Fall 11-2-2017

# Improving Large Scale Application Performance via Data Movement Reduction

Dewan M. Ibtesham
*University of New Mexico*

_____

*Candidate*

_____

*Department*

This dissertation is approved, and it is acceptable in quality and form for publication:

*Approved by the Dissertation Committee:*

_____, Chairperson

_____

_____

_____

_____

_____

_____

_____

_____

_____

# Improving Large Scale Application Performance via Data Movement Reduction

by

**Dewan Ibtesham**

B.Sc., Computer Science and Engineering, BUET, Bangladesh, 2007
M.Sc., Computer Science, UNM, 2014

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2017

# Dedication

*To Amma for always believing in me,*

*To Abba for being my role model,*

*To Shoshi for her steadfast support,*

*&*

*To the loving memory of my grandmother,*

*Zaman Ara (1925-2016)*

# Acknowledgments

I would like to express my sincere gratitude to the people without whom this thesis would not be possible. I would like to begin by acknowledging my advisor, professor Dorian Arnold for his extraordinary guidance, and support over these years. Dorian has taught me how an idea can be transformed into good research and can be properly communicated. His advice both on research as well as on my career have been invaluable.

I would also like to recognize the significant contributions of my co-advisor, Kurt Ferreira. Throughout my time at UNM, Kurt was always there with his help and guidance. He was always instrumental in reviewing my manuscripts, asked questions to challenge my assumptions and improve the research.

I would also like to extend my thanks to my committee members, professor Patrick Bridges and professor David Lowenthal. Patrick has always been generous with his time and advice. Many of the results in this dissertation was improved by his valuable feedback. David had also provided helpful and valuable feedback to improve the quality of this work.

Thank you to my fellow students in the Scalable Systems Lab for sitting through the practice talks, engaging in many paper discussion sessions and helping me to navigate graduate school. In particular, I would like to express my gratitude to: Taylor Groves, Scott Levy, Matthew Dosanjh, Whit Schonbien and Hans Weeks. Also many thanks to David DeBonis, Patrick Widener, Kevin Pedretti and Ryan grant from Sandia National Laboratories for engaging discussions on many occasions.

I would like to express my gratitude to my parents, Dewan Yamin and Amina Akhter, for shaping me the person I am today and my sister Nuzhat yamin, for teaching me how to remain strong in difficult situations. Special thanks to my mother-in-law, Mahmuda Hossain, for helping us whenever we needed her. Gratitude to Mr and Mrs. Ali for being our family away from home. I would also like to thank the Bangladeshi community in Albuquerque specially for their love and hospitality.

Finally, I would like to thank my wife Shoshi for her continuous support through out this journey. She was always encouraging and without her I could not have done it.

Last, but certainly not the least, I thank my daughter, Anuva, for being herself.

# Improving Large Scale Application Performance via Data Movement Reduction

by

**Dewan Ibtesham**

B.Sc., Computer Science and Engineering, BUET, Bangladesh, 2007
M.Sc., Computer Science, UNM, 2014

Ph.D., Computer Science, University of New Mexico, 2017

## Abstract

The compute capacity growth in high performance computing (HPC) systems is outperforming improvements in other areas of the system for example, memory capacity, network bandwidth and I/O bandwidth. Therefore, the cost of executing a floating point operation is decreasing at a faster rate than moving that data. This increasing performance gap causes wasted CPU cycles while waiting for slower I/O operations to complete in the memory hierarchy, network, and storage. These bottlenecks decrease application time to solution performance, and increase energy consumption, resulting in system under utilization. In other words, data movement is becoming a key concern for future HPC system-design.

