RELIABLE AND SCALABLE CHECKPOINTING SYSTEMS FOR DISTRIBUTED COMPUTING ENVIRONMENTS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Tanzima Zerin Islam

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2013

Purdue University

West Lafayette, Indiana

To my parents and my husband.

ACKNOWLEDGMENTS

First and foremost, I thank the Almighty for letting me come this far and for all of the blessings on the way. I would like to express my sincere gratitude to my advisor – professor Saurabh Bagchi, for his extraordinary guidance, understanding, and support over these years. He has taught me, both consciously and unconsciously, how good research is done. I appreciate all his contributions of time, ideas, and feedback to make my Ph.D. experience productive and stimulating. The enthusiasm he has for his research was contagious and motivational for me, even during tough times in the Ph.D. pursuit. His advice both on research as well as on my career have been invaluable.

A special thanks to my co-chair – professor Rudolf Eigenmann, for his valuable input, guidance, and funding during the entire period of my Ph.D. experience. "They no longer announce a winner at the Academic Awards (Oscar). Rather, it "goes" to an individual. Being nominated is a win in itself." – is what he would say after every time I returned disheartened without the Best Paper Award. Over the course of my graduate life, I made a number of different variants of this quote, preserving the basic idea but tailoring to suite different rejection scenarios. The teaching never fails to inspire me and I am sure, there will be ample opportunities for me to derive many more variations in future.

I would like to thank my dissertation committee members, professor Jan P. Allebach and Bronis R. de Supinski for serving as my committee members and letting my defense be an enjoyable moment.

I cannot thank my stars more for the fateful encounter with Tony Baylis, who is also known as – "he who makes everything happen", amidst thousands of people in a large career fair. Without Tony, I would not have gone for an internship at Lawrence Livermore National Laboratory (LLNL) in the first place. Also, my special thanks to

Bronis for giving me the opportunity to work with him during multiple summers at the lab and continuing the collaboration afterwards. I am thankful for the consistent access to the vast resources at LLNL, without which my research would not have been possible. I would also like to acknowledge my colleagues Adam Moody and Todd Gamblin for their input, time, and patience during our three years of remote collaboration, and especially Kathryn Mohror, who I also look up to as a female computer scientist. She has been my mentor for last three years and was always there to encourage me, guide me, and show her support with her kind words. She has been such an inspiration, and I would like to take this opportunity to say it again – Kathryn, you are the best!

Thank you to my fellow students in the Dependable Computing Systems Laboratory (DCSL) for sitting through numerous practice talks, and providing me with your ideas to improve my presentation skill. Also, it will not be complete if I do not acknowledge my NEES family – Gemez Marshall, Brian Rohler, and Dawn Weisman for being such accommodating supervisors and providing me with the most flexible work environment possible. The bar of my expectation has been raised.

My special thanks to the outstanding people of ECE Graduate Office and Purdue ISS for ensuring seemless administrative experience for all at Purdue.

The list of people, without whom this journey would have been impossible, is just too long. However, I would like to take this opportunity to name a few here. I have been blessed with a family – my parents, Hasina Islam and Mohammad Sirajul Islam, my brother Hasibul Islam, my sister-in-law Dr. Farhana Haque, my in-laws, Khaleda Akhter and Nazmul Haque, and my brother-in-law Khaled Hassan – who were always there to encourage me, cheer for me, and console me given the appropriate scenario. My special thanks to my lifetime hero – my maternal grandfather Dr. Mohammad Siddiqullah who has taught me to work with sincerity, to be persistent, and to pray for whatever occassion that may arise in life. I have had ample opportunities to practice all three during my graduate life, and appreciate the teaching much more than I did 5 years ago.

A big thanks to my Bangladeshi family here at Purdue for creating a home away from home. I would especially like to thank my brother Asaduzzaman Mohammad for being there through thick and thin. My sincere gratitude to the people of Bangladesh for providing me with a world-class education for free, even though their living standard is not high enough.

Last, but certainly not least – I would like to take this opportunity to thank my best friend in the whole wide world – my husband, Dr. Mohammad Sajjad Hossain, without whom, simply put – I could not have done it. Your courage in leaving your own Ph.D. in the middle to move to Purdue and continuing here instead, speaks volume about how supportive you have been throughout my journey. Could I have done the same for you? I am glad, I don't have to find out. You are the rock of my life.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

ABSTRACT xii

1 INTRODUCTION 1

1.1 Reliability in Distributed Computing Environments 3

1.2 Resilience Through Checkpoint/Recovery 5

1.2.1 Challenges in High-Throughput Systems 5

1.2.2 Challenges in High Performance Computing Clusters 6

1.2.3 Summary 7

1.3 Dissertation Contributions 7

1.3.1 Checkpointing System for Grid 7

1.3.2 Checkpointing System for Clusters 9

1.4 Dissertation Organization 10

2 RELIABLE CHECKPOINTING SYSTEM FOR FINE-GRAINED CYCLE SHARING SYSTEMS 11

2.1 Background on Failure-Aware Checkpointing 13

2.2 Design for Robust Checkpointing 15

2.2.1 Novel Multi-state Failure Model for Storage Hosts 16

2.2.2 Failure-aware Storage Selection 18

2.2.3 Single-Threaded Method for Checkpoint Recovery 22

2.2.4 Data Parallelism and Parallel Architecture 23

2.2.5 Design of Parallel Checkpoint-Recovery Scheme 24

2.3 Structure of FALCON 25

2.3.1 Compute Host Component (CHC) 27

Page

2.3.2 Storage Host Component (SHC) 28
2.3.3 History Server Component (HSC) 29
2.4 Evaluation 29
2.4.1 Macro Benchmark Experiments 31
2.4.2 Micro Benchmark Experiments 35
2.4.3 Single-threaded vs Multi-threaded Architecture of FALCON . 42
2.5 Related Work 46
2.6 Summary 47
3 SCALABLE CHECKPOINTING SYSTEM USING DATA-AWARE AGGREGATION AND COMPRESSION 48
3.1 Background 50
3.1.1 Application-level vs System-level Checkpointing 51
3.1.2 Checkpoint Writing 51
3.1.3 Checkpoint File Format 52
3.1.4 Checkpoint Data 54
3.2 Data-Aware Checkpoint Aggregation & Compression 56
3.2.1 Identifying Similarity Across Checkpoints 58
3.2.2 Merging Schemes 58
3.3 Structure of MCRENGINE 60
3.3.1 Checkpointing Phase 60
3.3.2 Restart Phase 63
3.4 Evaluation Methodology 64
3.4.1 Applications 64
3.4.2 Evaluation of Compression Schemes 66
3.5 Data-Aware Compression Effectiveness 66
3.5.1 Benefit of Multiple Passes of Compression 67
3.5.2 Change in Compression Ratio with Varying Group Size . . . 68
3.5.3 Impact of Interleaving Granularity on Compression Ratio . . 72

Page

3.5.4 Change in Compression Ratio as Simulation Progresses . . . 72
3.5.5 Summary of Data-Aware Compression Effectiveness 73
3.6 Performance of mcrEngine 73
3.6.1 Data-aware and Data-Agnostic I/O Performance 74
3.6.2 Benefit of Checkpoint Aggregation 75
3.6.3 Checkpoint and Restart Overheads 76
3.6.4 Discussion 78
3.7 Related Work 79
3.8 Summary 80
4 BENEFIT-AWARE CLUSTERING FOR PARALLEL APPLICATIONS 82
4.1 Background 83
4.2 Benefit-Aware Clustering 84
4.2.1 Number of Groups 85
4.2.2 New Similarity Metric 85
4.2.3 Sampling Method 87
4.3 Runtime Clustering 87
4.3.1 Reduce Dimension of β 87
4.3.2 Reduce Data to Cluster 88
4.3.3 Centralized vs Distributed Clustering 89
4.4 Structure of mcrCluster 89
4.4.1 Algorithm for Building Benefit-Matrix in Runtime 92
4.5 Evaluation of mcrCluster 93
4.5.1 Effectiveness of Benefit Metric in Finding Clusters 94
4.5.2 Impact of Benefit-Aware Clustering on Compression Ratio . 96
4.6 Discussion 96
5 Conclusion 98
LIST OF REFERENCES 100
VITA 106

LIST OF TABLES

Table — Page

2.1 Checkpoint sizes of different applications. 29

2.2 Statistical analysis of the eviction characteristics in DiaGrid. 32

2.3 Breakdown of different schemes evaluated in Figure 2.9. 41

2.4 Input sizes to different algorithms of FALCON and FALCON-P. 45

3.1 Module interactions during a checkpointing phase 61

3.2 Module interactions during a restart phase 64

3.3 Checkpoint characteristics 64

3.4 Relative improvement over *Agnostic* 73

3.5 Checkpoint/restart system configurations 74

4.1 Comparison of compression ratios for different clustering schemes. . . . 96

LIST OF FIGURES

Figure Page

1.1 Statistics based on DiaGrid. 2

1.2 Figure (a) shows how computing power is projected to grow; Figure (b) shows how processor count has increased over the years for the number 1 supercomputer since 1993 on the Top500 list. 4

2.1 New multi-state storage host failure model. 17

2.2 System level block diagram of our system. 26

2.3 Average job makespan of different applications. 33

2.4 Recovery overhead of four different checkpoint sizes. 34

2.5 Average execution time vs the number of concurrent clients. 36

2.6 Average checkpointing overhead vs the number of unavailable storage hosts. 38

2.7 Average checkpoint storing overhead of different schemes. 39

2.8 Parallel vs sequential retrieval during restart. 40

2.9 Contributions of compression, load balancing, and parallel network transfer during restart. 41

2.10 Improvement in compression overhead from using multiple cores. 43

2.11 Comparison of the checkpointing and the recovery overheads of FALCON and FALCON-P. 44

3.1 Two types of input data distribution 52

3.2 Different metadata annotations of checkpoint data 53

3.3 Volume rendering of the $3D$ variable: “Density” 54

3.4 Refined nested computation 55

3.5 Projections of Temperature, Pressure, and Density variables along X axis for Timestep 8. 56

3.6 Variable matching 57

3.7 Data-aware merging schemes 59

Figure Page

3.8 System-level interactions 62
3.9 Double compression vs single compression 68
3.10 Compression ratio vs group size. 70
3.11 ALE3D compression ratio ($Group_Size = 32$) 70
3.12 Compression ratio over time 71
3.13 I/O performance with different compression schemes 75
3.14 Benefit of aggregation for I/O performance 76
3.15 End-to-end checkpointing overhead 77
3.16 End-to-end restart overhead 78
4.1 Benefit matrices of the application Cactus. 88
4.2 Overall structure of benefit-aware clustering and data-aware compression. 91
4.3 Benefit matrix and silhouette plot of 32 checkpoints from Cactus 95

ABSTRACT

Islam, Tanzima Zerin Ph.D., Purdue University, May 2013. Reliable and Scalable Checkpointing Systems for Distributed Computing Environments. Major Professor: Saurabh Bagchi.

By leveraging the enormous amount of computational capabilities, scientists today are being able to make significant progress in solving problems, ranging from finding cure to cancer – to using fusion in solving world's clean energy crisis. The number of computational components in extreme scale computing environments is growing exponentially. Since the failure rate of each component starts factoring in, the reliability of overall systems decreases proportionately. Hence, in spite of having enormous computational capabilities, these ground breaking simulations may never run to completion. The only way to ensure their timely completion, is by making these systems reliable, so that no failure can hinder the progress of science.

On such systems, long running scientific applications periodically store their execution states in *checkpoint files* on stable storage, and recover from a failure by restarting from the last saved checkpoint file. Resilient high-throughput and high-performance systems enable applications to simulate scientific problems at granularities finer than ever thought possible. Unfortunately, this explosion in scientific computing capabilities generates large amounts of state. As a result, todays checkpointing systems crumble under the increased amount of checkpoint data. Additionally, the network I/O bandwidth is not growing nearly as fast as the compute cycles. These two factors have caused scalability challenges for checkpointing systems. The focus of this thesis is to develop scalable checkpointing systems for two different execution environments – high-throughput grids and high-performance clusters.

In grid environment, machine owners voluntarily share their idle CPU cycles with other users of the system, as long as the performance degradation of host processes remain under certain threshold. The challenge of such an environment is to ensure end-to-end application performance given the high-rate of unavailability of machines and that of guest-job eviction. Today's systems often use expensive, high-performance dedicated checkpoint servers. In this thesis, we present a system, falcon, that uses available disk resources of the grid machines as shared checkpoint repositories. However, an unavailable storage host may lead to loss of checkpoint data. Therefore, we model the failures of storage hosts and predict the availability of checkpoint repositories. Experiments run on production high-throughput system – DiaGrid show that falcon improves the overall performance of benchmark applications, that write gigabytes of checkpoint data, between 11% and 44% compared to the widely used Condor checkpointing solutions.

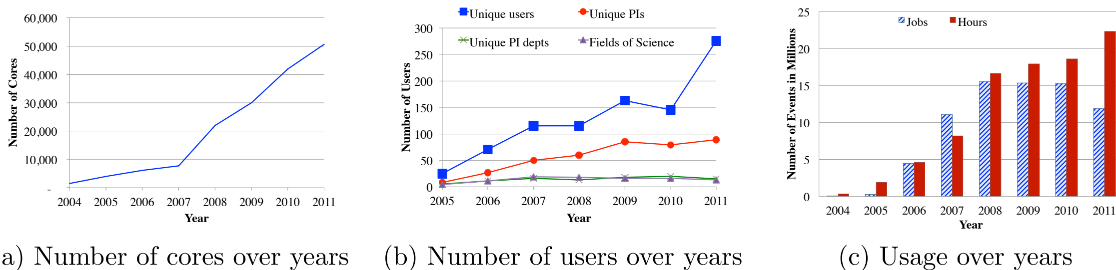
In high-performance computing (HPC) systems, applications store their states in checkpoints on a parallel file system (PFS). As applications scale up, checkpoint-restart incurs high overheads due to contention for PFS resources. The high overheads force large-scale applications to reduce checkpoint frequency, which means more compute time is lost in the event of failure. We alleviate this problem by developing a scalable checkpoint-restart system, mcrEngine. mcrEngine aggregates checkpoints from multiple application processes with knowledge of the data semantics available through widely-used I/O libraries, e.g., HDF5 and netCDF, and compresses them. Our novel scheme improves compressibility of checkpoints up to 115% over simple concatenation and compression. Our evaluation with large-scale application checkpoints show that mcrEngine reduces checkpointing overhead by up to 87% and restart overhead by up to 62% over a baseline with no aggregation or compression.

We believe that the contributions made in this thesis serve as a good foundation for further research in improving scalability of checkpointing systems in large-scale, distributed computing environments.

1. INTRODUCTION

```
"...You know you have a distributed system when the crash of a
computer you have never heard of stops you from getting any work
done..."
-- Leslie Lamport
```

With the explosion in the number of computational resources in large-scale distributed systems – failures will be continuous rather than exceptional events. The growth of high-performance computing (*HPC*) clusters and the recent paradigm shift towards harvesting idle cycles of hosts connected to the Internet (*cycle sharing systems*), made possible a lot of scientific applications. These applications vary from decoding the physics of universe to delivering the next break through in medicine. A couple of well known applications that use high-performance and high-throughput computing are – fusion reaction to generate clean energy for lighting up a star by National Ignition Facility at Lawrence Livermore National Lab (high-performance computing), and designing new proteins to fight against cancer by Rosetta@Home (high-throughput computing). These long running scientific applications use hundreds of thousands of hours of distributed computing to finish. Resources in cycle sharing systems have highly fluctuating availability since they are often owned by individuals who share their idle resources voluntarily. On the other hand, HPC clusters are much more reliable since these resources are owned and maintained by an organization whose mission is to provide uninterrupted computing environment to long running applications. However, related research shows that with exascale in the horizon, computational technology is approaching failure rates in the order of minutes. Due of their scale and complexity, the mean time between failure (MTBF) of today's



(a) Number of cores over years (b) Number of users over years (c) Usage over years

Fig. 1.1.: Statistics based on DiaGrid.

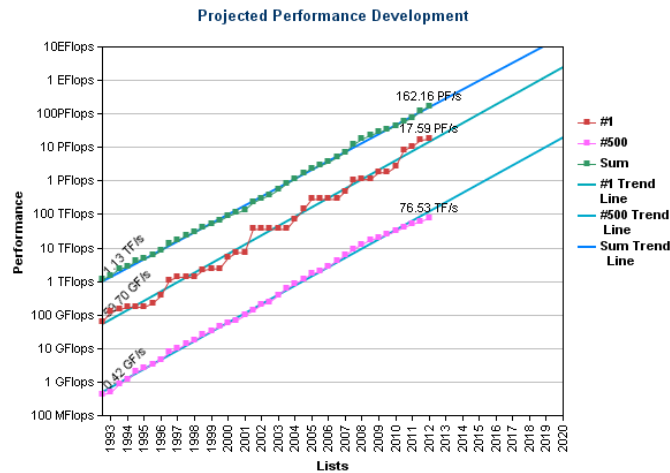
large-scale distributed systems is decreasing, making resilience and fault-tolerance in these environments a major challenge to be overcome.

Grid computing is the federation of computer resources from multiple locations to reach a common goal. The grid can be thought of as a distributed system with non-interactive workloads that involve a large number of files. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform large tasks. DiaGrid is an example of such a large, multi-university, high-throughput computing environment that is centered at Purdue University. In 2012, it included nearly 43,000 processors representing 301 teraflops of computing power. DiaGrid spans 10 different university campuses in the midwest. DiaGrid uses a popular middleware, called Condor, for managing resources and scheduling jobs. Grid resources can be configured so that foreign jobs can coexist with local jobs that are owned by the machine owner. In this environment, resources can become unavailable if slowdown of local jobs is observable. This type of system is also known as *Fine-Grained Cycle Sharing System (FGCS)*.

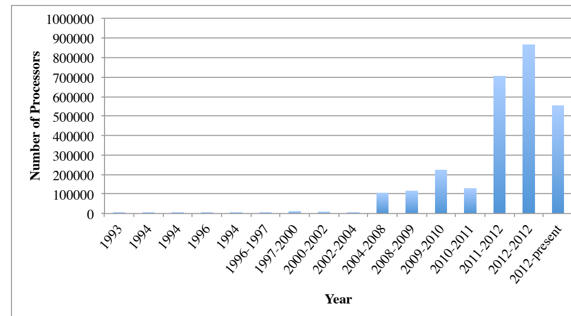
1.1 Reliability in Distributed Computing Environments

On one end of the spectrum, we have FGCS systems, where machine owners share their idle CPU cycles voluntarily with guest jobs. Recent paradigm shift towards volunteering computing has made grid computing quite popular. For example, Figure 1.1 shows the growth of DiaGrid, in terms of the number of cores, the number of users, and the number of hours spent by applications on these resources between 2004 and 2011. Given that more number of scientific applications are utilizing these virtually infinite resources for free, ensuring their reliable execution has become more challenging than ever. However, since these resources are mostly voluntarily shared, they can and often do become unavailable any time, due to reasons ranging from unexpected software, hardware failures to users unexpectedly turning off their machines. Without proper reliability measures, these guest jobs will never be able to run to completion, which beats the whole purpose of having such an environment in the first place. In Fine-Grained Cycle Sharing (FGCS) systems, machine owners voluntarily share their idle CPU cycles with guest jobs, as long as their (host processes) performance degradation is below certain threshold. However, for guest users, free resources come at the cost of highly fluctuating availability. Such unpredictable but frequent evictions from host machines lead to unpredictable completion times for guest applications. In this environment, *checkpoint-recovery* is a widely used technique for recovering from such "failures". Evictions may occur due to software or hardware failure, host work load increasing beyond a threshold or simply because owner of the machine has returned.

On the other hand, computational power of HPC systems is growing exponentially, which enables finer-grained scientific simulations. For example, Figure 1.2a, shows how aggregated computation power has increased since the first Top500 list came out in 1993. Since then, the number of components has grown leaps and bounds as well. For example, Figure 1.2b, shows the number of processors reported by the number 1 supercomputer of the corresponding year. The trend shows these systems to gather several millions of CPU cores, not including other hardware components. As software and hardware component counts in high performance computing (HPC) systems scale



(a) Projected performance development over time



(b) Number of processors over time

Fig. 1.2.: Figure (a) shows how computing power is projected to grow; Figure (b) shows how processor count has increased over the years for the number 1 supercomputer since 1993 on the Top500 list.

up, the likelihood grows of one failing while an application executes. For example, the 100,000 node BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) experiences an L1 cache parity error every 8 hours [1] and a hard failure every $7-10$ days. According to Schroeder *et al.*, exascale systems are projected to fail every $3-26$ minutes [2, 3]. Also, according to Sato *et al.*, in the year and a half from November 1st 2010 to April 6th 2012, TSUBAME2.0, ranking 5th in the Top500 list [4] (November 2011), experienced 962 node failures ranging from memory errors to whole rack failures. Thus, a failure occurred every 13.0 hours on average.

Furthermore, the MTBF (mean time between failure) of future systems is projected to shrink to tens of minutes [2]. Without a viable resilience strategy, applications will be unable to run for even one day on such a large machine. Thus, resilience in HPC has become more important than ever as we plan for future systems.

1.2 Resilience Through Checkpoint/Recovery

To achieve high performance in the presence of failures and resource volatility, applications use *checkpointing* and *rollback* [5, 6], which periodically saves application state in *checkpoint files* on stable storage. If a failure occurs, the application is restarted from the latest checkpoint, thus reducing repeated computation. To simplify checkpoint-restart implementations, many applications have all processes take checkpoints simultaneously [7]. This strategy avoids the complexities of message logging that uncoordinated checkpointing requires, as well as its possibility of cascading rollback.

1.2.1 Challenges in High-Throughput Systems

Most production FGCS systems, such as Condor, store checkpoints on dedicated checkpoint servers. These are few in number, well-provisioned, and maintained such that 24×7 availability is achieved. This solution works well when a cluster only belongs to a small administrative domain. However, storing checkpoints on a dedicated pool of servers will not scale with the growing size of grids having thousands of home users, multiple university campuses, and research labs as participants across the whole nation. A number of performance and feasibility issues must be considered.

- FGCS systems do not include a dedicated network that can efficiently handle the load of transferring potentially gigabytes of checkpoints between a *compute host* or *execution host* (the host on which the guest process is executing) or the *storage hosts* (hosts that have contributed storage). Moreover, for multi-university grids, current mechanism for storing checkpoints in dedicated servers

is inefficient. This is because the compute and storage hosts may be located at distant places.

- A mechanism based on round trip time (RTT) to choose the closest storage host for saving checkpoint files may result in huge network traffic during network transfer, making it less preferable than a distant one.

Real users using DiaGrid complained about poor performance of their applications due to the stated reasons.

1.2.2 Challenges in High Performance Computing Clusters

In HPC clusters, a large number of computing resources are generally connected to each other via high speed local area networks and a parallel file system (PFS) where generated output files such as visualization and restart dumps get stored.

The major problem that threatens to make today's checkpointing systems unscalable is the increase in the amount of data being generated. For example, pF3D, an application for studying laser-plasma instabilities [8], generates $2.15TB$ of checkpoint data for 4096 processes on 512 nodes [9]. The increase in computation capability in terms of CPU and memory enables applications to solve much more complex and intricate problems faster and at finer resolutions. Thanks to today's supercomputers, settling for a coarse granularity is a distant past. The underlying implication of large-scale computing is that these applications generate progressively larger volume of output files. These files could potentially be output files in the form of visualization dumps or memory snapshots, also known as "checkpoint files", that are used to recovery from failures. Related literature [10, 11] has already established that the increase in component count in large-scale clusters reduces the mean time between failure. Hence, failures and the need to recover the already completed computation is a common case.

However, since network bandwidth is not growing proportionately with the increase in the computational resources, this also indicates that with the increase in

the volume of data, any system with large volume of data transfer between different parts of a network will not scale as well. For example, visualization files for a climate modeling application can be several peta bytes depending on the resolution of the problem [12].

The implications of disproportionate growth of the volume of data and network bandwidth are two fold – (i) reduced checkpointing frequency – since it takes longer to store them on PFS, and (ii) increased overall application completion times since restarts, that take place in the critical path of applications, are becoming progressively more expensive.

1.2.3 Summary

Given that reliability is a big challenge for ever growing large-scale computing environments and the current solutions for providing fault-tolerance to applications will not scale, in this thesis we focus on developing scalable checkpointing systems for these computing environments. We propose analytical solutions to improve reliability and availability of checkpointing systems, designed and developed scalable systems to handle large volume of checkpoint data, and demonstrated the effectiveness of our solutions on production systems.

1.3 Dissertation Contributions

In this section, we summarize the contributions of this thesis for the projects we worked on so far.

1.3.1 Checkpointing System for Grid

We store checkpoints in a distributed manner on grid resources. This is different from the state-of-the-art checkpointing systems in grid since production systems such as Condor [13] store them on dedicated checkpoint servers. Checkpointing systems

with dedicated checkpoint storage will not scale well with the growing sizes of grids having machines in geographically distributed locations. However, since grid resources have high fluctuating availability, checkpoints stored on them may not be available when needed. To ensure reliable execution of applications, we develop a *multi-state failure model* for predicting availability and load on storage hosts. Previous work has only considered a failure model for compute hosts [14]. These two models are different since they consider two different types of resources – computational (CPU and memory) and storage (disk I/O).

Another mechanism available in Condor is to use multiple storage servers provisioned throughout the grid, selecting the closest storage server based on round trip time (RTT) for saving data. However, this solution will not work well since a physically close node may observe huge network traffic during checkpointing intervals, making it less preferable than a distant one. In contrast, we develop a *failure-aware storage selection technique* that selects a set of reliable and lightly loaded storage hosts for a compute host, based on their history of availability and load. Previous work has not considered the multiplicity of factors related to storage hosts that affect the performance of checkpointing and recovery [14, 15].

