPI process on live nodes and sends a process migration command to the replacement nodes;
6. **each MPI process on live nodes**: engages in job-pause; meanwhile,
7. **replacement nodes**: restart from the checkpoint file to migrate the process;
8. **each MPI process on live nodes/replacement nodes (migrated)**: continues/resumes execution.

The implementation details of this algorithm with respect to LAM/MPI are given in the next section.

2.2.1 Group Communication Framework and Fault Detector

In our prior work [17], we devised a scalable approach to reconfigure the communication infrastructure after node failures within the runtime system of the communication layer. A decentralized protocol maintains membership of nodes in the presence of faults. Instead of seconds for reconfiguration, our protocol shows overheads in the order of hundreds of microseconds and single-digit milliseconds over MPI on BlueGene/L nodes (BG/L) with up to 1024 processors.

We complemented lamd, the low-level communication daemon of LAM/MPI running on each node, with this scalable group communication framework from our prior work. We further designed a fault detector based on a single timeout mechanism. Excessive delay in response from any process to a message request is assumed to indicate process (node) failure. In our framework, link failures are handled similarly to node failures since the cause of a long communication delay is not distinguished, *i.e.*, different causes of failure can be

handled uniformly (and will be uniformly referred to as "node failures" or simply "failures"). When the detector determines a failure, it triggers the scheduler through a message containing the ID of the failed node.

2.2.2 Internal Schedule Mechanism in LAM/MPI

We next describe the mechanism to control a job's schedule upon node failure. This schedule mechanism is rooted within the LAM runtime environment, independent of the system-level batch scheduler governing job submission, such as Torque/OpenPBS (Portable Batch System). The protocol for the internal scheduler is as follows: It

- launches periodic checkpoint commands at a user-specified frequency;
- determines replacement nodes for failed nodes when a fault occurs; and
- launches job pause commands to *mpirun*s.

The mechanism has been implemented under the following design principles: The scheduler

- is integrated into the run-time environment;
- is decentralized, without any single point of failure; and
- provides good scalability due to the underlying group communication framework and its own functional layer on top.

The scheduler daemon is the main part of the schedule mechanism. In addition, *mpirun* also ties in to the schedule framework, most notably through the propagation of job pause commands.

In our current implementation, the schedule mechanism stores the information about the MPI job (such as *mpirun-pid@node-id*, *mpi-process-pid@node-id*, etc.) on a reliable shared storage or, in an alternative design, keeps them in memory. Both approaches have pros and cons. Nonetheless, the overhead of maintaining and utilizing such information is in the order of microsecond, regardless of the storage location. Hence, compared to overhead for C/R of up to tens of seconds, this bookkeeping overhead of the scheduler is insignificant. In our implementation, *mpirun* is responsible for logging the information about the MPI job on shared storage or in memory after a successful launch of the job. During finalization of the job, *mpirun* will disassociate this information.

2.2.3 Job-pause

As depicted in Figure 2.2, upon node failure, the job-pause mechanism allows the processes on live nodes to remain active by rolling back computation to the last checkpoint (rather than necessitating the traditional complete cold restart of an MPI job under LAM/MPI, which would incur long wait times in the job submission queues). At the LAM level, job-pause reuses the existing connections among the processes. Furthermore, at the C/R level (using BLCR), part of the state of the process is restored from the checkpoint state instead of a complete restart of the process. Currently, we use the existing process and its threads without forking or cloning new ones. Furthermore, we do not need to restore the parent/child relationships (in contrast to *cr_restart*), but we still restore shared resource information (*e.g.*, mmaps and files). LAM uses the module of *crtcp* to maintain the TCP connections (sockets) among the MPI processes.

BLCR restores the state of the MPI processes (*i.e.*, the sockets are kept open by the *crtcp* module during the *cr_pause*). Hence, we can safely reuse the existing connections among the processes. Even though BLCR does not support transparent C/R of socket connections, this does not adversely affect us here since LAM/MPI relies on communication *via crtcp* rather than through lower-level network sockets.

A pause of the MPI job process is initiated by *mpirun*. In response, the following sequence of events occurs:

1. **mpirun**: propagates the job-pause request to each MPI process on live nodes;
2. **BLCR on live nodes**: invokes the *cr_pause* mechanism;
3. **paused process**: waits for *mpirun* to supply the information about the migrated process;
4. **mpirun**: updates the global list with information about the migrated process and broadcasts it to all processes;
5. **paused process**: receives information about the migrated process from *mpirun*;
6. **paused process**: builds its communication channels with the migrated process;
7. **paused process**: resumes execution from the restored state.

Similar to the *checkpoint* and *restart* functionality of BLCR, the novel *pause* mechanism of BLCR also interacts with LAM through a threaded callback function. The callback is provided as part of our LAM enhancements and registered at the initialization of *crtcp*. The *pause* mechanism performs an ioctl() call to enter the pause state. As part of the pause

functionality, a process rolls back its state to that of a former checkpoint, typically stored on disk. This rollback is the inverse of checkpointing, *i.e.*, it restarts a process at the saved state. However, there are consequential differences. Pause/rollback reuses the existing process without forking a new one. Furthermore, existing threads in the process are reused. Only if insufficient threads exist will additional ones be created (cloned, in Linux terms). Hence, it becomes unnecessary to restore the Process ID (PID) information or re-create parent/child relationships from the checkpoint data.

2.2.4 Process Migration

When the scheduler daemon receives a node failure message from the membership daemon, it performs a migration to transfer processes, both at application and runtime level, from the failed node to the replacement nodes. This happens in a coordinated fashion between the *mpirun* processes and new processes launched at the replacement nodes,

Several issues need to be solved here: First, *mpirun* launches a *cr_restart* command on appropriate nodes with the relevant checkpoint image files. In our system, the scheduler daemon determines the most lightly loaded node as a migration target, renames the checkpoint file to reflect the change and then notifies *mpirun* to launch *cr_restart* from the relevant node with the right checkpoint file.

Second, the checkpoint files of all processes have to be accessible for replacement nodes in the system. This ensures that, at the fault time, the process can be migrated to any node in the system using the checkpoint file. We assume a shared storage infrastructure for this purpose.

Finally, knowledge about the new location of the migrated process has to be communicated to all other processes in the application. Since we operate within the LAM runtime environment, a node ID (instead of a node's IP address) is used for addressing information. Thus, migration within the LAM runtime environment becomes transparent, independent of external system protocols, such as Network Address Translation (NAT), firewalls, etc. Our system also updates the addressing information on-the-fly instead of scanning and updating all the checkpoint files, thereby avoiding additional disk access overhead for writes.

At the point of process migration, the following sequence of events occurs:

1. **mpirun**: sends a process migration command (*cr_restart*) to the replacement node;

2. **BLCR on the replacement node**: executes the *cr_restart* mechanism referencing the checkpoint file on shared storage;
3. **restarted process**: sends its new process information to *mpirun*;
4. **mpirun**: updates the global list with information about the migrated process and broadcasts it to all processes;
5. **restarted process**: builds its communication channels with all the other processes;
6. **restarted process**: resumes execution from the saved state.

Since all the information is updated at run-time, the normal *restart* operation of BLCR is executed from the replacement node without any modification.

2.3 Implementation Details

Our fault tolerance architecture is currently implemented with LAM/MPI and BLCR. Our components are implemented as separate modules to facilitate their integration into the run-time environment of arbitrary MPI implementations. Its design and implementation allows adaptation of this architecture to other implementations of MPI, such as MPICH (MPI Chameleon) [9] and OpenMPI [11].

2.3.1 Group Communication Framework and Fault Detector

Figure 2.3 depicts the framework for group communication implemented as a process of the lam daemon (lamd). The so-called membership daemons in the LAM universe communicate with each other and with the scheduler daemons through out-of-band communication channel provided by the LAM runtime environment.

2.3.2 Internal Schedule Mechanism in LAM/MPI

As main part of the schedule mechanism, the scheduler daemon is also implemented as a process of lamd communicating through the out-of-band communication channel, just as the membership daemon does (Figure 2.3).

When the scheduler daemon receives a failure message from the membership daemon, it consults the database to retrieve information about the MPI job and the nodes in the LAM universe. Based on this information, the most lightly loaded node is chosen as a migration target. Alternate selection policies to determine a replacement node can be

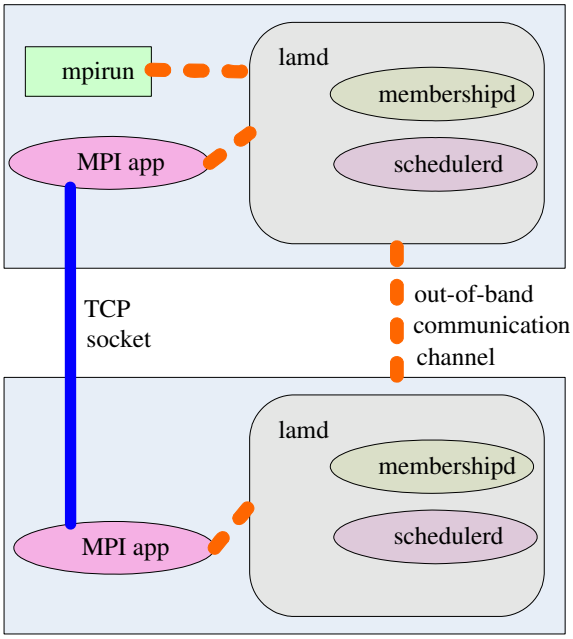


Figure 2.3: Group Membership and Scheduler

readily plugged in. Next, a job pause command is issued to *mpirun* through the scheduler daemon at the node on which *mpirun* is launched.

Triggered by the scheduler, *mpirun* retrieves information from an application schema file created during the last checkpoint. This application schema specifies the arguments required to initiate a *cr_restart* on specified nodes. We implemented an extension to *mpirun* that modifies and enhances the application schema at run-time with the following information received from the scheduler:

- For the processes on the live nodes, replace *cr_restart* with *cr_pause* using the respective arguments to the command;
- For the processes to migrate, replace the node number of the failed node with the replacement node. Update naming references referring to nodes with regard to this file to reflect any migrations.

2.3.3 Job-pause

The pause implementation in LAM/MPI relies on the out-of-band communication channel provided by lamd. Its design is not constrained to the interfaces of LAM/MPI and may be easily retargeted to other MPI or C/R implementations. In our case, *crtcp* [16] and *cr* [10] are utilized, which represent the interface to the most commonly used C/R mechanism provided by LAM (while other C/R mechanisms may coexist).

When a node fails or a communication link lapses, the MPI-related processes on other live nodes will block/suspend waiting for the failure to be addressed, which realizes the passive facet of the job pause mechanism. On the active side, we restore the failed process image from a checkpoint file. This checkpoint information encompasses, among other data, pending MPI messages. Hence, in-flight data on the network does not unnecessarily have to be drained at pause time. Consequently, a process can be individually paused at arbitrary points during its execution.

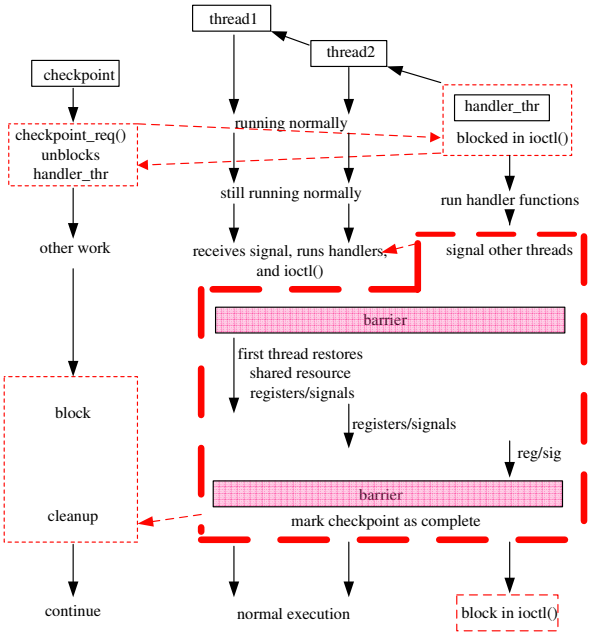


Figure 2.4: BLCR with Pause in Bold Frame

Figure 2.4 shows the steps involved during the job pause in reference to BLCR. A detailed account of the individual events during checkpointing and restarting is given in

the context of Figures 1 and 2 of [15] and is abbreviated here due to space constraints. Our focus is on the enhancements to BLCR (large dashed box).

In the figure, time flows from top to bottom, and the processes and threads involved in the pause are placed from right to left. Activities performed in the kernel are surrounded by dotted lines. The callback thread (right side) is spawned as the application registers a threaded callback and blocks in the kernel until a pause occurs. When *mpirun* invokes the *pause* command of BLCR, it provides the process id and the name of a checkpoint file to *pause* as an argument. In response, the *pause* mechanism issues an *ioctl* call, thereby resuming the callback thread that was previously blocked in the kernel. After the callback thread invokes the individual callback for each of the other threads, it reenters the kernel and sends a pause signal to each threads. These threads, in response, engage in executing the callback signal handler and then enter the kernel through another *ioctl* call.

Once in the kernel, the threads invoke the *thaw* command using VMADump to take turns reading their register and signal information from the checkpoint file. In our implementation, they no longer need to restore their process IDs and process relationships as we retain this information. After a final barrier, the process exits the kernel and enters user space, at which point the pause mechanism has completed.

Once the MPI processes have completed the pause and resume operation, they will wait for *mpirun* to supply information about the migrated process before establishing connections with the migrated process based on the received information. Once the connections have been established, the processes continue their normal execution.

2.3.4 Process Migration

Process migration after a failure is initiated on the replacement node when it receives a restart request from *mpirun*. Within the callback handler, the migrated process relays its addressing information to *mpirun*. This addressing information can later be identified as type *struct_gps* in the LAM universe. Next, *mpirun* broadcasts this information to all other processes and, conversely, sends information about all other processes (generated from the application schema) to the migrated process.

Once the processes receive the addressing information from *mpirun*, they reset the information in their local process list before establishing a TCP connection between each other. Notice that the connections among the paused processes remain active, *i.e.*, only the

connections between the migrated process and the paused processes need to be established.

Another enhancement with LAM/MPI for process migration is accomplished as follows. The LAM daemon establishes a named socket visible through the local file system (typically under the name `/tmp/lam-<username>@<hostname>/` or, alternatively, location-dependent on the TMPDIR environment variable or the LAM_MPI_SESSION_PREFIX variable). Since LAM initializes this location information at startup time and never refreshes it thereafter, we force a reset of the naming information at the replacement node resulting in an update of the directory reference inside the callback function of the migrated process.

The functionality to realize process migration discussed so far enhances LAM/MPI runtime system. In addition, upon restarting a BLCR-checkpointed job on a different node, we must ensure that the operating system on all nodes supplies the exact same libraries across migration. Any library reference by an executable that is migrated has to be portable across migrations. BLCR does not save state of shared libraries (*e.g.*, initialization state of library-specific variables). As of late, some distributions of Linux are using "prelinking" to assign fixed addresses for shared libraries in a manner where these fixed addresses are randomized to counter security attacks. We had to deactivate the prelinking feature in our system to provide cross-node compatible library addresses suitable for process migration.

2.4 Experimental Framework

We conducted our performance evaluations on a local cluster that we control. This cluster has sixteen compute nodes running Fedora Core 5 Linux x86_64 (Linux kernel-2.6.16) connected by a Gigabit Ethernet switch. Each node in the cluster is equipped with four 1.76GHz processing cores (2-way SMP with dual-core AMD Opteron 265 processors) and 2 GB memory. A 750 GB RAID5 array provides shared file service through NFS over the Gigabit switch, which is configured to be shared with MPI traffic in these experiments. We extended the latest versions of LAM/MPI (lam-7.2b1r10202) and BLCR (blcr-0.4.pre3_snapshot_2006_09_26) with our job pause mechanism for this platform.

2.5 Experimental Results

We assessed the performance of our system in terms of the time to tolerate faults for MPI jobs using the NAS Parallel Benchmarks (NPB) [18]. NPB is a suite of programs widely used to evaluate the performance of parallel systems. The suite consists of five kernels (CG, EP, FT, MG, and IS) and three pseudo-applications (BT, LU, and SP).

In the experiments, the NPB suite was exposed to class C inputs running on 4, 8, and 16 nodes. IS was excluded from experiments due to its extremely short completion time (≈10 seconds on 16 nodes), which did not reliably allow us to checkpoint and restart (with about the same overhead). All job pause/restart results were obtained from five samples with a confidence interval of ±0.1s for short jobs and ±3s for long jobs with a 99% confidence level.

Prior work already focused on assessing the cost of communication within the LAM/MPI and BLCR environments to checkpoint and restart MPI jobs [15, 19]. Our job pause and process migration mechanism decreases the communication by avoiding to re-establish the connections among the paused processes. Yet, the benefits of our approach lie in its applicability to proactive fault tolerance and incremental checkpointing. Our experiments are targeted at capturing the overhead of our approach, comparing it to prior approaches and analyzing the cause of its cost.

2.5.1 Checkpointing Overhead

Our system periodically takes snapshots of the MPI jobs at checkpoints yielding a transparent fault tolerance mechanism. Jobs can automatically recover by continuing from the most recent snapshot when a node fails. Figures 2.5, 2.6, and 2.7 depict the checkpoint overhead for 4, 8 and 16 nodes. As shown by these results, the overhead of job-pause C/R is uniformly small relative to the overall execution time of a job (benchmark), even for a larger number of nodes.

Figure 2.8 depicts the measured overhead for single checkpointing relative to the base execution time of each benchmark (without checkpointing). For most benchmarks, the ratio is below 10%. Figure 2.9 depicts the corresponding time for single checkpointing. This time is in the order of 1-12 seconds, except for MG and FT as discussed in the following.

MG has a larger checkpoint overhead (large checkpoint file), but the ratio is skewed

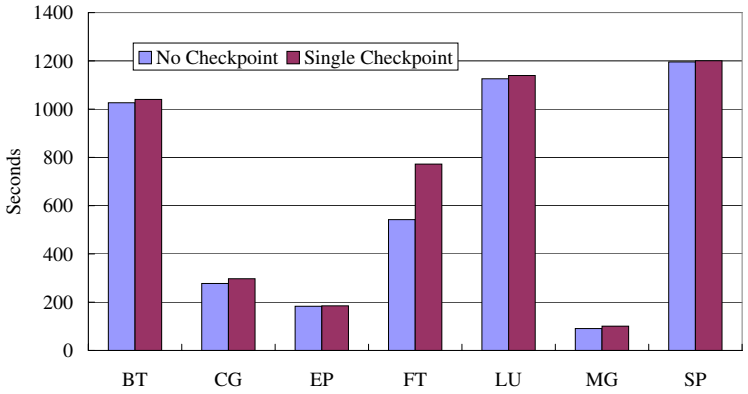


Figure 2.5: Single Checkpoint on 4 Nodes

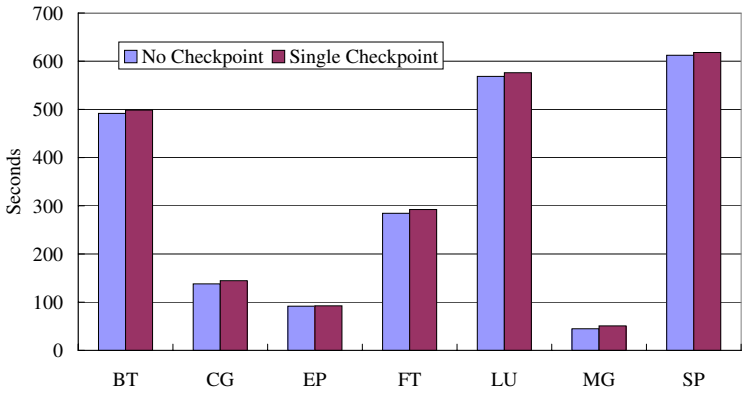


Figure 2.6: Single Checkpoint on 8 Nodes

due to a short overall execution time (see previous figures). In practice, with more realistic and longer checkpoint intervals, a checkpoint would not be necessitated within the application's execution. Instead, the application would have been restarted from scratch. For longer runs with larger inputs of MG, the fraction of checkpoint/migration overhead would have been much smaller.

As shown in prior work [15], checkpoint times increase linearly with the application's virtual memory (VM) size for processes that consume significantly less than half of the system's physical memory. However, when a process utilizes a VM size approaching or exceeding half the physical memory on the system, overheads increase up to an order of magnitude. This artifact explains the relatively high cost of FT for checkpointing (Figures 2.5, 2.8 and 2.9) and pause/restart (Figure 2.10) when it is run on just four nodes. For a larger number of nodes, the problem size is split such that smaller amounts of the VM

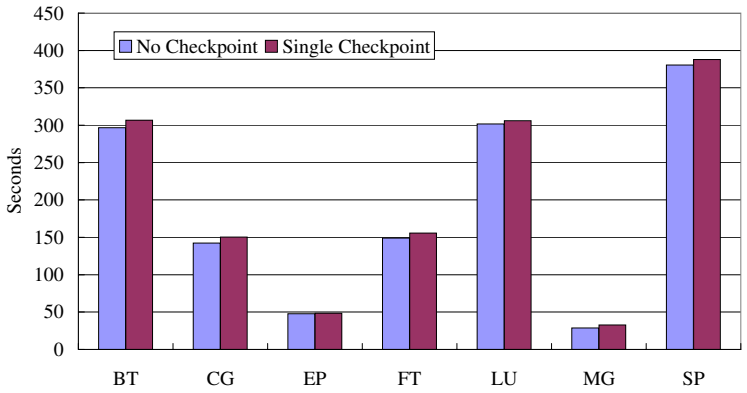


Figure 2.7: Single Checkpoint on 16 Nodes

are utilizes, which eliminates this problem. This property of checkpoint overhead relative to VM utilization is not inherent to our solution; rather, it is inherent to the checkpoint mechanism used in combination with the operating system.

Table 2.1 depicts the size of the checkpoint files for one process of each MPI application. The average file size is 135 MB on 16 nodes, 259 MB for 8 nodes, and 522.91 MB for 4 nodes. Writing many files of such size to shared storage synchronously may be feasible for high-bandwidth parallel file systems. In the absence of sufficient bandwidth for simultaneous writes, we provide a multi-stage solution where we first checkpoint to local storage. After local checkpointing, files will be asynchronously copied to shared storage, an activity governed by the scheduler. This copy operation can be staggered (again governed by the scheduler) between nodes. Upon failure, a spare node restores data from the shared file system while the remaining nodes roll back using the checkpoint file on local storage, which results in less network traffic.[1]

Table 2.1: Size of Checkpoint Files [MB]

Node #	BT	CG	EP	FT	LU	MG	SP
4	406.90	250.88	1.33	1841.02	185.51	619.46	355.27
8	186.68	127.17	1.33	920.82	99.50	310.36	170.47
16	111.12	63.50	1.33	460.73	52.61	157.31	100.39

[1] In our experiments, we simply copy exactly one checkpoint file to shared storage, namely the file of the node whose image will be migrated.

Figure 2.8: Single Checkpoint Overhead

Overall, the experiments show that the checkpoint/restart overhead of the MPI job

1. is largely proportional to the size of the checkpoint file (Table 2.1 and Figure 2.9); and
2. is nearly the same at any time of the execution of the job.

The first observation indicates that the ratio of communication overhead to computation overhead for checkpoint/restart of the MPI job is relatively low. Since checkpoint files are, on average, large, the time spent on storing/restoring checkpoints to/from disk accounts for most of the measured overhead. This overhead can be further reduced. We are currently investigating the potential for savings through incremental checkpointing and proactive diskless live migration, which would reduce the checkpoint and pause overheads, respectively.

In our experiments, communication overhead of the applications was not observed to significantly contribute to the overhead or interfere with checkpointing. This is, in part, due to our approach of restarting from the last checkpoint. Hence, our autonomic fault-tolerance solution should scale to larger clusters.

The second observation about checkpoint overheads above indicated that the size

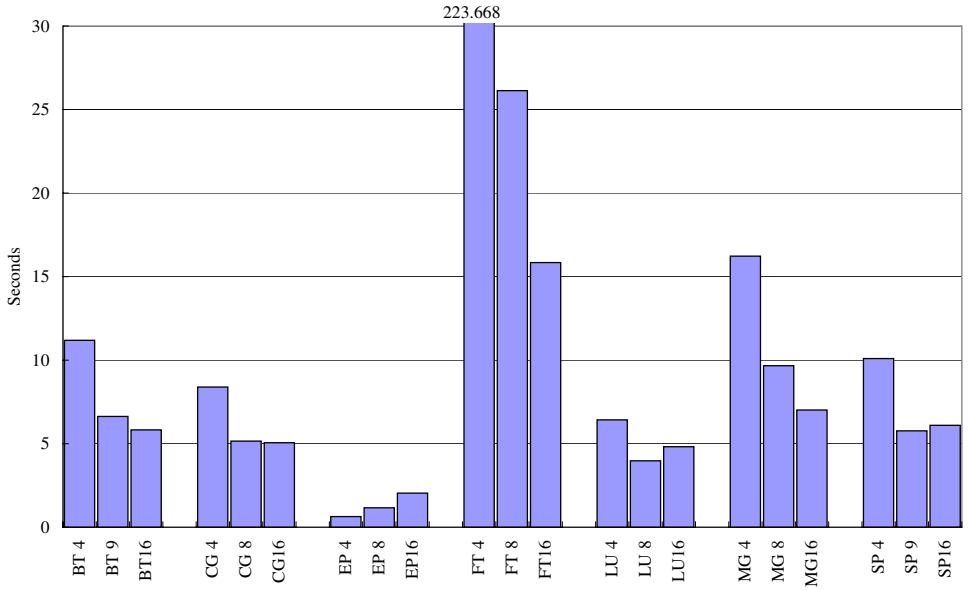


Figure 2.9: Single Checkpoint Time

of the checkpoint file remains stable during job execution. The NPB codes do not allocate or free heap memory dynamically within timesteps of execution; instead, all allocation is done during initialization, which is typical for most parallel codes (except for adaptive codes [20]). Thus, we can assume the time spent on checkpoint/restart is constant. This assumption is critical to determine the optimal checkpoint frequency [21].

2.5.2 Overhead of Job Pause

As mentioned in the implementation section, the job pause mechanism comprises two parts: *cr_pause* within BLCR and the pause work at the LAM level. Experiments indicated that the overhead of *cr_pause* is almost the same as that of *cr_restart*. Though we reuse existing processes and avoid restoring the parent/child relationship as *cr_restart* does, we still need to restore the entire shared state (mmaps, files, etc.), which is generally large and accounts for the main overhead of *cr_pause/cr_restart*. The overall effect on our scheme is, however, not bounded by the roll-back on operational nodes. Instead, it is constrained by process migration, which uses *cr_restart* to restore the process from the checkpoint file on the replacement node. The pause mechanism at the LAM level actually saves the overhead to reconnect sockets between the processes. This is reflected in Figures 2.10, 2.11, and

2.12, which show that our approach is performing nearly at par with with the job restart overhead for 4, 8 and 16 nodes. Yet, while job restart requires extra overhead for rebooting the LAM subsystem of processes, our approach does not incur this cost as it reuses existing processes.

2.5.3 Membership/Scheduler Performance

The decentralized group membership protocol, adopted from our prior work [17] and integrated in LAM, has been shown to yield response times in the order of hundreds of microseconds and single-digit milliseconds for any reconfiguration (node failure) under MPI. This overhead is so small that it may be ignored when considering the restart overhead in the order of seconds / tens of seconds.

The overhead of our new scheduler is also small. The impact of scheduling is actually spread over the entire execution of the MPI job. Yet, when considering fault tolerance, only the overheads to (a) determine the replacement node and (b) trigger *mpirun* need to be accounted for. Any global information about the MPI job is maintained by *mpirun* at job initiation and termination. The overhead to determine the replacement node and to trigger *mpirun* is also in the order of hundreds of microseconds and, hence, does not significantly contribute to the overall overhead of C/R.

2.5.4 Job Migration Overhead

The overhead of process migration represents the bottleneck of our fault tolerance approach. The job-pause mechanism of other (non-failed) nodes results in lower overhead, and the overhead of the group membership protocol and our novel scheduler is insignificant, as explained above. Let us consider the potential of our approach for job migration in contrast to a full restart. Figures 2.10, 2.11, and 2.12 show that the performance of job pause is only 5.6% larger than a complete job restart (on average for 4, 8 and 16 nodes). Yet, job pause alleviates the need to reboot the LAM run-time environment, which accounts for 1.22, 2.85 and 6.08 seconds for 4, 8 and 16 nodes. Hence, pause effectively reduces the overall overhead relative to restart.

Our approach has several operational advantages over a complete restart. First, job pause enables seamless and transparent continuation of execution across node failures.

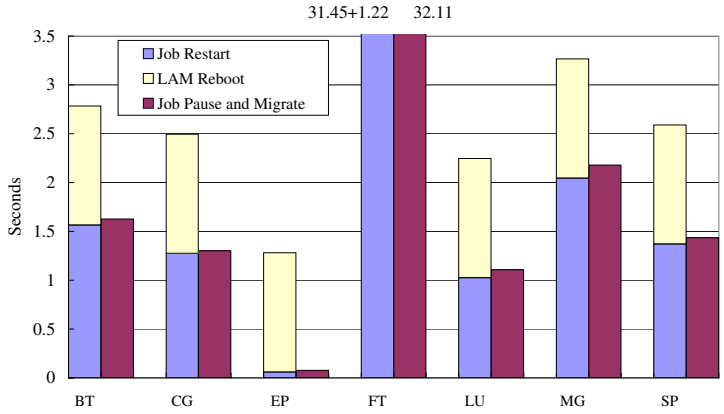


Figure 2.10: Pause and Migrate on 4 Nodes

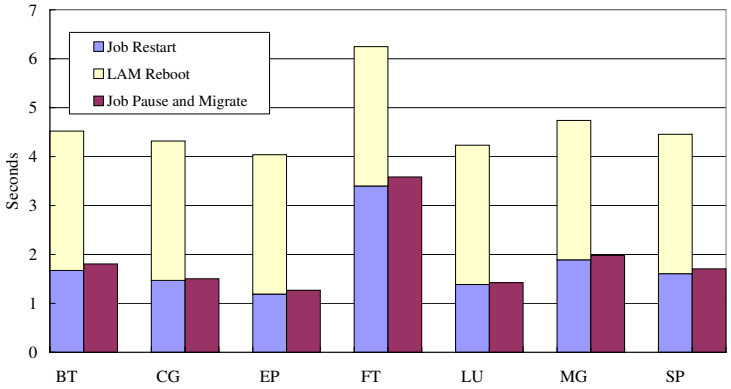


Figure 2.11: Pause and Migrate on 8 Nodes

In contrast, a complete restart may be associated with a lengthy requeuing overhead in the job submission system. Second, our approach is suitable for proactive fault tolerance with diskless migration. In such a scenario, healthy nodes would not need to roll back their computation. Instead, only the image of the unhealthy node is sent to a replacement node, which seamlessly picks up computation from there. The only effect on healthy nodes is that their socket connections have to be updated, which our scheme already supports. We are pursuing this approach, which with our novel job pause mechanism can now be realized.