Data volume reduction techniques (e.g. lossless data compression, information hiding approaches, difference-based patches etc.) have been useful in many contexts to reduce data movement. In this thesis, I study the use of such techniques to reduce data movement in the context of current and future HPC environments. I trade

off computation to reduce data volume, for faster completion of I/O operations. I identify three key data movement areas in HPC, intra-process, inter-process and inter-application data movement and investigate the impacts of various compression techniques on the data associated with each of these areas. To be specific, I introduce a compression-based paging system for HPC memory and demonstrate up to 78% capacity improvement with minimal runtime overhead (4%). Next, I propose and demonstrate a novel two-level diff-based approach that can reduce inter-process data movement by up to 99% although with potentially large runtime overhead. Finally, I reduce inter-application data movement by up to 90% using checkpoint/restart-based fault tolerance protocol as a case study. By doing so, I show that checkpoint data compression can improve application runtime efficiency by more than 50% and reduce energy expenditure by up to 90%.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

High-performance computing (HPC) systems have become an essential tool for science and engineering research. Modeling large complex physical processes enable scientists to gather valuable insights and to extend scientific knowledge. For example, some popular applications across different domains that use high performance computing include: climate and weather simulation, stellar atmosphere study, combustion/turbulence simulations, molecular dynamics simulations, quantum chemistry, material science and seismology [28].

These long running scientific applications divide the problems they are solving, across up to millions of processes. Each problem may require hundreds of thousands of node hours to complete. With more powerful computing capability, an application would complete faster and would be able to solve a problem at a higher resolution. Therefore, enabling research to be conducted much faster, providing investigators with more confident predictions.

The execution of an application can be broadly divided into three phases: 1)

computation, 2) data movement, and 3)storage[1]. During computation, an application executes the instructions in the processors. It is the key element in solving large scale problems. As a result, computation is one of the key areas of improvement to further increase these systems' capabilities. Data movement, on the other hand, includes fetching data to the processor through the memory hierarchy, exchanging messages among the processes, moving data for persistent storage and communicating with other applications such as, fault tolerance, data analytics, visualization, etc.

Today's HPC systems comprise of millions of compute cores [6, 127]. The next major advancement for HPC systems would be the ability to execute $10^{18}$ floating point operations per second, a so called exaflop system. To reach this goal, we expect future exascale class computing systems to have orders of magnitude more cores than our current systems [67]. However, all components in the system are not being improved at the same rate. For example, analysis of the fastest computers from the top500 list shows that: compute capacity growth is 2 times faster than per core memory capacity and bandwidth [127]; for storage, computational capacity grew 1000 times faster than storage bandwidth. Similar trends have been observed for network as well since the number of network interconnects remained fairly stable over time[68, 120]. Thus, data movement in HPC is increasingly becoming a bottleneck for application's runtime performance [7, 12, 67].

In this study, I investigate different areas in HPC system where data movement takes place and try to trade off computation by reducing the volume of data through applying compression-based techniques. My hypothesis is that reduced time spent in data movement can hide the added (de)/compression overhead and thereby improve an application's overall runtime performance.

[1]HPC systems typically use parallel file systems for storage. In this work, we will study the data movement associated with storage but will not dive deeply into studying storage.

## 1.1 Data movement in HPC systems

In an HPC system, the smallest processing units are the compute cores. When a parallel or distributed application is solving a problem, it typically breaks the problem into individual processes. The processes often are run in a symmetric multi-processing (SMP) mode that allows a core to run multiple processes or threads. In modern multi-core systems, each processor can have 2-64 cores and generally 2-8 processors are contained in a node. While computation takes place in the compute cores, data is moved between the cache hierarchy and memory.

In HPC, these processes and threads generally communicate across nodes by exchanging messages through the Message Passing Interface (MPI). Together, a large number of communicating nodes can solve a much larger problem and can finish the application code much faster than traditional computing systems.

Due to the limited availability and large procurement and operating costs of HPC systems, traditional HPC systems follow a shared usage model where multiple applications are running simultaneously to maximize resource usage. Some of these applications provide important services such as fault tolerance, visualization etc. In order to provide these services, large volumes of data are being transferred between these applications.

We can broadly classify HPC data movement into three categories:

- Intra-process data movement.
- Inter-process data movement.
- Inter-application data movement.