Grids do not include dedicated networks that can efficiently handle the load of transferring potentially gigabytes of data between compute hosts and storage servers. In contrast, our scheme selects a set of storage hosts based on their availability and available bandwidth between the compute machine and the storage hosts. Also, we develop *an efficient method* that provides fault-tolerance to the process of checkpointing data as well as uses parallelism offered by multiple fragments being stored on multiple storage hosts to reduce checkpoint and recovery overheads. This approach leverages prior work in erasure coding for fault-tolerance [16] while using it in a different context (shared grid environments) and using the parallelism afforded by it.

We have implemented and evaluated FALCON on the production Condor testbed of Purdue University – DiaGrid. The experiments ran on DiaGrid show that perfor-

mance of an application with FALCON improves between 11% and 44%, depending on the size of checkpoints and whether the storage server for Condor's solution was located close to the compute host. Also, we show that the performance of FALCON scales as the checkpoint sizes of different scientific applications increase.

Section 2.5 discusses studies that apply the knowledge of failure awareness in resource scheduling. However, none of them address the problem of selecting reliable checkpoint repositories in shared storage environment and balance load across them. In contrast, FALCON employs some well-known techniques, to improve fault-tolerance of data and to improve performance of guest processes. Distinct from prior work, we try to address the unanswered issues of choosing reliable storage nodes in a shared grid environment, balancing load across them, and finally using them for storing and retrieving checkpoints.

1.3.2 Checkpointing System for Clusters

On the other hand, for checkpointing systems in HPC, the problem is twofold: *first*, the increase in aggregate volume of data; *second*, the increase in contention due to a large-number of concurrent transfers. Since checkpoints are the primary source of today's large volume of data, we looked at the scalability issue of today's checkpointing systems as a case study. In order to achieve tolerable performance, applications may need to reduce their checkpointing frequency, which means more compute time is lost in the event of a failure. To address these challenges, we designed a novel technique, called "Data-Aware Compression", and developed and deployed our system, MCRENGINE, on a supercomputer at Lawrence Livermore National Laboratory. Other researchers have investigated reducing the size of checkpoints by writing less data in the checkpoints or by compressing the checkpoint files. Incremental checkpointing reduces the size of checkpoints by writing changes in application data between full checkpoints [17–20]. These approaches are orthogonal to our work, as incremental checkpoints can be compressed for further savings [21].

The state-of-practice in writing checkpoint is either $N \rightarrow 1$ (one process collects all checkpoints and writes to stable storage) or $N \rightarrow N$ (each process transfers checkpoint directly to stable storage). The best compromise is to implement $N \rightarrow M$ (where $M << N$) checkpoint transfer. However, this scheme is significantly complex for application developers to implement. Our solution is a checkpointing system that implements $N \rightarrow M$ scheme transparently to application developers.

To summarize, the major contributions of mcrEngine are–

- *transparently aggregate* checkpoints across processes in a data-aware manner,
- *applies data-aware compression* to gain the most reduction possible,
- *stores aggregated and compressed data* in a buffered manner to the PFS.

Our solution tackels both of the problems by reducing the number of concurrent transfers and the total size of data transferred between compute nodes and the PFS. Section 3.7 discusses a number of related work that looks at system-level checkpoint compression. These approaches all use data-agnostic compression while we investigate data-aware techniques. To the best of our knowledge, we are the first to investigate a data-aware compression approach and cross-process merge and compression for parallel applications with many processes.

1.4 Dissertation Organization

We present our solution to making checkpointing systems reliable and efficient in Chapter 2.6. Chapter 3.8 presents a novel technique of data semantic-aware aggregation and compression. In Chapter 4.6 we present preliminary results of benefit-aware clustering of checkpoints to improve overall compression ratio. Finally, we conclude in Chapter 5.

2. RELIABLE CHECKPOINTING SYSTEM FOR FINE-GRAINED CYCLE SHARING SYSTEMS

A Fine-Grained Cycle Sharing (FGCS) system [22] aims at utilizing the large amount of idle computational resources available on the Internet. In such a cycle sharing system, PC owners voluntarily make their CPU cycles available as part of a shared computing environment, but only if they incur no significant inconvenience from letting a foreign job (*guest process*) run on their own machines. To exploit available idle cycles under this restriction, an FGCS system allows a guest process to run *concurrently* with jobs belonging to the machine owner (*host processes*). However, for guest users, these free computation resources come at the cost of fluctuating availability due to "failures". Here we define failures to be due either to the eviction of a guest process from machines due to resource contention, or due to conventional hardware and software failures of machines. The primary victims of such resource volatility are large compute-bound guest applications, whose completion times fluctuate widely due to this effect. Most of these applications are either sequential or composed of several coarse-grained tasks with little communication in between.

To achieve high performance in the presence of resource volatility, *checkpointing* and *rollback* have been widely applied [5]. These techniques enable applications to periodically save checkpoints - snapshots of applications' states - onto stable storage that is connected to the computation node(s) through network. A job may get evicted from its execution machine any time and can recover from this failure by rolling back to the latest checkpoint. Evictions may occur due to software or hardware failure, host work load increasing beyond a threshold or simply owner of the machine has returned.

Most production FGCS systems, such as Condor, store checkpoints to dedicated storage servers. These are few in number, are well-provisioned, and maintained such

that 24×7 availability is achieved. This solution works well when a cluster only belongs to a small administrative domain or there is a large number of storage servers. However, it does not scale well with the growing sizes of grids having thousands of home users, geographically separated university campuses, and research labs as participants. For example, the production Condor pool at Purdue University (PU), called "DiaGrid" [23], is one of the largest such pools in the country with 36,000 processors. It has machines "flocking" from Indiana University (IU), University of Notre Dame (ND), and Purdue University Calumet (PUC). A number of performance and feasibility issues have been encountered by researchers running jobs on DiaGrid.

First, FGCS systems do not include dedicated networks that can efficiently handle the load of transferring potentially gigabytes of checkpoints between *compute hosts (CH)* (hosts on which guest processes execute) and *storage hosts (SH)* (hosts that contribute storage space). Moreover, for multi-university grids, the current mechanism for storing checkpoints on dedicated servers may cause large network latencies since the compute and storage hosts may be located at large network distances. We collected traces that show 12% of jobs submitted to DiaGrid between March 5th, 2009 and March 12th, 2009 from PU actually ran on ND's machines.

Second, even if multiple storage servers could be provisioned and made available throughout the grid, selecting the closest storage hosts based on round trip time (RTT) for saving data, may not perform well. The reason is that a physically close node may observe huge network traffic during checkpointing intervals, making it less preferable than a distant one. This mechanism is also available in Condor.

Third, a dedicated storage server becomes loaded as the number of concurrent transfers increases—which ultimately causes degradation in the performance of these guest applications.

Our work is motivated by these issues reported by scientists using DiaGrid for running their compute-intensive jobs with checkpoint-recovery. In this work, we have developed a framework for reliable execution of applications in a *shared storage environment*—an environment where a host can serve as both an execution node for a

guest job as well as a storage host for saving checkpoints of others—as opposed to dedicated checkpoint servers. For this, we first propose a novel multi-state failure model for the shared storage hosts. Then, we propose a failure prediction scheme and apply this to the multi-state failure model to choose reliable and less loaded storage nodes to serve as checkpoint repositories. Finally, we propose an algorithm for efficient checkpoint storage and recovery. This algorithm uses erasure encoding [16] to break checkpoint data into multiple fragments, such that the checkpoint data can be reconstructed from a subset of the fragments. We realize our algorithms in a practical system called FALCON.

The rest of the paper is organized as follows. Section 2.1 presents a comprehensive summary of our previous work on failure-aware checkpointing and resource availability prediction. Our major contributions and system designs for both FALCON and FALCON-P are described in detail in Section 2.2. Section 2.3 presents implementation details of FALCON. Then, experimental approaches and results are discussed in Section 2.4. Section 2.5 reviews the relevant literature.

2.1 Background on Failure-Aware Checkpointing

Failure-aware checkpointing builds on mechanisms that predict the availability of the involved compute and storage hosts. In our previously developed prediction techniques [24], we applied a *multi-state failure model* to predict the *Temporal Reliability*, TR, of compute hosts. TR is the probability that a host will be available throughout a given future time window. Quantitatively, $TR(x)$ is the probability that there will be no failure between now and time x in the future. To compute TR, we applied a Semi Markov Process (SMP) model, where the probability of transitioning to a state in the future depends on the current state and the time spent in this state. The parameters of this model are calculated from the host resource usages during the same time window on previous days, since in many environments, the daily pattern of host workloads are comparable to those in the most recent days [25].

In other work [14], we proposed two algorithms for selecting reliable storage hosts in an FGCS system, where non-dedicated host machines provide disk storage for saving checkpoint data. A checkpoint is taken periodically and contains the entire memory state of an application. A checkpoint is used by a job for recovery when it gets rescheduled to another machine after the current host fails. For reliable storage in an FGCS system, we considered two criteria: the network overhead due to saving and recovery of checkpoints, and the availability of the storage hosts. This work [14] applied the knowledge of network connectivity and of resource availability to predict reliability of a checkpoint repository from a set of storage hosts. This work also proposed a *one-step look ahead* heuristic to determine the optimal checkpoint interval. It compares the cost of checkpointing immediately with the cost of delaying that to a later time and uses that to adjusted checkpoint intervals.

Our prior work left some questions unanswered. First, it applied the failure-model for compute hosts to predict the availability of storage hosts. These two kinds of hosts offer resources with different characteristics; hence they will have different failure models. Second, there was no notion of load-balancing for storage hosts. Thus, a storage host that is predicted to have high availability in the near future will see a flash crowd of large checkpoints from several concurrently executing jobs. The checkpoints are often large in size (e.g., with the mcf benchmark application that we experiment with, the size is about 1.7 GB), and this load disbalance can cause significant perturbation to the FGCS system. For example, the machine owner can see slow I/O for his own jobs during such flash crowds. Third, for predicting reliability of a storage host, we used absolute temporal reliability even though *correlated temporal reliability* is the important criterion. By correlated temporal reliability, we mean what is the likelihood of the storage host being available, *when needed*, i.e., when the compute host has a failure. It is at that time that the checkpoint is needed for recovering the guest process on a different machine. Fourth, our prior work used a static bandwidth measure, given by the network specification, to estimate the network overhead. We find that the actual bandwidth available for a large checkpoint transfer

may vary significantly from the static measure. Finally, and most significantly from an implementation and deployment effort, our prior work performed a simulation of the checkpoint-based recovery scheme, using the GridSim toolkit. In this paper, we present a fully functional system executing on Purdue's DiaGrid.

We compare the performance of FALCON with two other checkpoint repository selection schemes:

- Dedicated: This scheme uses a pre-configured checkpoint server to store checkpoints. These are generally powerful machines with very high availability. This is a supported current mode of usage for checkpointing in DiaGrid, as in many other production Condor systems.
- Random: This scheme selects storage hosts randomly. Here, we assume that this scheme employs the same checkpoint store and retrieve methods as FALCON, except that it chooses storage hosts randomly at the beginning of each checkpoint interval.

2.2 Design for Robust Checkpointing

In this section, first we discuss our proposed novel multi-state failure model for storage hosts. The roles that a host assumes exhibit different characteristics - computation nodes execute guest jobs requiring CPU and memory resources whereas storage hosts handle I/O load. Therefore, failure models for these two types of resources are different. We propose a failure model specialized to storage nodes in a shared computing environment.

Second, we propose a new failure prediction technique to select reliable checkpoint repositories by considering correlation of failures between compute and storage hosts. Third, we present an algorithm that fragments checkpoints using erasure coding and concurrently saves them to multiple storage nodes. The coding introduces redundancy such that a subset of the fragments can be used for the recovery of the application. This section discusses the single-threaded design of the compression and

the erasure encoding algorithms. Section 2.2.5 presents the multi-threaded architecture of FALCON.

2.2.1 Novel Multi-state Failure Model for Storage Hosts

In an FGCS system, storage hosts are often non-dedicated, shared nodes contributing their unused disk spaces. Checkpoint data saved by guest jobs in these storage nodes may get lost when the storage nodes become unavailable due to resource contention or resource revocation. This is of particular concern for long-running compute-intensive applications. The model for recovering a guest job when it is evicted from a machine is that the guest process migrates to another compute host and uses the last checkpoint fragments to recover and re-execute from the checkpoint. In this situation, it is clearly advantageous to choose a storage host that:

- is going to be available with high probability *when a compute host becomes unavailable*.
- is less likely to have high I/O load. This ensures load-balancing across storage hosts and is crucial for an FGCS system, since creating load on an already busy node will reduce the performance of host and guest jobs.
- has large available bandwidth to the computation node so that the transfer of checkpoint fragments incurs lower network latency.

To predict failures of storage hosts, we propose a novel multi-state failure model. In our previous work [14], we developed a failure model for compute hosts and applied it to storage hosts. Since the underlying availability models of the two types of resources - CPU cycles and disk storage - are different, applying the same failure model to both these resources is inadequate.

Figure 2.1 presents our new five-state failure model for storage hosts. The states are defined as follows:

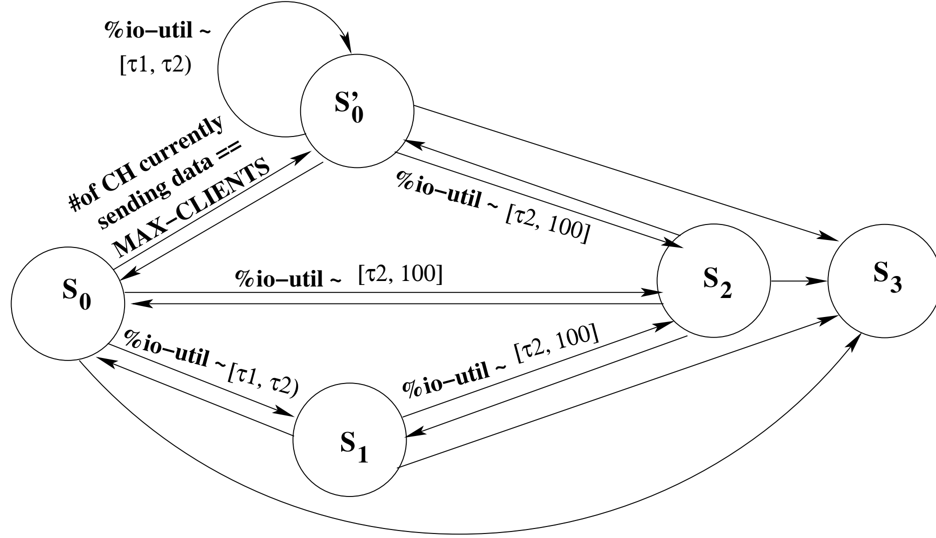


Fig. 2.1.: New multi-state storage host failure model.

(i) S_0: storage host is running with I/O load $< \tau_1$ and number of compute hosts sending checkpoint data concurrently is < MAX-CLIENTS, (ii) S_0': number of compute hosts sending checkpoint data concurrently is = MAX-CLIENTS, (iii) S_1: I/O load of storage host is between $[\tau_1, \tau_2)$, (iv) S_2: I/O load of storage host is between $[\tau_2, 100\%]$ (v) S_3: storage host is not available due to resource revocation.

Here, the states S_0' and S_2 ensure load-balancing since storage hosts in either of these states do not accept any more request for storing checkpoint data. Knowledge about states S_1 and S_2 is used during storage selection to rank storage hosts according to their likelihood of becoming loaded in the future. Note that, state S_0' has been separated from state S_1 and S_2 because this state represents a transient state of a storage host. A compute host only uses the knowledge of states running, loaded and temporarily unavailable to predict load on a storage host machine. The knowledge of S_0' is only used by the storage host to reject requests from new compute hosts for storing checkpoint and thus enforce load balancing. State S_3 is an absorbing state because we assume the failures are irrecoverable. Even if in practice the failure can be recovered, the time to recover is large enough and unpredictable enough to be useless for the current guest job.

$$CRLS(SH_k, CH_l) = \begin{cases} LI(SH_k, CH_l) + \gamma, & \text{if } CR(SH_k, CH_l) \geq \gamma; \\ CR(SH_k, CH_l), & \text{otherwise.} \end{cases} \tag{2.1}$$

where,

$$\begin{aligned} Pr_{CH_l}(i) &= Pr\{CH_l \text{ in state } i\} \\ Pr_{SH_k,CH_l}(i|j) &= Pr\{SH_k \text{ in state } i|CH_l \text{ in state } j\} \end{aligned}$$

$$CR(SH_k, CH_l) = \begin{cases} Pr_{SH_k,CH_l}(S_0|down) & \\ +Pr_{SH_k,CH_l}(S_1|down) & \\ +Pr_{SH_k,CH_l}(S_2|down), & \text{if } Pr_{CH_l}(down) > 0; \\ \gamma, & \text{otherwise.} \end{cases}$$

$$LI(SH_k, CH_l) = \begin{cases} \alpha \times Pr_{SH_k,CH_l}(S_0|up) & \\ +(1-\alpha) \times Pr_{SH_k,CH_l}(S_1|up), & \text{if } Pr_{CH_l}(up) > 0; \\ 0, & \text{otherwise.} \end{cases}$$

When a compute host requests a storage host to save checkpoint data, depending on which state the storage host is in, it replies back. When the storage host is in either S_0 or S_1, it replies "ok", and the compute host continues with sending data. Otherwise, if it is in either S_0' or S_2, it does not accept any new request.

2.2.2 Failure-aware Storage Selection

Temporal Availability

Similar to other resources in FGCS systems, checkpoint repositories are volatile. To predict availability of a storage host SH_k in a given time window with respect to a compute host CH_l, we define *Correlated Reliability Load Score (CRLS)* as:

In Equation 2.1, $Pr_{CH_l}(down)$ and $Pr_{CH_l}(up)$ denote the probability that the compute host CH_l is down (state S_0) and up (state S_1) respectively. Here, $CR(SH_k, CH_l)$ and $LI(SH_k, CH_l)$ denote the *correlated reliability* and the *load indicator* between SH_k and CH_l respectively. $CR(SH_k, CH_l)$ and $I(SH_k, CH_l)$ are probabilities while $CRLS$ is not—it is a score $\in [0, 2]$. In Equation 2.1, we first calculate $CR(SH_k, CH_l)$ as the total probability that the storage host SH_k remains up when the compute host CH_l is down. Note that, here we are adding up the probabilities corresponding to storge host SH_k being running, loaded, or temporarily unavailable. The intuition is that if the checkpoint data is needed (since the compute host has gone down), then the storage host will allow the read of the data, even if it is loaded. We consider storage hosts having $CR(SH_k, CH_l) \geq \gamma$ (γ is a configurable parameter, we chose $\gamma = 0.95$ for our experiments) as very reliable. The equation rounds the reliability component of $CRLS$ to γ since we consider that reliability scores greater than γ are high enough to be considered equivalent and also may be statistically indistinguishable due to the inherent noisiness of the measurements. Beyond this point, we would want to give weightage to the less loaded storage hosts and therefore add their load indicators to γ to make less loaded machines have high $CRLS$ value. For storage hosts having reliability ¡ γ, we do not consider their load because we want to ensure that the most reliable storage nodes get chosen first. We calculate the load indicator $LI(SH_k, CH_l)$ as a weighted probability of SH_k being in the two tolerably loaded states, namely, S_0 and S_1. The weight α is chosen as 0.75 in our experiments.

Available Bandwidth

In addition to failure-prediction, checkpoint transfer overhead is one of the key factors in storage repository selection. Our previous work [14] used effective bandwidth between a compute and a storage host to calculate network overhead. Effective bandwidth is the maximum possible bandwidth that a link can deliver. But the actual bandwidth available between a compute host and a storage host may be far less than

this quantity. So, it is more accurate to use available bandwidth between two hosts to access the overhead of transferring data between them. *Available bandwidth (ABw)* is the unused capacity of a link or end-to-end path in a network and is a time-varying metric. We define network overhead of transferring a checkpoint of size n with erasure coding parameters (m, k) from compute host CH_j to storage host SH_i in Equation 2.2. The parameters (m, k) mean that a total of $m + k$ checkpoint fragments are stored and any m of them may be used to recover the entire checkpoint.

$$\text{network overhead}, N_{i,j} = \frac{n/m}{ABw_{(SH_i,CH_j)}} \tag{2.2}$$

In Equation 2.2, $N_{i,j}$ represents the network overhead of sending a checkpoint from compute host CH_j to storage host SH_i. ABw_{SH_i,CH_j} is the available bandwidth between storage host SH_i and compute host CH_j.

Objective Function

We define an objective function in Equation 2.3 that tries to balance the checkpoint storing overhead with the re-execution cost if that checkpoint had not been taken. An application incurs overhead during a checkpoint storing phase while benefits from the fact that it does not have to re-execute from the very beginning and can easily restart from the state saved in the latest checkpoint. The difference of these two quantities is the ultimate price that the application pays. Clearly, a lower value is desirable. FALCON selects storage nodes such that they minimize this objective function. For a particular compute host CH_j, $m + k$ storage hosts are selected for storing that many erasure coded checkpoint fragments.

$$\mathcal{F} = \frac{MTTF_{cmp}}{CI} \times \sum_{i=1}^{V} (C_i \times N_{i,j}) - (T_{curr} + MTTF_{cmp}) \times \prod_{i=1}^{V} CRLS'(SH_i, CH_j) \tag{2.3}$$

$$\sum_{i=1}^{V} C_i = (m + k) \tag{2.4}$$

$$CRLS'(SH_i, CH_j) = max[1 - C_i, CRLS(SH_i, CH_j)] \tag{2.5}$$

Here, $MTTF_{cmp}$ is the mean time to failure of a compute host, CI is the length of a checkpoint interval and T_{curr} is the time units spent on performing useful computation for the job so far. V is the total number of storage hosts. The variable C_i is an indicator variable, set to 1 for the storage host that is selected and 0 for the one that is not. Our goal is to pick the $m + k$ storage hosts so as to minimize the objective function $\mathcal{F}$. The first term corresponds to the overhead of storing the checkpints. The term $\frac{MTTF_{cmp}}{CI}$ approximates the number of checkpoints generated within $MTTF_{cmp}$. The second term corresponds to the re-execution cost—a larger value means lower re-execution cost. Equation 2.5 forces the formulation to only consider the storage hosts that will be selected. We developed a similar objective function in our previous work [14]. However, Equation 2.3 uses different measures of network overhead and reliability score.

Storage Selection Algorithm

To choose storage nodes that minimize the objective function in Equation 2.3, we devise a greedy algorithm. Consider again that the storage selection is being done by compute host CH_j.

We first sort the storage hosts in decreasing order of $CRLS(SH_i, CH_j)$ and increasing order of $N_{i,j}$. If a storage host appears in the first $m + k$ elements of both the sorted lists, it is selected. Then, the value of $\mathcal{F}$ is calculated with $C_i = 1$ for the chosen hosts and 0 for others. This will be used as the baseline value of $\mathcal{F}$ when considering further nodes to add.

If the number of selected storage hosts is less than $m+k$, the objective function is calculated by including one unselected host at a time. The host causing the minimum increase to the objective function is selected. When the number of selected hosts is $m+k$, the algorithm terminates; otherwise the algorithm continues adding one storage host in each iteration. The relative ordering of the different hosts may change from

one iteration to the next and therefore the objective function has to be evaluated for all the unselected hosts at each iteration.

When a storage host becomes unavailable later during a checkpoint interval, the compute host needs to re-choose a new one to replace it. Re-choosing another host from previously unselected ones is also done based on minimizing the same objective function. Our system design is such that this storage selection process occurs in parallel to the actual application and to the algorithm that uses this decision to send checkpoint fragments to the selected repositories. Since checkpoint repository selection occurs out of the algorithm's critical path, this speeds up the checkpoint storing algorithm. But the tradeoff of this design choice is that there may be stale information being used to choose the storage hosts. This can be addressed by configuring the periodicity with which these measurements and list updates take place. Greater is the volatility in the underlying grid environment, smaller should be the setting of the period. The statistical analysis of the eviction characteristics in DiaGrid (as represented in Table 2.2) shows that on average 1.3 evictions occur per hour. For all our experiments, we have configured this periodicity of calculating the objective function to once every 10 seconds.

2.2.3 Single-Threaded Method for Checkpoint Recovery

In our previous work [14] we proposed two algorithms - *Optimistic* and *Pessimistic* for selection of storage repositories. The Optimistic scheme selects a set of storage hosts at the very beginning of a job's execution and uses this set to save checkpoint data during each checkpoint interval. This set is updated only when a job migrates to a different execution machine. On the other hand, the Pessimistic scheme selects a new set of storage hosts at the beginning of each checkpoint interval. While Optimistic ignores inherent dynamism that is present in resource availability, Pessimistic results in unnecessary overhead [14]. Here, we develop a new algorithm that chooses storage hosts on an as-needed basis, always keeping $m+k$ fragments. It releases the resource

availability assumptions from our prior work and updates the selection of storage hosts on an as-needed basis. It takes into account the changing load and fluctuating resource availability. Our algorithm for storing checkpoint data has the following steps:

1. Read chosen storage host list generated by algorithm described in Section 2.2.2
2. Read checkpoint from disk
3. Compress the checkpoint
4. Erasure encode the checkpoint into $m + k$ fragments (erasure coding with parameters (m, k))
5. Send fragments concurrently to storage hosts
6. If any of the chosen storage hosts is in state S_2 or S_3, re-choose another node from the list of unselected ones. The same greedy algorithm, described in Section 2.2.2, is used to rechoose. Send the remaining fragments concurrently.
7. Repeat step 6 until all the fragments are sent or a new checkpoint is generated. If a new checkpoint is generated then start from step 1 and abandon the remaining checkpoint fragments.