The behavior of FT in Figures 2.10, 2.11, and 2.12, which differs from other codes, can be explained as follows. As previously mentioned, the checkpoint file of FT for 4 nodes exceeds half the physical memory. As a result, its pause and restart time goes up by an order of magnitude, as documented previously in the BLCR work [15].

Furthermore, consider the overhead of EP in Figures 2.9, 2.10, 2.11, 2.12. The

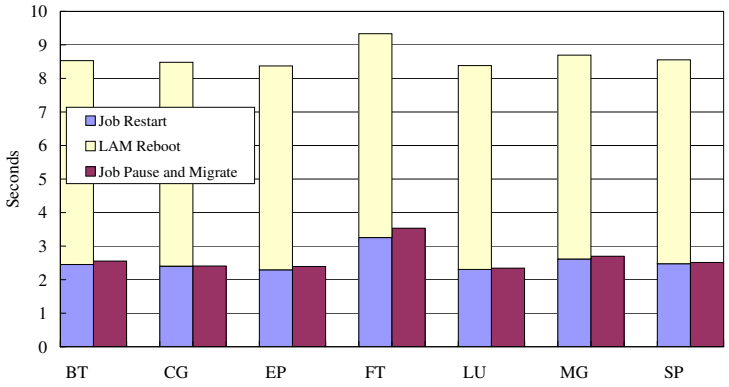


Figure 2.12: Pause and Migrate on 16 Nodes

checkpoint/pause&migrate/restart overhead of EP is becoming more dominant as we increase the number of nodes. This is caused by the small footprint of the checkpoint file (about 1 MB), which results in a relatively small overhead for storing/restoring the checkpoint file. Thus, the overhead of the migration combined with job pause mainly reflects the variance of the communication overhead inherent to the benchmark, which increases with the node count.

2.6 Conclusion

This chapter contributes a fresh approach for transparent C/R through our novel job pause mechanism. The mechanism, implemented within LAM/MPI+BLCR, allows live nodes to remain active and roll back to the last checkpoint while failed nodes are dynamically replaced by spares before resuming from the last checkpoint. Enhancements to LAM/MPI include (1) support of scalable group communication with fluctuating number of nodes, (2) transparent coordinated checkpointing, (3) reuse of network connections upon failures for operational nodes, and (4) a BLCR enhancement for the job pause mechanism. We have conducted experiments with the NAS Parallel Benchmark suite in a 16-node dual-processor Opteron cluster. Results indicate that the performance of job pause is comparable to that of a complete job restart, albeit at full transparency and automation. A minimal overhead of 5.6% is only incurred in case migration takes place while the regular checkpoint overhead remains unchanged. Yet, our approach alleviates the need to reboot the LAM run-time environment, which accounts for considerable overhead resulting in net savings

of our scheme in the experiments. Furthermore, job pause reuses existing resources and continues to run within the scheduled job, which can avoid staging overhead and lengthy requeuing in submission queues associated with traditional job restarts. Our experiments also indicate that, after the initialization phase, checkpoints are constant in size for a given application, regardless of the timing of checkpoints. Our job pause approach further offers an additional potential for savings through incremental checkpointing and proactive diskless live migration, which are discussed in Chapters 3 and 5.

Chapter 3

Proactive Process-level Live Migration

3.1 Introduction

This chapter promotes process-level live migration combined with health monitoring for a proactive FT approach that complements existing C/R schemes with self healing whose fault model is based on the work by Tikotekar *et al.* [22].

Health monitoring has recently become a wide-spread feature in commodity and, even more so, in server and HPC components. Such monitors range from simple processor temperature sensors to BMCs with a variety of sensing capabilities, including fan speeds, voltage levels and chassis temperatures. Similarly, the SMART disk standard provides the means to trigger events upon disk errors indicating disk problems, which can be saved in log files or which can trigger exception handling mechanisms. Aided by such capabilities, node failures may be anticipated when the health status of a node deteriorates, *i.e.*, when abnormal sensor readings or log entries are observed.

Health monitoring has been used to model failure rates and, in a reactive manner, to determine checkpoint intervals [23, 24]. In this work, we venture beyond reactive schemes by promoting a *proactive* approach that migrates processes away from "unhealthy" nodes to healthy ones. Such a self-healing approach has the advantage that checkpoint frequencies can be reduced as sudden, unexpected faults should become the exception. This requires the availability of spare nodes, which is becoming common place in recent cluster acquisitions.

We expect such spare nodes to become a commodity provided by job schedulers upon request. Our experiments assume availability of 1-2 spare nodes.[1]

The feasibility of health monitoring at various levels has recently been demonstrated for temperature-aware monitoring, *e.g.*, by using ACPI [25], and, more generically, by critical-event prediction [26]. Particularly in systems with thousands of processors, fault handling becomes imperative, yet approaches range from application-level and runtime-level to the level of OS schedulers [27, 28, 29, 30]. These and other approaches differ from our work in that we promote *live migration combined with health monitoring.*

We have designed an automatic and transparent mechanism for *proactive* FT of arbitrary MPI applications. The implementation, while realized over LAM/MPI+BLCR, is in its mechanisms applicable to any process-migration solution, *e.g.*, the Open MPI FT mechanisms [11, 31]. The original LAM/MPI+BLCR combination [10] only provides reactive FT and requires a complete job restart from the last checkpoint including job resubmission in case of a node failure. The work presented in the last chapter enhances this capability with a job pause/continue mechanism that keeps an MPI job alive while a failed node is replaced by a spare node. Paused, healthy tasks are rolled back to and spare nodes proceed from the last checkpoint in a coordinated manner transparent to the application.

The contribution of this chapter is to avoid roll-backs to prior checkpoints whenever possible. By monitoring the health of each node, a process is migrated as a precaution to potentially imminent failure. To reduce the impact of migration on application performance, we contribute a novel *process-level live migration* mechanism as an enhancement to the Linux BLCR module. Thus, execution proceeds while a process image is incrementally and asynchronously transferred to a spare node. This reduces the time during which the process is unresponsive to only a short freeze phase when final changes are transferred to the spare node before re-activating execution on the target node. Hence, MPI applications execute during much of process migration. In experiments, we assessed the trade-off between lower end-to-end wall-clock times of jobs subject to live migration *vs.* the slightly prolonged duration for migration as opposed to a traditional process-freeze approach. Depending on the estimated remaining up-time of a node with deteriorating health, one can choose

[1] Our techniques also generalize to task sharing on a node should not enough spare nodes be available, yet the cost is reduced performance for tasks on such shared nodes. This may result in imbalance between all tasks system-wide resulting in decreased overall performance. Such imbalance might be tolerable when faulty nodes can be brought back online quickly so that processes can migrate back to their original nodes.

between live and frozen migration schemes.

Our results further demonstrate that proactive FT complements reactive schemes for long-running MPI jobs. Specifically, should a node fail without prior health indication or while proactive migration is in progress, our scheme reverts to reactive FT by restarting from the last checkpoint. Yet, as proactive FT has the potential to prolong the mean-time-to-failure, reactive schemes can lower their checkpoint frequency in response, which implies that proactive FT can lower the cost of reactive FT. More specifically, experimental results indicate that 1-6.5 seconds of prior warning are required to successfully trigger live process migration while similar operating system (OS) virtualization mechanisms require 13-24 seconds. The approach further complements reactive FT by nearly twice as long a checkpointing interval due to proactive migration when 70% of the failures are predicted only a few seconds prior (derived from [26]).

The remainder of this chapter is structured as follows. Sections 3.2 and 3.3 present the design and implementation of our process live migration mechanism. Section 3.4 describes the experimental setup. Section 3.5 discusses experimental results. Section 3.6 summarizes the contributions.

3.2 Design

Figure 3.1 depicts the system components and their interaction, *i.e.*, the chronological steps involved in process migration of each MPI job and job dependencies with data exchanges. In the following, we discuss system support for live migration at two levels: (1) the synchronization and coordination mechanisms within an MPI job and (2) live migration with incremental update support at the process/MPI task level. We further consider the tradeoff between live and frozen migration options and develop approaches to manage "dirty" memory, *i.e.*, memory written since the last incremental update.

3.2.1 Live Migration at the Job Level

Figure 3.1 depicts steps 1-6, each of which are described in the following.

Step 1: Migration Trigger: In our system, the per-node health monitoring mechanism is realized on top of a BMC. It is equipped with sensors to monitor different properties, *e.g.*, sensors providing data on temperature, fan speed, and voltage. We also

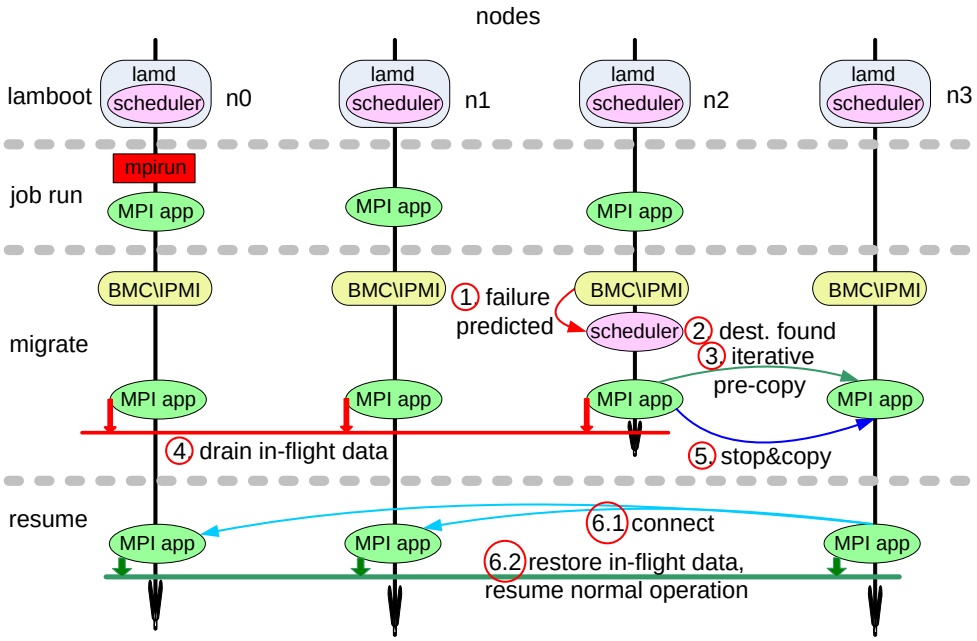


Figure 3.1: Job Live Migration

employ IPMI, an increasingly common management/monitoring component that provides a standardized message-based interface to obtain sensors readings for health monitoring. We further designed a decentralized scheduler, which can be deployed as a stand-alone component or as an integral process of an MPI daemon, such as the LAM daemon (lamd). The scheduler will be notified upon deteriorating health detected by BMC/IPMI, *e.g.*, due to a sensor reading exceeding a threshold value.

Step 2: Destination Node Determination: When the scheduler component on the health-decaying node receives the early warning issued in step 1, it first chooses a spare node as a replacement for the unhealthy node. Spare nodes are increasingly becoming a commodity in clusters. We expect that job schedulers may soon simply allocate a limited number of spares as a default or upon request during job submission. As of now, we still have to explicitly over-provision by setting a constant number of nodes aside during job submission.

These spare nodes comprise the destination for process migration. However, if no spare node was set aside during job submission or if all nodes were already used, we choose the most lightly loaded node as a migration destination, in which case the node

doubles for two MPI tasks. Sharing nodes may result in imbalance due to bearing twice the load of regular nodes, which generally results in lower overall application performance. Such imbalance might be tolerable when faulty nodes can be brought back online quickly so that processes can migrate back to their original nodes. Nonetheless, higher sustained application performance can be guaranteed when unloaded spares are available as migration targets.

Step 3: Memory Precopy: Once a destination node is determined, the scheduler initiates the migration of a process on both destination and source nodes. The objective of the first stage of migration is to transfer a memory snapshot of the process image to the spare node, yet to allow the application to execute during this stage, hence the name *live* migration. The transfer of the process image occurs at page granularity and is repeated for pages written to by the application between image scans. During the first process image scan (first iteration), all non-zero pages are transferred from the source node to the destination node. On subsequent scans/iterations, only the pages updated since the previous scan are transferred. When the number of such "dirty" pages between scans does not change significantly anymore, the scan loop terminates. System support for tracking dirty pages is discussed later in the context of memory management.

Step 4: In-flight Message Drainage: Before we stop the process and migrate the remaining dirty pages with the corresponding process state to the destination node, all MPI tasks need to coordinate to reach a consistent global state. Based on our LAM/MPI+BLCR design, message passing is dealt with at the MPI level while the process-level BLCR mechanism is not aware of messaging at all. Hence, we employ LAM/MPI's job-centric interaction mechanism for the respective MPI tasks to clear in-flight data in the MPI communication channels.

Step 5: Stop&Copy: Once all the MPI tasks (processes) reach a consistent global state, the process on the source node freezes (suspends) application execution but still copies the remaining dirty pages (written to since the last iteration in step 3) and the final process state (registers, signal information, pid, files etc.) to the destination node. All other MPI tasks are suspended at their point of execution.

Step 6: Connection Recreation, Messages Restoration and Job Continuation: When the process is ready on the destination node, it sets up a communication channel with all other MPI tasks. Subsequently, the drained in-flight messages are restored,

and all the processes resume execution from the point of suspension.

3.2.2 Live Migration at the Process Level

The incremental precopy and stop&copy (steps 3 and 5 in Figure 3.1) are performed at the process level involving only the destination and source nodes. Yet, there are trade-offs between simply stopping an application to engage in a frozen copy and the alternative of a live precopy with continued application progress. The latter, while generally resulting in shorter overall application wall-clock time, comes at the expense of background network activity, possibly even repeatedly transmitting dirtied pages. This also raises the question when the iterative precopy loop should terminate.

Figures 3.2 and 3.3 show migration with precopy (live) and without (stop&copy-only). Compared to a traditional job resubmission resuming execution from the last checkpoint in a reactive FT scheme, both of these proactive migration schemes lower the expected execution time of the MPI application in the presence of failures. This is especially the case for HPC environments where the MTBF is low, which typically implies that the number of compute nodes is high, the run-time of an application is long, and the memory footprint is large.

One of our objectives is to reduce the aggregate downtimes over all nodes, *i.e.*, the duration of the stop&copy step should be small. Live migration with incremental precopy results not only in shorter downtime on the local node (to transfer dirty pages plus other process state) but also in reduced suspension of all other nodes (once MPI message queues are drained) since fewer pages remain dirty after precopy. Another objective is to tolerate the imminent fault. The shorter the migration duration, the higher the probability that our proactive scheme makes a reactive restart from a prior checkpoint unnecessary. Frozen migration consisting only of the stop&copy step takes less overall time during migration, thereby increasing chances for successful migration. A compromise might even be to stop the precopy step prematurely upon receipt of another fault event indicating higher urgency of the health problem.

Two major factors affect the tradeoff between the downtime and the migration duration. First, the network bandwidth shared between the MPI job and the migration activity is limited. If an application utilizes less network bandwidth, more bandwidth can be consumed by the migration operation. Thus, the precopy step may only have minimum

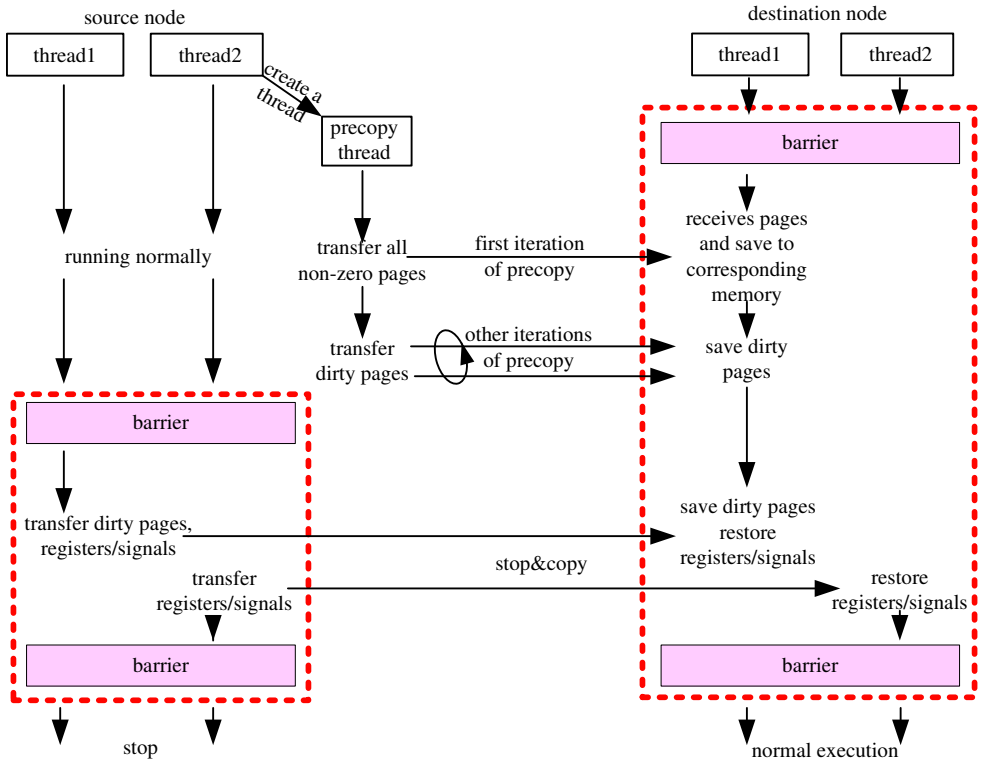


Figure 3.2: Process Migration with Precopy (Kernel Mode in Dotted Frame)

impact on application progress. More communication-intensive applications, however, may leave less unused bandwidth for live migration (during precopy). This could both slow down a local MPI task (with potentially globally unbalanced progress due to communication) and prolong precopy duration, ultimately requiring a premature termination of this step. High-end HPC installations may provide multiple network backbones to isolate MPI communication from I/O traffic, the latter of which covering checkpointing and migration traffic as well. In such systems, bandwidth issues may not be as critical (but cannot be fully discounted either).

Second, the page write rate (dirtying rate) affects the trade-off between downtime and migration duration. The dirtying rate is affected by the memory footprint (more specifically, the *rewrite* working set) of the application. A larger number of pages repeatedly written within a tight loop will prolong the precopy duration, which might lower the probability for successful migration (compared to non-live migration). In fact, the page

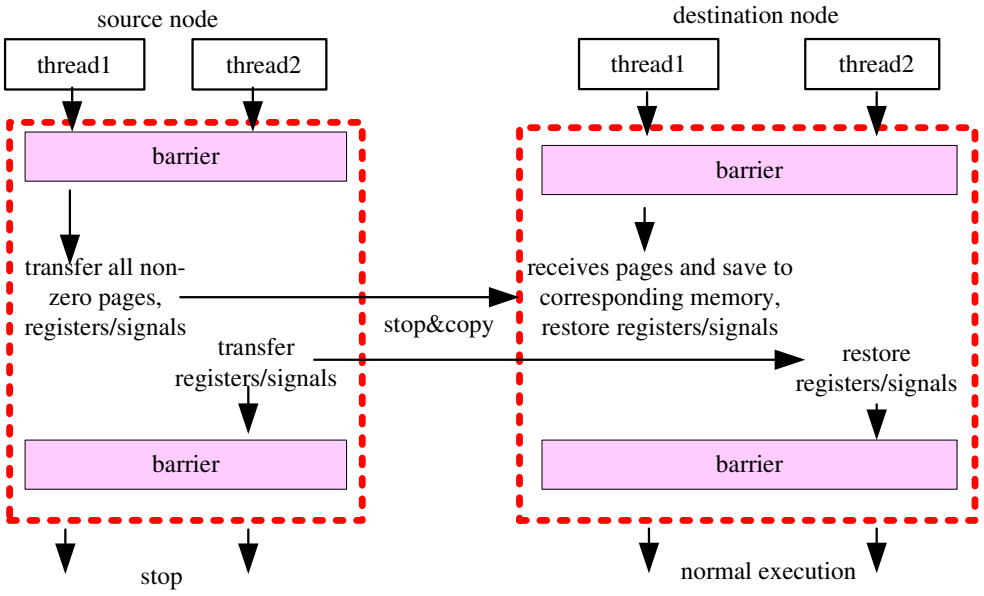


Figure 3.3: Process Migration without Precopy (Kernel Mode in Dotted Frame)

access patterns of the studied benchmarks will be shown to differ significantly. We further discuss this issue in the experimental section.

3.2.3 Memory Management

A prerequisite of live migration is the availability of a mechanism to track modified pages during each iteration of the precopy loop. Two fundamentally different approaches may be employed, namely page protection mechanisms or page-table dirty bits. Different implementation variants build on these schemes, such as the bookkeeping and saving scheme that, based on the dirty bit scheme, copies pages into a buffer [32].

Under the page protection scheme, all writable pages of a process are write-protected before each iteration occurs. When a page is modified (written to for the first time), a page fault exception is raised. The triggered exception handler enables write access to the page and records the page as dirty. Upon return from the handler, the write access is repeated successfully, and subsequent writes to the same page execute at full speed (without raising an exception). At the next iteration, pages marked as dirty are (1) write protected again before being (2) copied to the destination node. This ordering of steps is essential to

avoid races between writes and page transfers.

Under the dirty bit scheme, the dirty bit of the page-table entry (PTE) is duplicated. The dirty bit is set in response to handling by the memory management unit (MMU) whenever the first write to the page is encountered. At each iteration, the duplicate bit is checked and, if set, is first cleared before the page is transferred to the destination node. To provide this shadow functionality, kernel-level functions accessing the PTE dirty bit are extended to also set the duplicate bit upon the first write access.

The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected if an alternate signal stack is employed, which adds calling overhead and increases cache pressure. Furthermore, the overhead of user-level exception handlers is much higher than kernel-level dirty-bit shadowing. Thus, we selected the dirty bit scheme in our design. Implementation-specific issues of this scheme are discussed in detail in the next section.

3.3 Implementation

We next provide implementation details of our process-level live migration under LAM/MPI+BLCR. The overall design and the principle implementation methodology are applicable to arbitrary MPI implementations, *e.g.*, OpenMPI and MPICH.

3.3.1 Failure Prediction and Decentralized Scheduler

As outlined in Section 3.2, the capabilities of the BMC hardware and IPMI software abstraction were utilized by our framework to predict failures. In the following, we focus on the system aspects of the live migration mechanism to tolerate imminent faults and only to a lesser extent on fault prediction, a topic mostly beyond the scope of this chapter.

The scheduler is implemented as a process of lamd communicating through the out-of-band channel provided by LAM/MPI. When the scheduler daemon is notified by the BMC/IPMI health monitoring component, it consults the database to retrieve information about the MPI jobs and the nodes in the LAM universe. A spare node or, alternatively, the most lightly loaded node is chosen as the migration target. In our implementation, we also check the BMC/IPMI sensor readings with preset thresholds to determine if the degree of urgency allows accommodation of a precopy step. For example, if the temperature

is higher than a high watermark (indicating that failure is almost instantly imminent), the frozen migration (stop&copy-only) is launched, which guarantees a shorter migration duration. However, if the value is closer to the low watermark (indicating that failure is likely approaching but some time remains), the live migration (with its precopy step) is launched, which reduces overall application downtime. The design allows additional schemes, selected through a combination of sensor and history information, to be integrated as plug-ins in the future.

Next, we present implementation details of the process-level mechanisms, including dirty page tracking and process image restoration, as they were realized within BLCR. This is followed by the MPI-level implementation details based on the fundamental capabilities of BLCR, including the maintenance of consistency of a global state for the entire MPI job, as realized within LAM/MPI.

3.3.2 Process-Level Support for Live Migration

As part of our implementation, we integrated several new BLCR features to extend its process-level task migration facilities and to coordinate LAM/MPI's callback mechanism amongst all MPI processes. The scheduler discussed previously issues the *cr_save* and *cr_restore* commands on the source and destination nodes, respectively, when it deems live migration with precopy to be beneficial. Subsequently, once the precopy step has completed, LAM/MPI issues the *cr_stop* and *cr_quiesce* commands on the source node and all other operational nodes, respectively.

Precopy at the Source Node: *cr_save*

The *cr_save* command implements a sequence of steps specific to our live migration. It first sends a signal to the process on the source node where it triggers the creation of a new thread. This thread performs the precopy work at the user level. We chose a solution at the user level since a kernel-level precopy can block application threads, thereby hampering overall computational progress. During the first iteration of the precopy loop, all non-empty pages are transferred to the destination node. Subsequent iterations, in contrast, result in transfers of only those pages modified since the last iteration. The top half of Figure 3.2 at the source node depicts this procedure. Recall that the dirty bit approach implemented in

our system tracks if a page has been written to since the last transfer (iteration).

The Linux kernel maintains a dirty bit in the PTE. It is automatically set by the hardware (MMU) when the page is modified. Yet, we cannot utilize this dirty bit for our purposes since its state is tightly coupled with the Linux memory management, specifically the swap subsystem. We have to clear such a flag for each iteration of our page sweep, yet clearing the PTE dirty bit would indicate that a page need not be written on a subsequent swap-out, which is violating consistency. Instead of modifying the dirty bit semantics, we decided to shadow the dirty bit within the reserved bits of PTE. Updates to this shadow dirty bit may occur through system functions without much overhead (compared to user-level page protection).

This shadow dirty bit still needs to be preserved across swapping, which is a challenge since swapped PTEs are partially invalidated. One solution would be to simply disable swapping, which might be an option for HPC, but a more general solution is preferable. Linux actually preserves selected fields of a swapped out PTE, among other things to associate with it the swap disk device and the offset into the device where the page resides. This swap information includes a set of bits that are preserved, one of which we utilized as the shadow dirty bit. We implemented this approach in x86_64 and i386 Linux kernels similar to an orthogonal IA64 design by HP [33].

During live migration, a precopy thread iteratively transfers dirty pages to the destination node until one of the following conditions is satisfied:

- The aggregate size of dirty memory during the last iteration is less than a lower *memory threshold* (default: 1MB);
- the aggregate difference of transferred memory in consecutive iterations is less than a small *difference threshold* (indicating that a steady state of the rate in which pages are written to is reached);
- the remaining time left before the expected failure is below a lower *overhead threshold*.

As shown in Figures 3.5(a) and 3.5(b), the page dirty modification eventually stabilizes or its fluctuation is regular (repeats periodically). We could support an empirical factor in the algorithm to keep a profile history of the page modification rate and its regularity. Such model parameters could steer future application runs during live migration / precopy in

choosing a more sensitive termination condition. This could be especially beneficial for jobs with long runtime (a large number of timesteps) or repeated job invocations.

Once the precopy step terminates, the thread informs the scheduler, who then enters the stop&copy step in a coordinated manner across all processes of the MPI job. This includes issuing the *cr_stop* command on the source node.

Freeze at the Source Node: *cr_stop*

The redesigned *cr_stop* command signals the process on the source node to freeze execution, *i.e.*, to stop and copy the pages dirtied in the last iteration of the precopy step to the destination node. Threads of the process subsequently take turns copying their own state information (registers, signal information, etc.) to the destination node. This functionality is performed inside the Linux kernel (as an extension to BLCR) since process-internal information needs to be accessed directly and in a coordinated fashion between threads. The lower half of Figure 3.2 depicts these actions at the source node.

Between the precopy and freeze steps, the processes of an MPI job also need to be globally coordinated. The objective is to reach a consistent global state by draining all in-flight messages and delaying any new messages. This is accomplished in two parts at the source node, implemented by the *cr_save* and *cr_stop* commands. At the destination node, a single command implements the equivalent restore functionality.

Precopy and Freeze at the Destination Node: *cr_restore*

At the destination node, the *cr_restore* command with our extensions is issued by the scheduler. This results in the immediate creation of an equal number of threads as were existent on the source node, which then wait inside the Linux kernel for messages from the source node. A parameter to the command, issued by the scheduler, indicates whether or not a live/precopy step was selected on the source node. In either case, one thread receives pages from the source node and places them at the corresponding location in local memory. All threads subsequently restore their own state information received from the corresponding source node threads, as depicted in Figures 3.2 and 3.3 at the destination node. After the process image is fully transferred and restored, the user mode is entered. Next, the MPI-level callback function is triggered, which creates the connections with the

other MPI processes of the compute job and restores the drained in-flight messages discussed next.

Process Quiesce at Operational Nodes: *cr_quiesce*

In steps 4 and 6 of our design (cf. Section 3.2), processes on all other operational nodes drain the in-flight messages before the stop&copy step. They then remain suspended in this step, creating a connection with the new process on the destination (spare) node and ultimately restoring in-flight messages after. This sequence of actions is triggered through the newly developed *cr_quiesce* command. In our implementation, issuing this command signals the process (MPI task), which subsequently enters the MPI-level callback function to drain the messages, waits for the end of the stop&copy step of the faulty/spare nodes, and then restores its communication state before resuming normal execution.

3.3.3 Job Communication and Coordination Mechanism for Live Migration

In our implementation, we extended LAM/MPI with fundamentally new functionality provided through our BLCR extension to support live migration of a process (MPI task) within a job. Our approach can be divided into four stages introduced in their respective temporal order:

Stage 1: Live/Precopy: The scheduler of the LAM daemon, lamd, determines whether or not live/precopy should be triggered. If sufficient time exists to accommodate this stage, the *cr_save* and *cr_restore* commands are issued on the source and destination nodes, respectively. During the precopy step, the compute job continues to execute while dirty pages are sent iteratively by a precopy thread on the source node. If time does not suffice to engage in this live/precopy stage, the next stage is entered immediately.

Stage 2: Pre-Stop&Copy: In this stage, the scheduler issues a stop&copy command for *mpirun*, the initial LAM/MPI process at job invocation, which subsequently invokes the *cr_stop* and *cr_quiesce* commands on the source node and any other operational nodes, respectively. Once signaled, any of these processes first enters the callback function, which had been registered as part of the LAM/MPI initialization at job start. The callback forces a drain of the in-flight data to eventually meet a consistent global state for all

processes (MPI tasks).

Stage 3: Stop&Copy: Upon return from the callback function, the process on the source node stops executing and transfers the remaining dirty pages and its process state. Meanwhile, the processes on other operational nodes suspend in a waiting pattern, as discussed previously.

Stage 4: Post-Stop&Copy: Once all the state is transferred and restored at the destination (spare) node, the process is activated at this node and invokes the LAM/MPI callback function again from the signal handler. Within the callback handler, the migrated process relays its addressing information to *mpirun*, which broadcasts this information to all other processes. These other processes update their entry in the local process list before establishing a connection with the migrated process on the spare. Finally, all processes restore the drained in-flight data and resume execution from the stopped state.