Figure 1.1 illustrates a general overview of HPC data movement and the associated high level hardware architectures. Due to the increasing performance

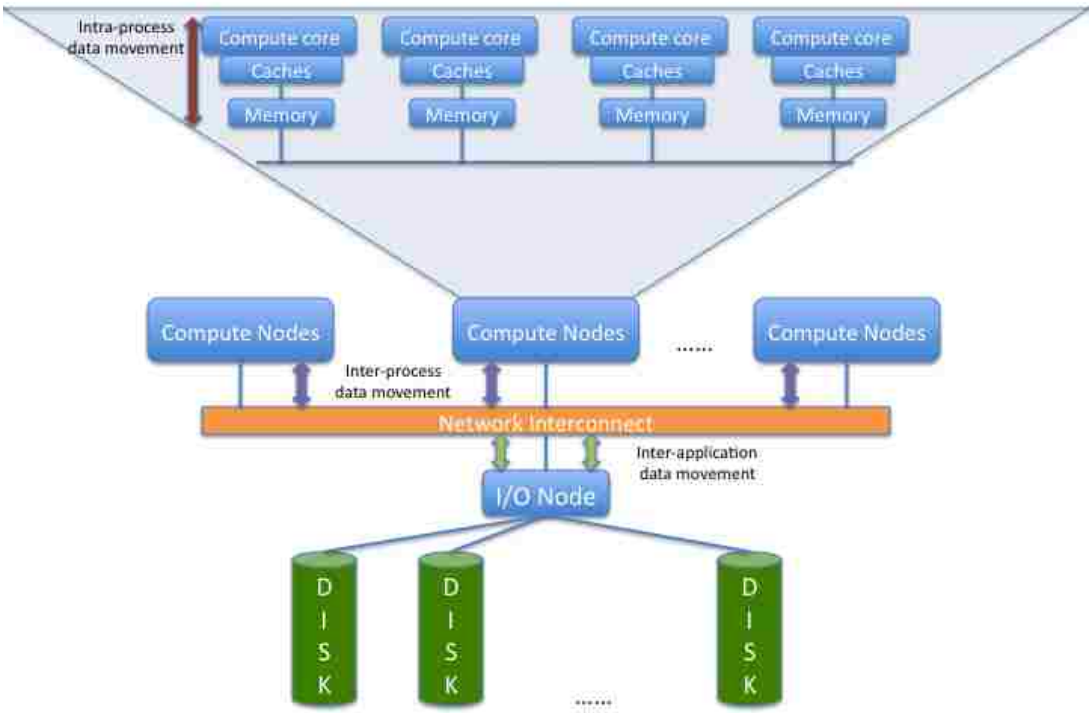



Figure 1.1: Data movement paths for HPC system. Double ended arrows represent the data movement paths studied in this thesis.

gap between processing and the rest of the systems, CPU cycles are wasted while waiting for data movement in the memory hierarchy, the disks and the network. These bottlenecks decrease an application's time to solution performance and as a result of that, we cannot utilize the full potential of the system.

### 1.1.1 Intra-process data movement

Data intensive HPC applications can process large external datasets and often require large working sets that can exceed memory capacity. For example, SCEC

Broadband platform's [83] per process working set is more than 1GB during 75% of it's total runtime [62]. In comparison, the fastest two systems in the TOP500 [127] list both have roughly 0.3GB memory per core. As a result, it is evident that some of the memory bound applications wouldn't be able to utilize top HPC systems to their full potential.

The decrease in memory capacity per core, also creates a constraint for simulation-time data analysis and data visualization within the computing node (in-situ analysis). Therefore, a larger memory capacity can avoid the cost of moving data to separate node or storage for co-analysis.

We can address the memory capacity problem by using a hardware-based approach. We can add more memory hardware to increase the per core memory capacity as well as can add more nodes to improve overall capacity. But the hardware-based solution increases both the procurement cost and the maintenance cost due to the increased energy requirements [47, 67, 116]. Scaling up horizontally, by adding more nodes, will also increase data movement, but it will as well have increased additional scaling complexities such as increased failure rates etc.

Hybrid software-hardware based approaches try to address this problem by adding locally attached non-volatile memory storage [134, 85] or new high-density memory technologies [60, 51]. But one of the side effect to that approach is the increase in data movement between cache hierarchy, DRAM and node local storage. As both the runtime and energy cost of data movement is a function of the distance between DRAM and backing store [67, 116], I propose that a compressed cache in the main memory can solve this problem.

Compression-based approaches have also been studied for HPC memory pages. For example, Levy et al. studied memory page similarities for recovering from memory faults [76, 75]. Studies have also reviewed memory page similarities to reduce memory footprint [16, 15]. Outside of the HPC context, compression has been explored to reduce paging to non-volatile storage by keeping a set of compressed cache in the main memory using both hardware- [128, 50, 40] and software-based compression [139, 30, 140, 26, 66, 131, 115]. However, due to the absence of node local storage in traditional HPC systems, paging to node local storage and a compression-based paging scheme has not been studied in HPC context.