2.2.4 Data Parallelism and Parallel Architecture

Since checkpoint sizes of the biology applications can potentially range from couple of megabytes to the order of gigabytes, processing them in their entirety may take a long time. But, by dividing a checkpoint data into a number of blocks and then processing each block in parallel may speed up the processing of the checkpoints in a computation host. Most of the commodity machines that are part of a grid, are multi-core machines. So, by taking advantage of thread level parallelism (TLP) to process the large sized checkpoint data, the processing overhead can be reduced

significantly. In this paper, we have redesigned the checkpoint storing and retrieving process of FALCON [26] to do exactly that. In the evaluation section, we will refer to this architecture as FALCON-P, differentiated from the FALCON architecture presented so far in the paper.

2.2.5 Design of Parallel Checkpoint-Recovery Scheme

Parallel Checkpointing

To take advantage of TLP, we divide a checkpoint into b blocks where b is a configurable parameter. The value of b should not be more than the number of cores available on that particular computation host. Then we spawn b number of threads where the ith of these reads from index $i \times \lceil n/b \rceil$ to $(i+1) \times \lceil n/b \rceil - 1$. Since multiple threads can read a file concurrently, the overhead of reading a file decreases. In addition to that, now each thread has to deal with a smaller sized data set. After reading a mutually exclusive block, each thread then goes on to compress the data, erasure encode each block into $m + k$ fragments and transfer them to the chosen storage hosts. We send the ith fragment of each block to the ith storage host chosen by our storage selection technique 2.2.2. The rationale behind storing one fragment from each block on a storage host is that an available storage host implies the availability of 1 fragment per block for each of the blocks. In this way, our analysis for selecting a set of reliable and efficient storage hosts still holds true.

Parallel Recovery

During the recovery phase, a number of threads equal to the number of blocks a checkpoint was divided into during the checkpoint storing phase, work in parallel to reconstruct the blocks. The ith thread does the following:

- fetch the $m + k$ fragments of the ith block in parallel
- erasure decode these fragments to build the compressed i^{th} block

- decompress the ith block and write to a file

While, the network transfer, decoding and decompression phases take the advantage of TLP, writing the blocks to the disk is a serial operation.

2.3 Structure of falcon

Figure 2.2 presents the system level block diagram of FALCON. In this figure, each large box represents one component and each small box represents a module. We have designed our system such that some modules run off the critical path of our checkpoint-recovery schemes.

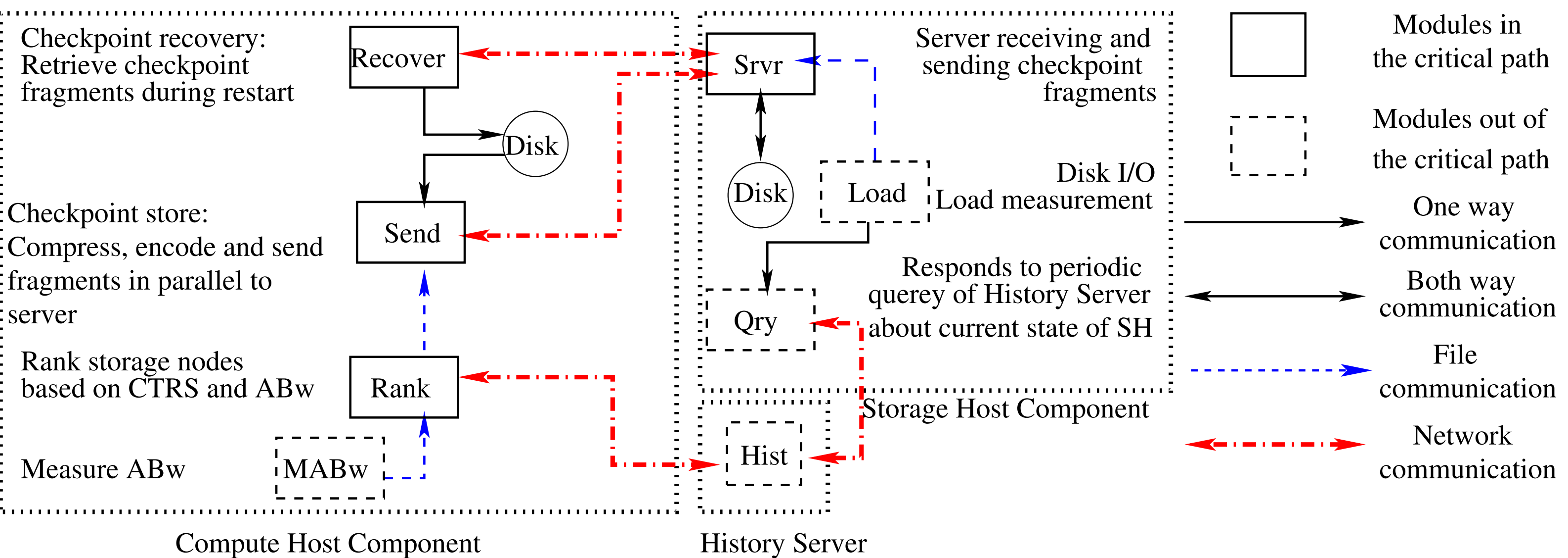


Fig. 2.2.: System level block diagram of our system.

FALCON consists of three major components:

- *Compute host component (CHC)* takes care of failure-aware checkpointing and is submitted along with the guest process to the compute host.
- *Storage host component (SHC)* is a user application that runs in storage hosts. This component implements the multi-state failure model explained in Section 2.2.1.
- *History server component (HSC)* runs on any machine and periodically collects states of compute and storage hosts. Modules in the CHC and the SHC communicate with this component to calculate $CRLS$.

FALCON is integrated with Condor and some of the design decisions were driven by the design of Condor. For example, all the modules are implemented as user-level processes. Detailed description of each module in each component is given in the following subsections.

2.3.1 Compute Host Component (CHC)

Compute host component is the part of FALCON that runs on a computation host. The CHC consists of four modules:

- Module `MABw` is a process that periodically measures available bandwidth between this computation host to all the storage hosts and appends them to a file. For available bandwidth measurement, we have used Spruce [27], a light-weight available bandwidth measurement tool. Spruce provides a server module that runs as a part of the CHC and a client module that runs as a part of the SHC. For our experiments, we have used a period of 10 seconds for measuring the available bandwidth.
- Module `Rank` is a process that implements the greedy algorithm described in Section 2.2.2. It iteratively ranks the storage hosts and produces an ordered list of storage hosts.

- Module `Send` implements an algorithm that is responsible for storing and retrieving the checkpoints. In the single threaded architecture of FALCON [26], this module reads the checkpoint data from disk, compresses it and then uses erasure coding to break the compressed checkpoint data into $m + k$ fragments where m and k are parameters of erasure coding. For erasure coding, we modified the `zfec` implementation [28] to convert it to use only C so that we can run on all the machines of DiaGrid. These checkpoint fragments are sent in parallel to storage host module `Srvr`. In the multi-threaded architecture, the checkpoint reading, compression and encoding schemes are conducted in blocks by parallel threads.

- Module `Recover` is responsible for retrieving checkpoint fragments during a rollback phase from storage repositories, then decoding the fragments to one compressed data and then uncompressing it to produce original checkpoint data - that a guest process uses to restart. Checkpoint fragments are fetched from storage hosts in parallel.

The modules are light-weight, requiring little CPU and memory resources.

2.3.2 Storage Host Component (SHC)

The SHC consists of three modules:

- Module `Load` measures disk I/O load periodically. This process runs in parallel to the actual server module `Srvr` and generates required load information that module `Srvr` uses to determine which state the storage server is in according to Figure 2.1.

- Module `Srvr` implements the server logic for receiving and sending checkpoint data to variable number of compute hosts. It updates the variable current-state at the beginning of each request received from Module `Send` of the CHC.

Table 2.1: Checkpoint sizes of different applications.

Applications	mcf	TIGR-I	TIGR-II	TIGR-III
Original Checkpoint Size (MB)	1677	946	500	170
Compressed Checkpoint Size (MB)	241	201	153	129
Compression Ratio	85.63%	78.75%	69.4%	24.12%

- Module `Qry` is a parallel process that responds to the history server's query about which state the storage host is currently in. It receives values of the state variables from module `Load` (I/O load) and module `Srvr` (the number of compute hosts currently being served).

2.3.3 History Server Component (HSC)

This can be run as a user process on any machine. This component pings each compute host to note if that machine is up or down and communicates with storage hosts to receive their current status. Note that, the HSC only takes 4 states of the storage hosts into account based on load - namely, S_0, S_1, S_2 and S_3. This information then is stored in log files as a {current time stamp, current state} tuple. The HSC computes $CRLS$ using Equation 2.1. Our current implementation uses a central server approach. This design can be extended to a distributed implementation.

2.4 Evaluation

We have developed a complete system, as described in Section 2.3. We ran experiments on a production Condor testbed—DiaGrid by integrating our work with standard benchmark applications. These applications were chosen from SPEC CPU 2006 and BioBench [29] benchmark suites. SPEC CPU 2006 is widely used for benchmarking CPU-intensive programs while BioBench consists of well known biomedical

applications. This section presents the experiments for evaluating the system in terms of its checkpoint recovery overheads and its effectiveness in improving job makespan. In Sections 2.4.1 and 2.4.2, we present the evaluation of FALCON (the single-core version) and in Section 2.4.3, we present the evaluation of FALCON-P (the multi-core version). Table 2.1 shows checkpoint sizes generated by different benchmark applications. MCF and TIGR are benchmark applications part of SPEC CPU 2006 and BioBench respectively. TIGR-I, TIGR-II and TIGR-III are runs of TIGR with different input sizes.

We have organized our experiments to measure both fine-grain (*micro benchmark experiments*) and coarse-grain (*macro benchmark experiments*) metrics. While the micro benchmark experiments compare overheads of different checkpoint-recovery schemes under controlled experimental conditions, the macro benchmark experiments evaluate the effectiveness in improving job makespan running on Purdue's condor environment, the DiaGrid. Schemes that we compare with are:

- Dedicated: Condor's scheme where a dedicated storage server is used for saving checkpoint data.
- Random: A scheme where the storage hosts for saving the erasure encoded checkpoints are randomly chosen from among all available storage hosts.
- Pessimistic: A scheme presented in [14] that assumes that resource fluctuation is very common and re-chooses all the storage hosts at the beginning of each checkpoint interval.

Note that, the default scheme of Condor is to send checkpoint back to the submitter machine (the machine from which the job was submitted). This scheme is similar to that of using a dedicated server and hence performs no better than our reference Dedicated algorithm (in fact, it can be significantly worse, if the submitter sits behind a low-bandwidth connection). We do not compare FALCON with the Optimistic scheme because the assumption of not having any fluctuation in the grid

environment by the Optimistic scheme does not hold in practice due to the volatility of the environment.

Checkpoint storing overhead includes time:

- for FALCON to: (i) Read chosen storage host list from disk (ii) Read checkpoint from disk (iii) Compress (iv) Erasure encode and write fragments to disk (v) Read fragments from disk (vi) Send checkpoint fragments in parallel to storage hosts
- for Dedicated to: (i) Read checkpoint from disk (ii) Send checkpoint to storage server
- for Random to: (i) Choose storage nodes randomly (ii) Follow steps (ii) - (vi) of FALCON

Recovery overhead includes time:

- for FALCON to: (i) Fetch the minimum required fragments from storage hosts. This time includes reading checkpoint at the storage host end, network transfer and writing to disk at the compute host end (ii) Erasure decode and write compressed checkpoint data to disk (iii) Decompress and write to disk
- for Dedicated to: (i) Fetch checkpoint data from storage server. This time includes reading checkpoint at the storage host end, network transfer and writing to disk at the compute host end
- For Random to: (i) Follow steps (i) - (iii) of FALCON

For all our macro and micro benchmark experiments, we set erasure coding parameters to $(3, 2)$ - meaning 3 fragments are required and 2 are redundant.

2.4.1 Macro Benchmark Experiments

This section presents results of our macro benchmark experiments - experiments that we ran by submitting scientific applications to DiaGrid and measuring their

Table 2.2: Statistical analysis of the eviction characteristics in DiaGrid.

N	μ	σ	*Range*
116	1.3130	0.2172	[1.0298,2.3931]

average makespan — the time difference between submission and completion of the job minus the time it spent in the idle or the suspended states. A job submitted to Condor remains idle until it gets scheduled to a suitable machine. Condor jobs can specify their requirements for disk space, memory, machine architecture, operating system etc. in a submission script and a scheduler matches these requirements with machines that are available for running Condor jobs. Since this idle time is in no way related to checkpoint-recovery scheme, we exclude it from calculating makespan.

Checkpointing in Condor is non-blocking for the applications - the only blocking part is till the checkpoint is locally stored. Condor then transfers this checkpoint to appropriate storage repository as configured. This non-blocking technique efficiently hides checkpoint transfer overhead from makespan of applications. It is during restart when applications need to fetch checkpoints to the execution machine and restart. This recovery overhead directly adds up to an application's makespan.

The recovery overhead is incurred as many times as there are evictions of the applications from the compute hosts. We empirically measured this in DiaGrid and present the failure characteristics in Table 2.2. We used 1.3 evictions per hour per job as the rate of eviction for our experiments in Section 2.4.1. The table shows number of jobs for which we collected data (N), average number of evictions per hour (μ), standard deviation (σ) and range.

In Section 2.4.1 we compare average job makespan of applications using different checkpoint repository techniques. The decompositions of checkpoint storing and recovery overheads are shown in Section 2.4.1 and Section 2.4.1 respectively. For all the macro benchmark experiments we have used the sequential checkpoint storing and recovery schemes of FALCON.

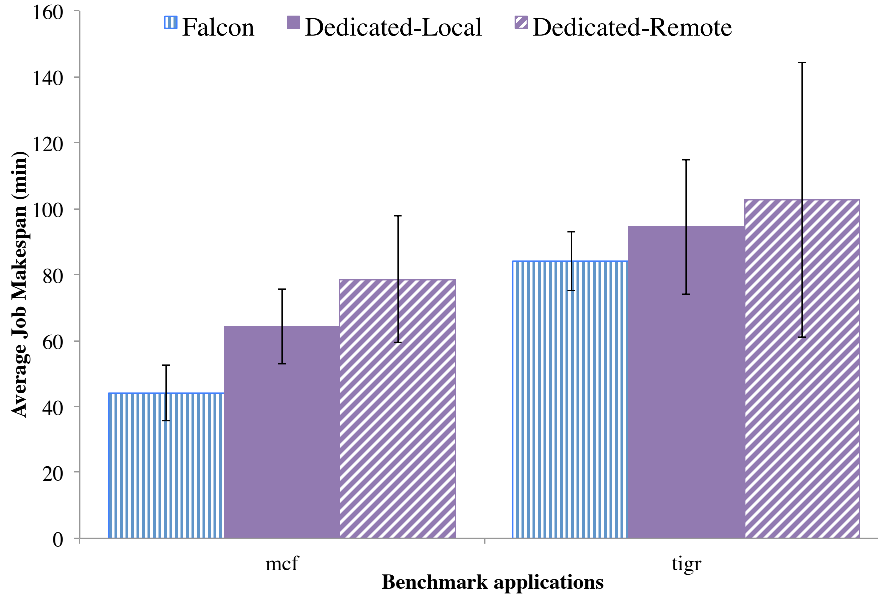
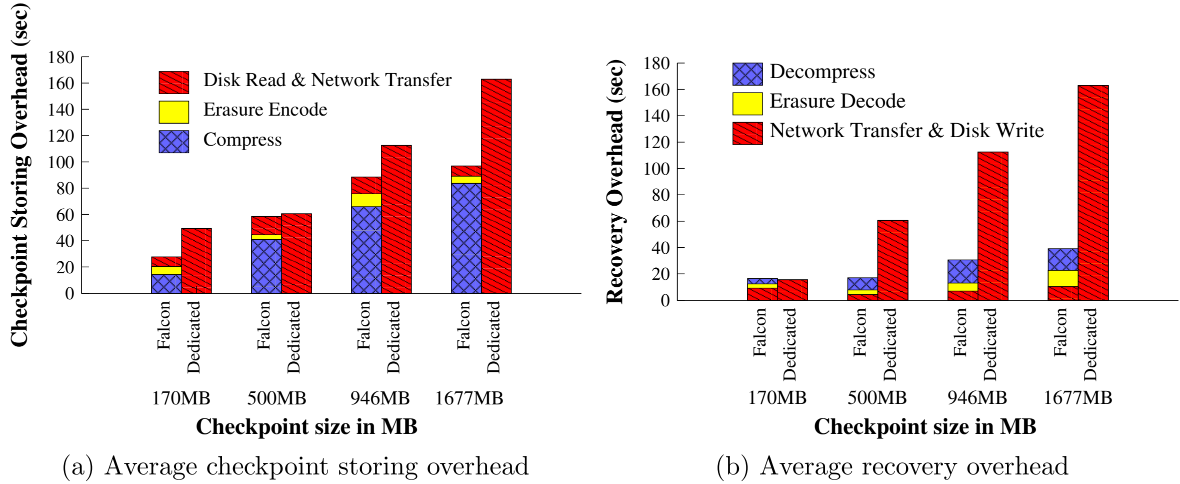


Fig. 2.3.: Average job makespan of different applications.

Overall Evaluation

For overall evaluation of different schemes, we collected average job makespan of two benchmark applications. We integrated three checkpoint schemes: FALCON, Dedicated with a local checkpoint server (lab machine connected to the campus-wide LAN at Purdue) and Dedicated with a remote checkpoint server (machine at University of Notre Dame connected to Internet) with these applications and submitted jobs in DiaGrid. Note that, University of Notre Dame is a part of this multi-university grid. For all cases, these applications took checkpoint once every 5 minutes. Here, using the Dedicated-Remote scheme represents the situation when jobs submitted from one university go to run at another university but the checkpoint server is at the first university. This is exactly the situation in Boiler-Grid for applications that run on other university machines.

From Figure 2.3, we see that FALCON outperforms Dedicated-Local and Dedicated-Remote in actual application runs. In actual runs on DiaGrid, applications on an average will see performance that lies between that of Dedicated-local and Dedicated-



(a) Average checkpoint storing overhead

(b) Average recovery overhead

Fig. 2.4.: Recovery overhead of four different checkpoint sizes.

Remote since applications do go to run on machines at other campuses. The reasons for performance improvement of FALCON are many-fold — (i) FALCON chooses storage repositories that are more efficient to access (ii) checkpoint fragments saved by FALCON are much smaller in size due to compression and encoding, compared to the checkpoint size that Dedicated schemes store. Section 2.4.1 explains that smaller checkpoint size results in lower recovery overhead and hence improved job makespan and (iii) the fragments are retrieved in parallel from the chosen storage hosts. More about the contribution of each of the techniques (compression, load balancing and parallel network transfer) in improving the recovery overhead is discussed in Section 2.4.2.

Checkpoint Storing Overhead

Figure 3.15 shows decomposition of overhead of FALCON and Dedicated for a single store operation. Dedicated scheme uses a remote checkpoint server.

One observation that can be drawn from Figure 3.15 is that as checkpoint size increases, increase in checkpoint storing overhead of Dedicated becomes much higher than FALCON. The overhead is dominated by the disk read and network transfer time, which increases with increasing checkpoint sizes. However, the design of compressing

the checkpoints and transferring the smaller checkpoint fragments in parallel speeds this up.

Recovery Overhead

In Figure 2.4b we plot recovery overhead incurred by FALCON and Dedicated schemes for a single recovery operation.

One observation that can be made from Figure 2.4b is that as checkpoint size increases, Dedicated scheme suffers due to large network transfer overhead. Compression by FALCON results in smaller checkpoint data and hence reduced network transfer overhead. As Table 2.1 shows, compression ratio increases as size of checkpoint data increases. This justifies our approach of incurring a little overhead at the compute host side for compression with the benefit of significant improvement in recovery overhead. Note that, lower recovery overheads directly translate to better performance for an application.

2.4.2 Micro Benchmark Experiments

The objective of the micro benchmark experiments is to show off specific features of FALCON under controlled experimental conditions. We conducted three sets of experiments to compare: (i) efficiency of different schemes in handling concurrent clients, (ii) efficiency in handling storage failures and (iii) performance improvement due to load balancing. For these experiments, we used checkpoint data of 500MB generated by application TIGR. As storage hosts for FALCON we used 11 - 1.86 GHz Intel Core 2 Duo machines with 80GB of hard disk connected to the campus-wide 100 Mbps LAN and 1 - 2.00 GHz laptop with 160GB of hard disk connected to a DSL modem. As dedicated storage server we used another lab machine with configuration 2.66 GHz Intel Core 2 Duo with 80GB of hard disk space and connected to the campus-wide LAN. This machine was always available.

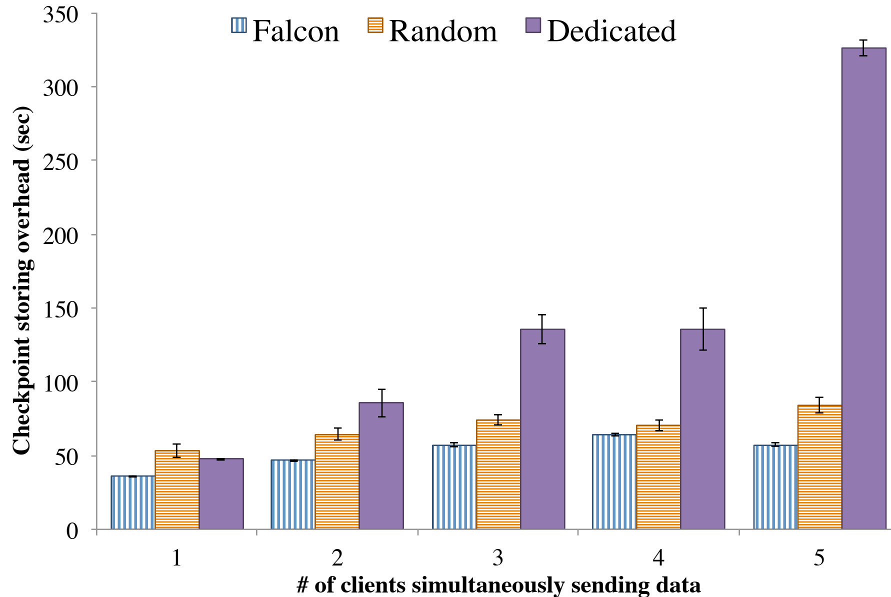


Fig. 2.5.: Average execution time vs the number of concurrent clients.

Efficiency in Handling Simultaneous Clients

The objective of this experiment is to show how the performance of different schemes scale with load imposed by multiple concurrent clients. In this experiment, the checkpoint storing overheads of different schemes, in addition to the factors described in Section 2.4, include time to write the checkpoint data to disk at the storage host end. We vary the number of compute hosts simultaneously sending data and measure the overhead for storing checkpoints.

Two observations that can be drawn from Figure 2.5 are:

1. As the number of clients simultaneously sending data increases, the checkpoint scheme with a dedicated server suffers more than FALCON. Even though with erasure coding FALCON introduces 40% more data, due to compression the total

amount actually sent by FALCON is less than that of Dedicated. Total amount of data sent by FALCON with compression and erasure coding is:

$$datasent = 5 \times \frac{153}{3} MB = 255MB < 500MB$$

2. Checkpoint storing overhead of Random is larger than that of FALCON because Random chose the laptop behind the slow network connection 8% of the time. Because of low available bandwidth between compute host and this laptop, FALCON never chose it.

Efficiency in Handling Storage Failures

Since storage hosts in FGCS are non-dedicated resources, a protocol must be able to handle unavailability of storage nodes efficiently. The objective of this experiment is to compare the added overhead of re-choosing storage nodes by FALCON with that of Random and the more conservative approach of Pessimistic [14]. For this experiment, we killed the storage daemons running in those storage hosts to make them appear unavailable.

One observation that can be drawn from Figure 2.6 is that the overhead of re-choosing storage hosts using history and available bandwidth is no worse than that of choosing them randomly. This shows that the design choice of measuring history and available bandwidth out of the critical path yields robustness at no extra cost. But it is clear from Section 2.4.2 that the scheme employed by FALCON chooses storage nodes wisely. Pessimistic however incurs large overhead due to measuring bandwidth between compute and storage hosts at the beginning of every checkpoint storing instance.

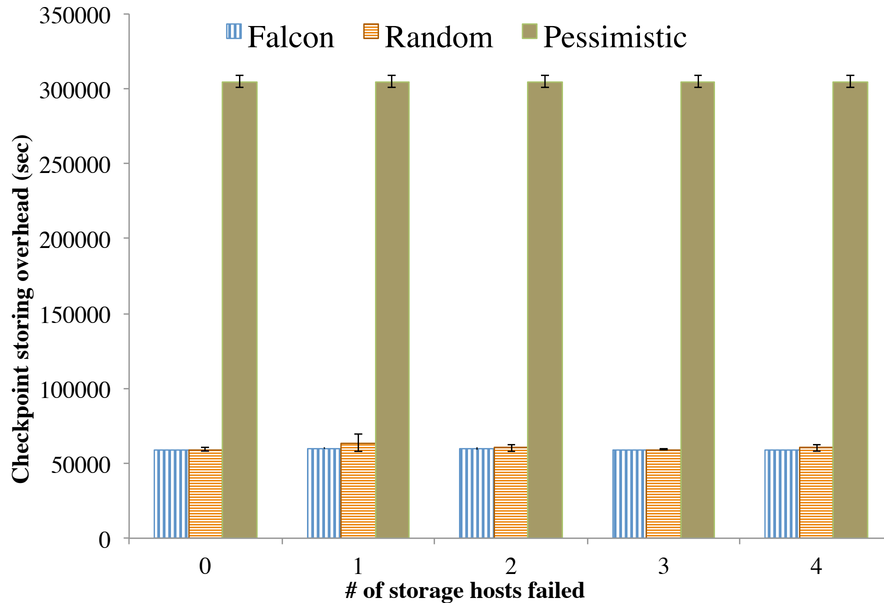


Fig. 2.6.: Average checkpointing overhead vs the number of unavailable storage hosts.