3.4 Experimental Framework

Experiments were conducted on the same cluster used in Chapter 2. The nodes are interconnected by two networks, both 1 Gbps Ethernet. The OS used is Fedora Core 5 Linux x86_64 with our dirty bit patch as described in Sections 3.2 and 3.3. One of the two networks was reserved for LAM/MPI application communication while the other supported process migration and other network traffic. In the next section, we present results for these two network configurations. As discussed in Section 3.2.2, the MPI job and the migration activity may compete for network bandwidth if only a single network is available. However, it is common for high-end HPC to install two separate networks, one reserved for MPI communication and the other for operations such as I/O, booting, system setup and migration. In our experiments, we also assessed the system performance with a single network responsible for both MPI communication and migration activities. The results are not significantly different from those with two networks, which shows that for the applications evaluated in our system communication and memory intensity do not coincide. For the future work, we will create and assess applications with varying communication rates and memory pressure to measure the tradeoff between live and frozen migrations and to provide heuristics accordingly, as discussed in the next section.

We have conducted experiments with several MPI benchmarks. Health deteriora-

tion on a node is simulated by notifying the scheduler daemon, which immediately initiates the migration process. To assess the performance of our system, we measure the wall-clock time for a benchmark with live migration, with stop&copy-only migration and without migration. The migration overheads are introduced by transferring the state of the process, including the dirty pages, and the coordination among the MPI tasks. In addition, the actual live migration duration can be attributed to two parts: (1) the overhead incurred by the iterative precopy and (2) the actual downtime for which the process on the source node is stopped. Accordingly, precopy durations and downtimes are measured.

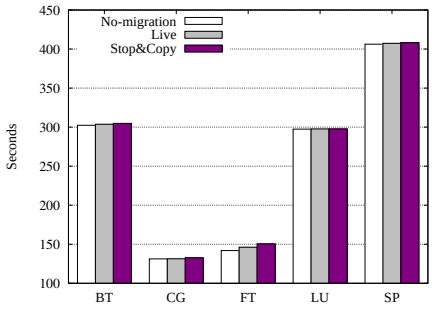
3.5 Experimental Results

Experiments were conducted to assess (a) overheads associated with the migration, (b) frozen and live migration durations, (c) scalability for task and problem scaling of migration approaches, and (d) page access patterns and migration times.

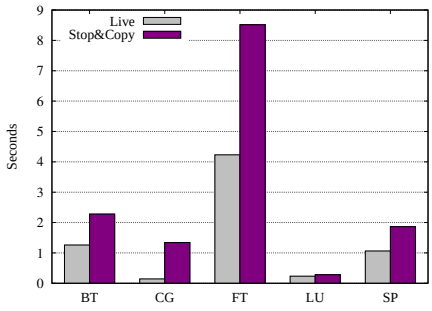
Results were obtained for the NPB version 3.2.1 [18], a suite of programs widely used to evaluate the performance of parallel systems. Out of the NPB suite, the BT, CG, FT, LU and SP benchmarks were exposed to class B and class C data inputs running on 4, 8 or 9 and 16 nodes. Some NAS benchmarks have 2D, others have 3D layouts for 2^3 or 3^2 nodes, respectively. In addition to the 16 nodes, one spare node is used, which is omitted (implicitly included) in later node counts in this chapter. The NAS benchmarks EP, IS and MG were not suitable for our experiments since they execute for too short a period to properly gauge the effect of imminent node failures. (With class B data inputs, completion times for EP, IS and MG are 11.7, 4.8 and 4.1 seconds, respectively, for 16-node configurations.)

3.5.1 Migration Overhead

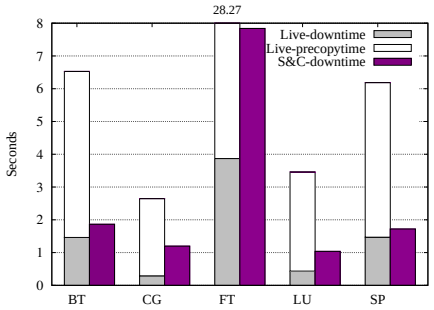
The first set of experiments assesses the overhead incurred due to one migration (equivalent to one imminent node failure). Figure 3.4(a) depicts the job execution time without any migration, with live migration and with frozen (stop&copy-only) migration under class C inputs on 16 nodes. The corresponding overheads per scheme are depicted in Figure 3.4(b). The results indicate that the wall-clock time for execution with live migration exceeds that of the base run by 0.08-2.98% depending on the application. The



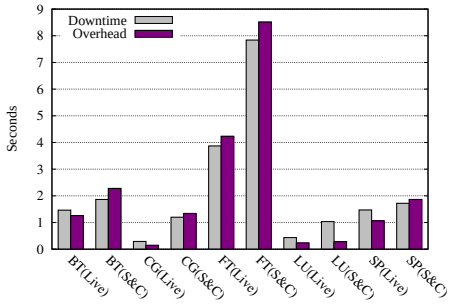
(a) Job Execution Time for NPB C-16



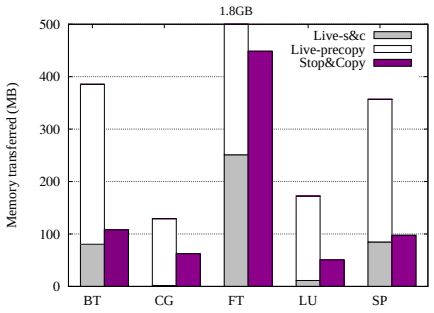
(b) Migration Overhead for NPB C-16



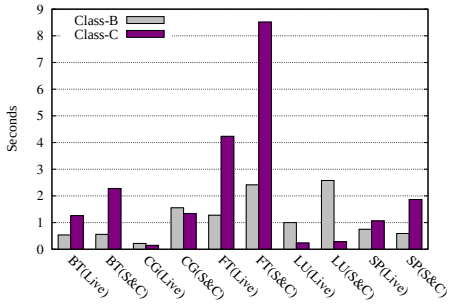
(c) Migration Duration for NPB C-16



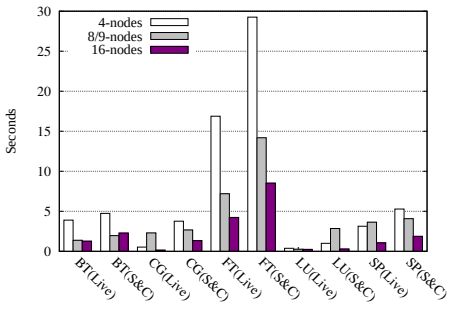
(d) Downtime vs. Overhead for NPB C-16



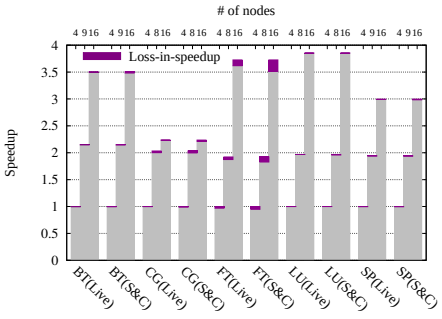
(e) Memory Transferred for NPB C-16



(f) Problem Scaling: Overhead NPB 16



(g) Task Scaling: Overhead of NPB C



(h) Speedup for NPB Class C

Figure 3.4: Evaluation with NPB (C-16: Class C on 16 Nodes)

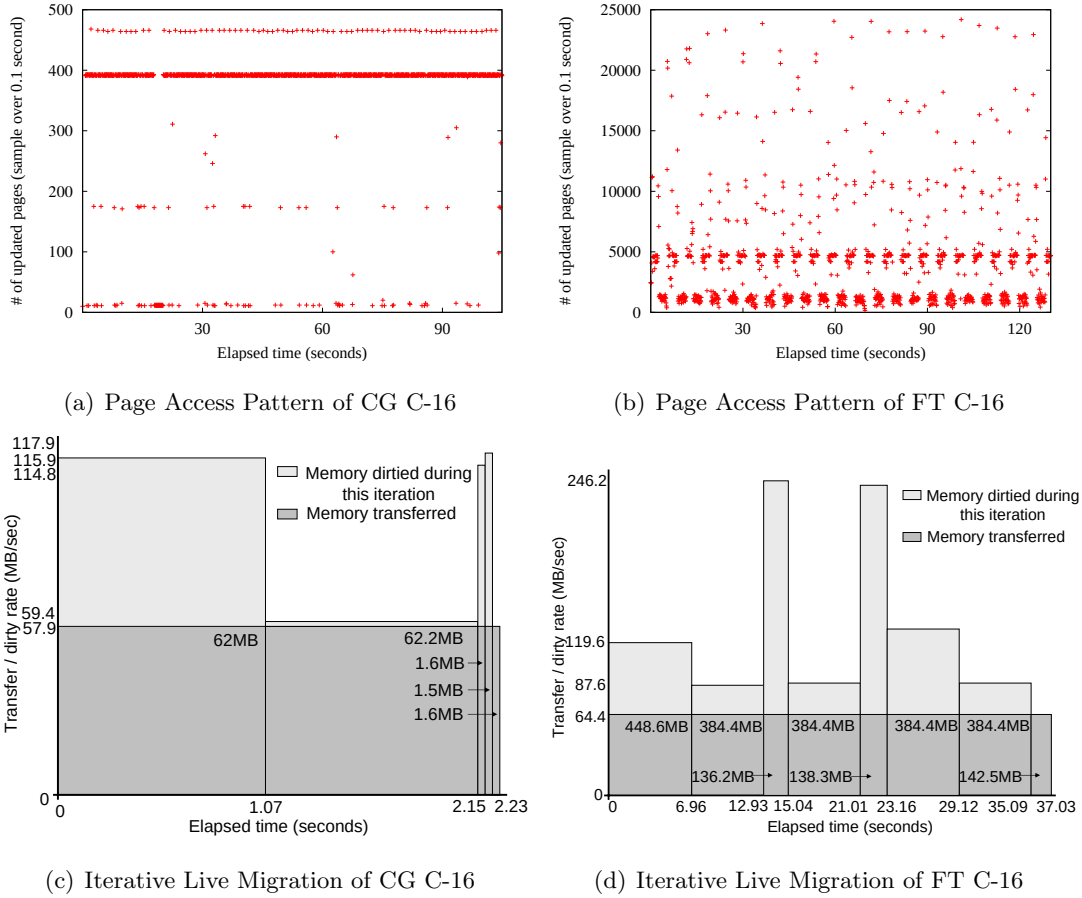
overhead of frozen migration is slightly higher at 0.09-6%. The largest additional cost of 6.7% was observed for FT under class B inputs for 16 nodes (not depicted here) due to its comparatively large memory footprint (113 MB) and relatively short execution time (37 seconds). Amongst BT, CG, FT, LU and SP under classes B and C running on 4, 8 or 9 and 16 nodes, the longest execution time is 20 minutes (for SP under class C inputs on 4 nodes, also not depicted here). Projecting these results to even longer-running applications, the overhead of migration becomes less significant (if not even insignificant) considering the equivalent checkpointing overhead under current MTTF rates.

3.5.2 Migration Duration

Besides overhead due to migration, we assessed the actual migration duration including live migration (precopy duration) and the downtime of other nodes during migration schemes. Figure 3.4(c) depicts (a) live migration with the precopy and downtime fractions and (b) frozen migration with only its downtime during the stop&copy step, both for NPB with class C inputs on 16 nodes. The precopy duration was measured from the issuing of the *cr_save* command to its completion. The stop&copy downtime was measured from issuing *cr_stop* on the source node / *cr_quiesce* on the operational nodes to the resumption of the job execution. Live downtime ranged between 0.29-3.87 seconds while stop&copy downtime was between 1.04-7.84 seconds. Live migration pays a penalty for the shorter downtime in that its precopy duration is prolonged. Precopy adds another 2.35-24.4 seconds. Nonetheless, the precopy stage does not significantly affect execution progress of the application as it proceeds asynchronously in the background. Figure 3.4(d) illustrates this fact for each scheme by comparing the downtime (from Figure 3.4(b)) with migration overhead for both frozen (stop&copy) and live migration. Both schemes show a close match between their respective downtime and overhead numbers. (Some benchmarks show shorter overhead than the absolute downtime due to larger variances in job execution times subsuming the shorter overheads, see Section 3.5.3.)

Figure 3.4(e) depicts the amount of memory transferred during migration. With frozen (stop&copy-only) migration, memory pages of a process cannot be modified while the process remains inactive in this stage. In contrast, live migration allows pages to be modified and consequently requires repeated transfers of dirty pages. Hence, both precopy duration and downtime are a function of the write frequency to disjoint memory pages.

Frozen (stop&copy-only) migration results in larger memory transfers (50.7-448.6MB) than just the stop&copy step of live migration (1.7-251MB), yet the latter incurs additional transfers (127.4-1565MB) during the precopy step. This result is consistent with the downtime observations of the two schemes discussed above. Our experiments also indicate an



(a) Page Access Pattern of CG C-16

(b) Page Access Pattern of FT C-16

(c) Iterative Live Migration of CG C-16

(d) Iterative Live Migration of FT C-16

Figure 3.5: Page Access Pattern vs. Iterative Live Migration

approximate cost of 0.3 seconds for MPI-level coordination and communication during live migration plus a cost of less than 0.1 seconds transferring process state information, *e.g.*, registers and signal information. Hence, both the precopy duration and the downtime are almost entirely due to the overhead of transferring memory pages. Hence, the overall trends and patterns of Figures 3.4(c) and 3.4(e) tend to be similar.

3.5.3 Effect of Problem Scaling

Figure 3.4(f) depicts the effect of migration on scaling the problem size with class B and C inputs on 16 nodes. For BT, FT and SP, we observe an increase in overhead as the task size increases from class B to class C. This behavior is expected as problem scaling results in larger data per node. However, the inverse behavior is observed for CG and LU. Though the actual downtime becomes longer as the task size increases from class B to class C, the absolute downtime (0.27-1.19 seconds) is so small that its effect is subsumed by the variance of overall job execution time (up to 11 seconds for CG and up to 8 seconds for LU), ultimately resulting in an overall decrease in overhead for increasing task sizes.

3.5.4 Effect of Task Scaling

We next examined migration under strong scaling by increasing the number of nodes. Figure 3.4(g) depicts the overhead for NPB codes with class C inputs on varying number of nodes (4, 8 or 9 and 16). In most cases, overheads tend to decrease as the number of nodes increases. Yet, BT(Stop&Copy), CG(Live), LU(Stop&Copy) and SP(Live) show no obvious trends. As with problem scaling, this can be attributed to the relatively minor migration downtime, which is effectively subsumed by variances in job execution times. Hence, no measurable effect on task scaling is observed in these cases.

Figure 3.4(h) depicts the speedup on 4, 8 or 9 and 16 nodes normalized to the wall-clock time on 4 nodes. The figure also shows the relative speedup of live migration, of frozen migration (stop&copy-only) and without migration. The lightly colored portion of the bars represents the execution time of the benchmarks in the presence of one migration. The aggregate value of light and dark stacked bars presents the execution time without migration. Hence, the dark portions of the bars show the loss in speedup due to migration. The largest loss in speedup is 0.21 with FT on 16 nodes. This can be attributed to FT's relatively large migration overhead (8.5 seconds) compared to its rather short execution time (150 seconds). While the overhead increases proportionately to the memory footprint, the memory footprint is limited by system hardware (total memory size), which also limits the migration overhead. Hence, our results indicate an increasing potential for strong scaling of the benchmarks.

3.5.5 Page Access Pattern & Iterative Live Migration

Next, page access patterns of the NPB application are analyzed with respect to their effect on live migration, specifically on precopy duration and downtime. Figures 3.5(a) and 3.5(b) depict the page access pattern of CG and FT with class C inputs on 16 nodes. We sampled the number of memory pages modified per 0.1 second interval. The page size is 4KB on our experimental system. The write frequency to memory pages of CG is lower (∼393/interval) than that of FT (between 1000/interval and 5000/interval). Both codes show a certain regularity in their write patterns, albeit quite an individual one each.

To compare the iterative live migration with the page access pattern, we further evaluated the aggregate amount of the transferred memory and the duration of each iteration of a full memory sweep, as depicted in Figures 3.5(c) and 3.5(d). During the first iteration, all non-empty pages are transferred (depicted as dark gray in the figures). In subsequent iterations, only those pages modified since the last iteration (light gray) are transferred. We observed that the write patterns of Figures 3.5(a) and 3.5(b) are in accord with those depicted in Figures 3.5(c) and 3.5(d). For CG, memory pages written in the last two iterations account for 1.6MB corresponding to a write frequency of 393 pages. However, the dirtied memory in the second iteration is only 1.6MB while that in the first iteration is 62.2MB overall, even though the iteration durations are the same. This happens because the first iteration occasionally coincides with the initialization phase of the application. There, higher per-page memory access frequencies (2000-8000 per 0.1 second) persist combined with more frequent cold misses in cache. (This effect was experimentally confirmed but is not explicitly reflected in Figure 3.5(c)). For FT, 138MB and 384MB overall of dirtied memory was transferred in an alternating pattern during consecutive iterations (Figure 3.5(d)) corresponding to the most significant clusters at 1200 and 4600 page updates in Figure 3.5(b).

These profile patterns of the page modification rate could be archived and subsequently utilized as an empirical factor for future application runs/precopy decisions to choose a more sensitive termination condition (see Section 3.2). Besides the page access pattern, the communication pattern also affects the precopy decision. For example, if the application is both communication intensive and memory intensive, the opportunity for the precopy operation to complete before an imminent failure is low for high migration over-

head and significant network contention. Three main criteria for trading off live and frozen migration and for precopy termination conditions are:

- thresholds, *e.g.*, temperature watermarks in Section 3.3.1, memory/difference/overhead thresholds in Section 3.3.2;
- available network bandwidth determined by dynamic monitoring; and
- size of the write set. (If the memory dirtying rate is faster than the available network rate, the precopy operation may be ineffective.)

Based on these conditions, a heuristics algorithm can be designed. However, the applications we evaluated are not sufficient to design such an algorithm. In future work, we plan to create and assess applications with varying communication rate and memory access pattern that are more suitable.

3.5.6 Process-Level Live Migration vs. Xen Virtualization Live Migration

We next provide the performance comparison of our approach to another solution at the OS virtualization layer in the context of proactive FT of MPI applications [34]. The common benchmarks measured with both solutions on the same hardware were NPB BT, CG, LU and SP. For these NPB codes with class C inputs on 16 nodes, the overhead of migrating an entire guest OS ranged between 4-14 seconds under Xen virtualization while the process-level solution caused only around 1 second overhead. The time taken from initiating migration to actual completion on 16 nodes ranged between 14-24 seconds for live migration as opposed to a near-constant cost of 13-14 seconds for frozen migration (stop&copy), both under Xen virtualization. In contrast, our process-level solution only requires 2.6-6.5 seconds for live migration and 1-1.9 seconds for frozen (stop&copy-only) migration. The main difference between the two solutions is that the Xen virtualization solution induced a 13 second minimum overhead to transfer the entire memory image of the inactive guest VM (with a guest memory cap of 1GB) while the process-level solution constrained migration to only the memory of the respective process. Hence, our solution (with 1-6.5 seconds of prior warning to successfully trigger live process migration) significantly outperforms the Xen virtualization solution (with 13-24 seconds of prior warning). One could argue that

only a subset of the OS image needs to be migrated, yet the strength of virtualization lies in its transparency, yet it comes at the cost of indiscriminate transfer of the entire virtual memory range.

3.5.7 Proactive FT Complements Reactive FT

We claim that proactive FT complements its reactive sister. This was already empirically shown in Figure 3.4(h) where we noted that scalability of live migration depends on the amount of transferred memory. Once local memory is exhausted by an application, the overhead of a single migration will remain constant irrespective of the number of nodes. Of course, the rate of failures in larger systems is bound to increase, but proactive FT supports larger systems while reactive schemes result in increased I/O bandwidth requirements, which can become a bottleneck. This supports our argument that proactive schemes are important since they can complement reactive schemes in lowering checkpoint frequency requirements of the latter.

An analytical argument for the complementary nature is given next. The objective here is to assess the ability of proactive FT to successfully counter imminent faults, which subsequently allows reactive schemes to engage in less frequent checkpointing. Let the time interval between checkpoints be T_c, the time to save checkpoint information be T_s, and the MTBF be T_f. Then, the optimal checkpoint rate is $T_c = \sqrt{2 \times T_s \times T_f}$ [21]. We also observed that the mean checkpoint time (T_s) for BT, CG, FT, LU and SP with class C inputs on 4, 8 or 9 and 16 nodes is 23 seconds on the same experimental cluster [35]. With a MTBF of 1.25 hours [4], the optimal checkpoint rate T_c is $T_c = \sqrt{2 \times 23 \times (1.25 \times 60 \times 60)} = 455$ seconds. Let us further assume that 70% of failures can be predicted [26] and can thus be avoided by our proactive migration. (Sahoo *et al.* actually reported that up to 70% of failures can be predicted without prior warning; with a prior warning window, the number of proactively tolerated failures could even exceed 70%.) Our solution can then prolong reactive checkpoint intervals to $T_c = \sqrt{2 \times 23 \times (1.25/(1 - 0.7) \times 60 \times 60)} = 831$ seconds. The challenge with proactive FT then becomes (1) to provide a sufficient number of spare nodes to avoid initial failures by live or frozen migration and (2) to repair faulty nodes in the background such that jobs running over the course of days can reclaim failed nodes for future proactive FT.

3.6 Conclusion

We have contributed a novel *process-level live migration* mechanism with a concrete implementation as an enhancement to the Linux BLCR module and an integration within LAM/MPI. By monitoring the health of each node, a process can be migrated away from nodes subject to imminent failure. We show live migrations to be beneficial to frozen migration due to a lower overall overhead on application wall-clock execution time. This lower overhead is attributed to the asynchronous transfer of a large portion of a migrated process image while the application stays alive by proceeding with its execution. Process-level migration is also shown to be significantly less expensive than migrating entire OS images under Xen virtualization. The resulting proactive approach complements reactive checkpoint/restart schemes by avoiding roll-backs if failures are predicted only seconds before nodes cease to operate. Thus, the optimal number of checkpoints of applications can be nearly cut in half when 70% of failures are addressed proactively.

Chapter 4

Process-level Back Migration

4.1 Introduction

A migrated task could present a bottleneck due to (1) increased hop counts for communication from/to the spare node, (2) reduced resources in heterogeneous clusters (lower CPU/memory/network speed), or (3) placement of multiple MPI tasks on a node if not enough spare nodes are available. We contribute **back migration** as a novel methodology. We have implemented the back migration mechanism within LAM/MPI and BLCR based on process-level live migration in reverse direction. Our results indicate that we can benefit from back migration when, on average, 10.19% of execution is still outstanding just for our set of benchmarks. For larger applications, benefits are projected to occur for even smaller remaining amounts.

4.2 Design

To determine if we can benefit from back migration before actually migrating back an MPI task when the original node is recovered, the performance of MPI tasks across all nodes must be monitored. Systems, such as Ganglia [36], htop [37] and PAPI [38], monitor the performance of entire nodes rather than MPI task-specific performance or even timestep-specific metrics. Per-node measurements tend to be inaccurate as they fail to capture the "velocity" of an MPI job *before and after* task migration. In our design, we eliminate node-centric draw-backs through self-monitoring within each MPI task.

During a job's execution, MPI tasks record the duration of a timestep and relay this information to the same decentralized scheduler that we designed for migration trigger as shown in Figure 3.1. The scheduler compares the "velocity" of the MPI job *before and after* task migration to decide whether or not to migrate an MPI task back to the original node once this node is brought back online in a healthy state. The decision considers (a) the overhead of back migration and (b) the estimated time for the remaining portion of the job, which is also recorded for the MPI job and communicated between the job and the scheduler. We assume back migration overhead to be symmetric to the initial migration overhead.

4.3 Implementation

Our approach requires that MPI tasks perform self-monitoring of execution progress along timesteps. One solution is to provide an interposition scheme that intercepts selected runtime calls of an application at the PMPI layer similar to mpiP [39], a lightweight profiling library for MPI applications to collect statistical information about MPI functions. However, to estimate the benefit of the potential back migration, we must need information about the number of active MPI tasks, which the MPI standard does not natively support. Thus, our implementation provides an API with calls that can be directly added to MPI application code within the timestep loop. These called routines subsequently communicate with the decentralized scheduler to allow timestep-centric bookkeeping of an MPI task's progress. Our back migration approach can be divided into three stages described in the following.

Stage 1: Pre-Live Migration: Instrumentation is added around the time step loop of the MPI application to measure the overhead of each time step. This information is subsequently sent together with the bound on the total number of time steps and the current time integration number to the scheduler of the LAM daemon, which records the last information it received.

Stage 2: Live Migration: When a health problem is predicted to result in a future node failure so that live migration is triggered, the scheduler measures and records the migration overhead.

Stage 3: Post-Live Migration: Just as prior to live migration, the application and scheduler continue to assess the overhead of time steps. The scheduler further compares the difference in "velocity" of the MPI job *before and after* the migration and compares it with the migration overhead to decide whether or not to migrate an MPI task back to the original node once this node is brought back online in a healthy state.

4.4 Experimental Results

The experimental framework for back migration is the same as the one used for live migration in Chapter 3.

Results were obtained for NPB as depicted in Figure 4.1. Figure 4.1(a) shows the downtime fraction of live migration from the original node to a spare for different CPU frequencies at the destination node. For BT, FT, LU and SP, we observe an increase in downtime as the CPU frequency on the spare node decreases, which is expected as slower CPU frequencies result in longer migration. For CG, no obvious increasing behavior is observed since the main computation of CG is matrix multiplication, which is dominated by memory accesses and communication rather than CPU frequency-centric effects. The corresponding overheads of one time step per CPU frequency are depicted in Figure 4.1(b). Figures 4.1(c) and 4.1(d) depict the savings of back migration for BT and LU. Figures 4.1(e) and 4.1(f) depict the cross-over region of the previous figures indicating the minimal number of required time steps to benefit from back migration BT and LU are representative for the remaining NPB codes. The results for CG, FT and SP are also measured and assessed but not depicted here since they follow the same trends.

Benefits from the back migration are obtained when

$$R \times (T_d - T_o) - T_m > 0$$

which implies

$$R > T_m/(T_d - T_o)$$

where R is the number of remaining time steps of the benchmark, T_d is the overhead of one time step on the spare/destination node, T_o is the overhead of one time step of the benchmark on the original node, and T_m is the back-migration overhead (assumed to be symmetric to the initial migration overhead to the spare node). T_m plays an important role in the calculation of benefits when the number of remaining time steps (R) is low, which

leads to a cross-over point as depicted in Figures 4.1(e) and 4.1(f). Benefits due to migration increase as the CPU frequency on the spare node decreases when R is large enough to cover up the effect of T_m.

Table 4.1 summarizes the time steps required to obtain benefits for BT, CG, FT, LU and SP class C on 16 nodes. The number below the benchmark name is the total number of time steps of the benchmark. The results in the table indicate that benefits from back migration occur when as little as 1.33% and as much as 65% of the MPI job execute time remains to be executed, depending on the benchmark, for an average of 10.19% of outstanding execution time. These results are highly skewed by the short runtime duration of the NPB codes, particularly by FT, which has a comparatively short job execution time and high migration overhead. For larger applications, benefits are projected to occur for even smaller fractions of execution time.

Table 4.1: Minimal Time Steps (Percentage) Remained to Benefit from Back Migration

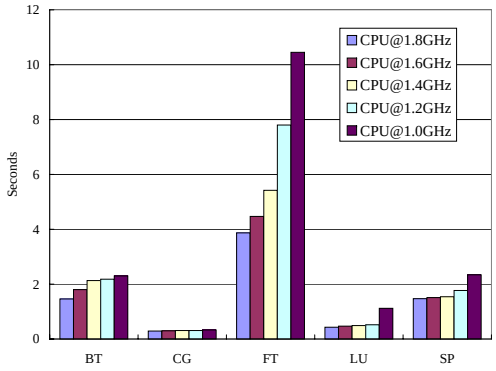
CPU Frequency of Spare Node	**BT (200)**	**CG (75)**	**FT (20)**	**LU (250)**	**SP (400)**
1.6GHz	14 (7%)	7 (9.3%)	13 (65%)	13 (5.2%)	26 (6.5%)
1.4GHz	13 (6.5%)	8 (10.67%)	4 (20%)	7 (2.8%)	9 (2.25%)
1.2GHz	6 (3%)	5 (6.67%)	5 (25%)	7 (2.8%)	9 (2.25%)
1.0GHz	3 (1.5%)	1 (1.33%)	4 (20%)	9 (3.6%)	10 (2.5%)

In general, the larger the amount of outstanding execution and the larger the performance difference between nodes, the higher the benefit due to back migration will be. These results illustrate a considerable potential of back migration particularly for large-scale clusters with heterogeneous nodes or multi-hop, non-uniform message routing. This is an aspect without studied investigation, to the best of our knowledge.

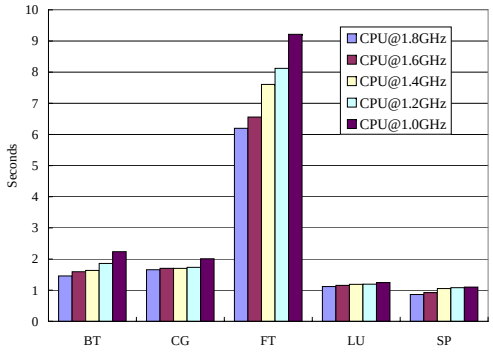
4.5 Conclusion

We contributed a mechanism to migrate an MPI task back to the original node once this node is brought back online in a healthy state to eliminate the potential load

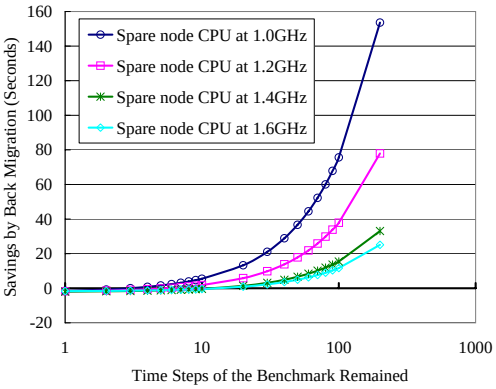
imbalance or bottlenecks caused by the initially task migrated, which is an unprecedented concept, to the best of our knowledge.