With regard to intra-process data movement, I have studied and compared runtime performances of an SSD-based paging scheme and a compressed paging scheme in the HPC context using a simulation-based approach. I show that using a compression-based approach to improve memory capacity will reduce the requirement of having additional hardware as this approach can provide similar capacity with minimal overhead.

### 1.1.2 Inter-process data movement

HPC applications divide the problem to be solved into smaller sub-problems and to distribute them across nodes. These sub-problems often require communication with other sub-problems because the intermediate results of one sub-problem may depend on the others. These dependencies are resolved either by shared memory access (intra-node) or through the interconnection network (inter-node).

Application processes that exchange information with other processes on a different node, generally use a message passing interface (MPI) [138] to communicate. These processes exchange large amounts of application data via point-to-point and/or group (collective) operations. As processing capabilities and job sizes

(process counts) increase, then typically so does inter-process congestion [68, 120]. Inter-process congestion can also come from inter-application communication saturating the network since different application jobs can run simultaneously in the typical space-shared HPC job scheduling model [13, 14]. Without proper control on congestion, application performance can be severely degraded [44, 48].

Improvement in network performance is lagging from the growth in HPC systems, thereby resulting in a decreased availability of per process network bandwidth. As a result, network bandwidth has become a precious commodity and both intra- and inter-application network contention and congestion have become important problems to understand and mitigate [38, 39, 42, 43, 55, 64].

Different approaches have been studied to reduce congestion and improve network bandwidth. One approach is to distribute the application process across the system to reduce some congestion [120]. For a job-sharing system, assigning computation heavy processes with communication heavy processes also improves application performance [68]. Another approach is to apply data reduction techniques to reduce volume of data that is exchanged by application processes.

In order to reduce inter-process data movement, I studied the similarities among the communication messages exchanged by application processes in HPC. The messages are generally exchanged using the Message Passing Interface (MPI) [138] which is the most common framework for inter-process communication. Previous studies have demonstrated performance improvements by applying compression on MPI messages [38, 39, 42, 43, 64]. However, none of these studies have quantified the message similarities among MPI message payloads. I show that my two-level diff-based approach can reduce message data volume by more than 90%.

### 1.1.3 Inter-application data movement

Application services such as, for example, *fault tolerance protocols* [87, 107, 112], *data analytics and visualization* [74, 81], move large volumes of data from compute cores to stable storage. These are basically memory snapshots of each of these processes that are moved to stable storage (checkpoints) or data analysis nodes (analytics/visualization). Due to the large volume of data and slower I/O, applications take a large portion of time waiting for these data transfers to be completed. To illustrate the problem, I will use the example of checkpoint/restart-based fault tolerance protocol as a proxy for all application services.

#### Data movement due to checkpoint/restart

Checkpoint/restart is currently the most common approach to handle faults in HPC systems. In checkpoint/restart, each application process periodically saves its state, a *checkpoint*, to a persistent storage facility [33]. Checkpoints can be taken in a coordinated or uncoordinated fashion or a hybrid of both ways. Currently the most widely-used mechanism for checkpointing is coordinated checkpointing.

Coordinated checkpointing works by having all processes take checkpoints at the same time, thus creating huge contention for network and I/O resources. During the time of taking a checkpoint, each process must wait and cannot move forward with application code until all other processes have completed committing their checkpoints. With an increase in system size, application's runtime efficiency is expected to decrease for the following reasons:-

- System mean time between failures (MTBF) will decrease. MTBF is inversely proportional to the total number of cores in the system [45, 109] assuming

every core has uniform failure rate. Hence, larger systems will experience more frequent failures and an optimal checkpoint algorithm will take checkpoints more frequently to reduce lost work due to failures.

- More processes will increase the overall checkpoint data volume as well as creating more contention for network and I/O.

Based on current projections for future exascale systems, an application could spend more than half of its wall clock time performing checkpoint/restart [35, 95].