Load Balancing vs Checkpointing Overhead

In this experiment, we compare the overheads of storing checkpoints when the workload in storage hosts varies. The objective of this experiment is to evaluate the effectiveness of the load balancing technique of FALCON. For this experiment, we generated background I/O load in a single storage host out of the 5 chosen ones using a file system benchmark application Bonnie [30]. The checkpointing overheads of both the techniques, in addition to the factors described in Section 2.4, include the time for the storage hosts to write data to disk. Additionally, the overhead of the scheme with load balancing includes time to rechoose a storage host to replace the overloaded one.

Since τ_2 is set as 80%, in Figure 2.7, the difference between the load balancing and no load balancing cases comes up when load on a storage host becomes $\geq$ 80% and FALCON with load balancing scheme re-chooses. As high I/O load may imply high CPU utilization as well, the model with load balancing benefits by not sending data to this host. Ensuring balanced load among the shared storage resources is utterly

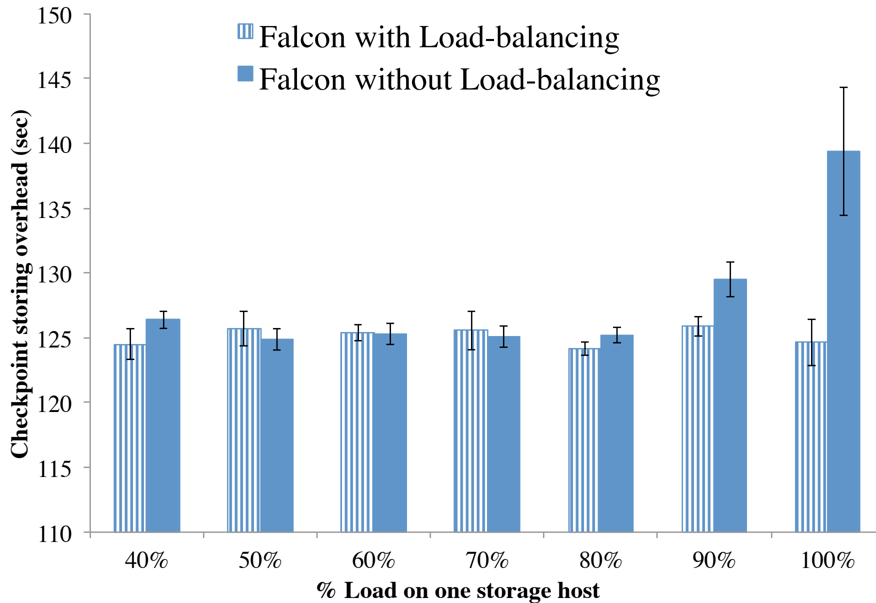


Fig. 2.7.: Average checkpoint storing overhead of different schemes.

important because these resources are shared by the owners voluntarily. Hence, taking advantage of these resources in a way so that the actual host's performance does not degrade beyond a threshold is as crucial a parameter as the performance benefit gained.

Parallel vs Sequential Retrieval of Checkpoints

In this experiment, we compare the network transfer overheads incurred by retrieving the checkpoint fragments sequentially with that of retrieving them in parallel. The size of the checkpoint fragments was 940MB each and we generated 100% I/O load on the loaded storage nodes using Bonnie [30]. There were 5 storage nodes ($m = 5$) including 1 that was loaded.

The objective of this experiment is to evaluate the effectiveness of the parallel data retrieval technique of FALCON when a subset of the storage nodes containing

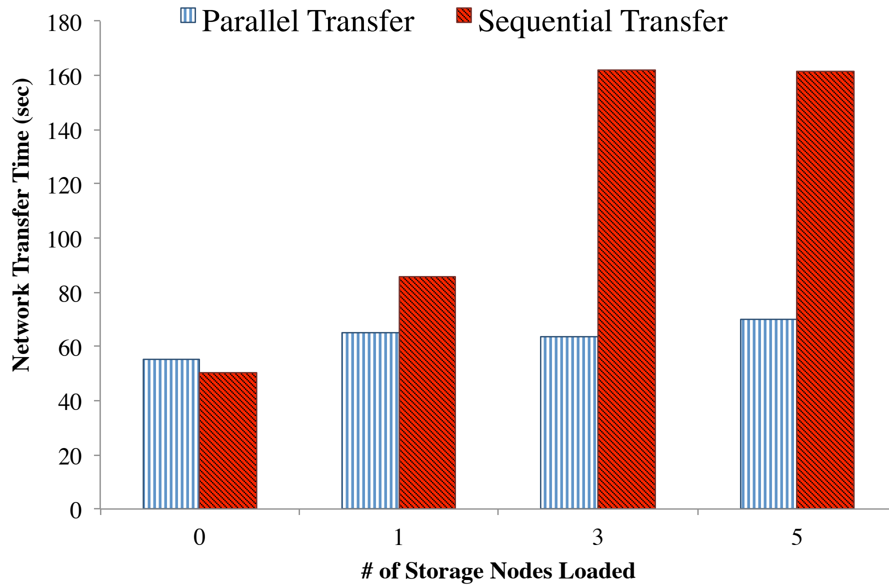


Fig. 2.8.: Parallel vs sequential retrieval during restart.

the checkpoint fragments becomes loaded. Here, the sequential scheme retrieves fragments from m storage nodes including all the loaded ones.

Figure 2.8 demonstrates the advantage of employing parallelism in retrieving the fragments from storage nodes. The sequential scheme performs poorly because it fetches the checkpoint fragments from the loaded nodes and can only complete after all the m fragments are fetched. So if even any one of the nodes is busy and the sequential scheme starts retrieving data from that node, it has to finish the transfer. In contrast, the parallel scheme is done as soon as any m of the fragments arrive. So, it often happens that there are m not so loaded nodes and the retrieval process finishes early.

Table 2.3: Breakdown of different schemes evaluated in Figure 2.9.

Scheme	Server Loaded	Parallel Network Transfer	Erasure Decoding	Decompress
I	✗	✓	✓	✓
II	✗	✗(Sequential)	✓	✓
III	✗	✓	✓	✗
IV	✓	✓	✓	✓

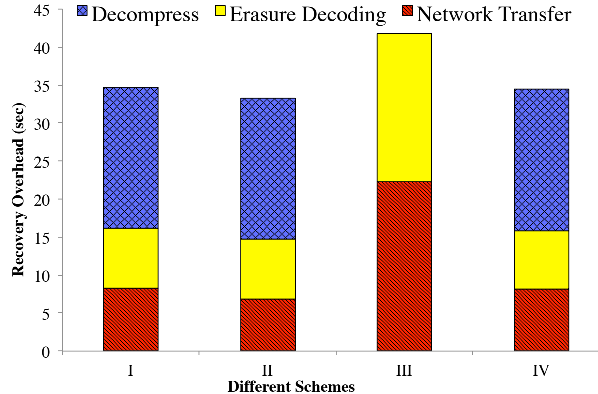


Fig. 2.9.: Contributions of compression, load balancing, and parallel network transfer during restart.

Contributions of Compression, Load balancing, and Network level Parallelism

In this experiment, we compare the contributions of each of the three schemes—compression, load balancing, and parallel retrieval of checkpoint fragments—in improving the recovery overhead and in turn, in improving the performance of the applications. For this, we successively remove one of the schemes from FALCON while keeping the other two schemes. The checkpoint used to run this experiment is that of the application TIGR-I (Table 2.1).

Figure 2.9 shows breakdown of schemes listed in Table 2.3. The first observation from Figure 2.9 is that the largest contribution in improving recovery overhead comes from compressing the checkpoint data. The highly compressible nature of

these checkpoint data can result in a compression factor as large as 86% (for mcf) and 79% for this application (Table 2.1). This in turn reduces the network transfer overhead. Also, if the checkpoint is not compressed, the decoding overhead increases. An important point to note is that even though the decompression overhead is the most dominating component in the recovery overhead, it is worthwhile to compress and decompress. Otherwise encoding very large checkpoints ($\geq$ 1GB) incurs very high memory cost and requires very long time, if at all possible. Second, due to small number of checkpoint fragments ((m,k) being $(3, 2)$), network transfer overhead of both the parallel and the sequential schemes are comparable. The advantage of using the parallel scheme over the sequential one in retrieving the checkpoint fragments is discussed in Section 2.4.2. Third, due to the smaller sizes of the checkpoint fragments, the overhead of retrieving the m fragments from storage nodes with 100% I/O load is comparable to that of retrieving them from non-loaded ones. But the point to note is that load-balancing while it does not have a very prominent contribution in lowering the recovery overhead in this experiment, it has an impact on the checkpoint transfer overhead (Section 2.4.2). Load balancing is also crucial because the storage nodes are shared resources. So, during the checkpoint storing phase, if compute hosts disregard the fact that a storage node is loaded and put more load on it by sending the bulk of checkpoint data, then the performance of the host jobs on that storage node may degrade considerably. This may cause the owner to remove his resource from the pool.

2.4.3 Single-threaded vs Multi-threaded Architecture of falcon

In this section, we discuss the experiments that we ran to compare the checkpoint storing and recovery overheads of the two different architectures of FALCON. At first, we ran an experiment to find out the value of b, the number of blocks that the checkpoint should be decomposed into, that should be used to get high overall performance improvement. Then, we used this value of b to run experiments by varying

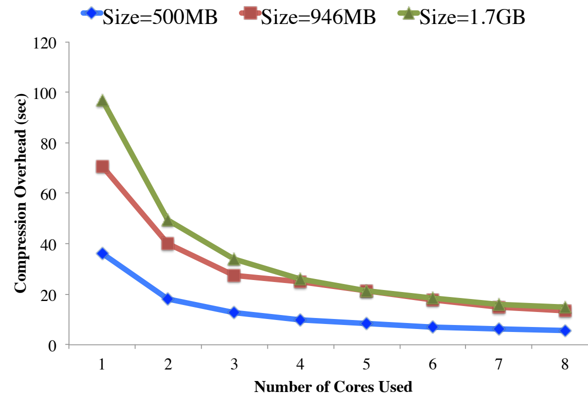


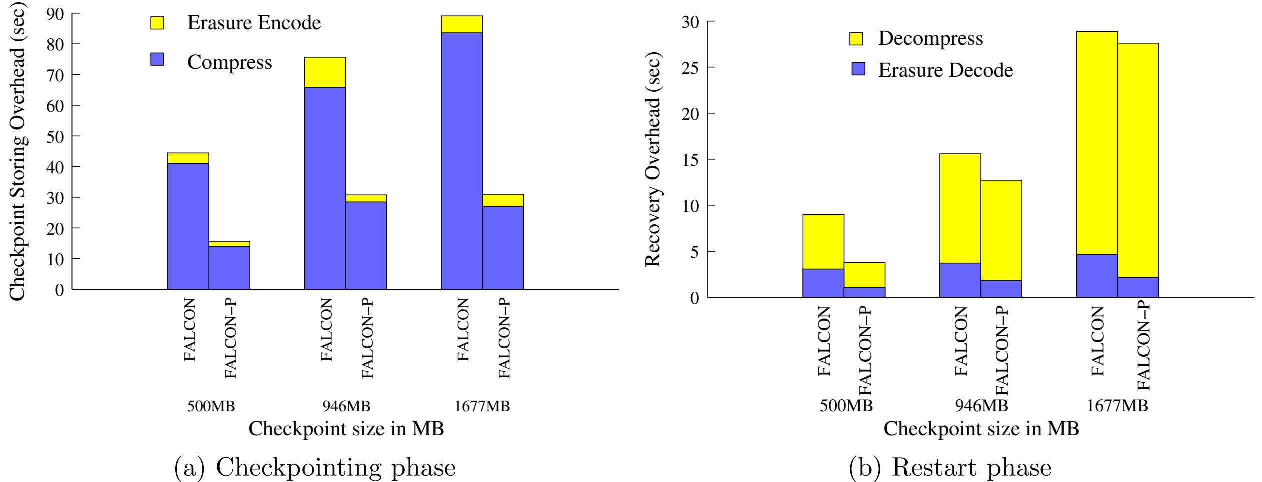
Fig. 2.10.: Improvement in compression overhead from using multiple cores.

the checkpoint sizes. The goal of these experiments is to compare the performance of FALCON architectures with and without the TLP option turned on. This new multi-threaded architecture works by breaking up the checkpoint data, compressing and creating $m + k$ fragments for each block in parallel. Transferring the checkpoint blocks in parallel over the network was already part of the single threaded architecture of FALCON. So, we only compare the overheads of parallel compression and encoding during the checkpoint storing phase and parallel decoding and decompression during the recovery phases of the two schemes. Lower compression and encoding overheads translate into lower overall checkpoint storing and recovery overheads.

Determine the Degree of Parallelism

In this experiment, for each size of checkpoint, we varied the number of threads working concurrently on the data. The machine that we used to run this experiment was an 8 core SMP (Symmetric Multi-processor) machine.

The observation that can be made from Figure 2.10 is that the decrease in compression overhead levels off as the number of cores concurrently used increases. In addition to that, the ability of a parallel program's performance to scale with the number of cores is the result of a number of interrelated factors. Some of the hardware



(a) Checkpointing phase

(b) Restart phase

Fig. 2.11.: Comparison of the checkpointing and the recovery overheads of FALCON and FALCON-P.

related limiting factors are memory-cpu bus bandwidth and the amount of memory available per core on a shared memory machine. As the number of cores used increases, the amount of memory available to each of the cores decreases and hence results in memory contention when more than 50% of the cores start being used. Based on this experiment we set the configurable parameter b to half of the total number of cores, i.e., 4 for the rest of the experiments.

Comparison of Checkpointing Overheads

In this experiment, we compare the overheads of compressing a checkpoint data and then encoding it into $m + k$ fragments with that of breaking up a checkpoint data into b blocks and then compressing and encoding each block in parallel. For this experiment, we set (m, k) to $(3, 2)$ and $b = 4$. In the figure, the scheme "FALCON" represents the single threaded architecture where as "FALCON-P" represents the multi-threaded one. The checkpoints were all generated by using different inputs to the same benchmark application TIGR.

Table 2.4: Input sizes to different algorithms of FALCON and FALCON-P.

Application	To Compression Algorithm(MB)		To Encoding Algorithm(MB)	
	FALCON	FALCON-P	FALCON	FALCON-P
mcf	1677	419.25	241	(65,64,62,62)
TIGR-I	946	236.5	201	(83,49,36,34)
TIGR-II	500	125	153	(40,40,39,34)

One observation that can be made from Figure 2.11a is that by breaking a large checkpoint up into multiple blocks and then working on each block in parallel reduces the overhead significantly. Table 2.4 shows a complete break down of the amount of data that each core had to deal with. The multi-core architecture of FALCON improves the compression overhead by up to 67% (higher gains for larger checkpoint sizes). Also, the encoding overhead reduces since now the size of the input data to each invocation of the erasure encoding algorithm is reduced by more than 50%.

Comparison of Recovery Overheads

This experiment compares the recovery overheads incurred by the two architectures.

The observations that can be made from Figure 2.11b are that:

- decoding overhead decreases since the fragment sizes to work with are smaller
- as checkpoint size increases, the difference between the decompression overheads decreases. The reason is that all the threads write to the same file and disk writes cannot be made in parallel. Hence with the increase in the size of a checkpoint data, there is not much improvement in decompression overhead.

Even though the improvement in the recovery overhead diminishes as checkpoint sizes increase, the reduction in the checkpoint storing overhead is significant. This justifies the use of thread level parallelism in the new architecture of FALCON (i.e., FALCON-P).

2.5 Related Work

Checkpoint-recovery is a widely used technique for providing fault-tolerance in high-performance parallel computing and distributed systems [5]. Related contributions include checkpointing facilities provided in production systems for MPI applications [31] and improving checkpointing performance. Production grid systems such as Condor [13], take checkpoints of applications periodically and store them in dedicated servers. However, relying on such dedicated servers does not leverage the idle storage resources in grid environment. Moreover, recent research [32] has shown that using non-dedicated storage can actually result in improved performance of guest applications if a reliable set of such resources can be chosen. These results motivate our work of applying resource availability prediction to select reliable, non-dedicated checkpoint repositories.

Erasure encoding for storing data in a distributed manner to tolerate failure is a well-known technique. Related work such as [16] discusses in detail a fault-tolerant method of checkpointing and recovery using erasure coding. On the other hand, [15] compares different techniques of introducing redundancy in checkpoint data to improve fault-tolerance of applications in a shared storage environment. Erasure coding is also a popular technique for providing reliable access to data in peer-to-peer networks [33]. The OceanStore project [34] creates massive scale redundant copies of data using (among other techniques) erasure coding. The work makes contributions in efficient read operation and Byzantine fault-aware replication. The model is not that of FGCS systems and therefore the notion of guest jobs and their evictions due to resource contention is not significant.

[35] is an empirical study based on actual Condor trace. It characterizes the reasons of resource unavailability in Condor and proposes a multi-state grid resource availability characterization. A few other studies use failure modeling of compute hosts for scheduling jobs on a grid [36].

2.6 Summary

We have designed, developed and evaluated FALCON, a system that provides fault-tolerant execution of applications in FGCS systems without any dedicated storage server. We present a load-balancing multi-state failure model for these shared storage resources and apply knowledge of this model to predict reliability. We present a new checkpoint store and retrieve technique that efficiently handles large-sized checkpoint data, of the order of gigabytes. Finally, we run experiments in DiaGrid, a multi-university production Condor system at Purdue University. Experiments show that FALCON provides consistency in running times and improves overall performance of jobs by 11% to 44% over the mechanisms of using dedicated checkpoint servers or choosing storage hosts randomly. We have also looked into the idea of utilizing multiple cores available on the computation hosts to reduce the checkpointing overhead. For this, we identify the most computationally expensive step of the whole checkpointing process (i.e. compression of the checkpoints) and parallelize this step by exploiting the data parallelism inherent to these checkpoints. In future, we would also like to explore the avenue of developing a compression technique for these checkpoints to improve the performance of the guest applications by reducing the overall checkpoint storing overhead.

FALCON demonstrates that the idea of compressing system-level checkpoints is beneficial. However, in high-performance computing (HPC) environments, applications use application-level checkpoints since the size of memory on those machines is orders of magnitude larger than the amount of state necessary to recover from a failure. Also, checkpointing systems in HPC environments store checkpoints on parallel file systems. So, in HPC environment, we focus on making checkpoint storing a scalable process since with the scale of these machines and the proportional growth of the amount of state, the network bandwidth between computation components and stable storage will become the key bottleneck.

3. SCALABLE CHECKPOINTING SYSTEM USING DATA-AWARE AGGREGATION AND COMPRESSION

As software and hardware component counts in high performance computing (HPC) systems scale up, the likelihood grows of one failing while an application executes. For example, the 100,000 node BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) experiences an L1 cache parity error every 8 hours [1] and a hard failure every 7-10 days. Exascale systems are projected to fail every 3-26 minutes [2,3]. Many applications tolerate failures through checkpoint-restart [6, 37], which periodically saves application state in *checkpoint* files on stable storage, such as a parallel file system (PFS). If a failure occurs, the application is restarted from the latest checkpoint, thus reducing repeated computation. To simplify checkpoint-restart implementations, many applications have all processes take checkpoints simultaneously [7]. This strategy avoids the complexities of message logging that uncoordinated checkpointing requires, as well as its possibility of cascading rollback.

Checkpointing causes 75-80% of the I/O traffic on current HPC systems [38, 39]. On future systems, checkpointing activities will dominate compute time and overwhelm file system resources [6, 11]. Checkpointing to a PFS is expensive at large scale: a checkpoint can take tens of minutes due to network bandwidth and PFS resource contention [40, 41]. Further, HPC computational capabilities are increasing more quickly than their I/O bandwidths. For example, BlueGene/L at LLNL and BlueGene/P at Argonne National Laboratory achieve less than 0.1 GB/s of I/O bandwidth per TeraFLOP of computational capability [40–42]. This high overhead leads to reduced checkpointing frequency, which implies more computation is lost in the event of failure.

We face two key challenges to checkpointing scalability: the number of checkpoint files written and their size. As application process counts grow, the number of check-

point files usually increases proportionally. Large counts of file writers degrades PFS performance and reliability due to contention [42–44]. Thus, application programmers are exploring techniques to combine checkpoints of multiple processes at the cost of application complexity. Further, the size of each file grows with process count under naive combining schemes; in any event, the size of the checkpoint files usually grows as processes compute larger sub-problems.

The challenges of checkpointing systems in grid and HPC environments are different. For a checkpointing system in grid, the challenge is to ensure reliable storage of checkpoints on grid resources that can become unavailable any time. These are often sequential applications taking system-level checkpoints. So, FALCON enables applications to achieve consistent performance by compressing these checkpoints using general-purpose compression algorithm and selecting reliable and lightly loaded grid machines to store them. On the other hand, applications run in HPC environment are MPI applications with a large number of processes working on different parts of a large input space. These processes generate scientific data in application-level checkpoints that are known to be hard to compress. The issues that checkpointing systems in HPC environment struggle with are – how to effectively reduce the amount of data and efficiently transfer it from compute nodes to stable storage.

We address these challenges unique to HPC environments through MCRENGINE, a library that aggregates multiple checkpoint files from different processes and compresses them. Further, we extract application-specific data semantics by analyzing the metadata contained in the checkpoint files. This knowledge enables us to merge the checkpoint fragments from the different processes intelligently, increasing the compressibility of the aggregated checkpoint, particularly when we apply data-specific compression algorithms (e.g., targeted to floating point data). These benefits also apply during restart, when all application processes must read the checkpoint fragments from the PFS, often within a short time window. We must keep this extra work small since restart is always on the application's critical path. We make the following major contributions:

- The concept of data-aware checkpoint compression, which significantly increases the compressibility of checkpoints;
- A thorough investigation of the application-specific impact of different aggregation schemes on compression ratio;
- The design, development, and thorough evaluation of an end-to-end checkpoint-restart system and its demonstration with three real-world, complex, and diverse applications.

Our evaluation uses ALE3D, Cactus, and Enzo. ALE3D is an arbitrary Lagrangian-Eulerian (ALE) multi-physics code. Cactus [45,46] solves Einstein's Equations. Enzo is a grid-based code that simulates cosmological structure formation [47]. Our significant results include: **1)** data-aware compression improves compression ratios between 27.72% and 115% on average over simple concatenation and compression; **2)** mcrEngine can reduce checkpointing intervals by up to 87%; **3)** mcrEngine decreases restart overhead by 62% compared to using uncompressed checkpoints; **4)** data-aware compression alone reduces the latency to write checkpoints to the PFS up to 92% and to read checkpoints up to 71%.

The paper is organized as follows. Section 3.1 provides background. Section 3.2 describes our solution and rationale. Section 3.3 details the structure of the implementation of mcrEngine. Section 3.4 presents our evaluation methodology. Sections 3.5 and 3.6 present key results.

3.1 Background

In this section, we discuss related background information on different checkpoint types, how checkpoints are written and the different file formats in which they are written.

3.1.1 Application-level vs System-level Checkpointing

HPC applications commonly take coordinated, application-level checkpoints. Coordination implies that all application processes agree when they will take the checkpoint, typically periodically. Application-level checkpointing uses application routines to save specific data structures in the checkpoint files. In contrast, system-level checkpoints are system-initiated snapshots that include the entire system memory. Application-level checkpointing provides several benefits. Application knowledge of the state needed to restart can significantly reduce checkpoint sizes. For example, for protein-folding applications on IBM Blue Gene, the size of an application-level checkpoint set is a few megabytes compared to terabytes for full system-level checkpoints [48]. Application-level checkpoints also increase portability.

3.1.2 Checkpoint Writing

Applications vary in how many processes write checkpoints. If an MPI application uses N processes, then each process writes its own state to a checkpoint file in the $N \rightarrow N$ pattern. This strategy performs poorly at large scales due to contention from the large number of writers. At the other extreme ($N \rightarrow 1$ checkpointing), a single process writes one large checkpoint that contains data from all processes, which creates a serialization bottleneck that also inhibits scaling. Thus, $N \rightarrow M$ checkpointing, in which a set of M processes write aggregated data for N processes ($M < N$), attempts to balance metadata costs and contention with the bandwidth advantages of multiple writers. However, $N \rightarrow M$ checkpointing increases application complexity, a trend that further optimizations continue. Thus, application developers need a strategy that provides the benefits of $N \rightarrow M$ checkpointing while hiding that complexity. mcrEngine is such a strategy.