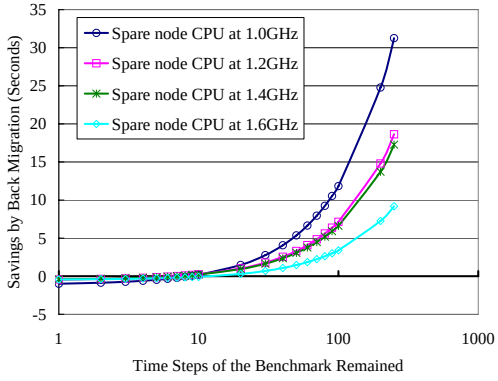
(a) Live Migration Downtime for NPB C-16



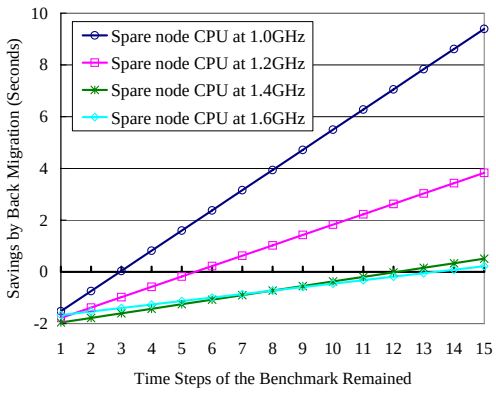
(b) Overhead of One Time Step for NPB C-16



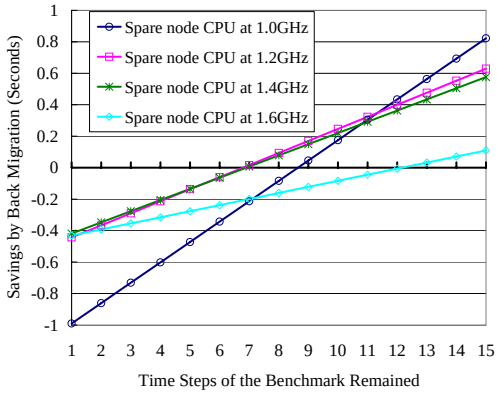
(c) Savings of Back Migration for BT C-16



(d) Savings of Back Migration for LU C-16



(e) Remaining Time Steps for BT C-16



(f) Remaining Time Steps for LU C-16

Figure 4.1: NPB Results for Back Migration (C-16: Class C on 16 Nodes)

Chapter 5

Hybrid Full/Incremental Checkpoint/Restart

5.1 Introduction

This Chapter contributes a hybrid full/incremental C/R solution to significantly reduce the size of the checkpoint file and the overhead of the checkpoint operations. Besides the original nodes allocated to an MPI job, it assumes the availability of spare nodes where processes (MPI tasks) may be relocated after a node failure. The Chapter further reduces the overhead of the restart operation due to roll-back after node failures, *i.e.*, restoration from full/incremental checkpoints on spare nodes, which only moderately increases the restart cost relative to a restore from a single, full checkpoint. After accounting for cost and savings, benefits due to incremental checkpoints significantly outweigh the loss on restart operations for our novel hybrid approach.

We conducted a set of experiments on an 18-node dual-processor (each dual core) Opteron cluster. We assessed the viability of our approach using the NAS Parallel Benchmark suite and mpiBLAST. Experimental results show that the overhead of our hybrid full/incremental C/R mechanism is significantly lower than that of the original C/R mechanism relying on full checkpoints. More specifically, experimental results indicate that the cost saved by replacing three full checkpoints with three incremental checkpoints is 16.64 seconds while the restore overhead amounts to just 1.17 for an overall savings of 15.47 seconds on average for the NAS Parallel Benchmark suite and mpiBLAST. The potential of

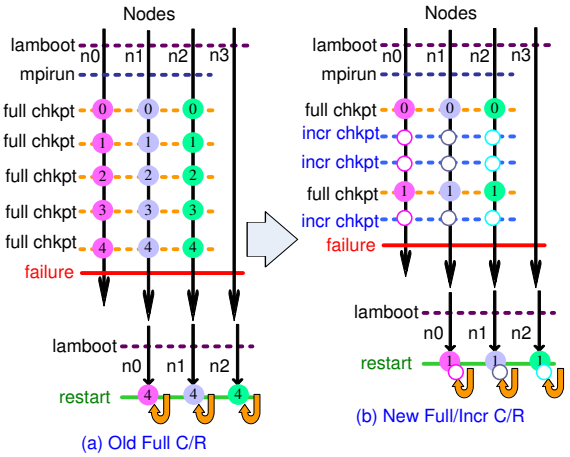


Figure 5.1: Hybrid Full/Incremental C/R Mechanism vs. Full C/R

savings due to our hybrid incremental/full C/R technique should even be higher in practice as (1) much higher ratios than just 1/3 for full/incremental checkpoints may be employed and (2) the amount of lost work would be further reduced if more frequent, lighter weight checkpoints were employed. Moreover, our approach can be easily integrated with other techniques, such as job-pause and migration mechanisms (discussed in Chapters 3 - 5) to avoid requeuing overhead by letting the scheduled job tolerate faults so that it can continue executing with spare nodes.

The remainder of this chapter is structured as follows. Section 5.2 presents the design of our hybrid full/incremental C/R mechanism. Section 5.3 identifies and describes the implementation details. Subsequently, the experimental framework is detailed and measurements for our experiments are presented in Sections 5.4 and 5.5, respectively. The work is then summarized in Section 5.6.

5.2 Design

This section presents an overview of the design of the hybrid full/incremental C/R mechanism with LAM/MPI and BLCR. We view incremental checkpoints as complementary to full checkpoints in the following sense. Every *n-th* checkpoint will be a full checkpoint to capture an application without prior checkpoint data while any checkpoints in between are incremental, as illustrated in Figure 5.1(b). Such process-based incremental checkpointing reduces checkpoint bandwidth and storage space requirements, and it leads to a lower rate of full checkpoints.

In the following, we first discuss the schedule of the full/incremental C/R. We then discuss system support for incremental checkpoints at two levels. First, the synchronization and coordination operations (such as the in-flight message drainage among all the MPI tasks to reach a consistent global state) at the job level are detailed. Second, dirty pages and related meta-data image information are saved at the process/MPI task level, as depicted in Figure 5.3. We employ filtering of "dirty" pages at the memory management level, *i.e.*, memory pages modified (written to) since the last checkpoint, which are utilized at node failures to restart from the composition of full and incremental checkpoints. Each component of our hybrid C/R mechanism is detailed next.

5.2.1 Scheduler

We designed a decentralized scheduler, which can be deployed as a stand-alone component or as an integral process of an MPI daemon, such as the LAM daemon (lamd). The scheduler will issue the full or incremental checkpoint commands based on user-configured intervals or the system environment, such as the execution time of the MPI job, storage constraints for checkpoint files and the overhead of preceding checkpoints.

Upon a node failure, the scheduler initiates a "job pause" mechanism in a coordinated manner that effectively freezes all MPI tasks on functional nodes and migrates processes of failed nodes [35]. All nodes, functional (paused ones) and migration targets (replaced failed ones), are restarted from the last full plus n incremental checkpoints, as explained in Section 5.2.5.

5.2.2 Incremental Checkpointing at the Job Level

Incremental Checkpointing at the Job Level is performed in a sequence of steps depicted in Figure 5.2 and described in the following.

Step 1: Incremental Checkpoint Trigger: When the scheduler decides to engage in an incremental checkpoint, it issues a corresponding command to the *mpirun* process, the initial LAM/MPI process at job invocation. This process, in turn, broadcasts the command to all MPI tasks.

Step 2: In-flight Message Drainage: Before we stop any process and save the remaining dirty pages and the corresponding process state in checkpoint files, all MPI

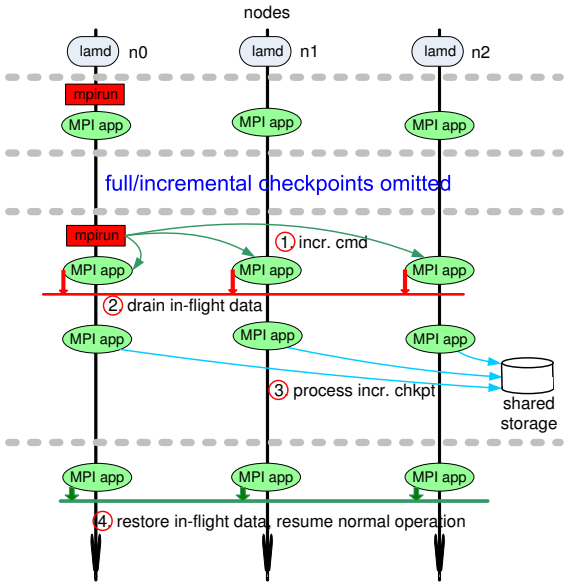


Figure 5.2: Incremental Checkpoint at LAM/MPI

tasks coordinate a consistent global state equivalent to an internal barrier. Based on our LAM/MPI+BLCR design, message passing is handled at the MPI level while the process-level BLCR mechanism is not aware of messaging at all. Hence, we employ LAM/MPI's job-centric interaction mechanism for the respective MPI tasks to clear in-flight data in the MPI communication channels.

Step 3: Process Incremental Checkpoint: Once all the MPI tasks (processes) reach a globally consistent state, all the MPI tasks perform the process-level incremental checkpoint operations independently, as discussed in Section 5.2.3.

Step 4: Messages Restoration and Job Continuation: Once the process-level incremental checkpoint has been committed, drained in-flight messages are restored, and all processes resume execution from their point of suspension.

5.2.3 Incremental Checkpointing at the Process Level

Incremental checkpointing of MPI tasks (step 3 in Figure 5.2) is performed at the process level, which is shown in detail in Figure 5.3. Compared to a full checkpoint, the incremental variant lowers the checkpoint overhead by saving only those memory pages modified since the last (full or incremental) checkpoint. This is accomplished via our BLCR enhancements by activating a handler thread (on right-hand side of Figure 5.3) that signals

compute threads to engage in the incremental checkpoint. One of these threads subsequently saves modified pages before participating in a barrier with the other threads, as further detailed in Section 5.3.2.

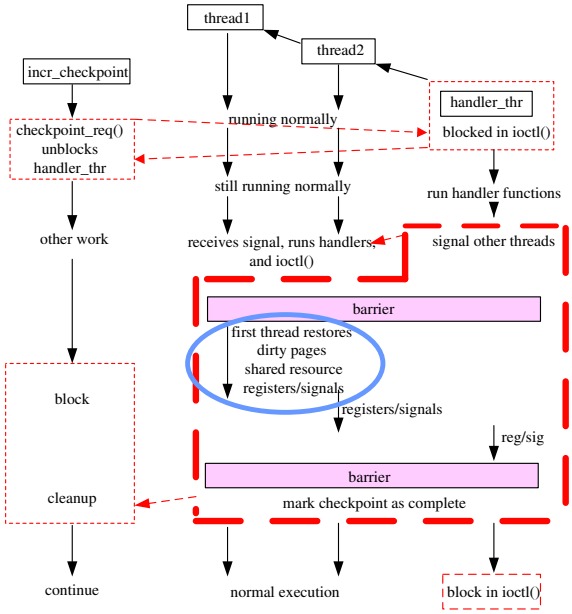


Figure 5.3: BLCR with Incremental Checkpoint in Bold Frame

A set of three files serve as storage abstraction for a checkpoint snapshot, as depicted in Figure 5.4:

1. *Checkpoint file a* contains the memory *page content*, *i.e.*, the data of only those memory pages modified since the last checkpoint.

2. *Checkpoint file b* stores memory *page addresses*, *i.e.*, address and offset of the saved memory pages for each entry in *file a*.

3. *Checkpoint file c* covers other *meta information*, *e.g.*, linkage of threads, register snapshots, and signal information pertinent to each thread within a checkpointed process / MPI task.

File a and *file b* maintain their data in a log-based append mode for successive incremental checkpoints. The last full and subsequent incremental checkpoints will only be

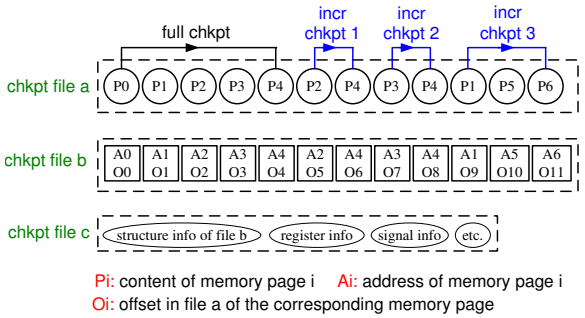


Figure 5.4: Structure of Checkpoint Files

discharged (marked for potential removal) once the next full checkpoint has been committed. Their availability is required for the potential restart up until a superseding checkpoint is written to stable storage. In contrast, only the latest version of *file c* is maintained since all the latest information is saved as one meta-data record, which is sufficient for the next restart.

In addition, memory pages saved in *file a* by an older checkpoint can be discharged once they are captured in a subsequent checkpoint due to page modifications (writes) since the last checkpoint. For example, in Figure 5.4, memory page 4 saved by the full checkpoint can be discharged when the first incremental checkpoint saves the same page. Later, the same page saved by the first incremental checkpoint can be discharged when it is saved by the second incremental checkpoint. In our on-going work, we are developing a garbage collection thread for this purpose. Similar to segment cleaning in log-structured file systems [40], the file is divided into segments (each of equal size as they represent memory pages) that are written sequentially. A separate garbage collection thread tracks these segments within the file, removes old segments (marked appropriately) from the end and puts new checkpointed memory data into the next segment. As a result, the file morphs into a large circular buffer as the writer thread adds new segments to the front and the cleaner thread removes old segments from the end toward the front (and then wraps around). Meanwhile, checkpoint *file b* is updated with the new offset information relative to *file a*.

5.2.4 Modified Memory Page Management

We utilize a Linux kernel-level memory management module that has been extended by a page-table dirty bit scheme to track modified pages between checkpoints as described in Chapter 3. This is accomplished by duplicating the dirty bit of the page-table entry (PTE) and extending kernel-level functions that access the PTE dirty bit so that the duplicate bit is set, which incurs negligible overhead (see Chapter 3 for details).

5.2.5 MPI Job Restart from Full+Incremental Checkpoints

Upon a node failure, the scheduler coordinates the restart operation on both the functional nodes and the spare nodes. First, the process of *mpirun* is restarted, which, in turn, issues the restart command to all the nodes for the MPI tasks. Thereafter, recovery commences on each node by restoring the last incremental checkpoint image, followed by the memory pages from the preceding incremental checkpoints in reverse sequence up to the pages from the last full checkpoint image, as depicted in Figure 5.5. The scan over all incremental checkpoints and the last full checkpoint allows the recovery of the last stored version of a page, *i.e.*, the content of any page only needs to be written once for fast restart. After process-level restart has been completed, drained in-flight messages are restored, and all the processes resume execution from their point of suspension. Furthermore, some pages saved in preceding checkpoints may be invalid (unmapped) in subsequent ones and need not be restored. The latest memory mapping information saved in *checkpoint file c* is also used for this purpose.

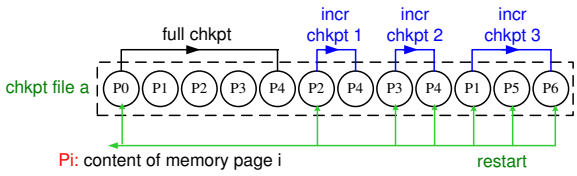


Figure 5.5: Fast Restart from Full/Incremental Checkpoints

5.3 Implementation Issues

Our hybrid full/incremental checkpoint/restart mechanism is currently implemented with LAM/MPI and BLCR. The overall design and implementation allows adaptation of this solution to arbitrary MPI implementations, such as MPICH and OpenMPI. Next, we present the implementation details of the full/incremental C/R mechanism, including the MPI-level communication/coordination realized within LAM/MPI and the process-level fundamental capabilities of BLCR.

5.3.1 Full/Incremental Checkpointing at the Job Level

We developed new commands *lam_full_checkpoint* and *lam_incr_checkpoint* to issue full and incremental checkpoint commands, respectively. The decentralized scheduler relays these commands to the *mpirun* process of the MPI job. Subsequently, *mpirun* broadcasts full/incremental checkpoint commands to each MPI tasks. At the LAM/MPI level, we also drain the in-flight data and reach to a consistent internal state before processes launch the actual checkpoint operation (see step 2 in Figure 5.2). We then restore the in-flight data and resume normal operation after the checkpoint operation of the process has completed (see step 4 in Figure 5.2).

5.3.2 Full/Incremental Checkpointing at the Process Level

We integrated several new BLCR features to extend its process-level checkpointing facilities, including the new commands *cr_full_checkpoint* and *cr_incr_checkpoint* to trigger full and incremental checkpoints at the process level within BLCR. Both of these commands write their respective portion of the process snapshot to one of the three files (see Section 5.2 and Figure 5.4).

Figure 5.3 depicts the steps involved in issuing an incremental checkpoint in reference to BLCR. Our focus is on the enhancements to BLCR (large dashed box). In the figure, time flows from top to bottom, and the processes and threads involved in the checkpoint are placed from right to left. Activities performed in the kernel are surrounded by dotted lines. A callback thread (right side) is spawned as the application registers a threaded callback and blocks in the kernel until a checkpoint has been committed. When *mpirun* invokes the newly developed *cr_incr_checkpoint* command extensions to BLCR, it provides

the process id as an argument. In response, the *cr_incr_checkpoint* mechanism issues an *ioctl* call, thereby resuming the callback thread that was previously blocked in the kernel. After the callback thread invokes the individual callback for each of the other threads, it reenters the kernel and sends a signal to each thread. These threads, in response, engage in executing the callback signal handler and then enter the kernel through another *ioctl* call.

Once in the kernel, the first thread saves the dirty memory pages modified since the last checkpoint. Then, threads take turns saving their register and signal information to the checkpoint files. After a final barrier, the process exits the kernel and enters user space, at which point the checkpoint mechanism has completed.

The command *cr_full_checkpoint* performs similar work, except that once the kernel is entered, the first thread saves all the non-empty memory pages rather than only the dirty ones.

5.3.3 Restart from Full+Incremental Checkpoints at Job and Process Levels

A novel command, *lam_fullplusincr_restart*, has been developed to perform the restart work at the job level with LAM/MPI. Yet another command, *cr_fullplusincr_restart*, has been devised to support the restart work at the process level within BLCR. In concert, the two commands implement the restart from the three checkpoint files and resume the normal execution of the MPI job as discussed in Section 5.2.

5.4 Experimental Framework

Experiments were conducted on a dedicated Linux cluster comprised of 18 compute nodes, each equipped with two AMD Opteron-265 processors (each dual core) and 2 GB of memory. The nodes are interconnected by two networks, both with 1 Gbps Ethernet. The OS used is Fedora Core 5 Linux x86_64 with our dirty bit patch as described in Section 5.2. We extended LAM/MPI and BLCR with our hybrid full/incremental C/R mechanism of this platform.

For all following experiments we use the MPI version of the NPB suite [18] (version 3.3) as well as mpiBLAST. NPB is a suite of programs widely used to evaluate the performance of parallel system, while the latter is a parallel implementation of NCBI BLAST,

which splits a database into fragments and distributes the query tasks to workers by query segmentation before the BLAST search is performed in parallel.

5.5 Experimental Results

Experiments were conducted to assess (a) overheads associated with the full and incremental checkpoints, (b) full and incremental checkpoint file size and memory checkpointed (which is the main source of the checkpointing overhead), (c) restart overheads associated with the full and incremental checkpoints, and (d) the relationship between checkpoint interval and checkpoint overhead.

Out of the NPB suite, the BT, CG, FT, LU and SP benchmarks were exposed to class C data inputs running on 4, 8 or 9 and 16 nodes, and to class D data inputs on 8 or 9 and 16 nodes. Some NAS benchmarks have 2D, others have 3D layouts for 2^3 or 3^2 nodes, respectively. The NAS benchmark EP is exposed to class C, D and E data inputs running on 4, 8 and 16 nodes. All the other NAS benchmarks were not suitable for our experiments since they execute for too short a period to be periodically checkpointed, such as IS, as depicted in Figure 5.7(a), or they have excessive memory requirement, such as the benchmarks with class D data inputs on 4 nodes.

Since the version of mpiBLAST we used assigns one process as the master and another to perform file output, the number of actual worker processes performing parallel input is the total process number minus two. Each worker process reads several database fragments. With our experiments, we set the mpiBLAST-specific argument *-use-virtual-frags*, which enables caching of database fragments in memory (rather than local storage) for quicker searches.

5.5.1 Checkpointing Overhead

The first set of experiments assesses the overhead incurred due to one full or incremental checkpoint. Figures 5.7(a), 5.8(a), 5.9(a) and 5.10(a) depict the base execution time of a job (benchmark) without checkpointing while Figures 5.7(b), 5.8(b), 5.9(b) and 5.10(b) depict the checkpoint overhead. As these results show, the checkpoint overhead is uniformly small relative to the overall execution time, even for a larger number of nodes. Prior work [35] already compared the overhead of full checkpointing with the base execution,

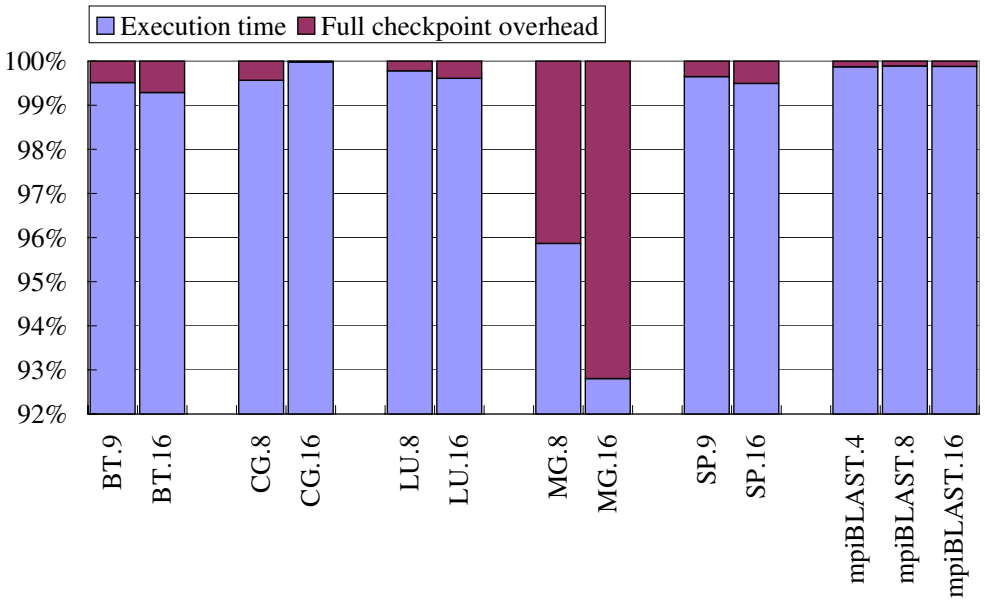
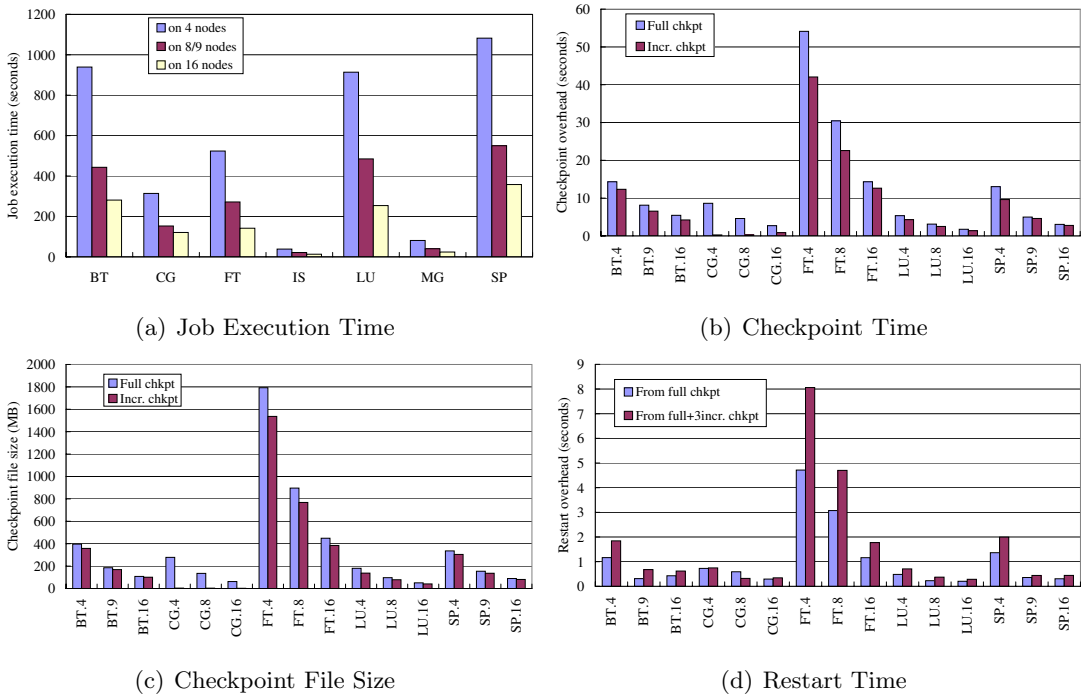


Figure 5.6: Full Checkpoint Overhead of NPB Class D and mpiBLAST

and the ratio is below 10% for most NPB benchmarks with Class C data inputs. Figure 5.6 depicts the measured overhead for single full checkpointing relative to the base execution time of NPB with Class D data inputs and mpiBLAST (without checkpointing). The ratio is below 1%, except for MG, as discussed in the following.

MG has a larger checkpoint overhead (large checkpoint file), but the ratio is skewed due to a short overall execution time (see Figure 5.8(a)). In practice, with more realistic and longer checkpoint intervals, a checkpoint would not be necessitated within the application's execution. Instead, the application would have been restarted from scratch. For longer runs with larger inputs of MG, the fraction of checkpoint/migration overhead would have been much smaller.

Figures 5.7(b), 5.8(b), 5.9(b) and 5.10(b) show that the overhead of incremental checkpointing is smaller than that of full checkpointing, so the overhead of incremental checkpointing is less significant. Hence, a hybrid full/incremental checkpointing mechanism reduces runtime overhead compared to full checkpointing throughout, i.e., under varying number of nodes and input sizes.



(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

(d) Restart Time

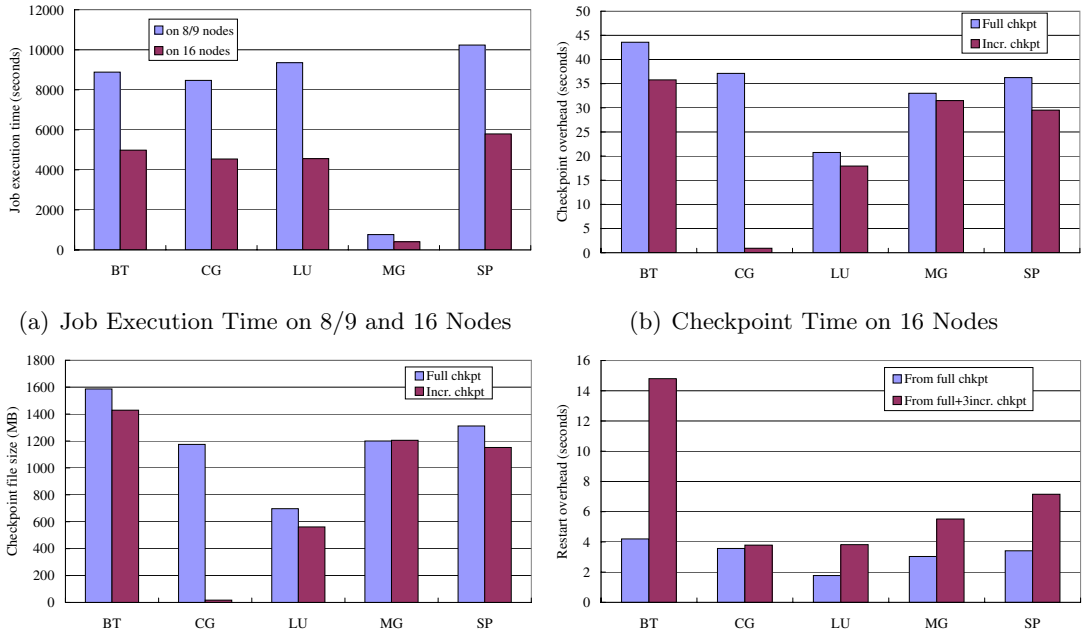
Figure 5.7: Evaluation with NPB Class C on 4, 8/9, and 16 Nodes

5.5.2 Checkpointing File Size

Besides overhead due to checkpointing, we assessed the actual footprint of the checkpointing file. Figures 5.7(c), 5.8(c), 5.9(c) and 5.10(c) depict the size of the checkpoint files for one process of each MPI application. Writing many files of such size to shared storage synchronously may be feasible for high-bandwidth parallel file systems. In the absence of sufficient bandwidth for simultaneous writes, we provide a multi-stage solution where we first checkpoint to local storage. After local checkpointing, files will be asynchronously copied to shared storage, an activity governed by the scheduler. This copy operation can be staggered (again governed by the scheduler) between nodes. Upon failure, a spare node restores data from the shared file system while the remaining nodes roll back using the checkpoint file on local storage, which results in less network traffic.

Overall, the experiments show that:

1. the overhead of full/incremental checkpointing of the MPI job is largely proportional to the size of the checkpoint file;



(a) Job Execution Time on 8/9 and 16 Nodes

(b) Checkpoint Time on 16 Nodes

(c) Checkpoint File Size on 16 Nodes

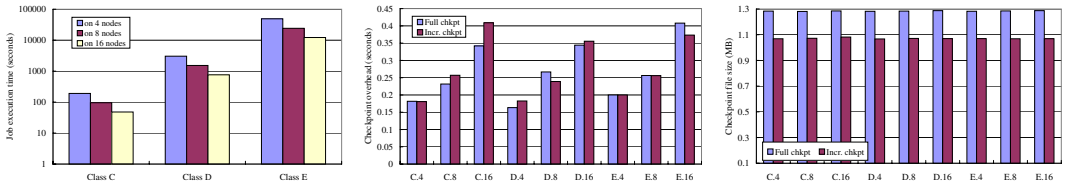
(d) Restart Time on 16 Nodes

Figure 5.8: Evaluation with NPB Class D

2. the overhead of full checkpointing is nearly the same at any time of the execution of the job;
3. the overhead of incremental checkpointing is nearly the same at any interval; and
4. the overhead of incremental checkpointing is lower than that of full checkpointing (except some cases of EP, which are lower than 0.45 seconds, which is excessively short. If required at this sort rate, one can employ full checkpointing only).

The first observation indicates that the ratio of communication overhead to computation overhead for C/R of the MPI job is relatively low. Since checkpoint files are, on average, large, the time spent on storing/restoring checkpoints to/from disk accounts for most of the measured overhead. This overhead is further reduced by the potential savings through incremental checkpointing.