A number of different checkpoint optimization strategies have been studied to reduce checkpoint data movement. These strategies include *memory exclusion* [100], *incremental checkpointing* [18, 36, 34, 102] and *checkpoint compression* [78, 89, 101, 103, 59]. Plank et al. demonstrated that checkpoint compression is not a viable approach [101, 103]; but, based on today's CPU/IO performance numbers, I demonstrate in this study the viability of checkpoint compression. I also compare this approach against other software/hardware-based checkpoint data reduction studies [102, 59], and study the impacts of different compression parameters in application runtime. Finally, I show that, checkpoint data movement reduction not only improves application's runtime performance but also improves application's energy performance.

## 1.2 Contributions

In this thesis, I examine the trade-off of computation and data movement in the context of highly scalable HPC systems. More specifically, I examine the viability of utilizing excess computation on current systems to reduce the volume of data movement. My hypothesis is that compression-based techniques can be effectively

used to reduce data movement in different areas in HPCs and can improve application runtime performance. I evaluate this hypothesis in the following ways:

- I propose a compression-based paging scheme for HPC application memory structure. Using a trace driven simulation-based approach for the real HPC workload, I have demonstrated that such an approach can effectively improve HPC application per-process memory capacity by up to 78% while keeping the overhead low (under 4%).

- I study intra- and inter-message similarities among MPI-based HPC applications. I demonstrate a two-level diff based approach to exploit these similarities to reduce overall network data volume by up to 99%, and to demonstrate an upper limit of the runtime overhead of our process.

- I develop a model with which to study the viability for checkpoint compression for future systems. Using an extended performance model, I evaluate the performance benefit of checkpoint compression for current systems and demonstrate application runtime efficiency improvements of more than 50% for future systems. I develop and validate an energy performance model for checkpoint compression and demonstrate that compression can reduce energy expenditure by up to 90%.

  I compare compression-based checkpoint optimization against other software/hardware-based checkpoint optimizations. Finally, I study the impact of different parameters in the checkpointing process to dictate future directions for improvements in checkpoint compression.

## 1.3 Document organization

The remainder of the document is organized as follows. In Chapter 2, I present background information and related research in the three data movement areas that are covered in this study. In Chapter 3, I study intra-process data movement reduction by improving the perceived per-core memory capacity. I use a simulation-based approach to introduce paging for HPC workloads and compare its performance against a compression-based memory paging solution. In Chapter 4, I study different ways to reduce inter-process data movement. Here I seek to identify the similarities that exists among inter-process messages using a diff-based approach and using simulation to study the overheads of my approach to reduce network congestion. In Chapter 5, I study inter-application data movement reduction by focusing on checkpoint/restart optimization that serve as a proxy for all inter-application data movement. I study the viability of checkpoint compression and demonstrate its application to improve overall application runtime performance. Finally, I conclude the dissertation with a discussion about future direction.

# Chapter 2

# Related Work

This chapter describes previous research that is related to the contributions presented in this document and summarize how the contributions of this document are novel and distinct from currently existing research. Section 2.1 summarizes existing research on intra-process data movement reduction. Section 2.2 describes existing research on inter-process data movement. Finally, Section 2.3 provides an overview of prior research on inter-application data movement reduction using checkpoint/restart optimization as a proxy for all inter-application data movement.

## 2.1 Intra-process data movement

Recent trends show that the memory capacity per core is decreasing [94, 67]. This limits our ability to solve large problem sizes without having to scale horizontally by adding more nodes. Adding more nodes will increase data movement, energy expenditure and procurement costs. Projections show that the cost of moving data from memory is two orders of magnitudes higher than the cost of computing a double-precision register-to-register floating point operation [67]. Since, computation is

getting cheaper than data movement, in this work, we want to investigate the use of compression to trade off computation and improve perceived memory capacity.

Outside of the HPC domain, the application of compression over L1/L2 caches and main memory pages has been studied extensively in the efforts to expand perceived memory capacity. I will discuss some recent approaches in this area and also different approaches in HPC to address the memory capacity problem.

### 2.1.1 Hardware-based compression between cache hierarchy and DRAM

Yang et al. [141] studied compression between the L1 and L2 cache lines. In their approach, they learned that each L1 cache line could store either an uncompressed cache or two compressed caches depending on the fact that a cache line must be compressed to at least 50% of its original size to have effective results. Their simulation based approach showed that compression could result in up to 36% savings in L1 miss rates. This leads to a reduction of up to 48% off chip data movement.