Applications also vary in how they distribute input data across processes. One approach is *segmented input distribution* (Figure 3.1a), in which data is naturally composed of segments that exhibit similarity. One segment of the input data is

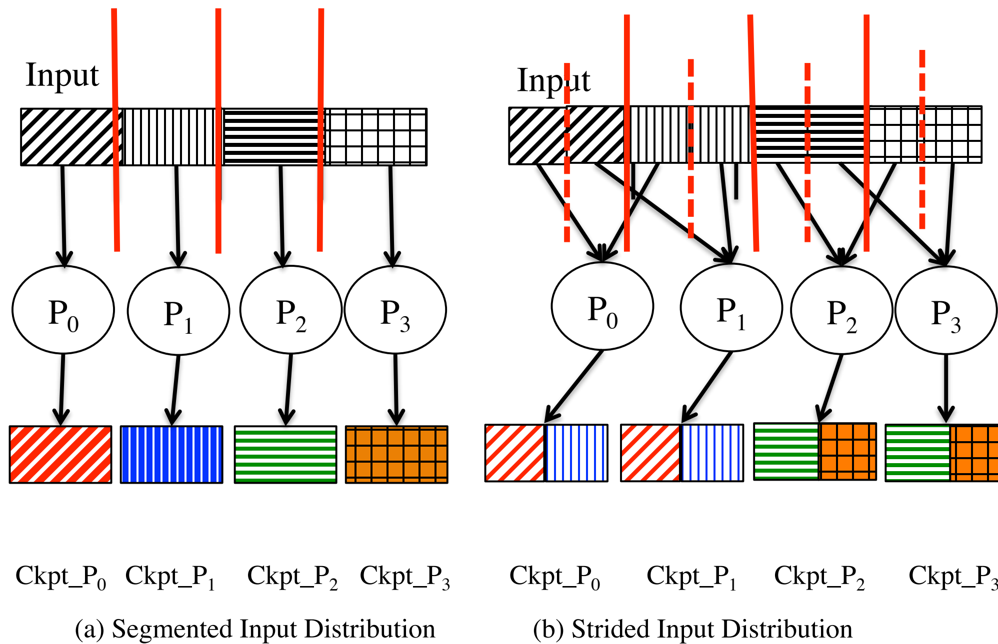


Fig. 3.1.: Two types of input data distribution

processed by one process. The second approach is *strided input distribution* (Figure 3.1b), in which one segment of the data is split into stripes and these stripes are distributed (or "strided") to different application processes.

3.1.3 Checkpoint File Format

Application-level checkpointing often builds on I/O libraries such as MPI-IO, POSIX, HDF5, netCDF, or Parallel-netCDF [49, 50]. While some applications write checkpoints in an application-specific format, using standard I/O libraries, such as POSIX or MPI-IO, many large-scale applications use standard self-describing I/O formats (e.g., HDF5 or netCDF). While the initial time to deploy POSIX or MPI-IO solutions can be low, it reduces portability due to issues such as endianness and thus increases maintenance cost. Using a descriptive data format to write structured checkpoints ensures portability across users, tools (such as visualizers), and systems.

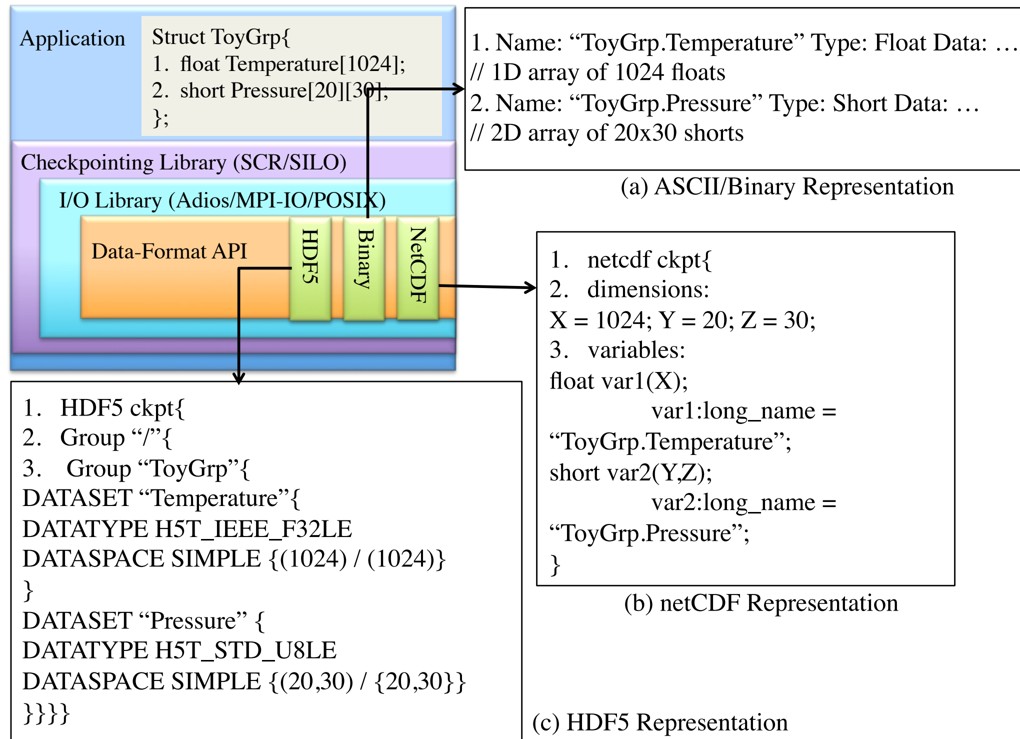
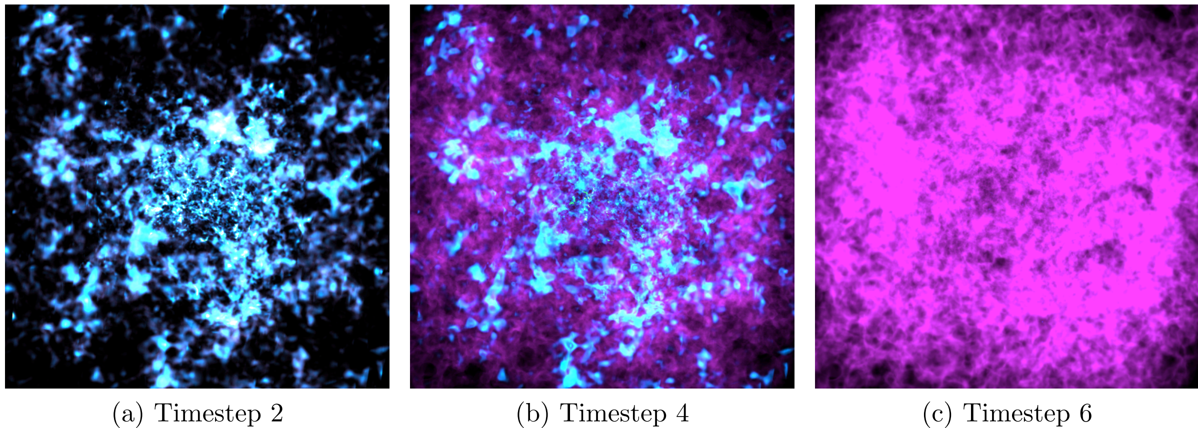


Fig. 3.2.: Different metadata annotations of checkpoint data

Thus, self-describing I/O formats are popular among large-scale applications, such as HDF5 [49] with its long list of users.

As Figure 3.2 shows, data structures in application-level checkpoints are described differently depending on the data format. Also checkpoint files are often used for visualization. Although the file format structure varies, the key point is that application data is annotated with descriptive metadata. The application (Figure 3.2a) or the data format library (Figure 3.2b and c) can provide the metadata.

mcrEngine uses data from application-level, globally coordinated checkpoints to implement an $N \rightarrow M$ scheme. To demonstrate our approach, we use HDF5 checkpoints. However, the design of mcrEngine provides the flexibility to use other data formats easily. Using the example in Figure 3.2, data with the name *Temperature* has the same meaning across all processes in an Single Program Multiple Data application (we may require additional information for a Multiple Program Multiple Data application to capture data relationships across processes). We refer to an individual



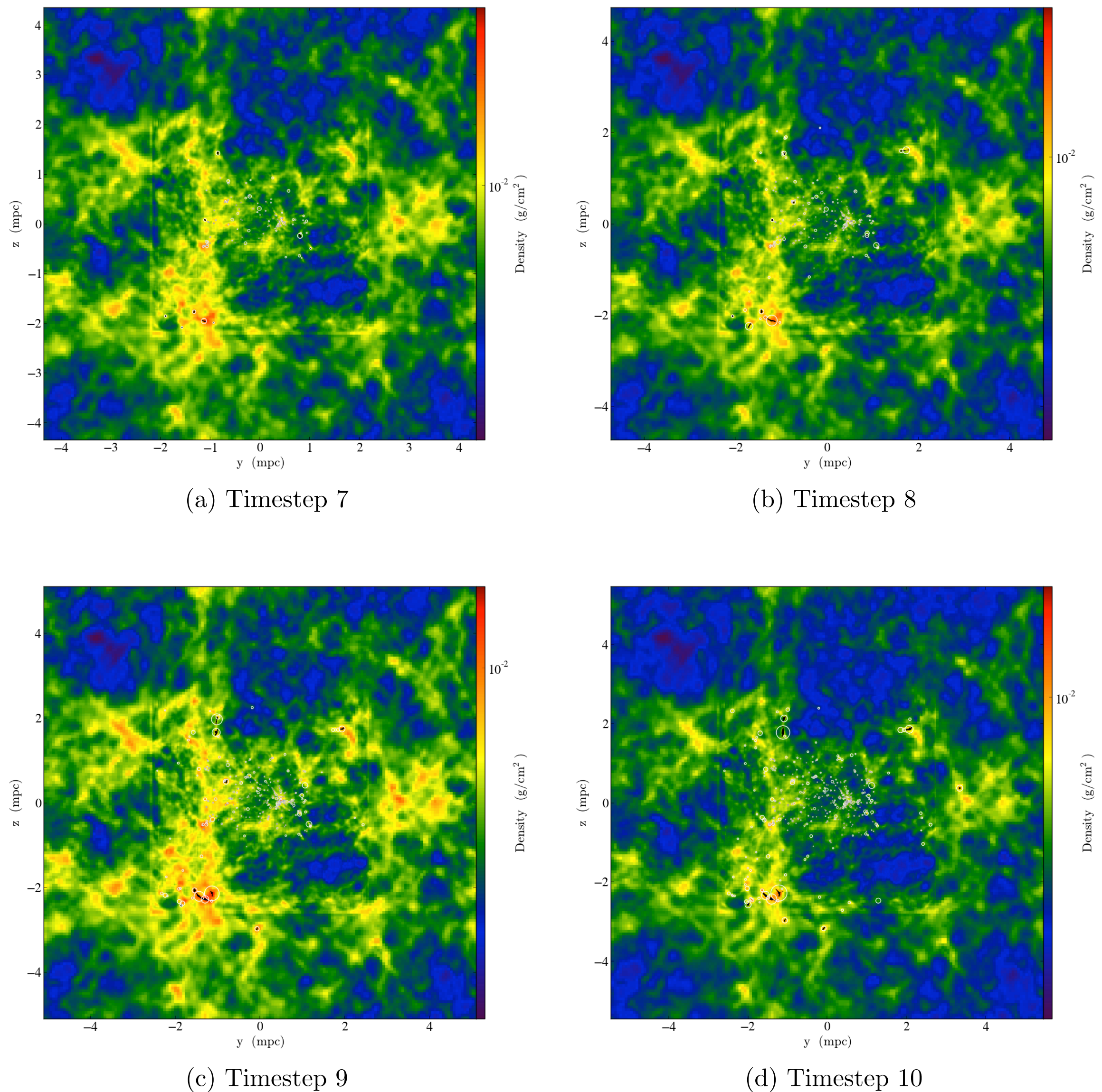
(a) Timestep 2 (b) Timestep 4 (c) Timestep 6

Fig. 3.3.: Volume rendering of the $3D$ variable: "Density"

file that a single process writes as a *checkpoint* and to a group of checkpoints that multiple processes write concurrently as a *checkpoint set*.

3.1.4 Checkpoint Data

To understand the characteristics of data that is written in application-level checkpoints, we visualized checkpoints from an application Enzo [47] that simulates cosmological formation using dark matter and hydro cosmology calculation. This particular simulation [51] zooms in on a large halo to simulate dark matter density at finer resolution. This simulation only refines in a small subvolume of the calculation, and includes radiative cooling, six species primordial chemistry, a uniform metagalactic radiation background, and prescriptions for star formation and feedback. Figure 3.3 shows the volume rendering of the density variable in these checkpoints representing density of particles during a cosmology simulation. This figure also shows that value of this variable changes as simulation progresses. Figure 3.4 shows the projection of the $3D$ variable density along X axis. The square in the middle shows the area where the simulation zooms in for higher precision. The primary observation from these figures is that, there is enough similarity within density variable across processes to gain good compression by arranging them together.

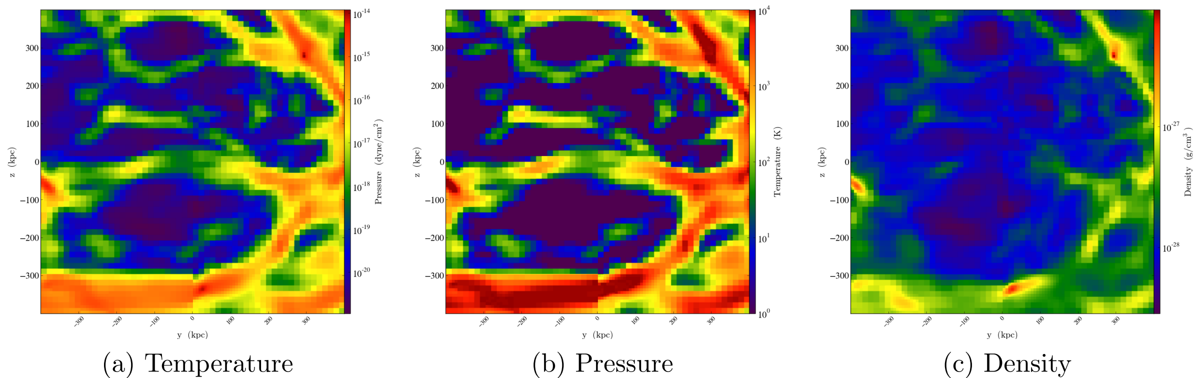


(a) Timestep 7 (b) Timestep 8

(c) Timestep 9 (d) Timestep 10

Fig. 3.4.: Refined nested computation

Figure 3.5 shows the projection of the $3D$ variables Temperature, Pressure, and Density along X axis. Different processes compute different parts of each of the variables. This figure shows that values in different variables (Temperature, Pressure, and Density) can be significantly different and it will be beneficial to rearrange these



(a) Temperature (b) Pressure (c) Density

Fig. 3.5.: Projections of Temperature, Pressure, and Density variables along X axis for Timestep 8.

variables so that all variables with the same meaning remain close together during compression.

3.2 Data-Aware Checkpoint Aggregation & Compression

Most parallel scientific applications distribute simulation data across multiple processes. These processes generally coordinate globally to take a consistent checkpoint. During an asynchronous checkpointing phase, once a process finishes checkpointing, it resumes its computation. Currently, with most checkpointing systems, the processes send their checkpoints directly to the PFS, although checkpointing libraries such as SCR [11] can store checkpoints locally instead.

In order to analyze the issues that will prevent scaling of current checkpointing systems, we conducted an experiment on a large Linux cluster. We present a complete discussion of the experiment in Section 3.6.2. Our key findings are:

- The average performance of operations (read and write) scales much better with fewer concurrent processes;
- Transfer overhead is lower with the same number of writers when the data volume is less.

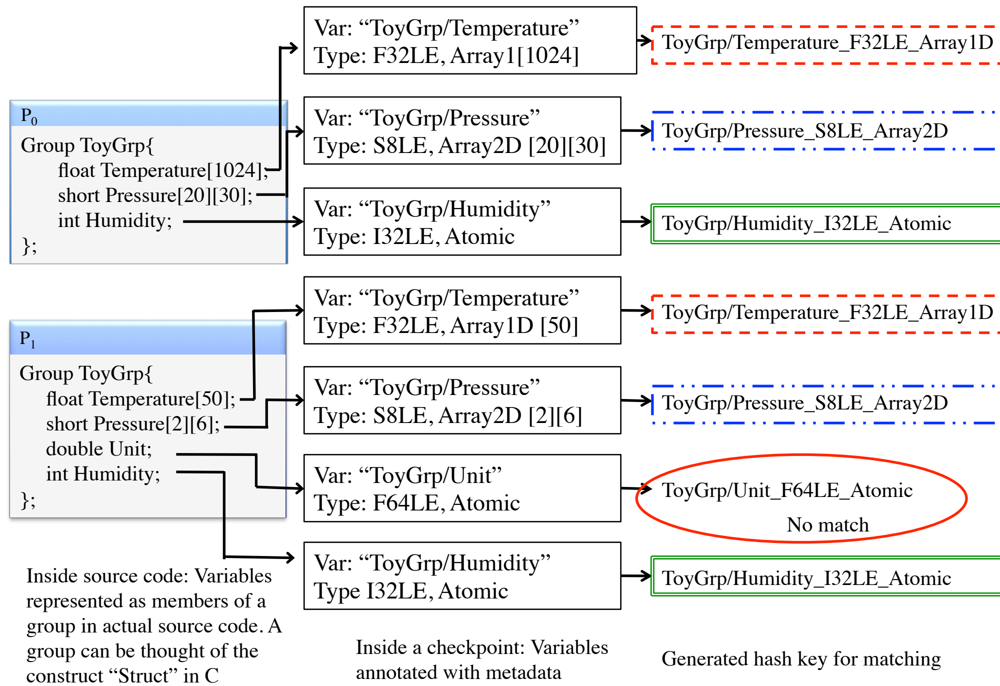


Fig. 3.6.: Variable matching

Thus, checkpointing systems should aggregate checkpoints to reduce the number of concurrent processes. Compressing checkpoints could provide additional benefits. We define *data-aware aggregation* as aggregating across process checkpoints such that data with similar meaning remain together in the merged checkpoint. mcrEngine uses *data-aware compression*, which dynamically selects from a set of compression algorithms and applies the best one for each similar data group. *Data-aware aggregation and compression* is the entire process of interleaving semantically similar data and using a dynamically selected compression algorithm.

All popular compression utilities use a finite window in which they look to find similarities between two data items. The window is kept reasonably small by default (e.g., 32KB for gzip) to keep the memory utilization of the compression utility low. By using data-aware aggregation, mcrEngine increases the likelihood that similar data appears in the same window, which increases compression and decreases the number of concurrent files written to the PFS.

3.2.1 Identifying Similarity Across Checkpoints

Intuitively, finding similarity across checkpoints is simple. Two variables with the same name and data type are likely to have the same meaning. mcrEngine uses metadata to locate checkpoint data that represent the same variables. mcrEngine interprets two variables in different checkpoints as *similar* if their names agree and their data representation is identical (same data type).

Figure 3.6 provides a step-by-step example. Processes P_0 and P_1 have groups of variables. To be general, the example shows different structures for the same group in the two checkpoints although in practice, two processes of the same application are unlikely to have different structures for the same group.

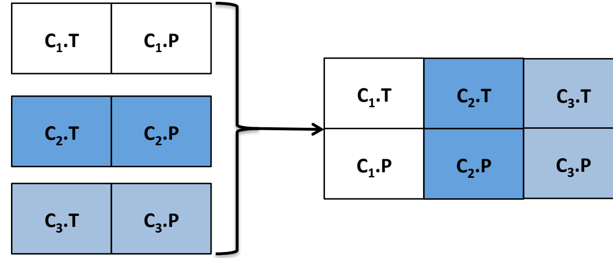
In our similarity detection, variables can be of any standard data type and can be of atomic or array class. We consider variables with the same name, data type and class to be similar even if they have different numbers of elements. Variable names can match either exactly or according to a regular expression; application developers can specify the regular expression as a configuration parameter. We locate similarity as follows:

- Match every element of a group separately;
- Annotate variables with their names, data types, array sizes, and classes in the checkpoints;
- Convert variable metadata into hash keys of type: `Group-name/Variable-name_Datatype_Class`.

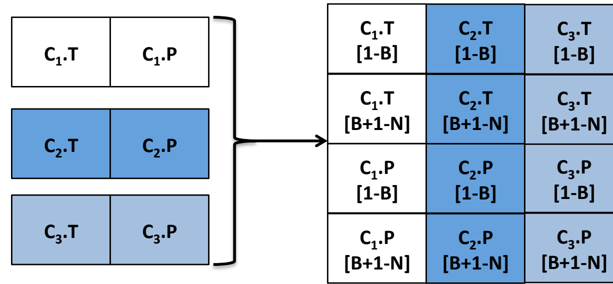
For example, mcrEngine generates `ToyGrp/Temperature_F32LE_Array1D` as the hash key for the variable `Temperature` in P_0. While Figure 3.6 illustrates our terminology, real application checkpoints have more complex structures.

3.2.2 Merging Schemes

We consider four schemes to aggregate checkpoints:



(a) Variable Concatenation



(b) Variable Blocking

Fig. 3.7.: Data-aware merging schemes

- **Checkpoint Concatenation** The simple *Agnostic* scheme concatenates entire checkpoints before compressing them;
- **Checkpoint Blocking** *Agnostic-Block* interleaves fixed-size data blocks instead of concatenating entire checkpoints;
- **Variable Concatenation** As Figure 3.7a shows (Cx.T denotes the temperature array of the checkpoint of rank x and Cx.P its pressure array), the *Data-Aware* or simply *Aware* scheme concatenates individual variables before compressing them;
- **Variable Blocking** As Figure 3.7b shows, the *Aware-Block* scheme interleaves variables in blocks of configurable sizes, instead of simply concatenating them.

The Aware and Aware-block schemes aim to increase compression ratios by grouping similar data.

3.3 Structure of mcrEngine

Figures 3.8a and 3.8b present the system level interactions among different modules of mcrEngine during the checkpoint and restart phases. In these figures, each large dotted box represents a component and each small box represents a module. A solid module indicates that it is on the critical path; a dotted one indicates it avoids the critical path by using a parallel thread. Finally, arrows represent different inter-module communication: a dashed arrow indicates network communication while a dotted one corresponds to a wake up signal to a thread. mcrEngine has two major components:

- *Compute node component (CNC)* offloads checkpoint processing from the compute nodes by sending checkpoints to the aggregator node component during checkpointing; restores checkpoints to the local disk during restart.
- *Aggregator node component (ANC)* processes (e.g., concatenates and compresses) checkpoints; stores and retrieves checkpoints from the PFS during restart.

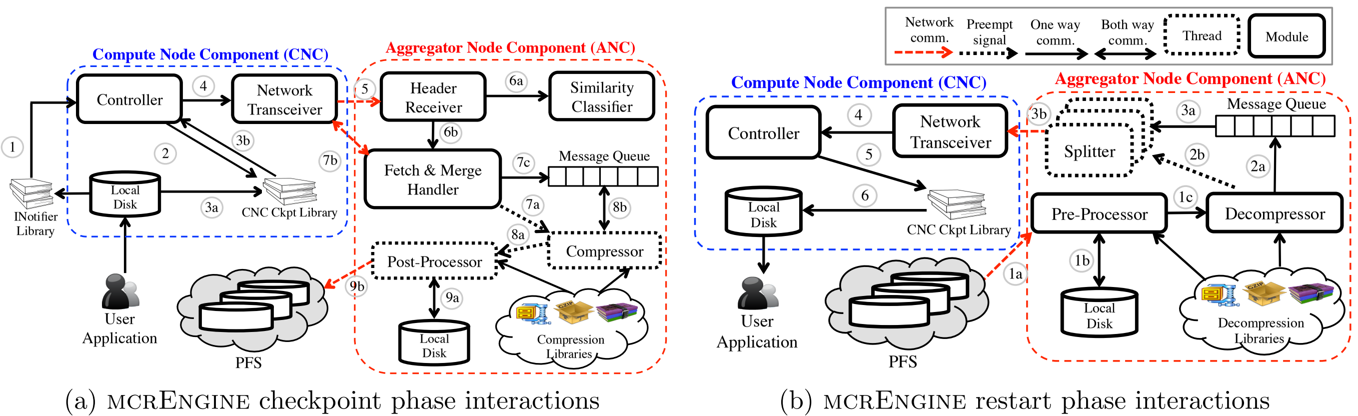
We now discuss interactions between CNC and ANC modules during the checkpointing and restart phases.

3.3.1 Checkpointing Phase

Table 3.1 summarizes the interactions between the CNC and ANC components. CNCs responsible for application processes in the same group (according to their rank) send data to the same ANC. Each CNC notifies the assigned ANC that a checkpoint is ready and transfers the data to it. The ANC concatenates and merges checkpoints based on the scheme in use and writes to the PFS. The message queue is a memory buffer in which each element points to a data block in memory. These lightweight modules require little CPU and memory resources. The CNC uses the Inotify blocking wait to reduce interference with the application.

Table 3.1: Module interactions during a checkpointing phase

ID	Communication Description
1	Inotify Library notifies the Controller of checkpoint creation
2	Controller initiates checkpoint reading through CNC interfaces
3	CNC Ckpt Library reads the checkpoints, written in application-specific formats, into memory (3a); returns header and data to the Controller (3b)
4	Controller passes reference to header/data buffer to Network Transceiver
5	Network Transceiver sends information about variables to Header Receiver
6	Header Receiver sends group name, variable name, data type, and class information to Similarity Classifier, which inserts them into a hash table, chaining variables with the same hash key in process rank order (6a); Header Receiver invokes Fetch & Merge Handler after all checkpoint headers of a group are classified (6b)
7	Fetch & Merge Handler signals Compressor to compress variables, if available (7a); traverses hash table to send data requests to Network Transceiver and fetches data for a pool of variables (7b); merges similar variables and enqueues them in a message queue (7c)
8	Compressor signals the Post-Processor thread to start buffering processed data, if available (8a); reads from the message queue; compresses merged variables according to data types (current implementation uses FPC [52], fpzip [53] and LZ [54] to compress doubles, floats, and other data types, respectively) and writes back to the message queue (8b)
9	Post-Processor flushes accumulated data to local disk (9a); applies Parallel-Gzip [55] after all variables across processes in the same group are merged and compressed; sends the final data to PFS (9b)



(a) MCRENGINE checkpoint phase interactions

(b) MCRENGINE restart phase interactions

Fig. 3.8.: System-level interactions

We design mcrEngine such that its aggregators fetch variables as needed in order to avoid requiring excessive aggregator disk space. By pulling data as needed, the ANC can limit the variables per process present at any time. We also exploit the inherent parallelism of accessing three kinds of resources (CPU, disk, and network) by using separate threads to transfer data between the CNC and ANC modules, to compress it and to write it to disk. Thus, when the ANC `Fetch & Merge` module starts a network transfer (fetch operation), it schedules the `Compressor` thread to use the CPU and the `Post-Processor` thread to finish disk I/O. Our results show that the network transfer time largely offsets the overheads of compression and local disk accesses.