For full/incremental checkpointing of EP (Figure 5.9(b)), incremental checkpointing of CG with Class C data inputs (Figure 5.7(b)) and incremental checkpointing of mpiBLAST (Figure 5.10(b)), the footprint of the checkpoint file is small (smaller than 13MB),



(a) Job Execution Time (b) Checkpoint Time (c) Checkpoint File Size

Figure 5.9: Evaluation with NPB EP Class C/D/E on 4, 8 and 16 nodes

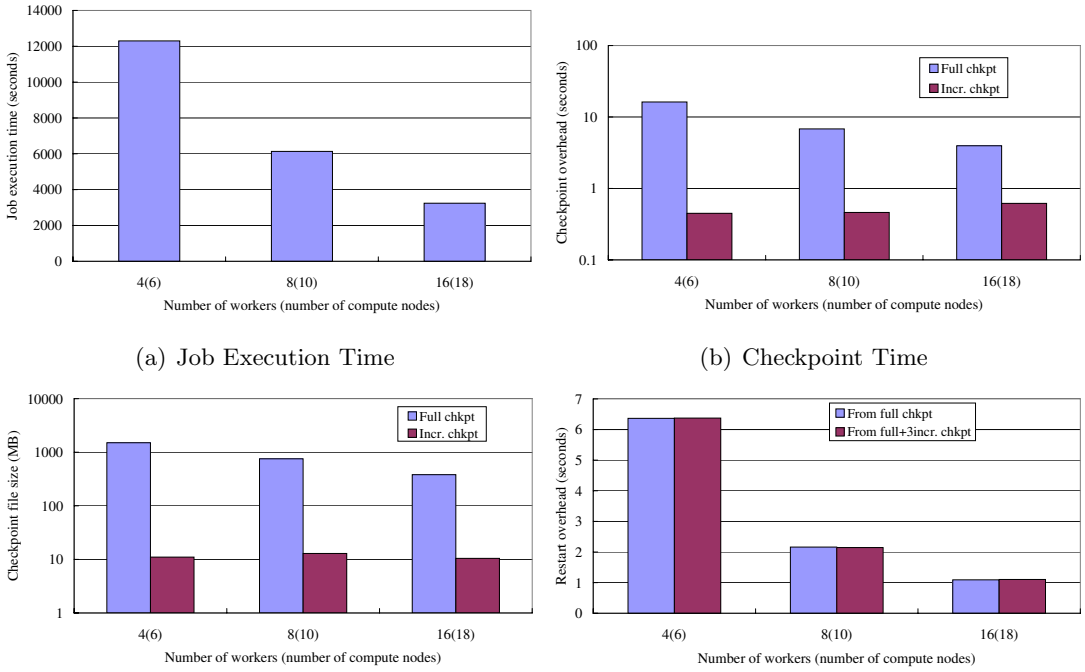
which results in a relatively small overhead. Thus, the checkpoint overhead mainly reflects the variance of the communication overhead inherent to the benchmark, which increases with the node count. However, the overall checkpoint overhead for these cases is smaller than 1 second. Hence, communication overhead of the applications did not significantly contribute to the overhead or interfere with checkpointing. This indicates a high potential of our hybrid full/incremental checkpointing solution to scale to larger clusters, and we have analyzed our data structures and algorithms to assure suitability for scalability. Due to a lack of large-scale experimentation platforms flexible enough to deploy our kernel modifications, new BLCR features and LAM/MPI enhancements, such larger scale experiments cannot currently be realized, neither at National Labs nor at larger-scale clusters within universities where we have access to resources.

The second observation about full checkpoint overheads above indicated that the size of the full checkpoint file remains stable during job execution. The benchmarks codes do not allocate or free heap memory dynamically within timesteps of execution; instead, all allocation is performed during initialization, which is typical for most parallel codes (except for adaptive codes [20]).

The third observation is obtained by measuring the checkpoint file size with different checkpoint intervals for incremental checkpointing, i.e., with intervals of 30, 60, 90, 120, 150 and 180 seconds for NPB Class C and intervals of 2, 4, 6, 8, 10 and 12 minutes for NPB Class D and mpiBLAST.

Thus, we can assume the time spent on checkpointing is constant. This assumption is critical to determine the optimal full/incremental checkpoint frequency.

The fourth observation verifies the superiority and justifies the deployment of our hybrid full/incremental checkpointing mechanism.



(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

(d) Restart Time

Figure 5.10: Evaluation with mpiBLAST

5.5.3 Restart Overhead

Figures 5.7(d), 5.8(d) and 5.10(d) compare the restart overhead of our hybrid full/incremental solution from one full checkpoint plus three incremental checkpoints with that of the original solution restarting from one full checkpoint. The results indicate that the wall clock time for restart from full plus three incremental checkpoints exceeds that of restart from one full checkpoint by 0-253% depending on the application, and it is 68% larger (1.17seconds) on average for all cases. The largest additional cost of 253% (10.6 seconds) was observed for BT under class D inputs for 16 nodes due to its comparatively large memory footprint of the incremental checkpointing. Yet, this overhead is not on the critical path as failures occur significantly less frequently than periodic checkpoints, *i.e.*, our hybrid approach reduces the cost along the critical path of checkpointing. For mpiBLAST and CG, the footprint of incremental checkpointing is comparatively so small that the overhead of restarting from full plus three incremental checkpoints is almost the

same as that of restarting from one full checkpoint. Yet, the time saved by three incremental checkpoints over three full checkpoints is 16.64 seconds on average for all cases. Even for BT under class D inputs for 16 nodes (which has the largest restart cost loss ratio), the saving is 23.38 seconds while the loss is 10.6 seconds. We can further extend the benefit by increasing the incremental checkpointing count between two full checkpoints.

We can also assess the accumulated checkpoint file size of one full checkpoint plus three incremental checkpoints, which is 185% larger than that of one full checkpoint. However, as just discussed, the overhead of restarting from one full plus three incremental checkpoint is only 68% larger. This is due to the following facts:

1. a page saved by different checkpoints is only restored once;
2. file reading for restarting is much faster than file writing for checkpointing; and
3. some pages saved in preceding checkpoints may be invalid and need not be restored at a later checkpoint.

5.5.4 Benefit of Hybrid Full/Incremental C/R Mechanism

Figure 5.11 depicts sensitivity results of the overall savings (the cost saved by replacing full checkpoints with incremental ones minus the loss on the restore overhead) for different number of incremental checkpoints between any adjacent full ones. Savings increase proportional to the number of incremental checkpoints (as the y axis in the figure is on a logarithmic base), but the amount of incremental checkpoints is still limited by stable storage capacity (without segment-style cleanup). The results are calculated by using the following formulae:

$$S_n = n \times (O_f - O_i) - (R_{f+n_i} - R_f)$$

where S_n is the saving with n incremental checkpoints between two full checkpoints, O_f is the full checkpoint overhead, O_i is the incremental checkpoint overhead, R_{f+n_i} is the overhead of restarting from full+n incremental checkpoints and R_f is the overhead of restarting from one full checkpoint. For mpiBLAST and CG, we may even perform only incremental checkpointing after the first full checkpoint is captured initially since the footprint of incremental checkpoints is so small that we will not run out of drive space at all (or, at least, for a very long time). Not only should a node failure be the exception over the set of all nodes, but the lower overhead of a single incremental checkpoint provides opportunities to

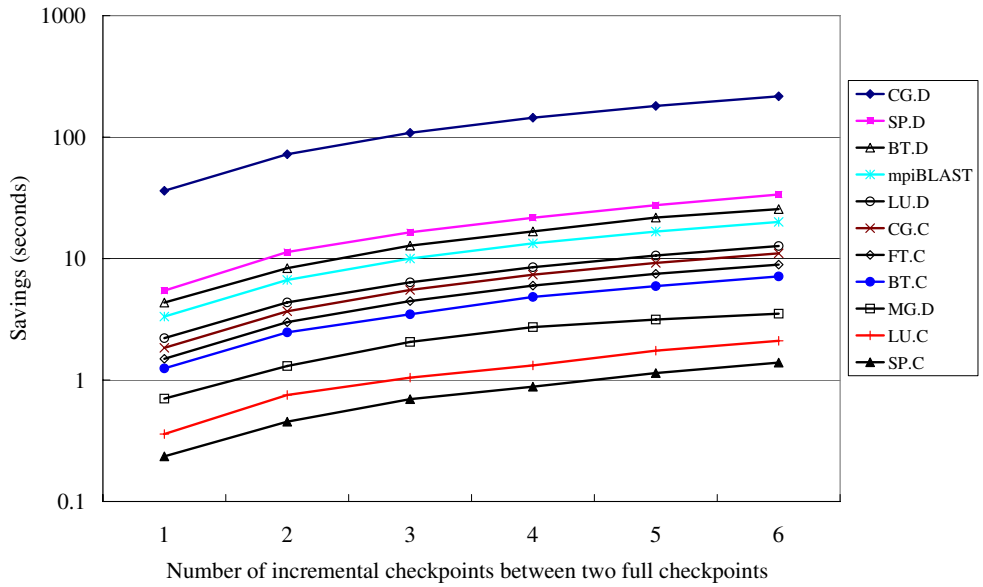


Figure 5.11: Savings of Hybrid Full/Incremental C/R Mechanism for NPB and mpiBlast on 16 Nodes

increase checkpoint frequencies compared to an application running with full checkpoints only. Such shorter incremental checkpoint frequencies reduce the amount of work lost when a restart is necessitated by a node failure. Hence, the hybrid full/incremental checkpointing mechanism effectively reduces the overall overhead relative to C/R.

Table 5.1 presents detailed measurements on the savings of incremental checkpointing, the overhead of restart from full plus incremental checkpoints, the relationship between the checkpoint file size and restart overhead, and the overall benefit from the hybrid full/incremental C/R mechanism. The benchmarks are sorted by the benefit. The table shows that (1) the cost caused by restart from one full plus one incremental checkpoints (which is R_{f+1_i} - R_f) is low, compared to the savings by replacing full checkpoints with incremental ones (which is O_f - O_i), and can be ignored for most of the benchmarks; (2) the restart cost is nearly proportional to the file size (except that some pages are checkpointed twice at both full and incremental checkpoints but later only restored once and thus lead to no extra cost); (3) for all the benchmarks, we can benefit from the hybrid full/incremental C/R mechanism, and the performance improvement depends on the memory access characteristics of the application.

Naksinehaboon *et al.* provide a model that aims at reducing full checkpoint over-

Table 5.1: Savings by Incremental Checkpoint vs. Overhead on Restart

Benchmarks	CG.D	SP.D	BT.D	mpiBLAST	LU.D	CG.C	FT.C	BT.C	MG.D	LU.C	SP.C
Savings of (O_f - O_i)	36.20	6.73	7.79	3.34	2.81	1.85	1.69	1.22	1.51	0.38	0.28
Restart overhead of (R_{f+1_i} - R_f)	0.03	1.28	3.45	0.01	0.59	0.01	0.20	-0.02	0.81	0.02	0.04
File increases caused by 1 incr. chkpt (MB)	17.26	1151.88	1429.14	10.45	561.46	2.10	384.41	100.67	1205.23	41.09	80.98
Benefit of hybrid C/R (S_1)	36.17	5.45	4.34	3.33	2.21	1.84	1.50	1.25	0.70	0.36	0.24

head by performing a set of incremental checkpoints between two consecutive full checkpoints [41]. They further develop a method to determine the optimal number of incremental checkpoints between full checkpoints. They obtain

$$m = \left\lceil \frac{(1-\mu) \times O_f}{P_i \times \delta} - 1 \right\rceil$$

where m is the number of incremental checkpoints between two consecutive full checkpoint, μ is the incremental checkpoint overhead ratio ($\mu = O_i/O_f$), P_i is the probability that a failure will occur after the second full checkpoint and before the next incremental checkpoint, and δ is additional recovery cost per incremental checkpoint. With the data from Table 5.1, we can determine

$$m = \left\lceil \frac{9.92}{P_i} - 1 \right\rceil$$

. Since $0 < P_i < 1$, a lower bound for m is 8.92, which indicates the potential for even more significant savings than just those depicted in Figure 5.11.

Overall, the overhead of the hybrid full/incremental C/R mechanism is significantly lower than the original periodical full C/R mechanism.

5.6 Conclusion

This work contributes a novel hybrid full/incremental C/R mechanism with a concrete implementation within LAM/MPI and BLCR with the following features: (1) It provides a dirty bit mechanism to track modified pages between incremental checkpoints; (2) only the subset of *modified* pages is appended to the checkpoint file together with page metadata updates for incremental checkpoints; (3) incremental checkpoints complement full checkpoints by reducing I/O bandwidth and storage space requirements while allowing lower

rates for full checkpoints; (4) a restart after a node failure requires a scan over all incremental checkpoints and the last full checkpoint to recover from the last stored version of a page, *i.e.*, the content of any page only needs to be written to memory once for fast restart; (5) a decentralized scheduler coordinates the full/incremental C/R mechanism among the MPI tasks. Results indicate that the performance of the hybrid full/incremental C/R mechanism is significantly lower than that of the original full C/R. For the NPB suite and mpiBLAST, the average savings due to replacing three full checkpoints with three incremental checkpoints is 16.64 seconds — at the cost of only 1.17 seconds if a restart is required after a node failure due to restoring one full plus three incremental checkpoints. Hence, the overall saving amounts to 15.47 seconds. Even more significant saving would be obtained if the rate of incremental checkpoints between two full checkpoints was increased. Our hybrid approach can further be utilized to (1) develop an optimal (or near-optimal) checkpoint placement algorithm, which combines full and incremental checkpoint options in order to reduce the overall runtime and application overhead; (2) create and assess applications with varying memory pressure to measure the tradeoff between full and incremental checkpoints and to provide heuristics accordingly; and (3) combine related job pause/live migration techniques with incremental checkpoints to provide a reliable multiple-level fault tolerant framework that incurs lower overhead than previous schemes. Overall, our hybrid full/incremental checkpointing approach is not only novel but also superior to prior non-hybrid techniques.

Chapter 6

Offline Reconstruction of Job Input Data

6.1 Introduction

As mentioned in Chapter 1, recovery operations are necessary to reconstruct pre-staged data, in the event of storage system failure before the job is scheduled. These operations are performed as periodic checks to ensure data availability for any given job. Further, the recovery is performed as part of system management even before the job is scheduled. Therefore, it is not charged against the users' compute time allocation. In the absence of reconstruction, user jobs are left to the mercy of traditional system recovery or put back in the queue for rescheduling.

To start with, this mode of recovery is made feasible due to the transient nature of job data and the fact that they have immutable, persistent copies elsewhere. Next, many high-performance data transfer tools and protocols (such as hsi [42] and GridFTP [43]) support partial data retrieval, through which disjoint segments of missing file data—specified by pairs of start offset and extent—can be retrieved. Finally, the costs of network transfer is decreasing much faster than that of disk-to-disk copy. These reasons, collectively, enable and favor offline recovery.

A staged job input data on the scratch parallel file system may have originated from the "/home" area or an HPSS archive [44] in the supercomputer center itself or from an end-user site. To achieve proactive data reconstruction, however, we need rich metadata

about data source and transfer protocols used for staging. In addition, we need sophisticated recovery mechanisms built into parallel file systems.

In the rest of this chapter, we describe the process of automatically extracting recovery metadata and instrumenting a parallel file system in Section 6.2. This provides a proof-of-concept implementation for our offline data recovery approach. In Sections 6.3 and 6.4, the experimental framework and results are presented, respectively. The work is subsequently summarized in Section 6.5.

6.2 Architecture

6.2.1 Metadata: Recovery Hints

To enable offline data recovery, we first propose to extend the parallel file system's metadata with *recovery information*, which can be intelligently used to improve fault tolerance and data/resource availability. Staged input data has persistent origins. Source data locations, as well as information regarding the corresponding data movement protocols, can be recorded as optional recovery metadata (using the extended attributes feature) on file systems. For instance, the location can be specified as a uniform resource index (URI) of the dataset, comprised of the protocol, URL, port and path (e.g., "http://source1/StagedInput" or "gsiftp://mirror/StagedInput"). In addition to URIs, user credentials, such as GSI (Grid Security Infrastructure) certificates, needed to access the particular dataset from remote mirrors can also be included as metadata so that data recovery can be initiated on behalf of the user. Simple file system interface extensions (such as those using extended attributes) would allow the capture of this metadata. We have built mechanisms for the recovery metadata to be automatically stripped from a job submission script's staging/offloading commands, facilitating transparent data recovery. One advantage of embedding such recovery-related information in file system metadata is that the description of a user job's data "source" and "sink" becomes an integral part of the transient dataset on the supercomputer while it executes. This allows the file system to recover elegantly without manual intervention from the end users or significant code modification.

6.2.2 Data Reconstruction Architecture

In this section, we present a prototype recovery manager. The prototype is based on the Lustre parallel file system [45], which is being adopted by several leadership-class supercomputers. A Lustre file system comprises of the following three key components: *Client*, *MDS* (MetaData Server) and *OSS* (Object Storage Server). Each OSS can be configured to host several *OST*s (Object Storage Target) that manage the storage devices (e.g., RAID storage arrays).

The reconstruction process is as follows. 1) Recovery hints about the staged data are extracted from the job script. Relevant recovery information for a dataset is required to be saved as recovery metadata in the Lustre file system. 2) The availability of staged data is periodically checked (i.e., check for OST failure). This is performed after staging and before job scheduling. 3) Data is reconstructed after OST failures. This involves finding spare OSTs to replace the failed ones, orchestrating the parallel reconstruction, fetching the lost data stripes from the data source, using recovery hints and ensuring the metadata map is up-to-date for future accesses to the dataset.

Extended Metadata for Recovery Hints: As with most parallel file systems, Lustre maintains file metadata on the MDS as inodes. To store recovery metadata for a file, we have added a field in the file inode named "recov", the value of which is a set of URIs indicating permanent copies of the file. The URI is a character string, specified by users during data staging. Maintaining this field requires little additional storage (less than 64 bytes) and minimal communication costs for each file. We have also developed two additional Lustre commands, namely *lfs setrecov* and *lfs getrecov* to set and retrieve the value of the "recov" field respectively, by making use of existing Lustre system calls.

I/O Node Failure Detection: To detect the failure of an OST, we use the corresponding Lustre feature with some extensions. Instead of checking the OSTs sequentially, we issue commands to check each OST in parallel. If no OST has failed, the probe returns quickly. Otherwise, we wait for a Lustre-prescribed timeout of 25 seconds. This way, we identify all failed OSTs at once. This check is performed from the head node, where the scheduler also runs.

File Reconstruction: The first step towards reconstruction is to update the stripe information of the target file, which is maintained as part of the metadata. Each

Lustre file f is striped in a round-robin fashion over a set of m_f OSTs, with indices $T_f = \{t_0, t_1, \ldots, t_{m_f}\}$. Let us suppose OST f_i is found to have failed. We need to find another OST as a substitute from the stripe list, T_f. To sustain the performance of a striped file access, it is important to have a distinct set of OSTs for each file. Therefore, in our reconstruction, an OST that is not originally in T_f will take OST f_i's position, whenever at least one such OST is available. More specifically, a list L of indices of all available OSTs is compared with T_f and an index $t_{i'}$ is picked from the set $L - T_f$. We are able to achieve this, since Lustre has a global index for each OST. The new stripe list for the dataset is $T'_f = \{t_0, t_1, \ldots, t_{i-1}, t_{i'}, t_{i+1}, \ldots, t_{m_f}\}$. When a client opens a Lustre file for the first time, it will obtain the stripe list from the MDS and create a local copy of the list, which it will use for subsequent read/write calls. In our reconstruction scheme, the client that locates the failed OST updates its local copy of the stripe and sends a message to the MDS to trigger an update of the master copy. The MDS associates a dirty bit with the file, indicating the availability of updated data to other clients. A more efficient method would be to let the MDS multicast a message to all clients that have opened the file, instructing them to update their local copies of the stripe list. This is left as future work.

When the new stripe list T'_f is generated and distributed, storage space needs to be allocated on OST $t_{i'}$ for the data previously stored on OST t_i. As an object-based file system [46], Lustre uses objects as storage containers on OSTs. When a file is created, the MDS selects a set of OSTs that the file will be striped on and creates an object on each one of them to store a part of the file. Each OST sends back the id of the object that it creates and the MDS collects the object ids as part of the stripe metadata. The recovery manager works in the same way, except that the MDS only creates an object in OST $t_{i'}$. The id of the object on the failed OST t_i is replaced with the id of the object created on OST $t_{i'}$. The length of an object is variable [46] and, therefore, the amount of data to be patched is not required to be specified at the time of object creation.

In order to patch data from a persistent copy and reconstruct the file, the recovery manager needs to know which specific byte ranges are missing. It obtains this information by generating an array of $\{offset, size\}$ pairs according to the total number of OSTs used by this file, m_f, the position of the failed OST in the stripe list, i, and the stripe size, $ssize$. Stripe size refers to the number of data blocks. Each $\{offset, size\}$ pair specifies a chunk of data that is missing from the file. Given the round-robin striping pattern used by Lustre,

it can be calculated that for a file with size $fsize$, each of the first $k = fsize \bmod ssize$ OSTs will have $\lceil \frac{fsize}{ssize} \rceil$ stripes and each of the other OSTs will have $\lfloor \frac{fsize}{ssize} \rfloor$ stripes. For each OST, it can be seen that in the j_{th} pair, $offset = j \times ssize$, and in each pair except the last one, $size = ssize$. If the OST has the last stripe of the file, then $size$ will be smaller in the last pair.

The recovery manager then acquires the URIs to remote permanent copies of the file from the "recov" field of the file inode, as we described above. Then, it establishes a connection to the remote data source, using the protocol specified in the URI, to patch each chunk of data in the array of $\{offset, size\}$ pairs. These are then written to the object on OST $t_{i'}$. We have built mechanisms so that the file reconstruction process can be conducted from the head node or from the individual OSS nodes (to exploit parallelism), depending on transfer tool and Lustre client availability.

Patching Session: At the beginning of each patching operation, a session is established between the patching nodes (the head node or the individual OSS nodes) and the data source. Many data sources assume downloads occur in an interactive session that includes authentication, such as GridFTP [43] servers using UberFTP client [47] and HPSS [44] using hsi [42]. In our implementation, we use Expect [48], a tool specifically geared towards automating interactive applications, to establish and manage these interactive sessions. We used Expect so that authentications and subsequent partial retrieval requests to the data source can be performed over a single stateful session. This implementation mitigates the effects of authentication and connection establishment by amortizing these large, one-time costs over multiple partial file retrieval requests.

6.3 Experimental Setup

In our performance evaluation, we assess the effect of storage node failures and its subsequent data reconstruction overhead on a real cluster using several local or remote data repositories as sources of data staging.

Our testbed for evaluating the proposed data reconstruction approach is a 40-node cluster at Oak Ridge National Laboratory (ORNL). Each machine is equipped with a single 2.0GHz Intel Pentium 4 CPU, 768 MB of main memory, with a 10/100 Mb Ethernet interconnection. The operating system is Fedora Core 4 Linux with a Lustre-patched kernel

(version 2.6.12.6) and the Lustre version is 1.4.7.

Since our experiments focus on testing the server-side of Lustre, we have setup the majority of the the cluster nodes as I/O servers. More specifically, we assign 32 nodes to be OSSs, one node to be the MDS and one node to be the client. The client also doubles as the head node. In practice, such a group of Lustre servers will be able to support a fairly large cluster of 512-2048 compute nodes (with 16:1-64:1 ratios seen in contemporary supercomputers).

We used three different data sources to patch pieces of a staged input file: (1) An NFS server at ORNL that resides outside the subnet of our testbed cluster ("Local NFS") (2) an NFS server at North Carolina State University ("Remote NFS") and (3) a GridFTP [49] server ("GridFTP") with a PVFS [50] backend on the TeraGrid Linux cluster at ORNL, outside the Lab's firewall, accessed through the UberFTP client interface [47].

6.4 Performance of Transparent Input Data Reconstruction

As illustrated in Section 6.2, for each file in question, the reconstruction procedure can be divided into the following steps. 1) The failed OSTs are determined by querying the status of each OST. 2) The file stripe metadata is updated after replacing the failed OST with a new one. 3) The missing data is patched in from the source. This involves retrieving the URI information of the data source from the MDS, followed by fetching the missing data stripes from the source and subsequently patching the Lustre file to complete the reconstruction. These steps are executed sequentially and atomically as one transaction, with concurrent accesses, to the file in question, protected by file locking. The costs of the individual steps are independent of one another, *i.e.*, the cost of reconstruction is precisely the sum of the costs of the steps. Below we discuss the overhead of each step in more detail.

In the first step, all the OSTs are checked in parallel. The cost of this check grows with the number of OSTs due to the fact that more threads are needed to check the OSTs. Also, the metadata file containing the status of all OSTs will be larger, which increases the parsing time. This step induces negligible network traffic by sending small status checking messages. The timeout to determine if an OST has failed is the default value of 25 seconds in our experiments. Shorter timeouts might result in false positives.

Figure 6.1 shows the results of benchmarking the cost of step 1 (detecting OST

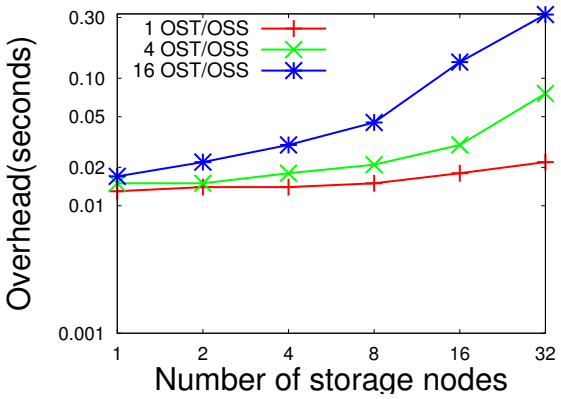


Figure 6.1: Cost of finding failed OSTs

failures) when there are no such failures. In other words, this is the cost already observed in the majority of file accesses, where data unavailability is not an issue. In this group of tests, we varied the number of OSTs per I/O node (OSS), which in reality is configured by system administrators. High-end systems tend to have multiple OSTs per OSS, medium-sized or smaller clusters often choose to use local disks and have only one OST per OSS. From Figure 6.1, it can be seen that the overhead of step 1 is fairly small with a moderate number of OSTs. In most cases, the step 1 cost is under 0.05 seconds. However, this overhead grows sharply as the number of OSTs increases to 256 (16 OSTs on each of the 16 OSSs) and 512 (16 OSTs on each of the 32 OSSs). This can be attributed to the network congestion caused by the client communicating with a large number of OSTs in a short span. Note that the current upper limit of OSTs allowed by Lustre is 512, which incurs an overhead of 0.3 seconds, which is very small considering this is a one-time cost for input files only.

The second recovery step has constant cost regardless of the number of OSTs as the only parties involved in this operation are the *MDS* and the *client* that initiates the file availability check (in our case, the head node). In our experiments, we also assessed this overhead and found it to be in the order of milliseconds. Further, this remains constant regardless of the number of OSSs/OSTs. The cost due to multiple *MDS* (normally two for Lustre) is negligible.

The overhead of the third recovery step is expected to be the dominant factor in the reconstruction cost when OST failures do occur. Key factors contributing to this cost

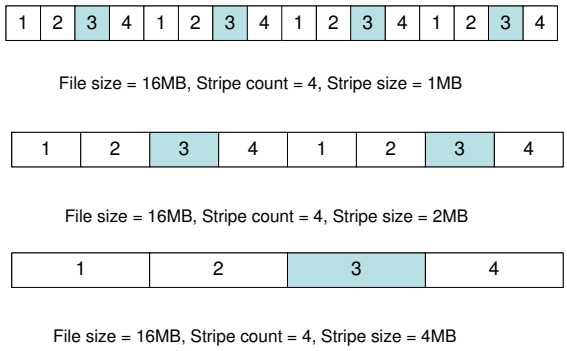
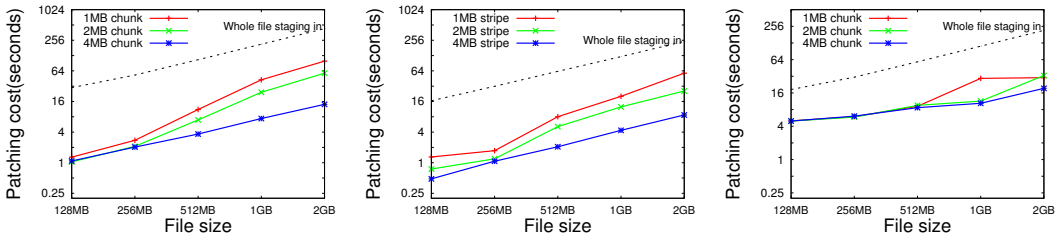


Figure 6.2: Round-robin striping over 4 OSTs



(a) Remote NFS (b) Local NFS (c) GridFTP

Figure 6.3: Patching costs for single OST failure from a stripe count 32. Filesize/32 is stored on each OST and that is the amount of data patched. Also shown is the cost of staging the entire file again, instead of just patching a single OST worth of data.

are the amount of data and the layout of missing data in the file. It is well known that non-contiguous access of data will result in lower performance than sequential accesses. The effect of data layout on missing data is more significant if sequential access devices such as tape drives are used at the remote data source. More specifically, factors that may affect patching cost include the size of each chunk of missing data, the distance between missing chunks, and the offset of the first missing chunk. Such a layout is mostly determined by the striping policy used by the file. To give an example, Figure 6.2 illustrates the different layout of missing chunks when OST 3 fails. The layout results from different striping parameters, such as the *stripe width* (called "stripe count" in Lustre terminology) and the *stripe size*. The shaded boxes indicate missing data stripes (chunks).

We conducted multiple sets of experiments to test the data patching cost using a variety of file striping settings and data staging sources. Figures 6.3(a)-(c) show the first

group of tests, one per data staging source (Remote NFS, Local NFS, and GridFTP). Here, we fixed the stripe count (stripe width) at 32 and increased the file size from 128MB to 2GB. Three popular stripe sizes were used with 1MB, 2MB, and 4MB chunks as the stripe unit, respectively. To inject a failure, one of the OSTs is manually disconnected so that 1/32 of the file data is missing. The *y-axis* measures the total patching time *in log scale*. For reference, a dotted line in each figure shows the time it takes to stage-in the whole file from the same data source (in the absence of our enhancements).

We notice that the patching costs from NFS servers are considerably higher with small stripe sizes (*e.g.*, 1MB). As illustrated in Figure 6.2, a smaller stripe size means the replacement OST will retrieve and write a larger number of non-contiguous data chunks, which results in higher I/O overhead. Interestingly, the patching costs from the GridFTP server remains nearly constant with different stripe sizes. The GridFTP server uses PVFS, which has better support than NFS for non-contiguous file accesses. Therefore, the time spent obtaining the non-contiguous chunks from the GridFTP server is less sensitive to smaller stripe sizes. However, both reconstructing from NFS and GridFTP utilize standard POSIX/Linux I/O system calls to seek in the file at the remote data server. In addition, there is also seek time to place the chunks on the OST. A more efficient method would be to directly rebuild a set of stripes on the native file system of an OST, which avoids the seek overhead that is combined with redundant read-ahead caching (of data between missing stripes) when rebuilding a file at the Lustre level. This, left as future work, should result in nearly uniform overhead approximating the lowest curve regardless of stripe size.