In the selective compressed memory system (SCMS) [73], it stored two adjacent compressed cache lines or an uncompressed cache in each cache line. SCMS managed the L1 cache as a regular first level cache but compressed L2 caches and DRAM pages. If two compressed cache lines can fit into a cache line, they are stored compressed; otherwise, they are stored uncompressed. SCMS also had an additional buffer between L1 and L2 caches. They demonstrated that their approach can reduce the miss ratio by up to 35%.

Alamelden and Wood [4] used a similar approach as Lee et al. [73]. In their proposed system design, L1 caches hold uncompressed data and L2 caches dynamically

decides whether to compress or not, and whether they can hold either 4 compressed cache lines or 8 uncompressed cache lines. Their 'Frequent Pattern Compression (FPC)' algorithm had a faster decompression rate, which was used by the L2 cache to determine whether compression can avoid a cache miss, given the additional decompression overhead. They demonstrated that the proposed system's performance improvement was up to 17% for some workloads while the performance penalty was never over 0.4% for the applications they tested.

After IBM's memory expansion technology (MXT) [128] was introduced with hardware based real time de(compression) support, some related research was concentrated on MXT-based implementation of main memory compression [40, 50]. Hallnor and Reinhardt used LZSS compression from the MXT system. They used a pool to store compressed cache lines, and allocate variable amounts of memory to compressed caches, depending on their compression ratio. They simulated benchmarks to demonstrate from -5% to 19% performance improvement for SPEC2000 benchmarks. It is important to note that due to the compression in MXT, the number of addressable memory is variable, and changes dynamically based on the compressibility of the pages in memory. The one-to-one mapping between physical page frame and virtual memory page is no longer valid and they need to be independently managed by the OS.

Although hardware-based approaches can provide better support for faster compression, they seem to have limited acceptance as they require changes to both hardware and software systems. Software-based compression on the other hand can be made readily available for existing systems and can be turned on or off selectively.

### 2.1.2 Software-based compression between DRAM and disk pages

The idea of software-based main memory page compression began in the early 1990s by Wilson [139] and Douglis [30]. As CPUs became increasingly faster than disks, compression speeds were also improving. So it became increasingly attractive to keep pages in memory in compressed form to avoid paging out to disks.

In their original proposal, memory contained a new level called compressed region, where pages were stored in their compressed form before paging them out to disk [30, 139]. So, if a page fault could be satisfied from the compressed region, that eliminated an expansive disk access. A compressed page also reduced the data to be transferred between main memory and disk when swapped out. Some other studies followed this design [26, 140]. In these designs, a page was either in the compressed region or in the uncompressed region, but not in both. A different approach which violated this property was proposed by Kjelso et al. [66].

Tuduce et al. proposed an adaptive main memory compression approach that dynamically managed memory fragmentation and was able to improve performance by up to 55% [131]. Memzip [115] provided a compressed memory architecture that targeted the energy, bandwidth and reliability of a system instead of the traditional capacity metric. The additional space that was due to compression was used for error correction or energy efficient data encoding, thereby resulting in up to 45% performance improvement and up to 57% memory energy reduction.

Another approach, data deduplication, keeps a single page for all duplicate memory pages, thereby reducing the memory footprint of the application. The difference engine [49] calculated hashes of page contents to identify identical pages in Xen Virtual machine systems (VM). Applying data deduplication based on these hash values, they were able to demonstrate from 65% (heterogeneous workload) to 90%

(homogeneous) space savings across VMs.

All of the related hardware-based and software-based memory compression research that I summarized in this section focused on commodity systems and applications, and not in the HPC context. Unlike commodity systems, HPC systems do not have locally attached storage with the compute cores and therefore do not support paging to local storage. For our work, I introduce paging to HPC and study the impact of compression-based paging in the HPC context.