The CNC also produces a metadata file that stores information about the last checkpoint set. The restart phase uses the information about the last set of checkpoints stored. An empty file indicates a new application run.

3.3.2 Restart Phase

As Figure 3.8b shows, during restart, the `Controller` module sends a request to the `Pre-Processor` module of the corresponding ANC to read a checkpoint file from the PFS. The `Pre-Processor` module then reads the checkpoint from the PFS, decompresses and separates the variables, and sends them to the `Controller`. Each `Splitter` thread communicates in parallel with a separate `Controller`, which must generate the original checkpoints for the application processes. Table 3.2 summarizes the communications between CNC and ANC modules during this phase.

The design of mcrEngine does not require node-local persistent storage on compute or aggregator nodes. This design feature allows mcrEngine to work on current large-scale systems (e.g., BG/L). On diskless nodes, the CNC uses in-memory buffering before streaming them to the ANC next stop without any intermediate storage (*checkpoint streaming*). However, upcoming technologies such as SSDs will replace the in-memory buffer. Our future work will implement *variable streaming*, in which

Table 3.2: Module interactions during a restart phase

ID	Communication Description
1	Pre-Processor reads processed checkpoint and metadata file from PFS (1a); applies Parallel-unzip and writes to the local disk (1b); invokes Decompressor on merged-compressed variables
2	Decompressor applies appropriate decompression algorithm to each variable; enqueues decompressed merged variables in the message queue (2a); signals Splitter threads to split variables, if available (2b)
3	Splitter reads and separates variables from message queue them (3a); uses a separate thread per process to send data to the Network Transceivers (3b)
4	Network Transceiver invokes the Controller module with received data
5	Controller invokes interfaces to write variables to checkpoint file
6	CNC writes variables to checkpoint file in application-specific format

Table 3.3: Checkpoint characteristics

	ALE3D	Cactus	Cosmology	Implosion
Write Pattern	N→M	N→N	N→N	N→N
Number of Checkpoints	32	32	128	64
Total Size (GB)	4.8	2.41	1.1	0.013
Individual Size: [Range] Average (MB)	[154.1, 154.5] 154.2	[58, 90] 77.15	[4.7, 13.4] 8.4	[0.07, 4] 0.2
Total Variables	56820	284800	33562	9996
Variable Count: [Range] Average	[1760, 1835] 1776	[8900, 8900] 8900	[76, 1090] 262	[28, 832] 156
Double Precision Floats (%)	88.8	33.94	24.3	0
Single Precision Floats (%)	2e-5	0	67.2	74.1
Other Data Types (%)	11.2	66.06	8.5	25.9

we buffer and stream large variables individually. That work will address issues such as synchronizing similar variable writes.

3.4 Evaluation Methodology

This section describes our methodology for evaluating mcrEngine with the schemes introduced in Section 3.2.2. We evaluate mcrEngine with checkpoints from four computational simulations of three real-world applications. Our evaluation criteria are the amount of compression, and checkpoint-restart end-to-end overhead.

3.4.1 Applications

We evaluate mcrEngine on ALE3D, Cactus, and Enzo (with two distinct inputs: *Cosmology* and *Implosion*). The simulations produce checkpoints with varying char-

acteristics. Table 3.3 summarizes our ALE3D checkpoint set and a checkpoint set from the midpoint of the Cactus and Enzo simulations. The write pattern indicates how the checkpoints are written. The number of checkpoints indicates how many individual checkpoints are in each set. For ALE3D, 1024 computation processes were run to generate 32 checkpoints. For the other applications, the number of checkpoints generated equals the number of computation processes. The total size is the sum of the individual checkpoint sizes. The individual size row shows the range and average size of the individual checkpoints. We report the total number of variables across a checkpoint set and the range and average of variable counts in the individual checkpoints. We also provide percentage breakdowns of variable data types.

ALE3D is a multiphysics numerical simulation that uses arbitrary Lagrangian-Eulerian (ALE) techniques. We do not have access to the ALE3D source code so we do not have information on its input distribution.

Cactus [45, 46] is a framework that numerically solves Einstein's equations [56]. It uses adaptive mesh refinement (AMR). During initialization, it divides input data among N processes (*top-level roots*). Each process then recursively divides its pieces among sub-grids. Each top-level root collects computation from sub-grid elements and writes a checkpoint. So, this application writes N checkpoints, while the number of sub-processes that perform actual computation is larger.

Enzo, an AMR, grid-based hybrid code (hydro + N-Body), simulates cosmological structure formation [47]. It uses the particle-mesh technique to solve dark matter N-body dynamics. Cosmology is a 3D Eulerian block-structured AMR simulation [57]. Implosion is a 2D converging shock problem in which a shock wave interacts with reflecting walls, undergoing a double Mach reflection [58]. While we generate both from the same code base, the nature of the execution and, thus, the checkpoints are significantly different.

3.4.2 Evaluation of Compression Schemes

Two metrics measure how well data-aware compression techniques reduce data size. *Compression ratio* is the ratio of the size of the (checkpoint's) uncompressed data to the size of the compressed data, including metadata. A compression ratio of 2 indicates that the uncompressed file is 2× larger than the compressed file. In Equation 3.1, $us(ckpt)$ is the uncompressed checkpoint size. For a scheme X, $cs_x(ckpt)$ is its compressed checkpoint size and $cr(X)$ is its compression ratio.

$$cr(X) = \frac{us(ckpt)}{cs_x(ckpt)} \tag{3.1}$$

Relative improvement compares the effectiveness of compression schemes as the compression ratio of schemes A and B:

$$\text{relative improvement} = \frac{cr(A) - cr(B)}{cr(B)} \times 100\% \tag{3.2}$$

Finally, we evaluate the performance of the different schemes, which we measure in terms of the time to complete the checkpointing and restart phases.

3.5 Data-Aware Compression Effectiveness

We now evaluate the effectiveness of our novel data-aware compression techniques. In particular, we explore:

- The benefit of multiple compression passes;
- The change in compression ratio with varying group size;
- The impact of interleaving granularity on compression ratio;
- The change in compression ratio as a simulation progresses.

We ran our experiments on LLNL's Sierra system [59], which is a 261.3 TFLOP/s Linux cluster running the CHAOS 4.4 operating system with an InfiniBand QDR interconnect. It has 1,944 nodes, each with 12 2.8 GHz cores and 24 GB of memory. Sierra is connected to a 1.3 PB Lustre file system with a maximum aggregate bandwidth of 30 GB/s.

To determine parameters that provide a high compression ratio with low time overhead we evaluated the compression libraries that we used. For each library, we choose settings for each algorithm that realize most of the possible compression without incurring significant overhead. Throughout the experiments, we set the compression levels of Parallel-Gzip to 6 (the default value), of FPC to 2, of fpzip to 1-dimensional, and of QuickLZ to 1 (the minimum value).

For this evaluation, we group application processes according to their rank order. For example, processes 1 to 32 and 33 to 64 are aggregated for a $GROUP_SIZE$ of 32. We run one application process per core on each compute node, and use all cores on each aggregator node for our experiments. Each aggregator process on a core handles checkpoints from $GROUP_SIZE$ computation processes. We use Equation 3.3 to determine the aggregator node count:

$$\#\text{ of Agg-Node} = \frac{cp}{ac \times GROUP_SIZE} \tag{3.3}$$

where cp is the number of computation processes and ac is the number of cores on each aggregator node.

3.5.1 Benefit of Multiple Passes of Compression

We first study the impact of multiple compression passes. We experiment with one and two compression passes, to which we refer as single (SC) and double (DC) compression. SC applies one pass, either with Gzip or data-aware compression. DC always applies Gzip in the second pass. Figure 3.9 shows the results for *Agnostic*, *Aware*, and *Aware-Block*. We do not show *Agnostic-Block* in Figure 3.9 because it

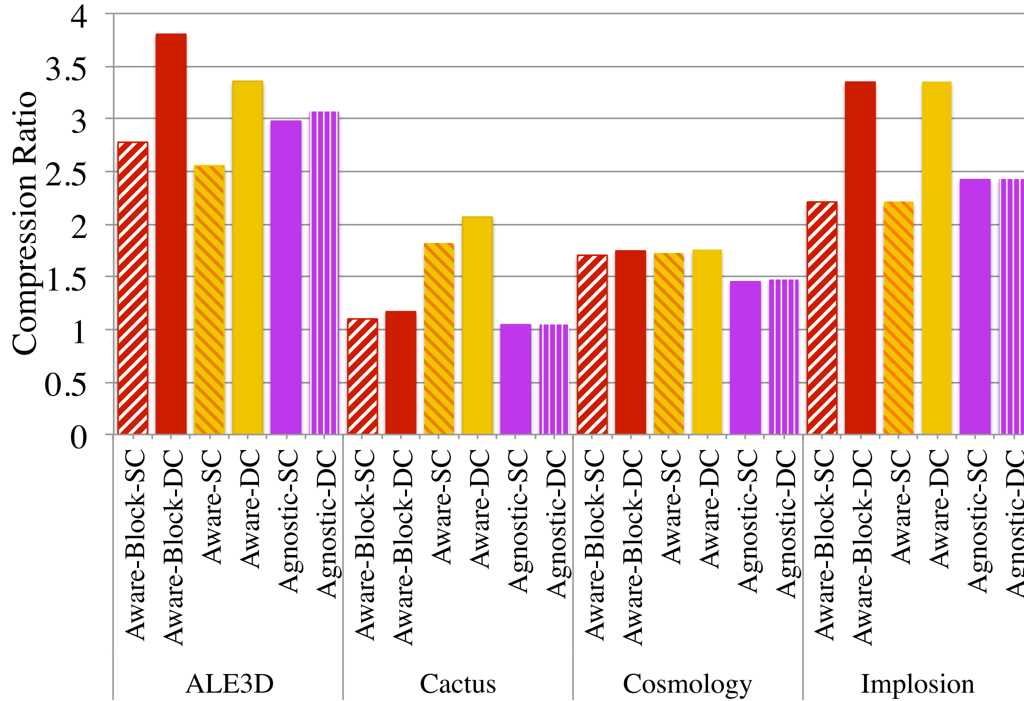


Fig. 3.9.: Double compression vs single compression

performs similarly to *Agnostic* for DC. DC never benefits the data-agnostic schemes. Alternatively, DC improves the compression ratio of the data-aware schemes considerably because the compression libraries that we use in the first pass convert the data to a more compressible format, which the second pass exploits. For example, FPC applies an XOR operation to a predicted value and the actual value. It encodes the resulting number of zeros, an integer, into the output, which can inflate the size of the output but enables Gzip to compress the transformed data better. In our remaining results, we use double compression for *Aware* and *Aware-Block*; single compression for *Agnostic*.

3.5.2 Change in Compression Ratio with Varying Group Size

In this section, we evaluate the impact of group size on the compression ratios of the different schemes. We make several observations from Figure 3.10 (*Aware* and

Aware-Block overlap in b, c, and d). Overall, we find that *Aware* obtains significant compression for all four simulations. It reduces the total data to about half or even to one-third of the amount compared with the uncompressed checkpoint. Additionally, we see that data-aware compression achieves higher compression ratios than data-agnostic compression. The most dramatic case is for Cactus in Figure 3.10b for which the gain is 115% by *Aware* compared to *Agnostic*. We also find that changing group size impacts the simulation that benefits most from the *Aware-Block* scheme (ALE3D) more than those that benefit most from the *Aware* schemes (Cactus, Cosmology, Implosion). For ALE3D, the *Aware-Block* scheme achieves an 8% improvement in compression ratio from group size 2 to 32.

We also observe that the best compression scheme varies across applications. For ALE3D, *Aware-Block* is the best, while *Aware* is the best for Cactus. Both data-aware schemes perform well for Cosmology and Implosion. The segmented data distribution and collection in Cactus, Cosmology and Implosion apparently emulates blocking for these simulations. To determine which scheme to apply to a given application, one could try all schemes offline on a sample checkpoint set and then configure mcrEngine to use the best option.

Our results for *Aware-Block* and *Agnostic-Block* use the best block size from our tests for each simulation. While *Agnostic-block* could emulate data-aware compression if variable layouts are similar across checkpoints, our results for Cactus, Cosmology and Implosion disprove this theory. Further, while *Agnostic-Block* achieves similar compression ratios to *Aware* for ALE3D, blocking the interleaved variables, as with *Aware-Block*, provides significant additional benefit.

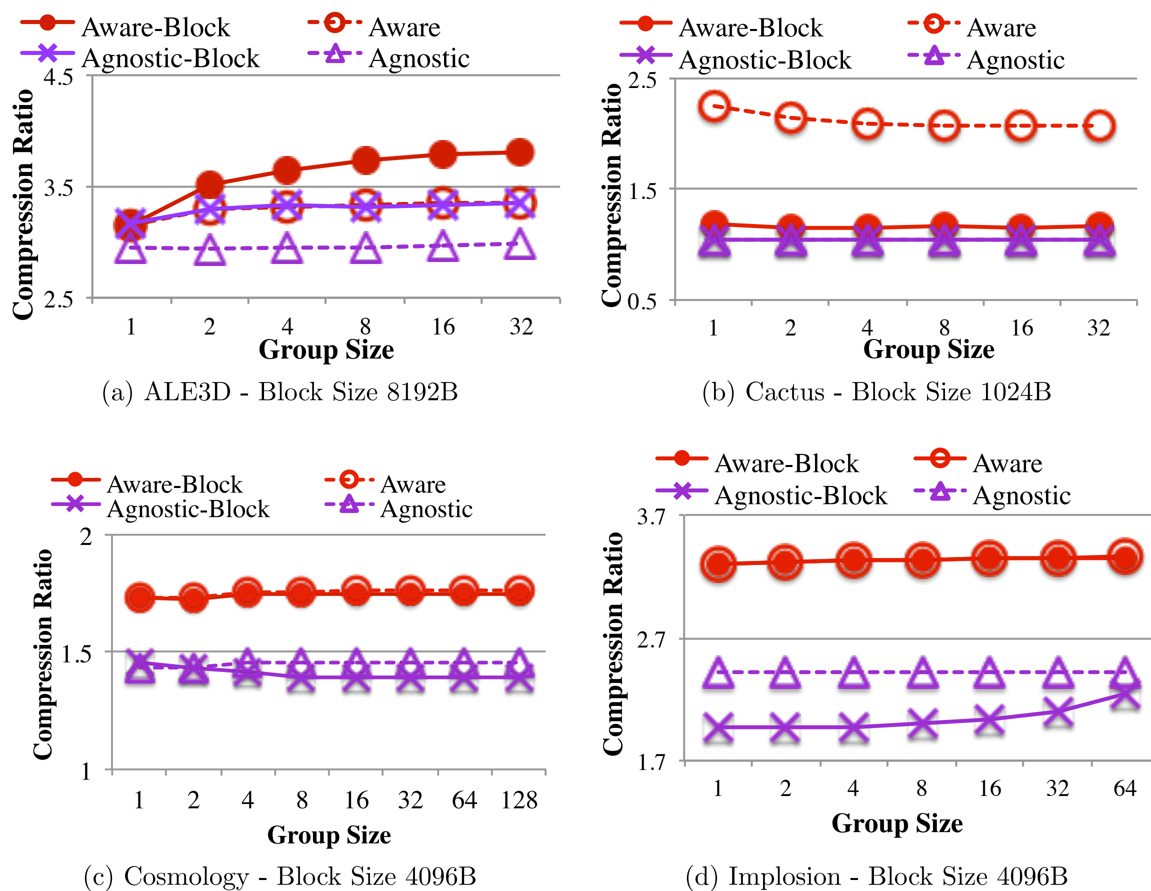


Fig. 3.10.: Compression ratio vs group size.

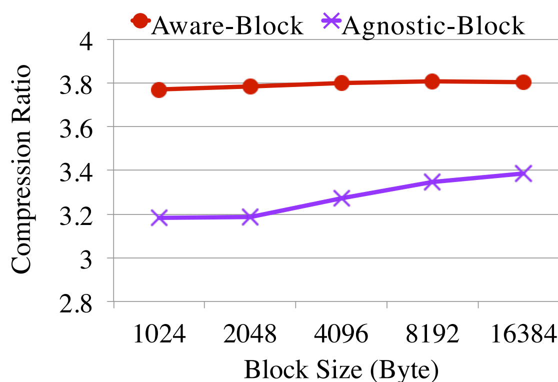
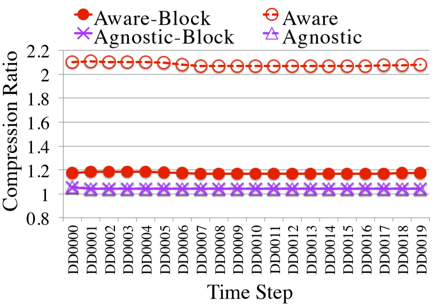
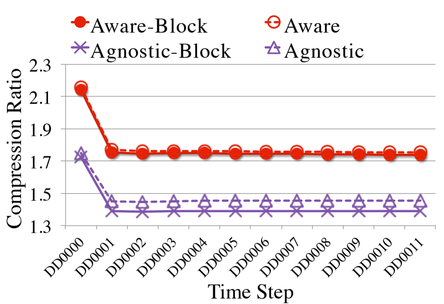


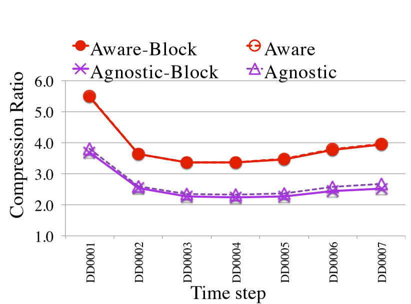
Fig. 3.11.: ALE3D compression ratio ($Group_Size = 32$)



(a) Cactus: Block Size 1024B, Group Size 32



(b) Cosmology: Block Size 4096B, Group Size 64



(c) Implosion: Block Size 1024B, Group Size 64

Fig. 3.12.: Compression ratio over time

3.5.3 Impact of Interleaving Granularity on Compression Ratio

ALE3D We now discuss the impact of varying block size on the compression ratios that the blocking schemes achieve. In this experiment, we only use checkpoints from ALE3D, since we observed in Figure 3.10, that only its checkpoints benefit from merging in blocks. Figure 3.11 shows that the compression ratio increases steadily with *Agnostic-Block*, while with *Aware-Block* it is fairly steady with a small initial increase.

3.5.4 Change in Compression Ratio as Simulation Progresses

To evaluate how the compression schemes fare as the simulation progresses through time, we plot the compression ratio for a series of consecutive checkpoint sets in Figure 3.12, which shows the results for the Cactus, Cosmology, and Implosion simulations. Cactus wrote 21 checkpoint sets, each 5 minutes apart. The Cosmology simulation wrote 12 checkpoint sets each 5 minutes apart. The Implosion simulation wrote 8 checkpoint sets, each 0.05 seconds apart. In all three simulations, we label the checkpoints as "`DD00XX`", where "`XX`" records the simulation time step.

We make three observations from Figure 3.12. First, the *Aware* and *Aware-Block* schemes yield higher compression ratios than *Agnostic* or *Agnostic-Block*. For example, the relative improvement in compression ratio of Cactus in Figure 3.12a is about 98%. Second, for both Enzo simulations, the compression ratio of the first checkpoint set is higher than for subsequent ones. The maximum compression ratio for Cosmology drops around 20% (from 2.15 to 1.78) after the first time step. For Implosion, it drops about 50% (from 5.4 to 3.6). These drops occur because Enzo initializes data structures with the same default values so the data is highly compressible across processes while Cactus does not always initialize data structures with default values. Third, the compression ratio quickly levels out as simulations progress. For Cosmology, the compression ratio only has a slight downward slope after the initial drop. Similarly, for Implosion, the compression ratio drops, levels out, and then begins to

Table 3.4: Relative improvement over *Agnostic*

Application	Measure	Aware-Block	Aware
ALE3D	Best	**27.72%** (32)	12.7% (32)
	Worst	6.6% (1)	6.6% (1)
Cactus	Best	11.85% (32)	**115%** (1)
	Worst	10.69% (2)	98% (32)
Cosmology	Best	20.59% (1)	**21.14%** (32)
	Worst	20.13% (32)	20.59% (1)
Implosion	Best	38.43% (32)	**38.83%** (32)
	Worst	36.26% (1)	36.26% (1)

edge higher as the Implosion simulation runs. We expect the compression ratios to behave in this manner for applications that converge to a result.

3.5.5 Summary of Data-Aware Compression Effectiveness

We summarize our evaluation of *Aware* and *Aware-Block* compared to *Agnostic* in Table 3.4, which shows the largest and smallest relative improvements, which occur with the group size specified in parentheses. While the group size that provides the greatest benefit varies, data-awareness always results in higher compression ratios. We observe that applications with higher interprocess checkpoint similarity gain more compression by interleaving variables in blocks (*Aware-Block*), while applications with higher intraprocess checkpoint similarity gain more by concatenating variables before compressing (*Aware*).

3.6 Performance of mcrEngine

This section evaluates mcrEngine overall performance. For checkpointing, mcrEngine merges and compresses checkpoints and then transfers them to the PFS. For restart, it reads checkpoints from the PFS and then decompresses and splits them into their original format. We demonstrate the efficiency of mcrEngine for the two largest checkpoint sets, ALE3D and Cactus. For each application, we use a group size

Table 3.5: Checkpoint/restart system configurations

NAME	DESCRIPTION
NoC+NoAgg	No compression, no aggregation
C+NoAgg	Individual compression, no aggregation
NoC+Agg	No compression, aggregation
Agnostic+Agg	MCRENGINE aggregation, data-agnostic compression
Aware+Agg	MCRENGINE aggregation, data-aware compression

of 32, and the data-aware merging scheme from Table 3.4 that provides the best compression ratio (*Aware-Block* for ALE3D and *Aware* for Cactus). Table 3.5 describes the different merging and compression schemes that we compare in this section.

The first compression pass yields little benefit for small but compressible variables so we omit them from it. In our framework, the minimum size is configurable; our experiments omit variables $< 100B$. Easily compressed data types include integers and characters. This choice reduces compression ratios by less than 1%, and improves performance.

We measure the times to transfer checkpoints to the parallel file system using IOR [44, 60], a benchmark that characterizes HPC I/O performance. We use the average respective file sizes for each of the compressed or uncompressed checkpoints.

3.6.1 Data-aware and Data-Agnostic I/O Performance

We compare the performance of data-aware and data-agnostic compression in terms of time to transfer checkpoints to the PFS. The objective is to study the benefit of having data-aware compression on I/O performance. Data-aware compression results in smaller files and thus requires less time to write the checkpoints to the PFS. The merged and compressed checkpoint sizes per writer for ALE3D were 1.3 GB and 1.5 GB for data-aware and data-agnostic compression respectively. For Cactus, the sizes per writer were 1.2 GB and 2.4 GB.

We show the results for Cactus in Figure 3.13. In the figure, because the group size is 32, the number of writers is $N/32$, where N is the number of processes in the

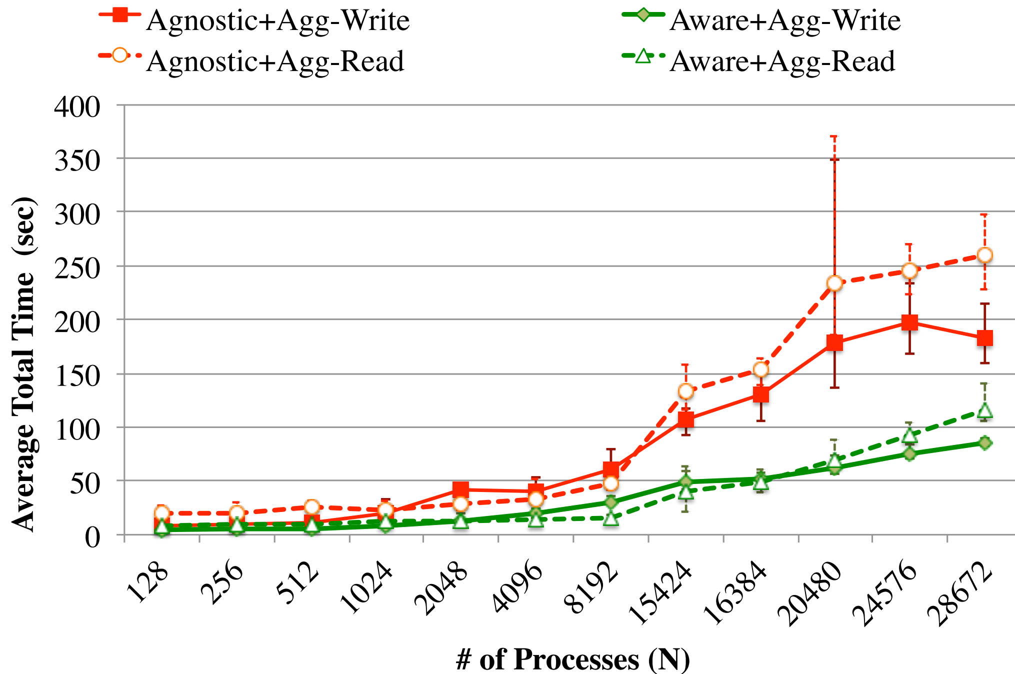


Fig. 3.13.: I/O performance with different compression schemes

job. MCRENGINE with data-aware compression reduces the data transfer overhead compared to data-agnostic compression for both reading and writing checkpoints.