Overall, we have found that the file patching cost scales well with increasing file sizes. For the NFS servers, the patching cost using 4MB stripes is about 1/32 of the entire-file staging cost. For the GridFTP server, however, the cost appears to be higher and less sensitive to the stripe size range we considered. The reason is that GridFTP is tuned for bulk data transfers (GB range) using I/O on large chunks, preferably tens or hundreds of MBs [51]. Also, the GridFTP performance does improve more significantly, compared with other data sources, as the file size increases.

Figures 6.4 and 6.5 demonstrate the patching costs with different stripe counts. In Figure, 6.4 we increase the stripe count at the same rate as the file size to show how the patching cost from the local NFS server varies. The amount of missing data due to an OST failure is $\frac{1}{stripe\_count} \times file\_size$. Therefore, we patch the same amount of data for

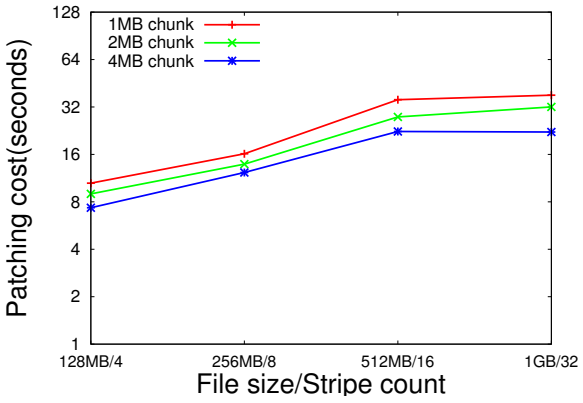
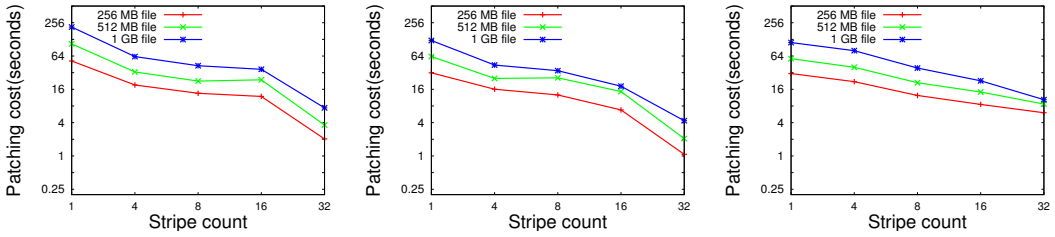


Figure 6.4: Patching from local NFS. Stripe count increases with file size. One OST fails and its data is patched.



(a) Patching from remote NFS server (b) Patching from local NFS server (c) Patching from GridFTP server

Figure 6.5: Patching costs with stripe size 4MB

each point in the figure. It can be seen that the patching cost grows as the file size reaches 512MB and remains constant thereafter. This is caused by the fact that missing chunks in a file are closer to each other with a smaller stripe count. Therefore, when one chunk is accessed from the NFS server, it is more likely for the following chunks to be read ahead into the server cache. With larger file sizes (512MB or more), distances between the missing chunks are larger as well. Hence, the server's read-ahead has no effect. Figure 6.5 shows the results of fixing the stripe size at 4MB and using different stripe counts. Note that a stripe count of 1 means transferring the entire file. From this figure, we can see how the cost of reconstructing a certain file decreases as the file is striped over more OSTs.

6.5 Conclusion

We have proposed a novel way to reconstruct transient, job input data in the face of storage system failure. From a user's standpoint, our techniques help reduce the job turnaround time and enable the efficient use of the precious, allocated compute time. From a center's standpoint, our techniques improve serviceability by reducing the rate of resubmissions due to storage system failure and data unavailability. By automatically reconstructing missing input data, a parallel file system will greatly reduce the impact of storage system failures on the productivity of both supercomputers and their users.

Chapter 7

On-the-fly Recovery of Job Input Data

7.1 Introduction

In HPC settings, data and I/O availability is critical to center operations and user serviceability. Petascale machines require 10,000s of disks attached to 1,000s of I/O nodes. Plans for 100k to 1M disks are being discussed in this context. The numbers alone imply severe problems with reliability. In such a setting, failure is inevitable. I/O failure and data unavailability can have significant ramifications to a supercomputer center at large. For instance, an I/O node failure in a parallel file system (PFS) renders portions of the data inaccessible resulting in either application stalling on I/O or being forced to be resubmitted and rescheduled.

Upon an I/O error, the default behavior of file systems is to simply propagate the error back to the client. Usually, file systems do little beyond providing diagnostics so that the application or the user may perform error handling and recovery. For applications that go through rigid resource allocation and lengthy queuing to execute on Petascale supercomputers, modern parallel file systems' failure to mask storage faults appears particularly expensive.

Standard hardware redundancy techniques, such as RAID, only protect against entire disk failures. Latent sector faults (occurring in 8.5% of a million disks studied [52]), controller failures, or I/O node failures can render data inaccessible even with RAID. Failover

strategies require spare nodes to substitute the failed ones, an expensive option with thousands of nodes. It would be beneficial to address these issues *within the file system* to provide graceful, transparent, and portable data recovery.

HPC environments provide unique fault-tolerance opportunities. Consider a typical HPC workload. Before submitting a job, users stage in data to the scratch PFS from end-user locations. After the job dispatch (hours to days later) and completion (again hours or days later), users move their output data off the scratch PFS (*e.g.*, to their local storage). Thus, job input and output data seldom need to reside on the scratch PFS beyond a short window before or after the job's execution. Specifically, key characteristics of job *input* data are their being (1) transient, (2) immutable, and (3) redundant in terms of a remote source copy.

We propose *on-the-fly data reconstruction* during job execution. We contribute an application-transparent extension to the widely used Lustre parallel file system [45], thereby adding reliability into the PFS by shielding faults at many levels of an HPC storage system from the applications. With our mechanism, a runtime I/O error (EIO) captured by the PFS instantly triggers the recovery of missing pieces of data and resolves application requests immediately when such data becomes available.

Such an approach is a dramatic improvement in fault handling in modern PFSs. At present, an I/O error is propagated through the PFS to the application, which has no alternative but to exit. Users then need to re-stage input files if necessary and resubmit the job. Instead of resource-consuming I/O node failover or data replication to avoid such failures, our solution does not require additional storage capacity. Only the missing data stripes residing on the failed I/O node are staged again from their original remote location. Exploiting Lustre's two-level locks, we have implemented a two-phase blocking protocol combined with delayed metadata updates that allows unrelated data requests to proceed while outstanding I/O requests to reconstructed data are served in order, as soon as a stripe becomes available. Recovery can thus be overlapped with computation and communication as stripes are recovered. Our experimental results reinforce this by showing that the increase in job execution time due to on-the-fly recovery is negligible compared to non-faulting runs.

Consider the ramifications of our approach. From a center standpoint, I/O failures traditionally increase the overall *expansion factor*, *i.e.*, *(wall_time + wait_time)/wall_time* averaged over all jobs (the closer to 1, the better). Many federal agencies (DOD, NSF, DOE)

are already requesting such metrics from HPC centers. From a user standpoint, I/O errors result in dramatically increased turnaround time and, depending on already performed computation, a corresponding waste of resources. Our method significantly reduces this waste and results in lower expansion factors.

The remainder of this chapter is structured as follows. Section 7.2 presents the design of the on-the-fly recovery mechanism. Section 7.3 identifies and describes the implementation details. Subsequently, the experimental framework is detailed and measurements for our experiments are presented in Sections 7.4 and 7.5, respectively. The work is then summarized in Section 7.6.

7.2 On-the-fly Recovery

The overarching goal of this work is to address file systems' fault tolerance when it comes to serving HPC workloads. The following factors weigh in on our approach.

(1) Mitigate the effects of I/O node failure: An I/O node failure can adversely affect a running job by causing it to fail, being requeued or exceeding time allocation, all of which impacts the HPC center and user. Our solution promotes continuous job execution that minimizes the above costs. *(2) Improve file system response to failure:* File system response to failure is inadequate. As we scale to thousands of I/O nodes and few orders of magnitude more disks, file systems need to be able to handle failure gracefully. *(3) Target HPC workloads:* The transient and immutable nature of job input data and its persistence at a remote location present an unique opportunity to address data availability in HPC environments. We propose to integrate fault tolerance into the PFS specifically for HPC I/O workloads. *(4) Be inclusive of disparate data sources and protocols:* HPC users use a variety of storage systems and transfer protocols to host and move their data. It is desirable to consider external storage resources and protocols as part of a broader I/O hierarchy. *(5) Be transparent to client applications:* Applications are currently forced to explicitly handle I/O errors or to simply ignore them. We promote a recovery scheme widely transparent to the application. *(6) Performance:* For individual jobs, on-the-fly recovery should impose minimal overhead on existing PFS functionality. For a supercomputing center, it should improve the overall job throughput compared to requeuing the job.

7.2.1 Architectural Design

To provide fault tolerance to PFS, the on-the-fly recovery component should be able to successfully trap I/O error of a system call resulting from I/O node failure. In a typical parallel computing environment, parallel jobs are launched on the numerous compute nodes (tens of thousands), and each one of those processes on the compute nodes perform I/O. Figure 7.1 depicts the overall design. Each compute node can act as a client to the parallel file system. Upon capturing an I/O error from any of these compute nodes, data recovery is set in motion. The calling process is blocked, and so is any other client trying to access the same unavailable data. The recovery process consults the MDS of the PFS to obtain remote locations where persistent copies of the job input data reside. (We discuss below how this metadata is captured.) It then creates the necessary objects to hold the data stripes that are to be recovered. Using the recovery metadata, remote patching is performed to fetch the missing stripes from the source location. The source location could be "/home", or an HPSS archive in the same HPC center, or a remote server. The patched data is stored in the PFS, and the corresponding metadata for the dataset in question is updated in the MDS. More specifically, missing stripes are patched in the client request order. Subsequently, blocked processes resume their execution as data stripes become available. Thus, the patching of missing stripes not yet accessed by the client is efficiently overlapped with client I/O operations to significantly reduce overhead.

7.2.2 Automatic Capture of Recovery Metadata

To enable on-demand data recovery, we extend the PFS's metadata with recovery information. Staged input data has persistent origins. Source data locations, as well as information regarding the corresponding data movement protocols, are recorded as optional recovery metadata (using the extended attributes feature) on file systems. Locations are specified as a URI of the dataset comprised of the protocol, URL, port and path (e.g., http://source1/StagedInput or gsiftp://mirror/StagedInput). Simple file system interface extensions (*e.g.*, extended attributes) capture this metadata. We have built mechanisms for the recovery metadata to be automatically stripped from a job submission script's staging commands for *offline* recovery [53] that we utilize here for *online* recovery. By embedding such recovery-related information in file system metadata, the description of a user job's

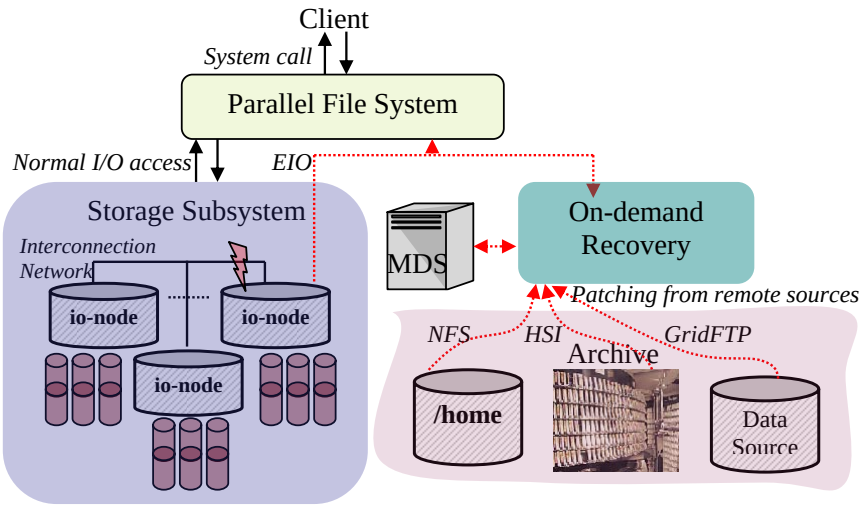


Figure 7.1: Architecture of on-the-fly recovery

data source and sink becomes an integral part of the transient dataset on the supercomputer while it executes. User credentials, such as GSI certificates, may be needed to access the particular dataset from remote mirrors. These credentials can also be included as file metadata so that data recovery can be initiated on behalf of the user.

7.2.3 Impact on Center and User

Performance of online recovery requires further analysis. PFS at contemporary HPC centers can support several Gbps of I/O rate. However, this requires availability of all data and absence of failures in the storage subsystem. When faced with a RAID recoverable failure (e.g., an entire disk failure), file systems perform in either "degraded" or "rebuild" mode, both of which incur perceivable performance losses [54]. In cases where standard hardware-based recovery is not feasible, the only option is to trigger an application failure.

As application execution progresses, the performance impact (and potential waste of resources) due to failures increases resulting also in substantially increased turnaround time when a job needs to be requeued. These aspects also impact overall HPC center serviceability.

On-the-fly recovery offers a viable alternative in such cases. With ever increasing network speeds, HPC centers' connectivity to high-speed links, highly tuned bulk transport

protocols are extremely competitive. For instance, ORNL's Leadership Class Facility (LCF) is connected to several national testbeds like TeraGrid (a 10Gbps link), UltrascienceNet, Lambda Rail, etc. Recent tests have shown that a wide-area Lustre file system over the TeraGrid from ORNL to Indiana University can offer data transfer speeds of up to 4.8 Gbps [55] for read operations bringing remote recovery well within reach.

Depending on how I/O is interspersed in the application, remote recovery has different merits. The majority of HPC scientific applications conduct I/O in a burst fashion by performing I/O and computation in distinct phases. These factors can be exploited to overlap remote recovery with computation and regular I/O requests. Once a failure is recognized and recovery initiated, the recovery process can patch other missing stripes of data that will eventually be requested by the application and not just the ones already requested. Such behavior can improve recovery performance significantly.

At other times, however, we may not be able to overlap recovery efficiently. In such cases, instead of consuming compute time allocation, a job might decide that being requeued is beneficial, thereby compromising on turnaround time. Thus, a combination of factors, such as I/O stride, time already spent on computation, cost of remote recovery and a turnaround time deadline, can be used to decide if and when to conduct remote data reconstruction. Nonetheless, the cause of I/O errors needs to be rectified before the next job execution. Although this is beyond the scope of this chapter, we have built the basis for a dynamic cost-benefit analysis. Our experiments analyze results and discuss their affect on job turnaround time in light of on-the-fly recovery.

7.3 Implementation

In this section, we illustrate how on-the-fly recovery has been implemented in Lustre FS. Should a storage failure occur due to an OSS or OST failure, the original input data can be replenished from the remote data source by reconstructing unavailable portions of files.

In supercomputers, remote I/O is usually conducted through the head or service nodes and, therefore, these nodes are likely candidates for the initiation of recovery. In our implementation, the head node of a supercomputer doubles as a recovery node and has a Lustre client installed on it. It schedules recovery in response to the requests received

from the compute nodes, which observe storage failures upon file accesses. The head node serves as a coordinator that facilitates recovery management and streamlines reconstruction requests in a consistent and non-redundant fashion. Figure 7.2 depicts the recovery scenario. Events annotated by numbers happen consecutively in the indicated order resulting in four distinct phases.

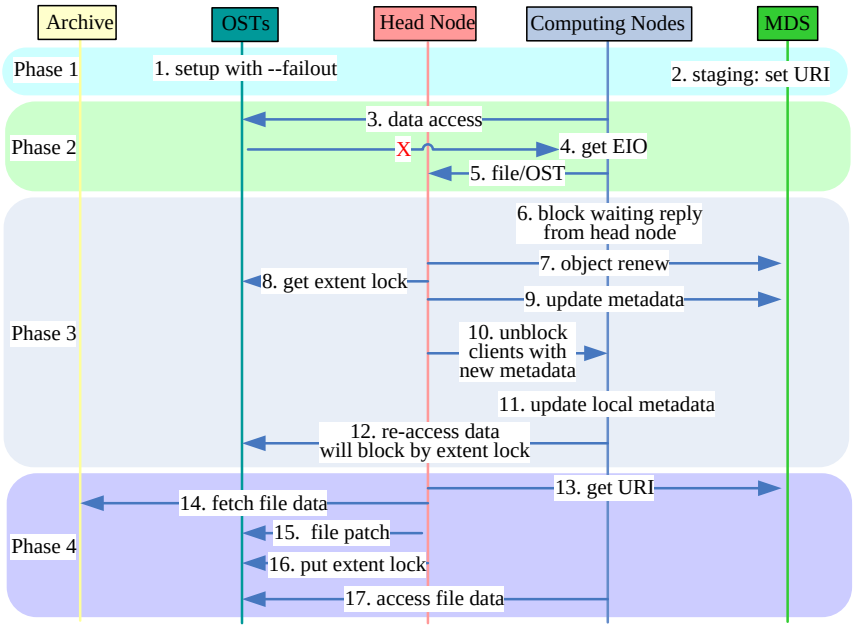


Figure 7.2: Steps for on-the-fly recovery

7.3.1 Phase 1: PFS Configuration and Metadata Setup

For on-the-fly recovery, the client needs to capture the OST failure case immediately. Hence, we configure all OSTs in Lustre's "fail-out" mode (step 1 of Figure 7.2). Thus, any operation referencing a file with a data stripe on a failed OST results in an immediate I/O error without ever blocking. In step 2, we further extend the metadata of the input files (at the MDS) with recovery information indicating the URI of a file's original source upon staging (see [53]).

7.3.2 Phase 2: Storage Failure Detection at Compute Nodes

To access the data of a file stored in the OST, the application issues calls *via* the standard POSIX file system API. The POSIX API is intercepted by the Lustre patched VFS system calls.

Due to the fail-out mode, both I/O node and data disk failures will lead to an immediate I/O error at the client upon file access (steps 3 and 4). By capturing the I/O error in the system function, we obtain file name and index of the failed OST or, in case of a disk failure, the location of the affected OST. In step 5, the client sends relevant information (file name, OST index) to the head node, which, in turn, initiates the data reconstruction. Hence, we perform online/real-time failure detection at the client for on-the-fly recovery during application execution, much in contrast to prior work on offline recovery that dealt with data loss prior to job activation [53].

7.3.3 Phase 3: Synchronization between Compute and Head Nodes

Upon receiving the data reconstruction request from the client, the head node performs two major tasks. First, it sends a request to the MDS, which locates a spare OST to replace the failed one and creates a new object for the file data on this spare. It next fetches the partial file data from the data source and populates the new object on the spare OST with it. When multiple compute nodes (Lustre clients) access the same data of this file, the head node only issues one reconstruction request per file per OST (even if multiple requests were received). At this point, compute nodes cannot access the object on the new OST as the data has not been populated. Once a stripe becomes available, compute nodes may access them immediately. To support such semantics, synchronization between the clients and OSTs is required. The fundamental mechanism for such synchronization is provided by Lustre locks.

Lustre Intent/Extent Lock Basics: Lustre provides two levels of locking, namely intent and extent locks. Intent locks arbitrate metadata requests from clients to MDS. Extent locks protect file operations on actual file data. Before modifying a file, an extent lock must be acquired. Each OST accommodates a lock server managing locks for stripes of data residing on that OST.

Synchronization Mechanism: We have implemented a centralized coordinator, a daemon residing on the head node. It consists of multiple threads that handle requests from clients and perform recovery. Upon arrival of a new request, the daemon launches the recovery procedure while the client remains blocked, just as other clients requesting data from this file/OST (step 6). Data recovery (step 7) is initiated by a novel addition to Lustre, the (*lfs objectrenew*) command. In response, the MDS locates a spare OST (on which the file does not reside yet) and creates a new object to replace the old one. Note that the MDS will not update its metadata information at this time. Instead, the update is deferred lazily to step 9 to allow accesses to proceed if they do not concern the failed OST.

In step 8, the daemon acquires the extent lock for the stripes of the new object. Since the (new) object information is hidden from other clients, there cannot be any contention for the lock. In step 9, the metadata information is updated, which utilizes the intent mechanism provided by Lustre again. In step 10, clients waiting for the patched data are unblocked and the new metadata is piggybacked. After clients update their locally cached metadata (step 11), they may already reference the new object. However, any access to the new object will still be blocked (step 12), this time due to their attempt to acquire the extent lock, which is still being held by the daemon on the head node.

Adjustment of the OST Extent Lock Grant Policy: In step 8, the daemon requests extent locks for all stripes of the recovery object. Consider the example in Fig. 7.3. Extent locks for stripes 2, 6, 10 and 14 are requested from OST 5. Upon a request for stripe 2, OST 5 grants the largest possible extent ([0,-1] where -1 denotes ∞) to the daemon. Afterward, requests for stripes 6, 10 and 14 match with lock [0,-1] resulting in an incremented reference count of the lock at the client without communicating with OST 5.

Our design modifies this default behavior of coarse-granular locking. We want to ensure that the extent lock to the stripes will be released one-by-one immediately after the respective stripe is patched. However, with Lustre distributed lock manager (DLM), the daemon only decrements the reference count on lock [0, -1] and releases it after all the stripes are patched.

To address this shortcoming, we adjust the extent lock grant policy at the OST server. Instead of granting the lock of [0,-1], a request from the daemon on the head node is granted only the exact range of stripes requested. This way, extent locks for different stripes differ (in step 8). Also, once a stripe is patched, the respective lock can be released so that

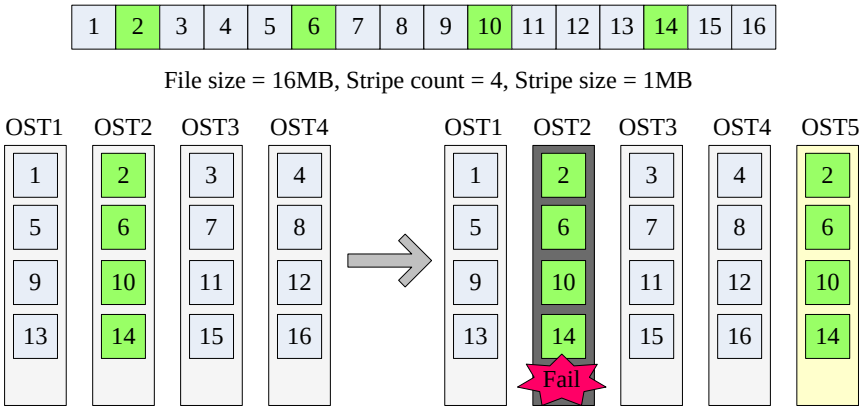
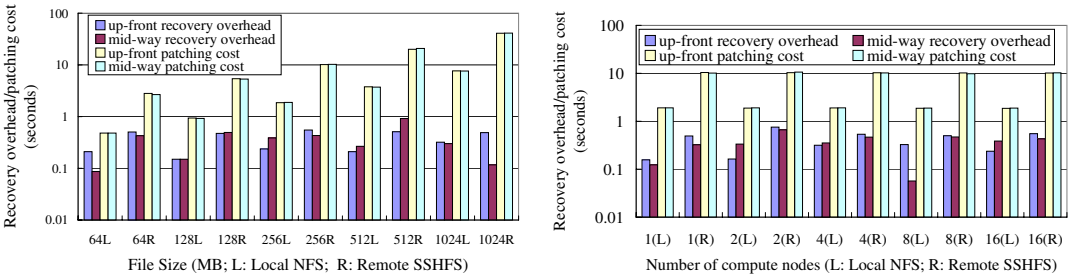


Figure 7.3: File reconstruction

other clients can access the patched data right away. Meanwhile, clients blocked on other stripes to be patched remain blocked on the extent locks. The extent lock policy is only updated for requests from the daemon on the head node without impacting the requests from other clients. Thus, it imposes no penalty in the non-failure case.

Such *metadata update delay* and *two-phase blocking of clients* provides the following properties: 1) Before any metadata update, clients can either access their cached data (which is consistent since stagein data is immutable) or request recovery (upon an I/O error). Either way, clients may still access the stripes of the old objects, but the new objects remain invisible to them until the head node has patched the data and notified the clients to update the metadata. 2) Before patching the actual file data, the head node obtains an extent lock for all stripes of the new object, thereby blocking other clients that access the data now or later. 3) After patching the data, the extent locks per stripe are immediately released so that other clients can access partial data (stripes). Meanwhile, the daemon continues to patch subsequent stripes to provide pipelined overlap between patching and application progress. 4) The extent lock is further utilized for the second phase of blocking. Thus, data patching becomes an independent task that can be offloaded to the OSSs to distribute the patching workload in a scalable manner. 5) An OSS failure only affects a subset of the computing nodes (the Lustre clients) even though all the clients participate in the parallel I/O operations. Furthermore, most of the affected clients are blocked by the extent locks without any communication with the centralized coordinator on the head node,

as discussed previously. Hence, the approach scales as communication with the centralized coordinator is limited to few nodes.



(a) Varied file size

(b) Varied # compute nodes

Figure 7.4: Matrix multiplication recovery overhead

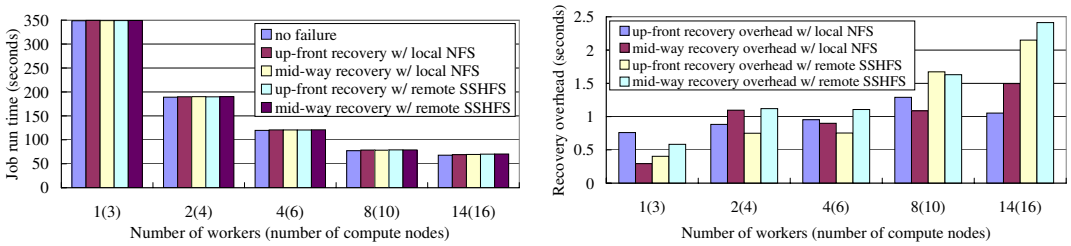
7.3.4 Phase 4: Data Reconstruction

In step 13, the URI of the remote file is obtained. In steps 14 and 15, stripes on the new object are populated. Due to per-stripe extent locks, stripes may be patched in any order. In our implementation, the clients subjected to I/O errors will supply the file range to access in their reconstruction request to the head node. The head node retains the order of the stripe requests and patches them accordingly. This speeds up application progress during reconstruction, particularly when files are accessed sequentially and a failure occurs in the middle of reading a file. In contrast, request-ignorant patching would hamper application progress by initiating a patch starting with the lowest indexed stripe of an OST, even though this stripe has already been read by clients.

To this end, we have implemented a new Lustre command, *lfs patch*. Since phase 3 already obtains the extent lock for all the stripes, the new command can update the data range directly. Also, we set the file position in the patch system function instead of invoking lseek() at the user level. This allows us to bypass the overhead associated with automatic read-ahead (due to VFS caching). The extent lock for each stripe is released immediately after patching so that clients can access the stripe instantly (step 16).

7.4 Experimental Framework

We used the same 17-node linux cluster at NCSU as our testbed. The OS on each node was Fedora Core 5 Linux x86_64 with a Lustre-patched RHEL5 2.6.18 Linux kernel (Lustre 1.6.3). In our experiments, the cluster nodes were setup as I/O servers, compute nodes (Lustre clients), or both, as indicated below. We used different data staging sources for the job input data: (1) "/home" on the local NFS file system at the same HPC center with patching cost at 34.41MB/sec; (2) a server at another campus accessed by a file system client, SSHFS, based on Filesystems in Userspace (FUSE) and secure shell with a patching cost of 6.31MB/sec. Other patching sources, *e.g.*, GridFTP servers, might incur further delay. However, since most of the patching cost is shown to be overlapped with computation or I/O operations, changes in patching cost remain largely hidden from applications.



(a) Job run time

(b) Recovery overhead

Figure 7.5: mpiBLAST performance

7.5 Experimental Results

We assessed overhead and patching cost of on-the-fly recovery using an MPI benchmark and an MPI application.

7.5.1 Performance of Matrix Multiplication

We first assessed an MPI kernel that performs dense matrix multiplication (MM) with the standard $C = A \times B$ matrix operations, where A, B and C are $n \times n$ matrices. A and B are stored consecutively in an input file. We vary n to manipulate the size of

the input file. Only one MPI task (the master) reads the input file before broadcasting the data to all the other tasks (workers). The matrix product $A \times B$ is distributed to all MPI processes. Since input occurs early during execution and since the code is more compute intensive, we focus on the recovery overhead, *i.e.*, the difference in job execution time of the jobs with and without failure.

Figure 7.4(a) shows the experimental results of matrix multiplication for increasing matrix dimensions, n (totaling 64MB, 128MB, 256MB, 512MB and 1GB). The MPI job runs on 16 compute nodes (one MPI task each). Figure 7.4(b) depicts the experimental results for varying number of compute nodes (1, 2, 4, 8 and 16) and a 256MB data input. For both of these tests, the *stripe count* (stripe width) for the input file was 4 and the *stripe size* was 1MB. We configured 5 OSTs (1OST/OSS) with the file residing on 4 OSTs and the spare OST for reconstruction. Some nodes double as both I/O and compute nodes. Since the configuration is the same, both with or without our solution, this provides a fair test environment.

To assess our system's capability to handle random storage failures, we varied the point in time where a failure occurred. In one experiment, we failed one of the OSTs up front, right as the MPI job started to run. This resulted in the master MPI task to experience an I/O error upon its first data access to the failed OST. In another experiment, we failed one OST mid-way during job execution. The master captures the I/O error immediately and sends a recovery request for the lost data to the daemon on the head node. Figures 7.4(a) and 7.4(b) indicate that the recovery overhead, from an application standpoint, is below 0.8 seconds for all cases. This is consistent in the sense that patching is overlapped with job I/O and hidden from the application. However, the actual time overlap between the patching and the job I/O varies. The recovery overhead for both up-front and mid-way recovery ranges from 0.06 to 0.75 seconds. Although the reconstruction cost in Figure 7.4(a) rises with file size, this is hidden from the application. While the patching cost from remote SSHFS is ~ 5 times that of local NFS, the recovery overhead for jobs patching from remote SSHFS is only slightly higher than local patching. The increase is dominated by the patching of the first stripe, which cannot be overlapped; subsequent stripes incur little extra cost.

7.5.2 Performance of mpiBLAST

We also assessed the performance of our solution using the mpiBLAST benchmark, a parallel implementation of NCBI BLAST, which splits a database into fragments and distributes the query tasks to workers by query segmentation before the BLAST search is performed in parallel.