### 2.1.3 Approaches to improve memory capacity in HPC

In HPC context, The PSMalloc [16] and its extension SBLLMalloc [15] are both user level libraries that use a data deduplication approach similar to the difference engine to increase memory capacity by identifying identical memory blocks. The PSMalloc exploits data similarity across MPI tasks while the SBLLMalloc extends it with the ability to identify zero pages across and within tasks. The recent work of Levy et al. on memory content similarity of HPC applications shows that significant similarities exist in HPC memory content [76]. We believe that compression can exploit these similarities and can reduce data movement for HPC applications.

Researchers in HPC have also investigated recent advancements into new high capacity, higher bandwidth memory hardware and sought to understand how they can improve memory system performance [51, 60, 136]. Hybrid memory cube (HMC), phase change memory (PCM), 3d stacked memory are some of the modern technology improvements that are being explored to provide lower effective latency, lower power, higher bandwidth and overall lower cost.

Hammond et al. studied the the impact of memory policies on a multi-level memory structure that included newer and faster memory technology [51]. They used

a simulation environment that is very similar to what we used for our experiments, and demonstrated that the addition policy (the decision to add a page to the fast memory) impacts application performance more than replacement policies. Jayaraj et al. studied memory access patterns and attempted to identify high density memory access areas in memory [60]. They proposed to keep high density access areas in faster memories for example, HCM and demonstrated that, although these faster memory hardwares are expensive (3 times more than DRAM), the overall relative costs can be reduced by using a combination of different kinds of memory hardware.

### 2.1.4 My research contribution

For this work, we sought to introduce software-based compressed-paging for HPC memory. Current HPC memory capacity is limited by the DRAM size since a typical compute node has no local non-volatile storage for swap space. We study software-based compression due to it's flexibility compared to the hardware-based approaches. Software-based approaches are also less expansive than hardware-based approaches [60] and can be applied selectively. In this study, we first studied the runtime performance impact of introducing SSD-based paging to the HPC context. Next, we compared SSD-based paging against our proposed compression-based paging and demonstrated reduced runtime overhead while increasing the perceived memory capacity. To the best of our knowledge, paging in general and a compression-based paging scheme has not been studied in a HPC context.

## 2.2 Inter-process data movement

HPC applications divide the problem space among smaller sub-problems, and assign processes to solve these sub-problems. Processes that are solving these sub-problems,

communicate with each other through Message Passing Interface (MPI) [138] over the network to resolve their inter-dependencies. MPI is the message passing library most widely used to provide messaging in parallel applications. MPI provides a complete interface for point-to-point and collective operations, synchronizations and I/O operations.

Due to the lack of improvement in networking subsystems, as compared to processing, the contention for resources and network congestion results in a bottleneck. For example, scientific applications that have distinct computation and communication phases have high sensitivity to network bandwidth. These applications have periods of high network utilization followed by idle networks thus requiring a high peak network bandwidth. The relative slow growth of network bandwidth therefore creates a bottleneck for the application and slows down the message rate. For this work, we focus on MPI message workloads and try to identify inter and intra-message similarities. Our focus is to improve the perceived aggregate network bandwidth by reducing the network data volume. In this section, I will summarize the background and related works in network contention/congestion and compression-based approaches that have been used in MPI messages to reduce the inter-process data movement.

### 2.2.1 Network Contention/Congestion in HPC Applications

Network congestion is a primary cause of performance degradation for communication heavy HPC applications [55, 42, 43, 64, 38, 39]. In this context, network congestion can arise from self-interference, when an application's own traffic is contending for limited network resources, or from cross-interference, when different applications and services contend for the network resources.

**Network contention and congestion due to an applications own processes**

In many core systems, it has been shown that contention for network interfaces and congestion from the reduced per-core network bandwidth can significantly degrade application performance [68, 120]. Soryani et al. [120] have shown the impact on communication performance of sending medium versus large message sizes . They showed that mapping communicating processes that send or receive larger messages within a node can improve performance. In addition, assigning processes that send large inter-node messages to different sockets can also improve performance. Studying both inter- or intra-node communications, the authors identified a threshold for optimal network utilization and proposed a process mapping scheme to distribute network activity in an attempt to avoid contention, thereby improving message rates by up to 19.6%.