3.6.2 Benefit of Checkpoint Aggregation

This experiment shows the PFS I/O performance impact of aggregation as we scale up the application. Figure 3.14 shows the average measurements and 95% confidence intervals for the time to write to the PFS using IOR for the Cactus checkpoints. For the schemes that aggregate, the number of writers is $N/32$ and the data transferred per reader/writer is 1.2 GB; for the schemes that do not aggregate, the number of writers is N and the data transferred is 87 MB per reader/writer. We find that aggregating the checkpoints in MCRENGINE results in better I/O scalability and reduces performance variability compared to non-aggregated I/O, particularly for read (i.e., restart) operations. Although unclear due to the overhead increase with

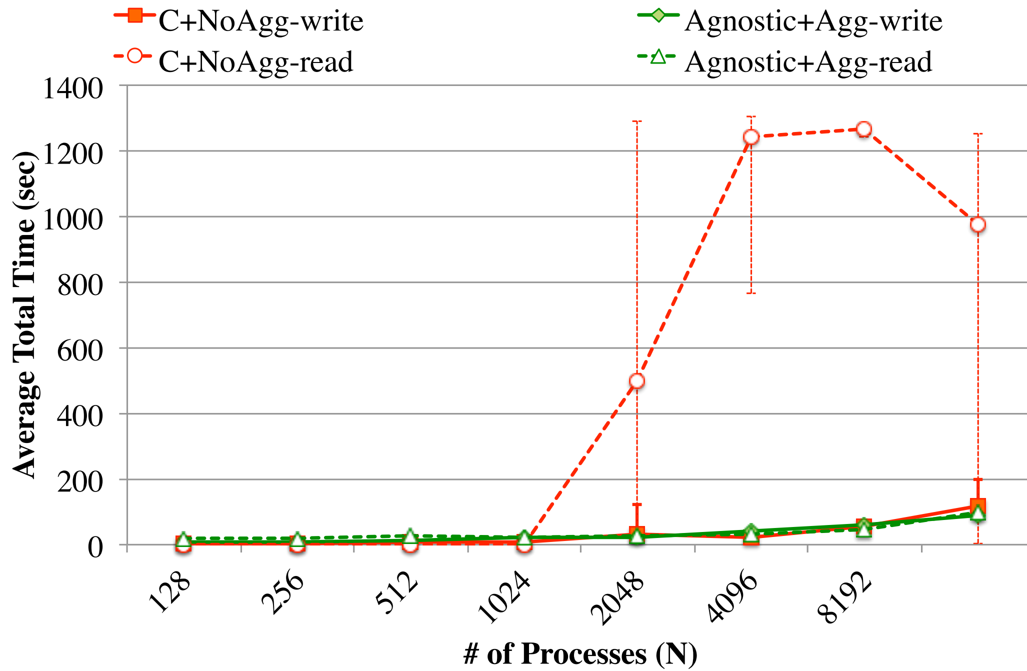


Fig. 3.14.: Benefit of aggregation for I/O performance

C+NoAgg-read, Figure 3.14 shows that aggregation improves write performance up to 26%.

3.6.3 Checkpoint and Restart Overheads

We now compare processing and transfer overhead using the mcrEngine checkpoint and restart schemes. Figures 3.15 and 3.16 show the best case average of the measured values of IOR for an application run with 15,408 processes, collected over several days at different times. The numbers beside each step correspond to the steps in Table 3.1 and 3.2.

We make several observations for checkpointing. First, data-aware compression takes more computation time than data-agnostic and the schemes that only use Gzip or do not compress. However, the checkpoint phase does not occur in the appliation's critical path so that overhead does not affect performance. Second, fetch time for

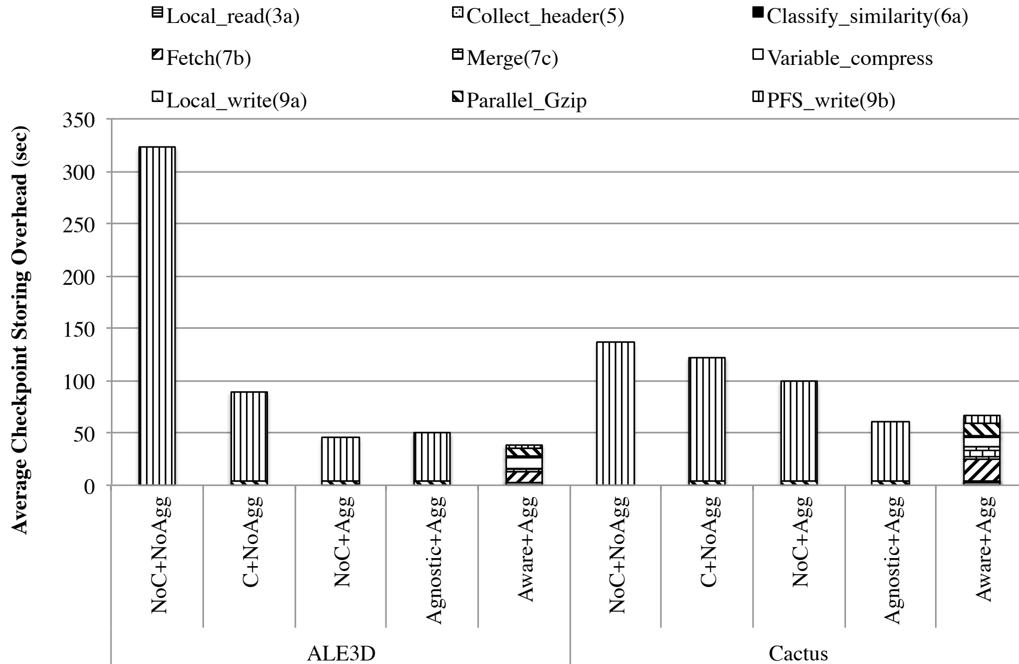


Fig. 3.15.: End-to-end checkpointing overhead

Cactus is higher than ALE3D because it has 5× more variables. Third, the smaller file sizes that result from data-aware compression mean that mcrEngine uses far less network resources when transferring these checkpoints. Data-aware compression reduces the I/O time by 92% and 86% over data-agnostic (Agnostic+Agg vs Aware+Agg) for ALE3D and Cactus. Finally, mcrEngine reduces checkpointing overhead by up to 87% over cases without aggregation (NoC+NoAgg vs Aware+Agg for ALE3D), and up to 32% for cases without compression (NoC+Agg vs Aware+Agg for Cactus). This reduction allows applications to checkpoint more frequently, which improves fault-tolerance.

Figure 3.16 shows the restart overhead results. Since restarts are on the application's critical path, lower overhead implies better end-to-end performance. We again make several observations. Data-aware compression reduces restart overhead by more than 62% compared to the current state of the practice (NoC+NoAgg vs, Aware+Agg) for ALE3D. It reduces overhead by more than 80% compared to only compressing individual checkpoints (C+NoAgg vs Aware+Agg). Second, network

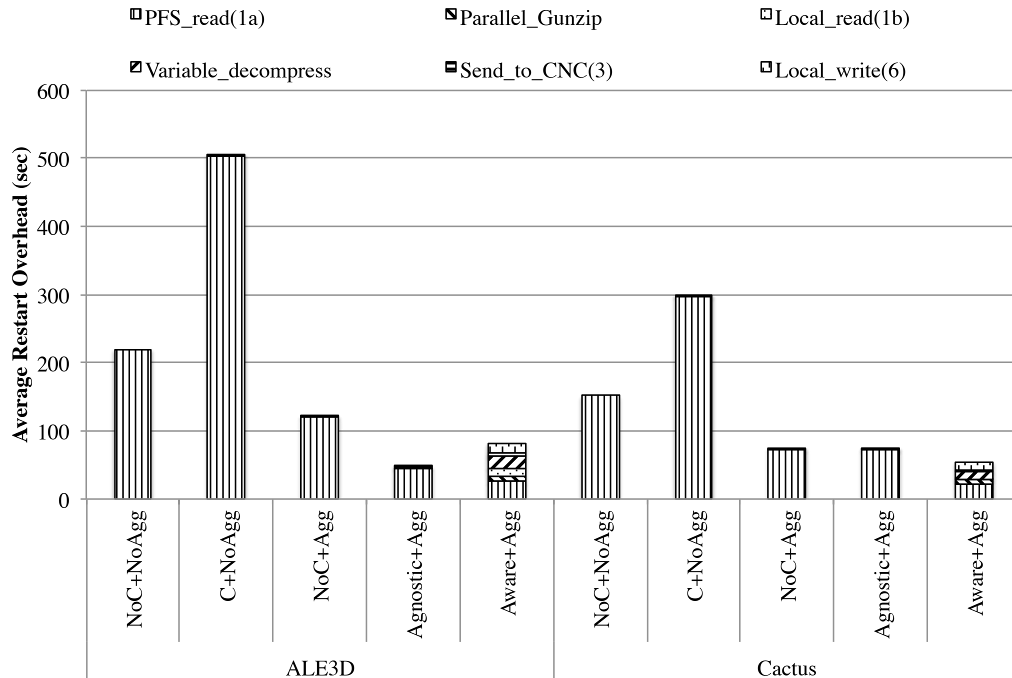


Fig. 3.16.: End-to-end restart overhead

transfer time dominates restart overhead for cases without aggregation, compared to CPU time for the mcrEngine data-aware scheme. Converting a network-bound problem to a CPU-bound problem improves scalability. Finally, data-aware compression significantly reduces network load by reading less data from the PFS compared with the data-agnostic case. The network I/O time for data-aware compression is 43% and 71% less than that of data-agnostic (Agnostic+Agg vs Aware+Agg) compression for ALE3D and Cactus.

3.6.4 Discussion

We observe writing to the PFS incurs the largest portion of the total overhead. Thus, our selection of the scheme that provides the highest compression ratio results in the least write time, which makes our scheme selection independent of system characteristics for large checkpoint files. However, system specifications such as I/O, network, and CPU characteristics, will affect the scheme's performance with smaller

checkpoint files when computation overhead dominates network overhead. In that case, another selection metric could be the ratio between compression and operation time overhead.

mcrEngine could be made *adaptive* by evaluating schemes periodically and selecting the best one for that checkpoint set. Also, mcrEngine currently selects the compression algorithm for each data type based on published results. An adaptive mcrEngine could sample their compression ratios to select these algorithms dynamically.

We used checkpoints from a variety of runs – both large and small. The ALE3D checkpoint set (the largest set), in particular, is from a production run and shows that the benefits of mcrEngine increase with checkpoint size. The smaller checkpoints (from Cosmology and Implosion) show less benefit. Currently, we use input sets provided by the community. Configuring these applications to run longer and to take larger checkpoints requires application-specific knowledge.

3.7 Related Work

Other researchers have investigated reducing the size of checkpoints by writing less data in the checkpoints or by compressing the checkpoint files. Incremental checkpointing reduces the size of checkpoints by writing changes in application data between full checkpoints [17–20]. These approaches are orthogonal to our work, as incremental checkpoints can be compressed for further savings [21].

Plank and Li compressed checkpoints to fit them in main memory [61], using a variation of Algorithm 1 [62]. They reported compression factors of 10 to 96 percent. They found that compression was beneficial with large and highly compressible checkpoints and when the compression factor exceeded the ratio of the disk write speed to compression speed. Li and Fuchs did not find any advantage in compressing checkpoints with an LZW algorithm on a uniprocessor system despite observing good compression factors since compression time exceeded uncompressed write

time [63]. Islam et. al. [64] found that compression and decompression overheads dominated checkpointing and restart times but the decreased network transfer time of the smaller files outweighed the higher overheads, especially with increasing checkpoint size. Ansel et. al. [65] also found that the time to compress checkpoints resulted in longer checkpoint times when writing to node-local disks although the disparity in restart times was smaller. These approaches all use data-agnostic compression while we investigate data-aware techniques. To the best of our knowledge, we are the first to investigate a data-aware compression approach and cross-process merge and compression for parallel applications with many processes.

Bautista-Gomez et. al. [66] developed a topology-aware reliable checkpointing library that provides reliability in cluster environments, which is different from the scalability challenge that we address. Wu et. al. [67] developed a compression scheme for bitmaps and demonstrated that it reduces bitmap index sizes and database query response time.

Several researchers have explored reducing the cost of checkpointing and writing to the file system. To reduce the number of writes to the file system, and ultimately to reduce checkpoint writing cost, Ouyang et. al. [68] cache writes of small and medium sizes within a node in order to aggregate them. Bautista-Gomez et. al. [69] avoid the I/O bottleneck of disk-based checkpointing and the issues of classic diskless checkpointing. These approaches are complementary to ours. They reduce the cost of writing data while our work reduces the amount of data written.

3.8 Summary

We presented mcrEngine, a novel scheme to compress checkpoints across processes. mcrEngine exploits semantic information in checkpoints to put similar data, i.e., with the same data type and name, close together to improve compression. We implemented several different schemes to interleave data from multiple checkpoints and found that different schemes provide the best results for different applications.

We evaluated mcrEngine and the different interleaving schemes through four real-world computational simulations in terms of the compression ratio and overhead. We found that the amount of compression gained from each scheme varied depending upon how the data was laid out across multiple processes. For some applications, similar data occur in the checkpoint from a single process, while for others, the similar data are striped in blocks and distributed across multiple processes. Thus, the best approach varies. Our compression schemes are as much as 115% better in reducing checkpoint size than simply applying Gzip compression to concatenated checkpoints. As expected, mcrEngine spends more computational time than simply applying compression; however, transferring less data overall with fewer concurrent writers reduced the overhead of writing to the parallel file system. mcrEngine converts network load to computational load; a fact that is crucial for making mcrEngine scale well with increasing application size. Since the parallel file system is often the bottleneck, this method reduces checkpointing overhead by up to 87%. Further, smaller files reduce restart times by up to 62%, a benefit that is particularly important since they occur on the application's critical path. In conclusion, by reducing the time to store and retrieve checkpoints, mcrEngine makes checkpointing-based fault-tolerance in large systems practical.

However, mcrEngine groups checkpoints according to their ranks. The question that we address in the next chapter is whether grouping checkpoints in a data-aware manner before compressing them, improves compressibility or not.

4. BENEFIT-AWARE CLUSTERING FOR PARALLEL APPLICATIONS

Writing to the Parallel File System (PFS) is very expensive. Specifically, if every single process writes directly to the PFS. This results in high contention on shared PFS and network resources. Parallel file systems, such as Lustre, are optimized for a small number of reads and writes of large amount of data, and perform poorly otherwise. In order to exploit this design of PFS, Islam *et. al* [70] proposed checkpoint aggregation on a set of nodes before transferring data to and from stable storage. Also, in order to achieve better compression, authors utilize semantic information written in checkpoints to align data so that the aggregated output compresses better. One question that still remains is, how should these processes be grouped. Our earlier work in [70] compresses members of each group in rank-wise order for demonstrating the benefit of data-aware aggregation and compression. However, applications that have separate groups of processes working on different parts of the input data, may not have such a straight forward mapping of rank to group. For example, applications that organize processes in a mesh, will have processes with non-contiguous ranks on the border computing similar tasks than processes in the middle. This may result in similarity in checkpoints from non-contiguous ranks. Thus, clustering them in groups with similar checkpoints will help such a system achieve better compression ratio.

The contributions of this work is to present preliminary results on synthetic data to support that clustering checkpoints based on benefit improves overall compression ratio. Our experimental results on synthetic data shows a 30% improvement in compression ratio compared to compressing checkpoints in rank-wise order.

The paper is organized as follows. Section 4.1 provides background. Section 4.2 describes benefit-aware clustering technique. Section 4.4 details the structure of the

implementation of mcrCluster. Section 4.5 presents our evaluation methodology and preliminary results.

4.1 Background

In this section, we discuss the related background information on different checkpoint types, how checkpoints are written, checkpoint data format, and data-aware compression technique developed in [70], named mcrEngine.

Most parallel scientific applications distribute simulation data across multiple processes. These processes generally coordinate globally to take a consistent checkpoint. During an asynchronous checkpointing phase, once a process finishes checkpointing, it resumes its computation. Currently, with most checkpointing systems, the processes send their checkpoints directly to the PFS, although checkpointing libraries such as SCR [11] can store checkpoints locally instead.

We define *data-aware aggregation* as aggregating across process checkpoints such that data with similar meaning remain together in the merged checkpoint. mcrEngine uses *data-aware compression*, which dynamically selects from a set of compression algorithms and applies the best one for each similar data group. *Data-aware aggregation and compression* is the entire process of interleaving semantically similar data and using a dynamically selected compression algorithm.

mcrEngine interprets two variables in different checkpoints as *similar* if their names agree and their data representation is identical (same data type). It uses metadata to locate checkpoint data that represent the same variables.

mcrEngine then merges similar variables in one of the following two ways:

- **Variable Concatenation** As Figure 3.7a shows (Cx.T denotes the temperature array of the checkpoint of rank x and Cx.P its pressure array), the *Data-Aware* or simply *Aware* scheme concatenates individual variables before compressing them;

- **Variable Blocking** As Figure 3.7b shows, the *Aware-Block* scheme interleaves variables in blocks of configurable sizes, instead of simply concatenating them.

In our previous work, we found that compressing variables with data-type specific algorithm converts non-uniform, not so compressible data to a more uniform format. Also, different merging schemes work well for different applications depending on their input distribution pattern. mcrCluster builds on top of our data-aware checkpoint aggregation and compression technique and improves compression ratio further by combining checkpoints from processes depending on a novel "similarity" metric that we present in this paper.

4.2 Benefit-Aware Clustering

As applications grow in scale, sending checkpoint data directly to the PFS will hinder the scalability of future checkpointing systems. Also, to play along the strength of PFS in handling small number of large data transfers, aggregation and compression has been shown to be a beneficial method in reducting the number of checkpoint files and overall data. Further investigation shows that grouping checkpoints randomly has impact on overall compression ratio. The goal is to break checkpoints in multiple groups in order to balance load on aggregator nodes and do so without impacting compression ratio.

Some additional design questions that we address are:

- how many groups and members per group to select?
- what should be the metric for deciding "similarity" or "distance"?
- how to make clustering practical for using in an online checkpointing system?

The following sections discuss our design and some other alternate choices.

4.2.1 Number of Groups

Typical clusters consist of compute nodes and gateway or I/O nodes. I/O nodes are the interface to disk resources. All I/O performed on compute nodes is routed to the I/O nodes over the internal switch network (such as InfiniBand). Individual files are stored as a series of "blocks" that are striped across the disks of different I/O nodes. This permits concurrent access by a multi-task application when tasks read/write to different segments of a common file. Data transfer through these gateway nodes can achieve the best parallelism if the number of aggregator nodes equals the number of gateway nodes. Each aggregator handles 1 group. So, the number of groups should equal to the number of aggregator nodes in turn.

The naive way of grouping is to equally (if possible) divide these checkpoints in rank-wise order. However, for applications that have subgroups of processes doing separate things and processes in the same group not necessarily having consecutive ranks – this method will not work well.

Our idea of grouping checkpoints is to utilize the knowledge of how beneficial it is to pair up two processes, and based on that knowledge create a group. The ultimate goal is to gain the most compression possible out of these groups.

4.2.2 New Similarity Metric

The ultimate goal of clustering checkpoints is to minimize the total size of compressed data. To do that, we want to pick the pairings of checkpoint files that give the most improvement in compression ratio. That is, we want to pick files that compress better together than they do by themselves or with other files.

We define the benefit of compressing two checkpoints together as the percentage improvement in size reduction over their individual compression ratios.

$$\alpha(i) = \frac{||i||}{||compress(i)||} \tag{4.1}$$

$$\alpha(j) = \frac{||j||}{||compress(j)||} \tag{4.2}$$

$$\alpha_{AwareBlock}(i,j) = \frac{||i|| + ||j||}{||AwareBlock(i,j)||} \tag{4.3}$$

$$Similarly, \tag{4.4}$$

$$\alpha_{Aware}(i,j) = \frac{||i|| + ||j||}{||Aware(i,j)||} \tag{4.5}$$

Benefit matrix, β is an $N \times N$ matrix, where N is the number of processes and $\beta_{Scheme}(i,j)$ is the benefit of compressing checkpoint i with j using $Scheme$ as the merging scheme. $Scheme$ can be either $Aware$ or $AwareBlock$ as presented in Section 3.2.2. Equation 4.6 gives us the % of reduction in size if we pair checkpoint i and j together, instead of compressing them separately and then concatenating.

$$\beta_{Scheme}(i,j) = [1 - \frac{||Scheme(i,j)||}{||Scheme(i)|| + ||Scheme(j)||}] \times 100\% \tag{4.6}$$

$$\text{where, } \beta_{Scheme}(i,j) = \begin{cases} -ve, & \text{Pairing these two checkpoints} \\ & \text{is not advantageous} \\ 0, & \text{Pairing these two checkpoint} \\ & \text{is not better than compressing} \\ & \text{each individually} \\ +ve, & \text{There is similarity across} \\ & \text{these two checkpoints and} \\ & \text{pairing them reduces total size} \end{cases}$$

Given a set of N checkpoint files C, $c_0 ... c_{n-1} \epsilon C$, we can define a benefit matrix $\beta^{Timestep}_{Scheme}$ such that $\beta^{T}_{Scheme}(i,j)$ represents the benefit matrix computed by applying a scheme $Scheme \in (AwareBlock, Aware)$ on the checkpoints i and j at time step T.

If there is a wide range of values in the benefit matrix, then we can reasonably expect that choosing the best pairings from this matrix to guide groupings of checkpoint files will come close to minimizing the total compressed size of C. If there is not much variability here, then we know that there is no "best" grouping of checkpoints and it does not really matter how we group things when we ship them off to the intermediate compression nodes.

4.2.3 Sampling Method

Variables can be sampled in one of four different ways:

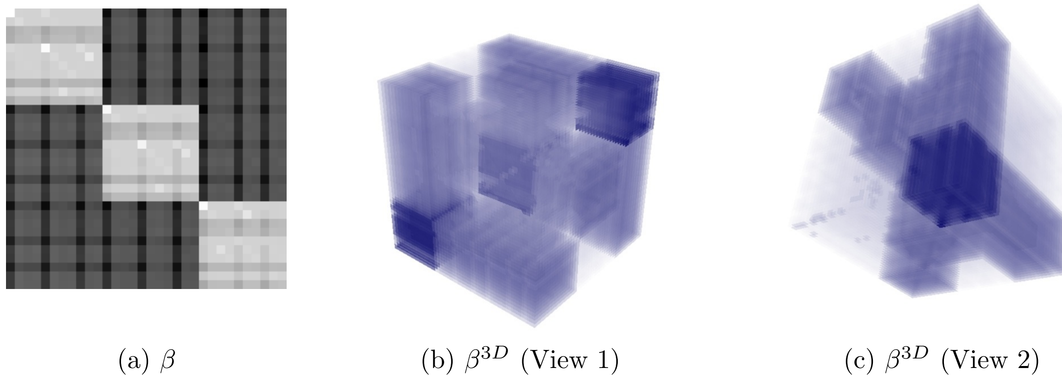
- *Random sampling:* sampling s elements from random positions of a variable
- *Chunking:* using the first s elements of a variable
- *Wavelet:* applying Wavelet compression to reduce data. The level of compression can be determined based on the configurable parameter s.
- *Complete:* use the entire variable for computing benefit.

4.3 Runtime Clustering

Ideally, for any checkpoint file C_i, S_i be the set of similar checkpoints to C_i, where similar means that the benefit of compressing the two checkpoints is high. That is, S_i contains all values of k such that $\beta(i, k)$ is beyond certain threshold. The most important point in designing a practical runtime clustering system is to ensure that we can approximate the correct benefit by only sampling a fraction of the entire data. We do the following to ensure this goal is met.

4.3.1 Reduce Dimension of β

One way to keep clustering practical is by reducing the dimension of the benefit matrix for each application. In order to see if pair-wise benefit is sufficient, we



(a) β (b) β^{3D} (View 1) (c) β^{3D} (View 2)

Fig. 4.1.: Benefit matrices of the application Cactus.

generated a 2D benefit matrix using pair-wise benefit values and a 3D benefit matrix using three-way benefit values. In 3D, S_i is the set of all unique j, k for which $\beta^{3D}(i, j, k)$ is high. Figures 4.1b and 4.1c show two different views of the benefit matrix computed using three-way compression on checkpoints from a real application, Cactus. Figure 4.1a shows that S_i in 2D is the same as S_i in 3D, at least for the Cactus application. This means that if checkpoints are clustered using their $2D$ benefit, the same group membership will be observed.

4.3.2 Reduce Data to Cluster

Computing an $N \times N$ benefit matrix includes $O(N^2)$ compressions. If not done in an efficient manner, computing clusters will not scale for even a small set of checkpoints. We reduce the data to cluster by:

- reducing the number of variables to cluster on
- reducing the amount of data to compress

From Table 3.3, we can observe that applications tend to have certain data-types that dominate over others. From our observations, actual computed values are either of single- or double-precision floats. We can configure our variable selection filter to only consider those variables that are of certain data types, larger than certain size,

and until a certain percentage of data is considered. For example, in our experiments, we selected 100 double-precision floating point variables with more than 1024 elements to be the variables to consider, and it covered 88% of actual data written in these checkpoints. The rationale is that certain data-types are less compressible but dominate the actual data space. If we can stage them in such a way that we can gain the most compression, then it will actually improve the overall compression ratio.

4.3.3 Centralized vs Distributed Clustering

We find the "distance" between checkpoints i and j by looking at their "similarity" relationships. The assumption is, if i and j result in better compression, and k does not compress well with any one of i or j, it is better to keep k separate. In other words, k will have a *larger* distance from i and j. Based on the benefit matrix, β, clustering algorithms compute Euclidean distance between row i and row j of β, in order to compute their "distance". The rationale is that checkpoint i and j will be grouped together if their benefit profiles are similar.