Since mpiBLAST is more input-intensive, we discuss the impact of failure on the overall performance. Figure 7.5(a) shows the job run time. Figure 7.5(b) depicts the recovery overhead. mpiBLAST assigns one process to perform file output and another to schedule search tasks. Hence, the number of actual workers is the number of all the MPI processes minus two. Each worker accesses several files.

We configured 9 OSTs and increased compute nodes from 3 to 16 so that some double as server nodes (since our testbed has a total of 17 nodes). We distributed each input file to four of the OSTs by the Lustre stripe distribution policy and then failed one OST. As the number of worker processes increases, more files need to be accessed, *i.e.*, more files reside on the failed OST and require recovery so that the recovery overhead also increases (see Figure 7.5(b)). The number of failed files grows at the same rate as the workers. Compared to the overall runtime, the increase in recovery overhead is moderate. This is due to (1) parallel recovery of failed files referenced by disjoint workers and (2) reduced per-file patching cost for more workers as file sizes decrease due to work sharing. Figure 7.5(b) shows that the recovery overhead for jobs patching from remote SSHFS is higher than for local patching due to the slower data source. Also, with more workers, more failed files exist. Consequently, recovery becomes more costly, yet at a moderate growth rate due to the aforementioned overlap. For the benchmarks we used, such moderate recovery overhead is negligible compared with the job runtime. We expect that the same holds true for most supercomputing jobs as large jobs tend to run much longer and as input files are typically only read in the job initialization phase. Wallclock time estimates generally cover such negligible overhead. Hence, additional time need not be budgeted for the job due to our techniques.

7.6 Conclusion

We have presented the design of a novel on-the-fly recovery framework as a means to address fault tolerance within parallel file systems in HPC centers. The recovery framework provides a seamless way for a running job's input data to be reconstructed from its remote source in case of I/O errors. We have designed the system to take advantage of key characteristics of HPC I/O workloads such as their immutable input data, sequential access and persistent remote copy. We have further implemented this design into the Lustre parallel file system commonly used in supercomputer centers. Results with I/O-intensive MPI benchmarks suggest that the recovery mechanism imposes little overhead. Both HPC centers and users stand to benefit from improved serviceability, data availability and reduced job turnaround time in the face of storage system failure.

Chapter 8

Temporal Replication of Job Input Data

8.1 Introduction

Currently, the majority of disk failures are masked by hardware solutions such as RAID [13]. However, it is becoming increasingly difficult for common RAID configurations to hide disk failures as disk capacity is expected to grow by 50% each year, which increases the reconstruction time. The reconstruction time is further prolonged by the "polite" policy adopted by RAID systems to make reconstruction yield to application requests. This causes a RAID group to be more vulnerable to additional disk failures during reconstruction [14].

According to recent studies [56], disk failures are only part of the sources causing data unavailability in storage systems. RAID cannot help with storage node failures. In next-generation supercomputers, thousands or even tens of thousands of I/O nodes will be deployed and will be expected to endure multiple concurrent node failures at any given time. Consider the Jaguar system at Oak Ridge National Laboratory, which is on the roadmap to a petaflop machine (currently No. 5 on the Top500 list with 23,412 cores and hundreds of I/O nodes). Our experience with Jaguar shows that the majority of whole-system shutdowns are caused by I/O nodes' software failures. Although parallel file systems, such as Lustre [45], provide storage node failover mechanisms, our experience with Jaguar again shows that this configuration might conflict with other system settings. Further, many supercomputing centers hesitate to spend their operations budget on replicating I/O

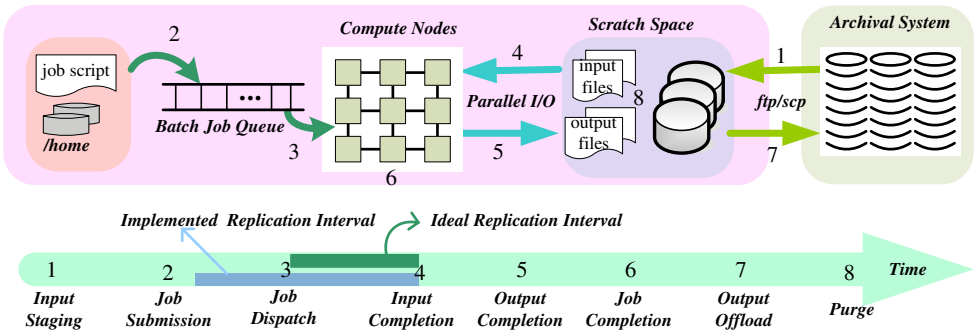


Figure 8.1: Event timeline with ideal and implemented replication intervals

servers and instead of purchasing more FLOPS.

Figure 8.1 gives an overview of an event timeline describing a typical supercomputing job's data life-cycle. Users stage their job input data from elsewhere to the scratch space, submit their jobs using a batch script, and offload the output files to archival systems or local clusters. For better space utilization, the scratch space does not enforce quotas but purges files after a number of days since the last access. Moreover, job input files are often read-only (also read-once) and output files are write-once.

Although most supercomputing jobs performing numerical simulations are output-intensive rather than input-intensive, the input data availability problem poses two unique issues. First, input operations are more sensitive to server failures. Output data can be easily redirected to survive runtime storage failures using *eager offloading* [57]. As mentioned earlier, many systems like Jaguar do not have file system server failover configurations to protect against input data unavailability. In contrast, during the output process, parallel file systems can more easily skip failed servers in striping a new file or perform restriping if necessary. Second, loss of input data often brings heavier penalty. Output files already written can typically withstand temporary I/O server failures or RAID reconstruction delays as job owners have days to perform their stage-out task before the files are purged from the scratch space. Input data unavailability, on the other hand, incurs job termination and resubmission. This introduces high costs for job re-queuing, typically orders of magnitude larger than the input I/O time itself.

Fortunately, unlike general-purpose systems, in supercomputers we can anticipate *future* data accesses by checking the job scheduling status. For example, a compute job is

only able to read its input data during its execution. By coordinating with the job scheduler, a supercomputer storage system can selectively provide additional protection only for the duration when the job data is expected to be accessed.

We proposed temporal file replication, wherein a parallel file system performs transparent and temporary replication of job input data. This facilitates fast and easy file reconstruction before and during a job's execution without additional user hints or application modifications. Unlike traditional file replication techniques, which have mainly been designed to improve long-term data persistence and access bandwidth or to lower access latency, the temporal replication scheme targets the enhancement of short-term data availability centered around job executions in supercomputers.

We have implemented our scheme in the popular Lustre parallel file system and combined it with the Moab job scheduler by building on our previous work on coinciding input data staging alongside computation [53]. We have also implemented a replication-triggering algorithm that coordinates with the job scheduler to delay the replica creation. Using this approach, we ensure that the replication completes in time to have an extra copy of the job input data before its execution.

We then evaluate the performance by conducting real-cluster experiments that assess the overhead and scalability of the replication-based data recovery process. Our experiments indicate that replication and data recovery can be performed quite efficiently. Thus, our approach presents a novel way to bridge the gap between parallel file systems and job schedulers, thereby enabling us to strike a balance between an HPC center resource consumption and serviceability.

8.2 Temporal Replication Design

Supercomputers are heavily utilized. Most jobs spend significantly more time waiting in the batch queue than actually executing. The popularity of a new system ramps up as it goes towards its prime time. For example, from the 3-year Jaguar job logs, the average job wait-time:run-time ratio increases from 0.94 in 2005, to 2.86 in 2006, and 3.84 in 2007.

Table 8.1: Configurations of top five supercomputers as of 06/2008

System	# Cores	Aggr-egate Memory (TB)	Scratch Space (TB)	Memory to Storage Ratio	Top 500 Rank
RoadRunner(LANL)	122400	98	2048	4.8%	1
BlueGene/L(LLNL)	106496	73.7	1900	3.8%	2
BlueGene/P(Argonne)	163840	80	1126	7.1%	3
Ranger(TACC)	62976	123	1802	6.8%	4
Jaguar(ORNL)	23412	46.8	600	7.8%	5

8.2.1 Justification and Design Rationale

A key concern about the feasibility of temporal replication is the potential space and I/O overhead replication might incur. However, we argue that by replicating selected "active files" during their "active periods", we are only replicating a small fraction of the files residing in the scratch space at any given time. To estimate the extra space requirement, we examined the sizes of the aggregate memory space and the scratch space on state-of-the-art supercomputers. The premise is that with today's massively parallel machines and with the increasing performance gap between memory and disk accesses, batch applications are seldom out-of-core. This also agrees with our observed memory use pattern on Jaguar (see below). Parallel codes typically perform input at the beginning of a run to initialize the simulation or to read in databases for parallel queries. Therefore, the aggregate memory size gives a bound for the total input data size of active jobs. By comparing this estimate with the scratch space size, we can assess the relative overhead of temporal replication.

Table 8.1 summarizes such information for the top five supercomputers [6]. We see that the memory-to-storage ratio is less than 8%. Detailed job logs with per-job peak memory usage indicate that the above approximation of using the aggregate memory size significantly overestimates the actual memory use (discussed later in this subsection). While the memory-to-storage ratio provides a rough estimation of the replication overhead, in reality, however, a number of other factors need to be considered. First, when analyzing the storage space overhead, queued jobs' input files cannot be ignored, since their aggregate size can be even larger than that of running jobs. In the following sections, we propose additional optimizations to shorten the lifespan of replicas. Second, when analyzing the

bandwidth overhead, the frequency of replication should be taken into account. Jaguar's job logs show an average job run time of around 1000 seconds and an average aggregate memory usage of 31.8 GB, which leads to a bandwidth consumption of less than 0.1% of Jaguar's total capacity of 284 GB/s. For this reason, we primarily focus on the space overhead in the following discussions.

Next, we discuss a supercomputer's usage scenarios and configuration in more detail to justify the use of replication to improve job input data availability.

Even though replication is a widely used approach in many distributed file system implementations, it is seldom adopted in supercomputer storage systems. In fact, many popular high-performance parallel file systems (e.g., Lustre and PVFS) do not even support replication, mainly due to space concerns. The capacity of the scratch space is important in (1) allowing job files to remain for a reasonable amount of time (days rather than hours), avoiding the loss of precious job input/output data, and (2) allowing giant "hero" jobs to have enough space to generate their output. Blindly replicating all files, even just once, would reduce the effective scratch capacity to half of its original size.

Temporal replication addresses the above concern by leveraging job execution information from the batch scheduler. This allows it to only replicate a small fraction of "active files" in the scratch space by letting the "replication window" slide as jobs flow through the batch queue. Temporal replication is further motivated by several ongoing trends in supercomputer configurations and job behavior. First, as mentioned earlier, Table 8.1 shows that the memory to scratch space ratio of the top 5 supercomputers is relatively low. Second, it is rather rare for parallel jobs on these machines to fully consume the available physical memory on each node. A job may complete in shorter time on a larger number of nodes due to the division of workload and data, resulting in lower per-node memory requirements at a comparable time-node charge. Figure 8.2 shows the per-node memory usage of both *running* and *queued* jobs over one month on the ORNL Jaguar system. It backs our hypothesis that jobs tend to be in-core, with their aggregate peak memory usage providing an upper bound for their total input size. We also found the actual aggregate memory usage averaged over the 300 sample points to be significantly below the total amount of memory available shown in Table 8.1: 31.8 GB for running jobs and 49.5 GB for queued jobs.

8.2.2 Delayed Replica Creation

Based on the above observations about job wait times and cost/benefit trade-offs for replication in storage space, we propose the following design of an HPC-centric file replication mechanism.

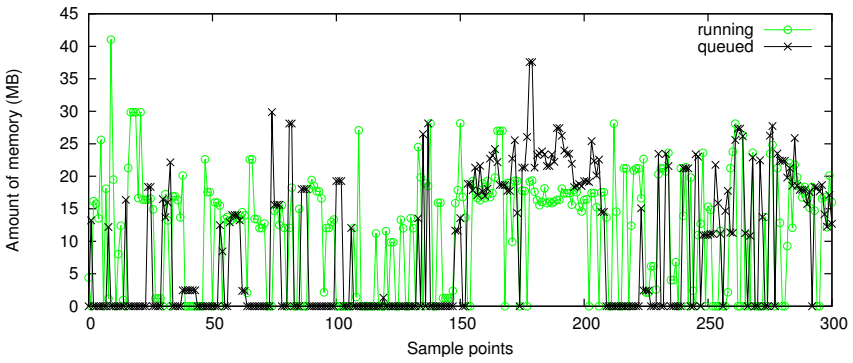


Figure 8.2: Per-node memory usage from 300 uniformly sampled time points over a 30-day period based on job logs from the ORNL Jaguar system. For each time point, the total memory usage is the sum of peak memory used by all jobs in question.

When jobs spend a significant amount of time waiting, replicating their input files (either at stage-in or submission time) wastes storage space. Instead, a parallel file system can obtain the current queue status and determine a *replication trigger point* to create replicas for a given job. The premise here is to have enough jobs near the top of the queue, stocked up with their replicas, such that jobs dispatched next will have extra input data redundancy. Additional replication will be triggered by job completion events, which usually result in the dispatch of one or more jobs from the queue. Since jobs are seldom interdependent, we expect that supplementing a modest prefix of the queued jobs with a second replica of their input will be sufficient. Only one copy of a job's input data will be available before its replication trigger point. However, this primary copy can be protected with periodic availability checks and remote data recovery techniques previously developed and deployed by us [53].

Completion of a large job is challenging as it can activate many waiting jobs requiring instant replication of multiple datasets. As a solution, we propose to query the queue status from the job scheduler. Let the replication window, w, be the length of the prefix of jobs at the head of the queue that should have their replicas ready. w should be

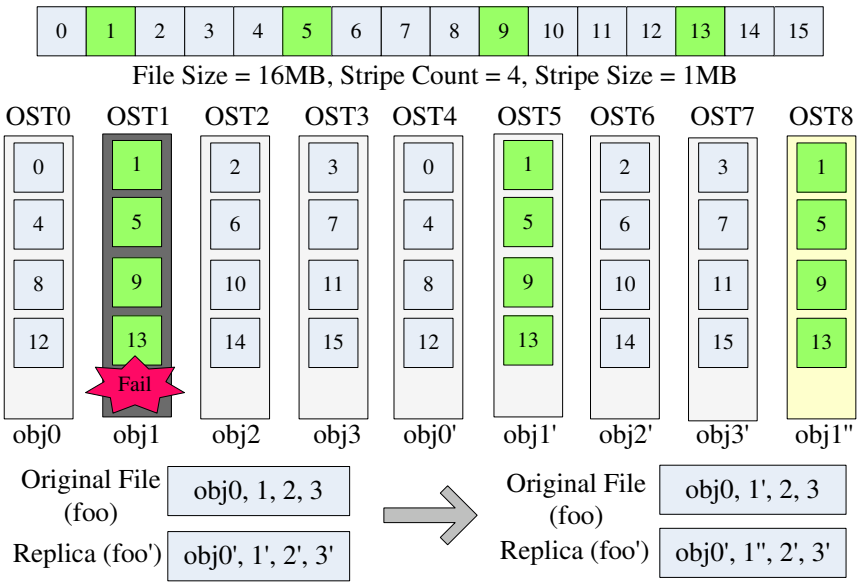


Figure 8.3: Objects of an original job input file and its replica. A failure occurred to OST1, which caused accesses to the affected object to be redirected to their replicas on OST5, with replica regeneration on OST8.

the smallest integer such that:

$$\sum_{i=0}^{w} |Q_i| > max(R, \alpha S),$$

where $|Q_i|$ is the number of nodes requested by the ith ranked job in the queue, R is the number of nodes used by the largest running job, S is the total number of nodes in the system, and the factor $\alpha(0 \leq \alpha)$ is a controllable parameter to determine the eagerness of replication.

One problem with the above approach is that job queues are quite dynamic as strategies such as backfilling are typically used with an FCFS or FCFS-with-priority scheduling policy. Therefore, jobs do not necessarily stay in the queue in their arrival order. In particular, jobs that require a small number of nodes are likely to move ahead faster. To address this, we augment the above replication window selection with a "shortcut" approach and define a threshold T, $0 \leq T \leq 1$. Jobs that request $T \cdot S$ nodes will have their input data replicated immediately regardless of the current replica window. This approach allows jobs that tend to be scheduled quickly to enjoy early replica creation.

8.2.3 Eager Replica Removal

We can also shorten the replicas' life span by removing the extra copy once we know it is not needed. A relatively safe approach is to perform the removal at job completion time. Although users sometimes submit additional jobs using the same input data, the primary data copy will again be protected with our offline availability check and recovery [53]. Further, subsequent jobs will also trigger replication as they progress toward the head of the job queue.

Overall, we recognize that the input files for most in-core parallel jobs are read at the beginning of job execution and never re-accessed thereafter. Hence, we have designed an *eager replica removal* strategy that removes the extra replica once the replicated file has been closed by the application. This significantly shortens the replication duration, especially for long-running jobs. Such an aggressive removal policy may subject input files to a higher risk in the rare case of a subsequent access further down in its execution. However, we argue that reduced space requirements for the more common case outweigh this risk.

8.3 Implementation Issues

All our modifications were made within Lustre and do not affect the POSIX file system APIs. Therefore, data replication, failover and recovery processes are entirely transparent to user applications.

In our implementation, a supercomputer's head node doubles as a replica management service node, running as a Lustre client. Job input data is usually staged via the head node making it well suited for initiating replication operations. Replica management involves generating a copy of the input dataset at the appropriate replication trigger point, scheduling periodic failure detection before job execution, and also scheduling data recovery in response to reconstruction requests. Data reconstruction requests are initiated by the compute nodes when they observe storage failures during file accesses. The replica manager serves as a coordinator that facilitates file reorganization, replica reconstruction, and streamlining of requests from the compute nodes in a non-redundant fashion.

Replica Creation and Management: We use the copy mechanism of the underlying file system to generate a replica of the original file. In creating the replica, we ensure that

it inherits the striping pattern of the original file and is distributed on I/O nodes disjoint from the original file's I/O nodes. As depicted in Figure 8.3, the objects of the original file and the replica form pairs (objects (0, 0′), (1, 1′), etc.). The replica is associated with the original file for its lifetime by utilizing Lustre's extended attribute mechanism.

Failure Detection: For persistent data availability, we perform periodic failure detection before a job's execution. This offline failure detection mechanism was described in our previous work [53]. The same mechanism has been extended for transparent storage failure detection and access redirection during a job run. Both I/O node failures and disk failures will result in an I/O error immediately within our Lustre patched VFS system calls. Upon capturing the I/O error in the system function, Lustre obtains the file name and the index of the failed OST. Such information is then sent by the client to the head node, which, in turn, initiates the object reorganization and replica reconstruction procedures.

Object Failover and Replica Regeneration: Upon an I/O node failure, either detected by the periodic offline check or by a compute node through an I/O error, the aforementioned file and failure information is sent to the head node. Using several new commands that we have developed, the replica manager will query the MDS to identify the appropriate objects in the replica file that can be used to fill the holes in the original file. The original file's metadata is updated subsequently to integrate the replicated objects into the original file for seamless data access failover. Since metadata updates are inexpensive, the head node is not expected to become a potential bottleneck.

To maintain the desired data redundancy during the period that a file is replicated, we choose to create a "secondary replica" on another OST for the failover objects after a storage failure. The procedure begins by locating another OST, giving priority to one that currently does not store any part of the original or the primary replica file.[1] Then, the failover objects are copied to the chosen OST and in turn integrated into the primary replica file. Since the replica acts as a backup, it is not urgent to populate its data immediately. In our implementation, such stripe-wise replication is delayed by 5 seconds (tunable) and is offloaded to I/O nodes (OSSs).

[1] In Lustre, file is striped across 4 OSTs by default. Since supercomputers typically have hundreds of OSTs, an OST can be easily found.

Streamlining Replica Regeneration Requests: Due to parallel I/O , multiple compute nodes (Lustre clients) are likely to access a shared file concurrently. Therefore, in the case of a storage failure, we must ensure that the head node issues a single failover/regeneration request per file and per OST despite multiple such requests from different compute nodes. We have implemented a centralized coordinator inside the replica manager to handle the requests in a non-redundant fashion.

8.4 Experimental Results

To evaluate the temporal replication scheme, we performed real-cluster experiments. We assessed our implementation of temporal replication in the Lustre file system in terms of the online data recovery efficiency.

8.4.1 Failure Detection and Offline Recovery

Before a job begins to run, we periodically check for failures on OSTs that carry its input data. The detection cost is less than 0.1 seconds as the number of OSTs increases to 256 (16 OSTs on each of the 16 OSSs) in our testbed. Since failure detection is performed when a job is waiting, it incurs no overhead on job execution itself. When an OST failure is detected, two steps are performed to recover the file from its replica: object failover and replica reconstruction. The overhead of object failover is relatively constant (0.84-0.89 seconds) regardless of the number of OSTs and the file size. This is due to the fact that the operation only involves the MDS and the client that initiates the command. Figure 8.4 shows the replica reconstruction (RR) cost with different file sizes. The test setup consisted of 16 OSTs (1 OST/OSS). We varied the file size from 128MB to 2GB. With one OST failure, the data to recover ranges from 8MB to 128MB causing a linear increase in RR overhead. Figure 8.4 also shows that the *whole file reconstruction (WFR)*, the conventional alternative to our more selective scheme where the entire file is re-copied, has a much higher overhead. In addition, RR cost increases as the chunk size decreases due to the increased fragmentation of data accesses.

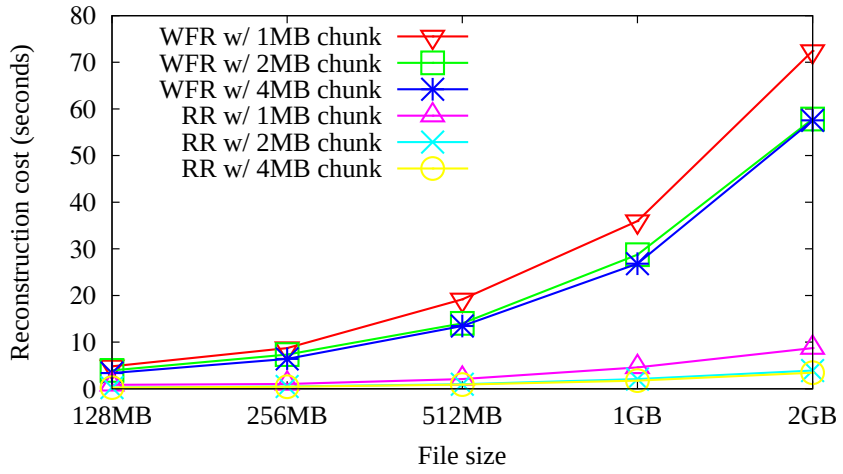


Figure 8.4: Offline replica reconstruction cost with varied file size

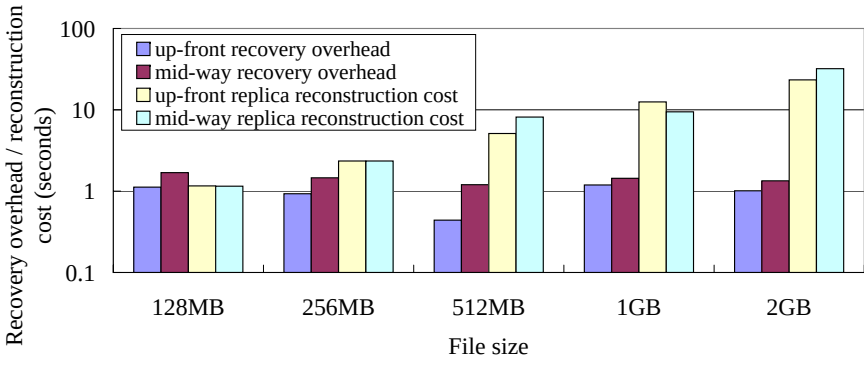


Figure 8.5: MM recovery overhead vs. replica reconstruction cost

8.4.2 Online Recovery

Application 1: Matrix Multiplication (MM)

To measure on-the-fly data recovery overhead during a job run with temporal replication, we used MM, an MPI kernel that performs dense matrix multiplication. It computes the standard $C = A * B$ operation, where A, B and C are $n * n$ matrices. A and B are stored contiguously in an input file. We vary n to manipulate the problem size. Like in many applications, only one master process reads the input file, then broadcasts the data to all the other processes for parallel multiplication using a BLOCK distribution.

Figure 8.5 depicts the MM recovery overhead with different problem sizes. Here, the MPI job ran on 16 compute nodes, each with one MPI process. The total input size was varied from 128MB to 2GB by adjusting n. We configured 9 OSTs (1 OST/OSS), with the original file residing on 4 OSTs, the replica on another 4, and the reconstruction of the failover object occurring on the remaining one. Limited by our cluster size, we let nodes double as both I/O and compute nodes.

To simulate random storage failures, we varied the point in time where a failure occurs. In "up-front", an OSTs failure was induced right before the MPI job started running. Hence, the master process experienced an I/O error upon its first data access to the failed OST. With "mid-way", one OST failure was induced mid-way during the input process. The master encountered the I/O error amidst its reading and sent a recovery request to the replica manager on the head node. Figure 8.5 indicates that the application-visible recovery overhead was almost constant for all cases (right around 1 second) considering system variances. This occurs because only one object was replaced for all test cases while only one process was engaged in input. Even though the replication reconstruction cost rises as the file size increases, this was hidden from the application. The application simply progressed with the failover object from the replica while the replica itself was replenished in the background.

Application 2: mpiBLAST

To evaluate the data recovery overhead using temporal replication with a read-intensive application, we tested with mpiBLAST [58], which splits a database into fragments and performs a BLAST search on the worker nodes in parallel. Since mpiBLAST is more input-intensive, we examined the impact of a storage failure on its overall performance. The difference between the job execution times with and without failure, i.e., the recovery overhead, is shown in Figure 8.6. Since mpiBLAST assigns one process as the master and another to perform file output, the number of actual worker processes performing parallel input is the total process number minus two.

The Lustre configurations and failure modes used in the tests were similar to those in the MM tests. Overall, the impact of data recovery on the application's performance was small. As the number of workers grew, the database was partitioned into more files. Hence, more files resided on the failed OST and needed recovery. As shown by Figure 8.6, the

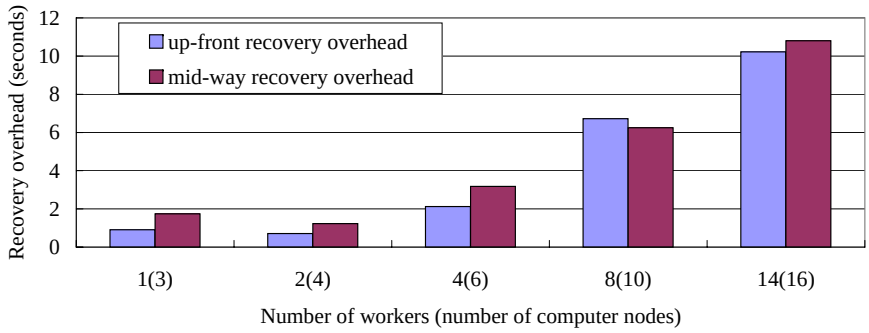


Figure 8.6: Recovery overhead of mpiBLAST

recovery overhead grew with the number of workers. Since each worker process performed input at its own pace and the input files were randomly distributed to the OSTs, the I/O errors captured on the worker processes occurred at different times. Hence, the respective recovery requests to the head node were not issued synchronously in parallel but rather in a staged fashion. With many applications that access a fixed number of shared input files, we expect to see a much more scalable recovery cost with regard to the number of MPI processes using our techniques.

8.5 Conclusion

We have presented a novel temporal replication scheme for supercomputer job data. By creating additional data redundancy for transient job input data, we allow fast online data recovery from local replicas without user intervention or hardware support. This general-purpose, high-level data replication can help avoid job failures/resubmission by reducing the impact of both disk failures or software/hardware failures on the storage nodes. Our implementation, using the widely used Lustre parallel file system and the Moab scheduler, demonstrates that replication and data recovery can be performed efficiently.

Chapter 9

Related Work

This dissertation aims to present FT techniques for MPI jobs and job input data to realize significant advances in fault resilience of HPC jobs as presented in Chapters 2 - 8. This chapter on related work is also split into two parts: Section 9.1 presents research related to FT for MPI jobs and Section 9.2 presents the work on FT for job input data.

9.1 Fault Tolerance Techniques for MPI Jobs

A wide range of methods and systems to support FT for MPI jobs have been developed in the past.

Checkpoint/Restart: C/R techniques for MPI jobs are frequently deployed in HPC environments and can be divided into two categories: coordinated (LAM/MPI+BLCR [10, 15], CoCheck [59], etc.) and uncoordinated (MPICH-V [9, 60]). Coordinated techniques commonly rely on a combination of OS support to checkpoint a process image (*e.g.*, via the BLCR Linux module [15]) or user-level runtime library support. Collective communication among MPI tasks is used for coordinated checkpoint negotiation [10]. Uncoordinated C/R techniques generally rely on logging messages and possibly their temporal ordering for asynchronous non-coordinated checkpointing, *e.g.*, MPICH-V [9, 60] that uses pessimistic message logging. The framework of OpenMPI [61, 11] is designed to allow both coordinated and uncoordinated types of protocols.

Our focus is on LAM/MPI+BLCR, which requires a complete system restart

where target node information, such as IP addresses, cannot be changed and checkpointing is not fully automated. This severely limits its applicability. Prior work extended this LAM/MPI+BLCR functionality to support migration of selected checkpoint images to new nodes [19]. Similar work extended the HA-OSCAR (High Availability Open Source Cluster Application Resources) distribution not only with compute-node failover (equivalent to migration with a complete start) but also to head-node failover (active-standby) in clusters [62]. This required modifications to LAM's hard-coded internal addressing information within the checkpointed file images followed by a complete job restart. Job submission frameworks, such as Torque, are oblivious to such changes if the new job is carefully constructed such as to resemble the originally submitted one. In contrast to this work, we take LAM/MPI+BLCR to yet another level with our novel job pause mechanism (as discussed in Chapter 2) that supports migration without restart, *i.e.*, by retaining functioning process images and rolling back to the last checkpoint. Our approach is independent and transparent of any higher-level frameworks, such as job submission facilities. More importantly, users do not lose their allocated time of a running job, *i.e.*, instead of requeuing the restarted job and waiting for its execution, execution commences from the last checkpoint without noticeable interruption.

Furthermore, conventional C/R techniques checkpoint the entire process image leading to high checkpoint overhead, heavy I/O bandwidth requirements and considerable hard drive pressure, even though only a subset of the process image of all MPI tasks changes between checkpoints. With our hybrid full/incremental C/R mechanism, as discussed in Chapter 5, we mitigate the situation by checkpointing only the modified pages and at a lower rate than required for full checkpoints.