4.4 Structure of mcrCluster

We have designed a system, mcrCluster, in order to study the benefit of clustering on application checkpoints. Figure 4.2 shows the overall structure of how the entire system works. The system has two components, *Compute Node Component* and *Aggregator Node Component*. `Clustering` modules in the `Compute Node Components` of application processes collaborate to determine which cluster each of the processes belong to, and then send data to the respective `Aggregator Node Component` with rank equal to the cluster id. The `Aggregator Node Component` applies data-aware compression described in our previous work in [70] and in Sections 3.2 and 3.3.

Figure 4.2 shows that mcrCluster is a small component within the larger `Compute Node Component`. The mcrCluster component consists of the following four modules:

Algorithm 1: mcrCluster building benefit matrix for realizing clusters among checkpoints.

```
   Data: Checkpoint taken by process with rank r
   Result: Cluster assignment, produced by Rank 0
 1 initialization;
 2 loadConfigurations; /* Defines properties of variables that should be
   selected */
 3 read variables that match defined properties from local checkpoint file;
 4 Compute keys with <variable_names,types,sizes>;;        /* O(N × V × s) */
 5 Perform distributed sort on keys ;
 6 MPI_Allgather to exchange sampled data from filtered variables;
   /* O(log(N) + (N − 1) × V × s) */
 7 for i = 1 → V do
 8 |   Merge and compress i^th variable from rank r through N using data-type
   |   specific compression algorithm;                      /* fpc,fpzip */
 9 end
10 Compress merged-compressed data using general purpose compression
   algorithm;                     /* Parallel Gzip, O(V + 1) compression */
11 Compute β(r, p) by using Equation 4.6;
12 MPI_Allgather to collect the entire benefit matrix β at every process;
   /* O(log(N) */
13 for p = 1 → N do
14 |   for i = 1 → N do
15 |   |   δ(r, p) += (β(r, i) − β(p, i))^2
16 |   end
17 |   δ(r, p) = sqrt(δ(r, p));                                 /* O(N^2) */
18 end
19 MPI_Allgather to distribute entire distance matrix δ among all processes;
   /* O(log(N) */
```

- the `Filter` module selects variables that are of certain types and larger than a threshold size. If the configuration file does not specify any type, then all data-types are considered. It is important to note that our design decision of selecting a small number of variables is based on our observation with 4 different real world applications that less than 10% of variables constitute more than 80% of data.

Algorithm 2: mcrCluster building benefit matrix for realizing clusters among checkpoints.

Data: Checkpoint taken by process with rank r
Result: Cluster assignment, produced by Rank 0

1 initialization;
2 loadConfigurations; `/* Defines properties of variables that should be selected */`
3 read variables that match defined properties from local checkpoint file;
4 Compute keys with <variable_names,types,sizes>;
5 Perform distributed sort on keys ;
6 MPI_Allgather to exchange sampled data from filtered variables;
`/*` $O(log(N) + (N-1) \times V \times s)$ `*/`
7 **for** $i = 1 \rightarrow V$ **do**
8 Merge and compress i^{th} variable from rank r through N using data-type specific compression algorithm; `/* fpc,fpzip */`
9 **end**
10 Compress merged-compressed data using general purpose compression algorithm; `/* Parallel Gzip,` $O(V+1)$ `compression */`
11 Compute $\beta(r,p)$ by using Equation 4.6;
12 Call CAPEK [71] to cluster data in a distributed manner in $O(logN)$ (since $Vs = N$); `/*` $O(log(N))$ `*/`

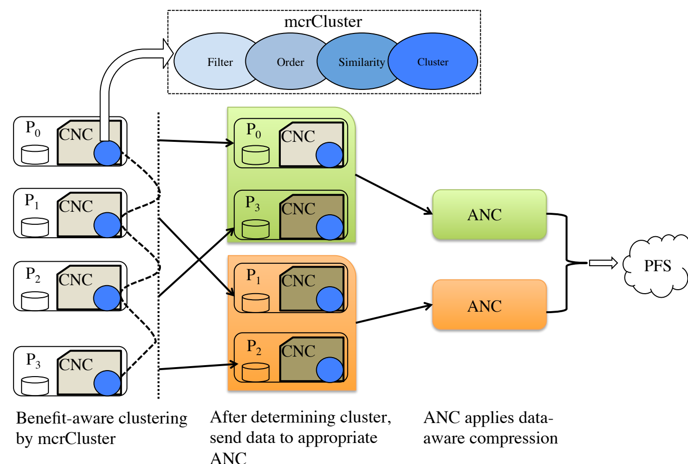


Fig. 4.2.: Overall structure of benefit-aware clustering and data-aware compression.

- the `Order` module sorts selected variables in a distributed manner, according to their names and data-types. The idea is to order variables such that the ith variable for each process has the matching name and data-type to be considered

"similar". This module also samples data based on one of random sampling, chunking, or wavelet transform methods to reduce the amount of data to send, and then performs an MPI collective communication to send data to the other processes.

- the `Similarity` module of process i builds the ith row of the entire benefit matrix, $\beta(i)$.
- the `Cluster` module either implements a centralized or a distributed clustering algorithm, depending on a configuration parameter. In our current implementations, we made PAM and CLARA available as centralized clustering algorithms and CAPEK [71] as the distributed clustering algorithm. Implementations of these algorithms have been taken from the library Muster [72].

4.4.1 Algorithm for Building Benefit-Matrix in Runtime

Algorithm 1 summarizes the steps that mcrCluster takes in implementing runtime clustering. Lines 1 through 4 denotes tasks that the `Filter` module on each process performs locally, namely – initializing variables, reading configuration file, filter variables and create hash key for distributed ordering of variables. Lines 5 and 6 are performed by the `Order` module to arrange variables in a particular order and distribute data among processes. The `Similarity` module performs tasks listed from line 7 to 19. Lines 7 through 11 are performed by each process locally in parallel. Line 12 is another MPI collective communication. Lines 13 through 19 calculates and distributes the Euclidean distance for each process in parallel so that the clustering algorithm can access distance between rank r and any other process in $O(1)$ time. This goes a long way in improving scalability of the clustering step.

The worst-case runtime complexity of Algorithm 1 can be determined by computing the cost of the dominating operations in this algorithm. The most expensive steps are:

- line 6: if the configuration parameter for the size of sample is s, then the total amount of data exchanged in this step for all processes is $N \times V \times s$.

- For N number of processes, the complexity of the `MPI_Allgather` operation is $lgN + (N-1)Vs$ [73], where $V \times s >> N$. So, the overall complexity is $O(NVs)$.

- lines 7 to 12: the empirical overhead is proportional to the cost of $V+1$ number of compressions.

- line 12: $O(N + lgN)$, since the amount of data is $O(N)$ benefit values.

- lines 13 - 17: $O(N^2)$ complexity of line 19 is $O(lgN + N)$.

Since, the overall complexity of Algorithm 1 is $O(N^2)$, it will not scale well as the number of checkpoints and the amount of data increases. However, Algorithm 1 is a good choice to study the benefit of clustering on applications at small scale. Algorithm 2 shows another scalable approach that uses CAPEK instead of PAM or CLARA to cluster on benefit values. Each process contributes N number of benefit pairs and this makes the overall complexity of CAPEK to be $O(log(N))$. CAPEK samples processes and uses Euclidean distance to compute the distance between benefit values of pairs of processes.

To make mcrCluster scale up to large-scale applications, Algorithm 3 lists the steps for clustering checkpoints in $O(\frac{Vs}{N} log(N))$ using CAPEK. Here, V is the number of variables to sample and s is the number of elements from each variable to use for clustering. Both of these parameters are configurable and can be changed to adjust the amount of data to cluster on.

4.5 Evaluation of mcrCluster

To evaluate the effectiveness of grouping checkpoints correctly, we conducted several experiments on synthetic data and real application checkpoint sets.

Algorithm 3: mcrCluster using distributed clustering for large-scale applications.

Data: Checkpoint taken by process with rank r
Result: Cluster assignment, produced by Rank 0

1. initialization;
2. loadConfigurations; `/* Defines properties of variables that should be selected */`
3. read variables that match defined properties from local checkpoint file;
4. Compute keys with <variable_names,types,sizes>;
5. Perform distributed sort on keys ;
6. Sample variables to create data to cluster on using one of the sampling methods described in 4.2.3.
7. Call CAPEK [71] to cluster data in a distributed manner in $\frac{Vs}{N}O(logN)$.

For proof of concept, we generated 32 checkpoints where processes with even ranks wrote 0s and those with odd ranks wrote $rank|1234567$ as double-precision floating point values. For generating these checkpoints, we used the file system benchmark IOR [44]. Each of these checkpoints contained 1 variable each. We also applied benefit-aware clustering on checkpoint sets for multiple simulation steps of the AMR application Cactus.

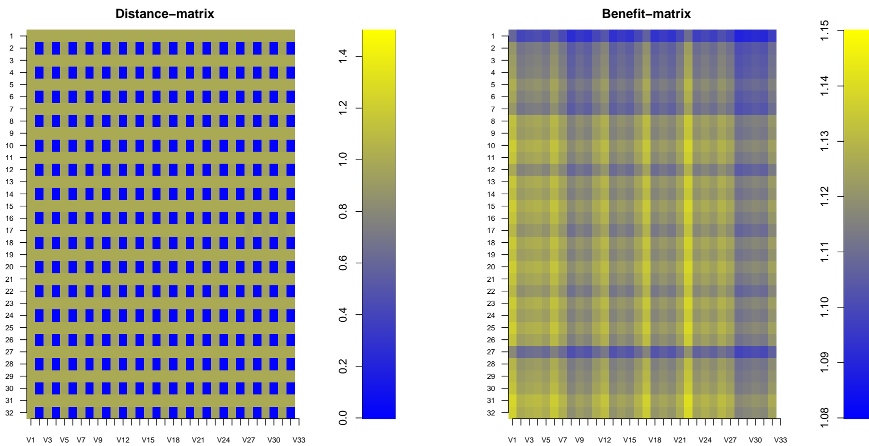
We evaluate the effectiveness of our novel benefit-aware clustering technique. In particular, we explore:

- Effectiveness of benefit metric in finding clusters
- The impact of benefit-aware clustering on compression ratio

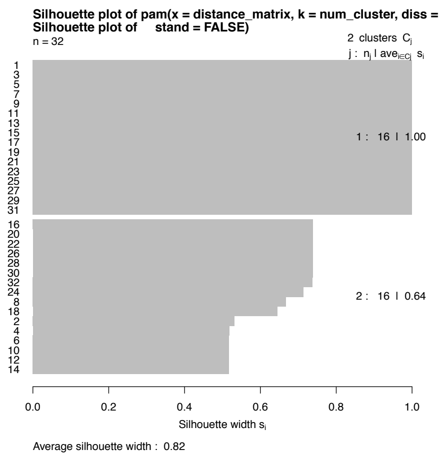
For our experiments, we used the entire variable to build the benefit matrix.

4.5.1 Effectiveness of Benefit Metric in Finding Clusters

Figure 4.3a and Figure 4.3b are the visual representations of how benefit values vary for a time step in the middle of a simulation for two applications IOR and Cactus. The observations that can be made from Figure 4.3 are:



(a) Benefit matrix of IOR (synthetic data) (b) Benefit matrix of Cactus (AMR)



(c) Silhouette plot showing natural clustering in benefit values of IOR checkpoints

Fig. 4.3.: Benefit matrix and silhouette plot of 32 checkpoints from Cactus

- the range of benefit values is really close for AMR applications in 4.3b. This indicates that using benefit-aware clustering may not be beneficial for these checkpoint sets. Table 4.1 summarizes that benefit-aware clustering results in the same compression ratio as grouping them in rank-wise order.
- since the synthetic data from IOR had distinct groups in data, the benefit matrix clearly shows that. This indicates that if checkpoint sets have distinct

Table 4.1: Comparison of compression ratios for different clustering schemes.

Applications	Number of Clusters	Number of checkpoints	Random	Rank-wise	Benefit-aware
IOR	2	32	509	509	665
Cactus	4	128	1.25	1.29	1.29

inherent clusters in them, then our novel benefit metric will be able to realize them.

4.5.2 Impact of Benefit-Aware Clustering on Compression Ratio

To study the impact of benefit-aware clustering on the overall compression ratio, we applied mcrCluster on the synthetic checkpoint set generated from IOR. Figure 4.3c shows the silhouette plot of the benefit matrix computed by mcrCluster and realized by the statistical software R. Figure 4.3c shows that our novel benefit metric does a good job in realizing the natural clustering among checkpoints, if there exists any. Table 4.1 summarizes compression ratios of applications and shows that with data-aware clustering, compression ratio improves by X% compared to random ordering, and 30% compared to rank-wise grouping. This improvement is significant enough to make us excited about finding real applications with significant variation in data.

4.6 Discussion

In this paper, we developed a new technique for grouping checkpoints from different processes of an application depending on their data in order to improve the overall compression ratio. To quantify benefit of clustering checkpoints together, we developed a new metric for measuring "similarity" between checkpoints. To study the impact of our new clustering mechanism, we design and develop mcrCluster, a system that uses data-aware compression technique for computing "similarity". Experiments with mcrCluster and mcrEngine on synthetic data shows that signifi-

cant benefit can be obtained by clustering checkpoints with diverse data. Visualizing the generated benefit matrices give us a good idea about when clustering is beneficial and when simple rank-wise ordering is sufficient. We are looking for real applications with groups of processes responsible for distinctly different computation to apply our technique and study its benefit.

5. CONCLUSION

Checkpoint-restart is the most widely used technique for making large-scale distributed systems resilient and fault-tolerant in the face of increasing failure rate. As applications scale with the ever increasing amount of computational resources, the amount of data produced by them increases proportionately. More than 80% of data transferred between compute nodes and stable storage is checkpoint data. Today's checkpointing systems will not be able to handle such insurgency in data since network bandwidth is not scaling proportionately with the amount of computational resources. To solve this problem, in this thesis we focus on checkpointing systems in two different types of computational environments – high-through systems such as grids and high-performance clusters such as supercomputers.

Todays grid systems often use expensive, high-performance dedicated checkpoint servers. However, in a geographically distributed environment, this solution incurs high checkpoint transfer latency. Additionally, transferring gigabytes of checkpoint data to distant locations through shared network adversely impacts the utilization of the network as well as the performance of users of the entire system. To solve this problem, in this thesis, we present a novel failure-model and fault-tolerant technique for storing checkpoints in a distributed manner on shared grid resources. Our system, FALCON, selects storage resources periodically based on their predicted reliability and load, and stores them using fault-tolerant techniques. Our experiments on DiaGrid shows that FALCON improves application performance between 11% and 44% compared to Condor's checkpointing systems with local and remote storage servers, respectively.

The major challenge of checkpointing systems in HPC is that network bandwidth is not scaling proportionately with the increase in the amount of checkpoint data. In addition to that, the current practice of storing checkpoint files by each process

individually results in contention on parallel file system resources and network bandwidth. This ultimately results in checkpointing systems not being able to scale with the growth of applications. To solve this problem, we propose a new technique called — *data-aware checkpoint aggregation and compression*. Our system, mcrEngine, aggregates checkpoints in rank-wise order, merges "similar" variables, and performs data-aware compression. For real-world applications, mcrEngine achieves 3.8× compression compared to uncompressed data; improves compression ratio by 115% compared to concatenating them and compressing using general purpose algorithm. Our preliminary results on synthetic data shows that grouping checkpoints in "benefit-aware" manner can improve compression ratio even further. Our observation in Chapter 4.6 is that our benefit metric is able to realize any inherent clusters these checkpoints may have. Results show that mcrCluster achieves significant improvement in compression ratio, compared to grouping them in rank-wise order.

In summary, this thesis contributes in solving the scalability challenge of checkpointing systems in large-scale, distributed computing environments. We hope that techniques presented in this thesis will go a long way in making the exascale dream a reality.

LIST OF REFERENCES

LIST OF REFERENCES

[1] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. Gunnels, and F. Streitz, "Extending Stability Beyond CPU Millennium: a Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, p. 58, ACM, 2007.

[2] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," in *Journal of Physics: Conference Series*, vol. 78, p. 012022, IOP Publishing, 2007.

[3] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, and K. Hill, "ExaScale Software Study: Software Challenges in Extreme Scale Systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[4] "Top500 Supercomputing Site." `http://www.top500.org/`.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[6] E. N. Elnozahy and J. S. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 97 – 108, April-June 2004.

[7] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello, "Coordinated Checkpoint Versus Message Log for Fault Tolerant MPI," *International Journal of High Performance Computing and Networking*, vol. 2, no. 2, pp. 146–155, 2004.

[8] R. L. Berger, L. M. Divol, S. H. Glenzer, D. E. Hinkel, R. K. Kirkwood, A. B. Langdon, J. D. Moody, C. H. Still, L. J. Suter, E. A. Williams, *et al.*, "Modeling with Backscatter and Transmitted Light of High-Power Smoothed Beams with pF3D: A Massively Parallel Laser Plasma Interaction Code," in *Proceedings of SPIE*, vol. 4424, p. 206, 2001.

[9] A. Moody, "Overview of the Scalable Checkpoint / Restart (SCR) Library." `http://www.csm.ornl.gov/srt/conferences/ResilienceSummit/2009/pdf/moody_slides.pdf` .

[10] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–351, 2010.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, pp. 1 –11, November 2010.

[12] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 19:1–19:10, IEEE Computer Society Press, 2012.

[13] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience.," *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[14] X. Ren, R. Eigenmann, and S. Bagchi, "Failure-aware checkpointing in fine-grained cycle sharing systems," in *Proceedings of the 16th international symposium on High performance distributed computing*, pp. 33–42, 2007.

[15] de Camargo, R. Y., Cerqueira, Renato, and K. Fabio, "Strategies for storage of checkpointing data using non-dedicated repositories on grid systems," in *MGC*, pp. 1–6, 2005.

[16] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp. 336–345, 2005.

[17] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pp. 277–286, 2004.

[18] S. I. Feldman and C. B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, pp. 112–123, 1988.

[19] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 783–788, 2008.

[20] K. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-Based Incremental Checkpointing Using GPUs," *Recent Advances in the Message Passing Interface*, pp. 272–281, 2011.

[21] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," Tech. Rep. CS-95-302, University of Tennessee, August 1995.

[22] K. Ryu and J. Hollingsworth, "Resource Policing to Support Fine-Grain Cycle Stealing in Networks of Workstations," *IEEE Transactions on Parallel And Distributed Systems*, pp. 878–892, 2004.

[23] "http://www.dia-grid.org/members/purdue.cfm."

[24] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi, "Resource Failure Prediction in Fine-Grained Cycle Sharing Systems," in *Proc. of Fifteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-15)*, pp. 19–23, 2006.

[25] X. Ren and R. Eigenmann, "Empirical studies on the behavior of resource availability in fine-grained cycle sharing systems," in *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pp. 3–11, 2006.

[26] T. Z. Islam, S. Bagchi, and R. Eigenmann, "Falcon: a system for reliable checkpoint recovery in shared grid environments," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–12, ACM, 2009.

[27] D. K. Jacob Strauss and F. Kaashoek, "A measurement study of available bandwidth estimation tools," in *IMC*, pp. 39–44, 2003.

[28] Z. Wilcox-O'Hearn, "Zfec Homepage," *Located at http://allmydata.org/trac/zfec*, 2008.

[29] A. K., J. A., X. Wu, F. M., J. B., C.-W. Tseng, and Y. D., "Biobench: A benchmark suite of bioinformatics applications," in *ISPASS '05*, pp. 2–9, 2005.

[30] T. Bray, "The Bonnie home page," *Located at http://www.textuality.com/bonnie*, 1996.

[31] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant mpi," in *ICS '03*, pp. 234–243, 2003.

[32] J. Walters and V. Chaudhary, "A Comprehensive User-level Checkpointing Strategy for MPI Applications," tech. rep., TR 2007-1, The State University of New York, Buffalo, NY, 2007.

[33] R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication," in *Peer-to-Peer Systems IV 4th International Workshop IPTPS 2005*, 2005.

[34] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore prototype," in *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[35] B. Rood and M. J. Lewis, "Multi-state grid resource availability characterization," in *GRID '07*, pp. 42–49, 2007.

[36] B. Rood and M. Lewis, "Scheduling on the Grid via multi-state resource availability prediction," in *Grid '08*, pp. 126–135, 2008.

[37] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[38] F. Petrini, "Scaling to Thousands of Processors with Buffer Coscheduling," in *Scaling to New Height Workshop*, (Pittsburgh, PA), 2002.

[39] "The ASC Sequoia Draft Statement of Work." https://asc.llnl.gov/sequoia/rfp/02_SequoiaSOW_V06.doc, 2008.

[40] K. Iskra, J. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 153–162, ACM, 2008.

[41] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer, "Parallel I/O on the IBM Blue Gene/L System," *Blue Gene/L Consortium Quarterly Newsletter, Tech. Rep., First Quarter*, 2006.

[42] R. Hedges, B. Loewe, T. McLarty, and C. Morrone, "Parallel File System Testing for the Lunatic Fringe: The Care and Feeding of Restless I/O Power Users," in *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 3–17, IEEE Computer Society, 2005.

[43] T. H. Cormen and D. Kotz, "Integrating Theory and Practice in Parallel File Systems," in *Proceedings of the 1993 DAGS/PC Symposium*, vol. 7, 1993.

[44] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance of XT3," in *Proceedings of the 49th Cray User Group (CUG) Conference*, 2007.

[45] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," in *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, (Berlin), Springer, 2003.

[46] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H. C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, "Cactus Tools for Grid Applications," *Cluster Computing*, vol. 4, no. 3, pp. 179–188, 2001.

[47] B. O'Shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR Cosmology Application," *Adaptive Mesh Refinement – Theory and Applications*, pp. 341–349, 2005.

[48] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs," in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 2–13, 2006.

[49] "Who Uses HDF?." http://www.hdfgroup.org/users.html.

[50] "The NetCDF Users Guide." `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf.pdf`.

[51] "Enzo N-Body Simulation with Hydro+Dark Matter Nested Cosmology Simulation,"

[52] M. Burtscher and P. Ratanaworabhan, "FPC: A High-speed Compressor for Double-Precision Floating-Point Data," *IEEE Transactions on Computers*, pp. 18–31, 2008.

[53] P. Lindstrom and M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 1245–1250, 2006.

[54] L. Reinhold, "QuickLZ," 2009.

[55] "A Parallel Implementation of GZIP for Modern Multi-processor, Multi-core Machines." `http://zlib.net/pigz/`.

[56] E. Seidel and W. Suen, "Numerical Relativity as a Tool for Computational Astrophysics," *Journal of Computational and Applied Mathematics*, vol. 109, no. 1-2, pp. 493–525, 1999.

[57] J. Li, W. Liao, A. Choudhary, and V. Taylor, "I/O Analysis and Optimization for an AMR Cosmology Application," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 119–126, IEEE, 2002.

[58] R. Liska and B. Wendroff, "Comparison of Several Difference Schemes on 1D and 2D Test Problems for the Euler Equations," *SIAM Journal of Scientific Computing*, vol. 25, no. 3, pp. 995–1017, 2003.

[59] "Capability Cluster Sierra at LLNL." `https://computing.llnl.gov/?set=resources&page=OCF_resources#sierra`.

[60] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, p. 42, 2008.

[61] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, no. 2, pp. 62–67, 1994.

[62] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line Data Compression in a Log-Structured File System," *ACM SIGPLAN Notices*, vol. 27, pp. 2–9, September 1992.

[63] C. Li and W. Fuchs, "CATCH - Compiler-Assisted Techniques for Checkpointing," in *20th International Symposium on Fault-Tolerant Computing*, pp. 74–81, June 1990.

[64] T. Z. Islam, S. Bagchi, and R. Eigenmann, "FALCON: A System for Reliable Checkpoint Recovery in Shared Grid Environments," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, ACM, 2009.

[65] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *23rd IEEE International Parallel and Distributed Processing Symposium*, (Rome, Italy), May 2009.

[66] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, IEEE, 2011.

[67] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing Bitmap Indices with Efficient Compression," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 1–38, 2006.

[68] X. Ouyang, K. Gopalakrishnan, and D. K. Panda, "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems," in *International Conference on Parallel Processing*, pp. 34–41, IEEE, 2009.

[69] L. A. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 63–72, IEEE, 2010.

[70] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "Mcrengine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 17:1–17:11, IEEE Computer Society Press, 2012.

[71] T. Gamblin, B. R. De Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Clustering performance data efficiently at massive scales," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 243–252, ACM, 2010.

[72] "Muster Library." `http://tgamblin.github.com/muster/main.html`.

[73] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

VITA

VITA

Tanzima Zerin Islam received her Bachelors in Computer Science and Engineering from Bangladesh University of Engineering and Technology in November, 2006. Before starting as a Ph.D. student at Purdue in August 2007, she briefly worked for a leading research and development company in Bangladesh. In her short stay, she co-developed the first software solution to service-independent telecommunication network, also known as Intelligent Network (IN). The most significant achievement of her work has been the deployment of this technology in improving the overall communication system of major cities in Bangladesh. During her Ph.D., she worked on many different practical challenges that persist in making distributed systems more reliable. Her software solution improves the reliability of Purdue's largest opportunistic Condor system – DiaGrid and as a result, improves the performance of applications significantly.

Tanzima's research interests are in the broad areas of fault-tolerance and reliability in large-scale distributed environments. Her research internships during Summers of 2010, 2011, and 2012 at Lawrence Livermore National Laboratory led to significant publications. She co-authored multiple papers that were nominated for the Best Student Paper awards in premier conferences in the area of High-performance computing. She received the 2nd place in ACM Student Research Competition in 2010, and the Best Female Team award in the National Collegiate Programming Contest (2004) in Bangladesh, among others. Tanzima is an active member of the Bangladesh Student Association (BDSA) at Purdue University and chaired many important roles in the organization. She is also a member of the Society of Women Engineers and mentored several undergraduate, M.S. and first-year Ph.D. students over the years.