Incremental Checkpointing: Recent studies focus on incremental checkpointing [32, 63, 64]. TICK (Transparent Incremental Checkpointer at Kernel Level) [32] is a system-level checkpointer implemented as a kernel thread. It supports incremental and full checkpoints. However, it checkpoints only sequential applications running on a single process that do not use inter-process communication or dynamically loaded shared libraries. *In contrast, our hybrid full/incremental C/R solution transparently supports incremental checkpoints for an* **entire MPI job** *with all its processes. Pickpt* [63] is a page-level incremental checkpointing facility. It provides space-efficient techniques for automatically removing useless checkpoints

aiming to minimizing the use of disk space, which differs from our garbage collection thread technique. Yi *et al.* [65] develop an adaptive page-level incremental checkpointing facility based on the dirty page count as a threshold heuristic to determine whether to checkpoint now or later, a feature complementary to our work that we could adopt within our scheduler component. However, Pickpt and Yi's adaptive scheme are constrained to C/R of a single process, just as TICK was, while we cover an entire MPI job with all its processes and threads within processes. Agarwal *et al.* [66] provide a different adaptive incremental checkpointing mechanism to reduce the checkpoint file size by using a secure hash function to uniquely identify changed blocks in memory. Their solution not only appears to be specific to IBM's compute node kernel on BG/L, it also requires hashes for each memory page to be computed, which tends to be more costly than OS-level dirty-bit support as caches are thrashed when each memory location of a page has to be read in their approach.

A prerequisite of incremental checkpointing is the availability of a mechanism to track modified pages during each checkpoint. Two fundamentally different approaches may be employed, namely page protection mechanisms or page-table dirty bits. Different implementation variants build on these schemes. One is the bookkeeping and saving scheme that, based on the dirty bit scheme, copies pages into a buffer [32]. Another solution is to exploit page write protection, such as in *Pickpt* [63], to save only modified pages as a new checkpoint. The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected with application-level control if an alternate signal stack is employed, which adds calling overhead and increases cache pressure. Furthermore, the overhead of user-level exception handlers or, alternatively, kernel-level read/write protection changes is much higher than kernel-level dirty-bit shadowing. Thus, we selected the dirty bit scheme in our live migration and incremental checkpointing mechanisms, yet in our own implementation within the Linux kernel. *Our hybrid full/incremental checkpoint/restart approach is unique among this prior work in its ability to capture and restore an* **entire MPI job** *with all its tasks, including all relevant process information and OS kernel-specific data.* Hence, our scheme is more general than language specific solutions (as in Charm++), yet lighter weight than OS virtualization C/R techniques.

Checkpoint Interval Model: Aiming at optimality for checkpoint overhead and roll-back time over a set of MPI jobs, several models have been developed to determine job-

specific intervals for full or incremental checkpoints. Yong [21] presented a checkpoint model and obtained a fixed optimal checkpoint interval. Based on Young's work, Daly [67, 68] improved the model to an optimal checkpoint placement from a first order to a higher order approximation. Liu *et al.* provide a model for an optimal full C/R strategy toward minimizing rollback and checkpoint overheads [69]. Their scheme focuses on the fault tolerance challenge, especially in a large-scale HPC system, by providing optimal checkpoint placement techniques that are derived from the actual system reliability. Naksinehaboon *et al.* provide a model to perform a set of incremental checkpoints between two consecutive full checkpoints [41] and a method to determine the optimal number of incremental checkpoints between full checkpoints. While their work is constrained to simulations based on log data, our work on hybrid full/incremental checkpoint/restart mechanism focuses on the design and implementation of process-level incremental C/R for MPI tasks. Their work is complementary in that their model could be utilized to fine-tune our incremental C/R rate. In fact, the majority of their results on analyzing failure data logs show that the full/incremental C/R model outperforms full checkpointing. Furthermore, our reverse scanning restart mechanism is superior to the one used in their model.

Proactive FT and Migration: The feasibility of proactive FT has been demonstrated at the job scheduling level [30], within OS virtualization [17] and in Adaptive MPI [27, 28, 29] using a combination of (a) object virtualization techniques to migrate tasks and (b) causal message logging [70] within the MPI runtime system of Charm++ applications. In contrast to Charm++, our process-level live migration solution is coarser grained as FT is provided at the process level, thereby encapsulating most of the process context, including open file descriptors, which are beyond the MPI runtime layer.

Two facets on proactive FT are intensively studied. First, there are a number of research efforts on failure prediction [26, 71, 72]. These papers report high failure prediction accuracy with a prior warning window, which is the premise for our process migration mechanism proposed in Chapter 3. Second, various migration mechanisms have been studied extensively in the past [73, 74, 75, 76, 77, 78, 15]. Furthermore, MPI-Mitten [79], an MPI library between the MPI layer and the application layer, provides proactive fault tolerance to MPI applications. It uses HPCM [80] as a middleware to support user-level heterogeneous process migration. These two facets are integrated in approaches that combine

prediction and migration in proactive FT systems and evaluate different FT policies. In [81], the authors provide a generic framework based on a modular architecture allowing the implementation of new proactive fault tolerance policies/mechanisms. An agent-oriented framework [82] was developed for grid computing environments with separate agents to monitor individual classes or subclasses of faults and proactively act to avoid or tolerate a fault. Sun *et al.* provide fault-aware systems, such as FARS [83] and FENCE [84], to increase the accuracy of fault prediction and improve system resilience to failures with different fault management mechanisms including process migration. They also model the migration cost and introduce a dynamic scheduling mechanism accordingly [85]. In their paper, Tikotekar *et al.* also present a simulation framework that evaluates different FT mechanisms and policies, including a combination of reactive FT and proactive FT to decrease the number of checkpoints [22], which obtained the best results among all the real and simulated FT mechanisms and policies. These prior works with their fault models, FT mechanisms for fault occurrences and their evaluation simulations, confirm that the process migration is a suitable approach for proactive FT with lower cost than OS virtualization, which reinforces the significance of our solution.

Approaches for transparent real-time performance monitoring of MPI parallel programs exploit information on wall-clock time of MPI events collected in prior work [86, 87] with the intent to automatically detect deviations of a process from its expected behavior and subsequently enable real-time scheduling of MPI programs. We, in contrast, contribute a monitoring API that meshes directly with MPI applications to log overheads of time steps, as discussed in Section 4.3.

Our proactive live migration solution orchestrates BMC/IPMI health monitoring, fundamentally new BLCR capabilities and the extended communication mechanism at LAM/MPI through a decentralized scheduler. The framework is simple and applicable to arbitrary MPI implementations in HPC environments. Furthermore, our approach provides a *live migration mechanism* that supports continued execution of MPI applications *during* much of the migration time. This solution parallels live migration at the OS virtualization layer [88], which has been studied in the context of proactive FT of MPI applications [34], an approach that supports integrated health-based monitoring and proactive live migration over Xen guests. We contribute *process-level* live migration and demonstrate its superior efficiency to of OS-level virtualization. In HPC, process-level solutions are more widely

accepted than OS virtualization, not the least because of potential performance penalties of network virtualization or additional driver development for virtualization-bypass technologies [89, 90].

9.2 Fault Tolerance Techniques for Job Input Data

Next, we will discuss related efforts in FT for job input data.

RAID Recovery: Disk failures can often be masked by standard RAID techniques [13]. However, RAID is geared toward whole disk failures and does not address sector-level faults [52, 91, 92]. It is further impaired by controller failures and multiple disk failures within the same group. Without hot spares, reconstruction requires manual intervention and is time consuming. With RAID reconstruction, disk arrays either run in a degraded (not yielding to other I/O requests) or polite mode. In a degraded mode, busy disk arrays suffer a substantial performance hit when crippled with multiple failed disks [93, 54]. This degradation is even more significant on parallel file systems as files are striped over multiple disk arrays and large sequential accesses are common. Under a polite mode, with rapidly growing disk capacity, the total reconstruction time is projected to increase to days subjecting a disk array to additional failures [14]. Furthermore, RAID technology cannot handle a media error, also know as an unrecoverable read error (URE), which occurs during dailed disk reconstruction [94]. With 50 GB SATA disk drives, MTBF is about one error every 10^{14} bits (12.5 terabytes), so that a media error was very unlikely to occur. When it does occur during reconstruction, a 50 GB disk takes only a few hours to recover from tape. However, for terabyte disks, the disk failure plus media error scenario becomes almost inevitable. Recovering the storage array from backup tape could take a month. To address this problem, Panasas has implemented a significant extension, called "tiered parity" [95], to RAID. In this model, Panasas has built "vertical parity" and "network parity" on top of existing "horizontal parity". With vertical parity, they have added RAID within each disk. According to Panasas, vertical parity reduces the error rate to between one in 10^{18} and one in 10^{19} bits written, which is 1000 to 10,000 times better than the URE rate. The extra parity information uses 10 percent of the disk capacity. On top of horizonal and vertical parity schemes, Panasas also adds an additional layer of network parity protection. At this

level, parity checking is done on the client side. Our approach complements RAID systems by providing fast recovery protecting against non-disk and multiple disk failures.

Recent work on popularity-based RAID reconstruction [96] rebuilds more frequently accessed data first, thereby reducing reconstruction time and user-perceived penalties. However, supercomputer storage systems host transient job data, where "unaccessed" job input files are often more important than "accessed" ones. In addition, such optimizations cannot cope with failures beyond RAID's protection at the hardware level.

Replication: Data replication, a commonly used technique for persistent data availability, creates and stores redundant copies (*replicas*) of datasets. Various replication techniques have been studied [97, 98, 99, 100] in many distributed file systems [101, 102, 103]. Most existing replication techniques treat all datasets with equal importance and each dataset with static, time-invariant importance when making replication decisions. An intuitive improvement would be to treat datasets with different priorities. To this end, BAD-FS [104] performs selective replication according to a cost-benefit analysis based on the replication costs and the system failure rate. Similar to BAD-FS, our temporal replication approach (as discussed in Chapter 8) also makes on-demand replication decisions. However, our scheme is more "access-aware" rather than "cost-aware". While BAD-FS still creates static replicas, our replication approach utilizes explicit information from the job scheduler to closely synchronize and limit replication to jobs in execution or soon to be executed.

Parallel File Systems: There are two classes of parallel file systems, namely those whose architectures are based on I/O nodes/servers managing data on directly attached storage devices (such as PVFS [50] and LUSTRE [45]) and those with centralized, shared storage devices that are shared by all I/O nodes (such as GPFS [105]). For the former category, node failure implies that a partition of the storage system is unavailable. Since parallel file systems usually stripe datasets for better I/O performance, failure of one node may affect a large portion of user jobs. Moreover, unlike specialized nodes/servers such as metadata servers, token servers, etc., I/O nodes in parallel file systems may not be routinely protected through failover. I/O node failover does not help when the underlying RAID recovery is impaired (as mentioned above) as the data is seldom replicated. Our offline and online recovery solution for job input data mitigates these situations by exploiting the source

copies of user job input data.

I/O shepherding [91] introduces a reliability infrastructure for file systems by executing I/O requests using user-specified FT mechanisms including retries, sanity checking, checksums, and mirrors or parity protection to recover from lost blocks or disks. This work is similar in the sense that it attempts to introduce fault-tolerant behavior into file systems by reliably executing I/O requests. However, we are concerned with HPC job input data and rely on external sources for I/O node failure recovery.

Erasure Coding: Another widely investigated technique is erasure coding [106, 107, 108]. With erasure coding, k parity blocks are encoded into n blocks of source data. When a failure occurs, the whole set of $n+k$ blocks of data can be reconstructed with any n surviving blocks through decoding.

Erasure coding reduces the space usage of replication but adds computational overhead for data encoding/decoding. In [109], the authors provide a theoretical comparison between replication and erasure coding. In many systems, erasure coding provides better overall performance balancing computation costs and space usage. However, for supercomputer centers, its computation costs will be a concern. This is because computing time in supercomputers is a precious commodity. At the same time, our data analysis suggests that the amount of storage space required to replicate data for active jobs is relatively small compared to the total storage footprint. Therefore, compared to erasure coding, our replication approach is more suitable for supercomputing environments, which is verified by our experimental study.

Chapter 10

Conclusion

The hypothesis of this dissertation was that while fault of HPC jobs increases as the number of computing and I/O nodes in HPC environments goes up, fault resilience can be significantly improved by employing a combination of multiple FT techniques. The work presented in Chapters 2 - 8 has shown this hypothesis to be true in evaluating process-level job healing and input data recovery solutions. These contributions are summarized in the following.

10.1 Contributions

In the area of process-level job healing techniques, we provide a novel proactive FT scheme and also significant enhancements to reactive FT, as presented in Chapters 2 - 5.

Chapter 2 presents a transparent job pause mechanism. It pauses the job when a process fails and restarts the failing process on another processor to prevent the job from having to re-enter the job queue. In Chapters 3 and 4, we present process-level live migration and back migration solutions for proactive fault-tolerance in HPC environments. This complements reactive fault-tolerant mechanisms, such as checkpointing, resulting in a reduction of the number of checkpoints when a majority of the faults can be handled proactively. An incremental checkpointing mechanism, presented in Chapter 5, was developed and combined with full checkpointing. This reduces the overhead of checkpointing by infrequently performing periodic full checkpoints interspersed by several incremental

checkpoints. All these techniques can be deployed together to (1) handle the majority of the faults proactively, (2) reduce the number of full checkpoints required, (3) reduce the checkpoint operation overhead, and (4) alleviate the need to requeue the jobs.

Experimental results with the NPB suite for the job pause service show that a minimal overhead of 5.6% is only incurred in case migration takes place while the regular checkpoint overhead remains unchanged without a need to reboot the LAM run-time environment. Experiments indicate that 1-6.5 seconds of prior warning are sufficient to successfully trigger live process migration while similar operating system virtualization mechanisms require 13-24 seconds. This self-healing approach complements reactive FT by nearly cutting the number of checkpoints in half when 70% of the faults are handled proactively. Experiments also indicate that savings due to replacing full checkpoints with incremental ones average 16.64 seconds while restore overhead amounts to just 1.17 seconds.

This part of the contributed work resulted in several publications [17, 35, 110, 111, 112].

In the area of FT, techniques for job input data preservation, remote patching and temporal replication, presented in Chapters 6 - 8, were developed to improve the reliability, availability and performance of HPC I/O systems.

In Chapters 6 and 7, we investigate offline and online approaches for reconstructing missing pieces of datasets from data sources where the job input data was originally staged from. The two approaches complement each other. We have shown that supercomputing centers' data availability can be drastically enhanced by periodically checking and reconstructing datasets for queued jobs while the reconstruction overheads are barely visible to users. The approach provides a seamless way for a running job's input data to be reconstructed from its remote source in case of I/O errors. In Chapter 8, we have presented a novel temporal replication scheme by creating additional data redundancy for transient job input data to avoid job failures/resubmission.

Both remote patching and temporal replication will be able to help with storage failures at multiple layers. While remote patching poses no additional space overhead, the patching costs depend on the data source and the end-to-end network transfer performance. It can be hidden from applications during a job's execution. Temporal replication, on the other hand, trades space (which is relatively cheap at supercomputers) for performance. It provides high-speed data recovery and reduces the space overhead by only replicating the

data when it is needed. Our optimizations presented in Chapter 8 aim at further controlling and lowering the space consumption of replicas.

Experimental results with I/O-intensive MPI benchmarks indicate that the recovery mechanism of the file remote patching imposes little overhead and scales well with increasing file sizes. This includes three independent steps: checking all the OSTs in parallel, updating file stripe metadata and reconstructing the missing data. Furthermore, online remote patching can be overlapped with computation and communication as data stripes are recovered. Our experimental results reinforce this by showing that the increase in job execution time due to on-the-fly recovery is negligible compared to non-faulting runs. Experimental results also indicate the impact of data recovery from temporal replication on the application's performance is small since applications simply progress with failover objects from the replicas, while replicas themselves are replenished in the background.

The work for job input data resulted in several publications [53, 113, 114].

Overall, the offline/online recovery and temporal replication approaches for job input data tolerate a majority of the failures with storage and I/O. They are complemented by the previously contributed process-level FT techniques to tolerate the remaining storage or I/O failures as well as failures due to resources such as CPU, memory etc.

10.2 Future Work

The research presented in this dissertation suggests many interesting research directions for future work.

Scheduling and Orchestration: We now have various FT techniques for process-level job healing and job input data. The next direction is to systematically devise a scheduler to orchestrate all the approaches with the following major steps:

1. First, we deploy offline recovery of job input data for queued jobs.

2. During job execution, we deploy health monitoring and fault detection mechanisms, maintain a temporal replica for the job input data, and collect the information about the application, *e.g.*, size of the write set and runtime of the timesteps.

3. Meanwhile, we issue full or incremental checkpoints saving the checkpointing files to multi-level storage systems including memory, local disk, scratch space and remote

storage space (*i.e.*, scheduling of diskless checkpoints and multi-level checkpoints).

4. Upon an I/O error, temporal replica or online recovery supports the reconstruction of job input data.

5. For "unhealthy" nodes, either live migration, frozen migration or a job pause plus rollback is issued based on heuristics algorithms.

6. Then, we will apply back migration.

7. During all times, recovery mechanisms for job output data are deployed.

Fault detection mechanisms are beyond the scope of this dissertation. We simulated node failures by notifying the scheduler daemon to immediately initiate the job pause plus rollback mechanism. Since fault detection cannot be perfect in practice, one could miss a subset failures that ultimately might lead to job execution termination. This illustrates the need for a new task within the scheduler, *i.e.*, in this case, to restart the whole job from the last checkpoint using live nodes and spare nodes.

Scheduling of Diskless Checkpoints and Multi-level Checkpoints: The checkpointing schemes discussed in Chapters 2 and 5 rely on disks to store checkpoint images, *i.e.*, they require a high-bandwidth interconnect and either local storage or a highly efficient parallel file system, such as Lustre. Thus, checkpoints are either written to local disks of neighboring nodes (to ensure redundancy) or to the parallel file system, which itself provides redundancy (*e.g.*, through RAID technology). An alternate and faster option is provided memory-based checkpointing. Such diskless (in-memory) checkpoints are an interesting avenue for future research. Diskless checkpointing is also attractive in the absence of local disks, such as on BG/L. Furthermore, disk-based and diskless checkpoint mechanisms can be integrated with our full and incremental checkpointing approaches to provide a distributed multi-level scheduling mechanism to further improve the fault resiliency for HPC jobs.

Heuristics Algorithms for Migration: In Chapter 3, three main criteria for trading off live and frozen migration and for precopy termination conditions are discussed: (1) thresholds, *e.g.*, temperature watermarks; (2) available network bandwidth; and (3) size of the write set. Based on these conditions, a heuristics algorithm can be designed. In future work, we could (1) create and assess applications with varying communication rate

and memory access pattern to measure the tradeoff between live and frozen migrations and (2) provide heuristic algorithms to automate this task. Specifically, if the degree of urgency neither allows live migration nor frozen migration, a job pause plus rollback to the last checkpoint (or full+incremental checkpoints) with a spare node replacing the unhealthy one is a next choice. This would be orchestrated by the scheduler as discussed previously. The algorithm should cover this case and allow additional schemes, selected through a combination of sensor and history information, to be integrated as plug-ins.

We could also support an empirical factor in the algorithm to keep a profile history of the page modification rate and its regularity. Such model parameters could steer future application runs during live migration / precopy in choosing a more sensitive termination condition. This could be especially beneficial for jobs with long runtime (a large number of timesteps) or repeated job invocations. The same approach would further aid migration in an effort to better balance the load between nodes as downtimes would be shortened even though the migration duration may slightly increase.

Job Output Data: The remote patching and temporal replication techniques presented in Chapters 6 - 8 are constrained to job input data. However, during the parallel execution, the application performs periodic I/O and writes result files every few timesteps, which amounts to many TBs. These files are offloaded to the end-user's local cluster. Such job output data needs to be offloaded from center scratch space in timely fashion. Otherwise, it is exposed to center purge policies, which make room for data of incoming jobs. Further, a delayed offload wastes precious scratch space and increases the probability of data losses due to storage failures. In addition, end-users need to visualize the data within a deadline or use it as input for another job. Thus, areas of research to be explored include the development of approaches to perform "eager offloading" for job output data.

Bibliography

[1] Chung-H. Hsu and Wu-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, 2005.

[2] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[3] Oak Ridge National Laboratory. Resources - national center for computational sciences (nccs). http://info.nccs.gov/resources/jaguar, June 2007.

[4] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of HPCA-11*. IEEE Computer Society, 2005.

[5] J. Gray and A. Szalay. Scientific data federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2003.

[6] Top 500 list. http://www.top500.org/, June 2002.

[7] Message Passing Interface Forum. *MPI: Message-Passing Interface Standard*, June 1995.

[8] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*, July 1997.

[9] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, November 2002.

[10] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *LACSI*, October 2003.

[11] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical report, Indiana University, Computer Science Department, 2006.

[12] www.openmp.org. *Official OpenMP Specification*, May 2005.

[13] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, 1988.

[14] B. Schroeder and G. Gibson. Understanding failure in petascale computers. In *SciDAC Conference*, 2007.

[15] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[16] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[17] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault-tolerant membership for MPI tasks on hpc systems. In *International Conference on Supercomputing*, pages 219–228, June 2006.

[18] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing*, 1999.

[19] Jiannong Cao, Yinghao Li, and Minyi Guo. Process migration for MPI applications based on coordinated checkpoint. In *ICPADS*, pages 306–312, 2005.

[20] Andrew Wissink, Richard Hornung, Scott Kohn, and Steve Smith. Large scale parallel structured amr calculations using the samrai framework. In *Supercomputing*, November 2001.

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

[22] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Stephen L. Scott, and Chokchai Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *IEEE Cluster*, September 17-20, 2007.

[23] Hertong Song, Chokchai Leangsuksun, and Raja Nassar. Availability modeling and analysis on high performance cluster computing systems. In *ARES*, pages 305–313, 2006.

[24] Sunil Rani, Chokchai Leangsuksun, Anand Tikotekar, Vishal Rampure, and Stephen Scott. Toward efficient failre detection and recovery in HPC. In *High Availability and Performance Computing Workshop*, 2006.

[25] Advanced configuration & power interface. http://www.acpi.info.

[26] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD '03*, 2003.

[27] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of HPCA-11*, 2005.

[28] S. Chakravorty, C.Mendes, and L. Kale. Proactive fault tolerance in MPI applications via task migration. In *HiPC*, 2006.

[29] S. Chakravorty, C. Mendes, and L.Kale. A fault tolerance protocol with fast fault recovery. In *IPDPS*, 2007.

[30] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for BlueGene/L systems. In *IPDPS*, 2004.

[31] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *DPDNS*, March 2007.

[32] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, 2005.

[33] Readable dirty-bits for IA64 linux. https://www.gelato.unsw.edu.au/archives/gelato-technical/2005-November/001080.html.

[34] A. B. Nagarajan and F. Mueller. Proactive fault tolerance for HPC with Xen virtualization. In *ICS*, June 2007.

[35] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *IPDPS*, April 2007.

[36] Ganglia. http://ganglia.sourceforge.net/.

[37] htop. http://htop.sourceforge.net/.

[38] Performance application programming interface. http://icl.cs.utk.edu/papi/.

[39] mpip: Lightweight, scalable mpi profiling. http://mpip.sourceforge.net/.

[40] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Trans. on Computer Systems, Vol. 10, No. 1*, February 1992.

[41] 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 *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783–788, Washington, DC, USA, 2008. IEEE Computer Society.

[42] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/.

[43] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.

[44] R.A. Coyne and R.W. Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium*, 1995.

[45] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/whitepaper.pdf, 2002.

[46] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[47] Ncsa gridftp client. http://dims.ncsa.uiuc.edu/set/uberftp/index.html, 2006.

[48] D. Libes. The expect home page. http://expect.nist.gov/, 2006.

[49] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. In *Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[50] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[51] X. Ma, S. Vazhkudai, V. Freeh, T. Simon, T. Yang, and S. L. Scott. Coupling prefix caching and collective downloads for remote data access. In *Proceedings of the ACM International Conference on Supercomputing*, 2006.

[52] Lakshmi Bairavasundaram, Garth Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, pages 289 – 300, June 2007.

[53] Z. Zhang, C. Wang, S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller. Optimizing center performance through coordinated data staging, scheduling and recovery. In *Supercomputing*, November 2007.

[54] Alexander Thomasian, Gang Fu, and Chunqi Han. Performance of two-disk failure-tolerant disk arrays. *IEEE Transactions on Computers*, 56(6):799–814, 2007.

[55] Stephen C. Simms, Gregory G. Pike, and Doug Balog. Wide area filesystem performance using lustre on the teragrid. In *TeraGrid*, 2007.

[56] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4(3):1–25, 2008.

[57] H. Monti, A.R. Butt, and S. S. Vazhkudai. Timely Offloading of Result-Data in HPC Centers. In *Proceedings of 22nd Int'l Conference on Supercomputing ICS'*08, June 2008.

[58] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiblast. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution '03*, June 2003.

[59] G. Stellner. CoCheck: checkpointing and process migration for MPI. In *Proceedings of IPPS '96*, 1996.

[60] Bouteiller Bouteiller, Franck Cappello, Thomas Herault, Krawezik Krawezik, Pierre Lemarinier, and Magniette Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Supercomputing*, 2003.

[61] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in Open MPI. In *European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[62] A. Tikotekar, C. Leangsuksun, and S. L. Scott. On the survivability of standard MPI applications. In *LCI International Conference on Linux Clusters: The HPC Revolution*, May 2006.

[63] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

[64] Shang-Te Hsu and Ruei-Chuan Chang. Continuous checkpointing: joining the checkpointing with virtual memory paging. *Softw. Pract. Exper.*, 27(9):1103–1120, 1997.

[65] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of*

the 2006 ACM symposium on Applied computing, pages 1472–1476, New York, NY, USA, 2006. ACM.

[66] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[67] J. T. Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *International Conference on Computational Science*, pages 3–12, 2003.

[68] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[69] Yudan Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and Stephen Scott. A reliability-aware approach for an optimal checkpoint/restart model in hpc environments. *Cluster Computing, 2007 IEEE International Conference on*, pages 452–457, Sept. 2007.

[70] 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, 1992.

[71] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *IEEE ICDCS*, 2008.

[72] Prashasta Gujrati, Yawei Li, Zhiling Lan, Rajeev Thakur, and John White. A meta-learning failure predictor for BlueGene/L systems. In *ICPP*, September 2007.

[73] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[74] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Symposium on Operating Systems Principles*, October 1983.

[75] Marvin Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *SOSP*, pages 2–12, 1985.

[76] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[77] Amnon Barak and Richard Wheeler. MOSIX: An integrated multiprocessor UNIX. In *Proceedings of the Winter 1989 USENIX Conference*, pages 101–112, Berkeley, CA, USA, 1989. USENIX.

[78] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21(8):757–785, 1991.

[79] Cong Du and Xian-He Sun. MPI-Mitten: Enabling migration technology in MPI. In *IEEE CCGrid*, 2006.

[80] Cong Du, Xian-He Sun, and Kasidit Chanchio. HPCM: A pre-compiler aided middleware for the mobility of legacy code. In *IEEE Cluster*, 2003.

[81] Geoffroy Vallée, Kulathep Charoenpornwattana, Christian Engelmann, Anand Tikotekar, Chokchai (Box) Leangsuksun, Thomas Naughton, and Stephen L. Scott. A framework for proactive fault tolerance. In *ARES*, pages 659–664, 2007.

[82] M. T. Huda, H. W. Schmidt, and I. D. Peake. An agent oriented proactive fault-tolerant framework for grid computing. In *International Conference on e-Science and Grid Computing*, 2005.

[83] Yawei Li, Prashasta Gujrati, Zhiling Lan, and Xian-He Sun. Fault-driven re-scheduling for improving system-level fault resilience. In *ICPP*, 2007.

[84] Xian-He Sun, Zhiling Lan, Yawei Li, Hui Jin, and Ziming Zheng. Towards a fault-aware computing environment. In *HAPCW*, March 2008.

[85] Cong Du, Xian-He Sun, and Ming Wu. Dynamic scheduling with process migration. In *IEEE CCGrid*, May 2007.

[86] Samuel H. Russ, Rashid Jean-Baptiste, Tangirala S. Kumar, and Marion Harmon. Transparent real-time monitoring in mpi. In *Springer*, 1999.

[87] German Florez, Zhen Liu, Susan M. Bridges, Anthony Skjellum, and Rayford B. Vaughn. Lightweight monitoring of mpi programs in real time. In *Concurr. Comput.: Pract. Exper.*, 2005.

[88] C. Clark, K. Fraser, S. Hand, J.G. Hansem, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, May 2005.

[89] A. Menon, A. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Conference*, June 2006.

[90] J. Liu, W. Huang, B. Abali, and D. Panda. High performance vmm-bypass I/O in virtual machines. In *USENIX Conference*, June 2006.

[91] H. Gunawi, V. Prabhakaran, S. Krishnan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *Symposium on Operating Systems Principles*, October 2007.

[92] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi abd Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Symposium on Operating Systems Principles*, pages 206 – 220, October 2005.

[93] Q. Xin, E. Miller, and T. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, pages 172–181, June 2004.

[94] Panasas Invents 'Tiered Parity'. feature, HPCwire, 2007.

[95] Panasas tiered parity architecture. white paper, Panasas, 2008.

[96] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zhikun Wang, and Zhenlei Song. Pro: a popularity-based multi-threaded reconstruction optimization for raid-structured storage systems. In *USENIX Conference on File and Storage Technologies*, pages 32–32, Berkeley, CA, USA, 2007. USENIX Association.

[97] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[98] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the ACM SIGCOMM Conference*, 2002.

[99] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[100] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Operating Systems Design and Implementation*, November 2006.

[101] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.

[102] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Symposium on Operating Systems Principles*, 2003.

[103] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

[104] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation*, March 2004.

[105] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[106] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM Conference*, 1998.

[107] J. Plank, A. Buchsbaum, R. Collins, and M. Thomason. Small parity-check erasure codes - exploration and observations. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[108] Jay J. Wylie and Ram Swaminathan. Determining fault tolerance of xor-based erasure codes efficiently. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP Interna-*

tional Conference on Dependable Systems and Networks, pages 206–215, Washington, DC, USA, 2007. IEEE Computer Society.

[109] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[110] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration in hpc environments. In *Supercomputing*, 2008.

[111] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Hybrid Full/Incremental Checkpoint/Restart for mpi jobs in hpc environments. In *TR 2009-14, Dept. of Computer Science, North Carolina State University*, 2009.

[112] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration and back migration in hpc environments. In *TR 2009-15, Dept. of Computer Science, North Carolina State University*, 2009.

[113] C. Wang, Z. Zhang, S. Vazhkudai, X. Ma, and F. Mueller. On-the-fly recovery of job input data in supercomputers. In *ICPP*, 2008.

[114] C. Wang, Z. Zhang, X. Ma, S. Vazhkudai, and F. Mueller. Improving the availability of supercomputer job input data using temporal replication. In *ISC*, 2009.