Koop et al. [68] demonstrated performance degradation that was caused by network congestion by comparing the execution times of applications executed on shared nodes versus on dedicated nodes. By assigning communication heavy processes to multiple nodes compared to single nodes, while keeping the process-to-core assignment and total number of processes constant, the authors demonstrated the negative performance impact of network congestion and contention. They proposed to mix a workload by pairing computation heavy processes with communication heavy processes. Their results showed that by reducing network contention, runtime could be improved by 20%

**Network congestion due to other applications or application services**

Bhatele et al. [13] applied supervised learning algorithms to network components in order to identify which has the greatest impact on performance. For their test applications MILC and pF3D, they demonstrated that the average number of bytes

passing through the network is an important indicator of congestion and recommended a process mapping to reduce the average load per link. In a separate work, the same group observed that network performance variability that was induced by neighboring jobs can lead to significant performance variations for identical jobs [14]. They demonstrated that execution time of a communication heavy application can vary from 28% faster to 41% slower than the average observed performance due to interference from other jobs that were running nearby.

### 2.2.2 MPI Message Compression

The previously mentioned works primarily used different process mapping schemes to reduce network congestion and contention. Other works have explored compressing MPI messages to reduce network data volumes.

#### Compressing Application Messages

The potential benefits of message compression have been demonstrated in various MPI-based studies [42, 43, 64, 38, 39]. PACX-MPI compressed all transmitted messages uniformly (independent of type or content) and reduced the amount of data transmission by a factor of 3X [42]. MiMPI leveraged different compression algorithms and used message size thresholds to decide when to use compression [43]. They compared their results against IBM's multithreaded MPI implementation and demonstrated up to 20% improved message throughput.

cMPI applied a value-prediction based message scheme to MPI messages that calculated the difference between predicted and actual data values in messages and encoded that the difference using highly-compressible leading zero counts [64]. They tested cMPI with NAS parallel benchmarks and were able to provide from 3% to 10% speedup.

Lastly, CoMPI and its extension, adaptive CoMPI, selected the best message compression algorithm at runtimes based on data types [38, 39]. CoMPI also decided whether or not to compress a message using speedup or slowdown estimates. They were able to speed up two HPC benchmarks by 1.2 and 1.4 times respectively with CoMPI. One of the extensions of adaptive CoMPI also accounts for the MPI ranks to check whether the message will be sent over the network or within processes on the same node. It then used that information to decide whether to compress the message or not.

### 2.2.3 My research contribution

This work is distinct from previous compression-based approaches because we consider message payload similarity explicitly, both within a message and across messages. We apply a diff-based approach to identify message similarities and provide a novel two-phase diff application to reduce message volume by more than 90% in some cases. To the best of our knowledge, no other work has considered a similar approach.

## 2.3 Inter-application data movement

HPC applications also exchange large volumes of data with other applications that perform important services such as, for example, fault tolerance services, data visualization/analytic etc. Fault tolerance services can reduce the amount of lost work due to software/hardware failures. So, one common approach is to store a memory snapshot into a stable storage (checkpoint) [33], so that we can use that information to restore computation in the event of a failure. For large-scale applications, checkpoint data movement can lead to performance bottlenecks due to excessive data

volumes and contention for network and storage devices.

Data visualization and data analytic provide insights on the simulation results and sometimes intermediate results are verified to make sure that simulation is making progress towards the correct results. Typically, we take these results and then move them to another node for further data processing causing large volumes of data movement across the network.

For this work, I used the checkpoint/restart-based fault tolerance protocol as a proxy application for other inter-application data movements. Currently, the most widely used mechanism for checkpointing is coordinated checkpointing. Hence, this study is focused on coordinated checkpointing only.

### 2.3.1 Software-based checkpoint/restart data movement optimizations

Checkpoint/restart performance optimizations that target the checkpoint data movement challenge can be divided into two classes. The first class of optimizations attempts to hide or reduce (perceived) commit latencies without actually reducing the amount of checkpoint data. These strategies include:

- *diskless, remote and buddy checkpointing*: Diskless checkpoint/restart protocols [104] and remote checkpoint/restart protocols [24, 122, 142, 119] leverage the higher bandwidths available to the network or other storage media like RAM in order to mitigate the performance of slower storage media like magnetic disks. Additionally, remotely stored checkpoints allow systems to survive non-transient node failures.

  Plank et al. [104] proposed to keep additional processes that will store checkpoint parity data, while the application process will allocate a portion of its

memory to store the checkpoint. Parity data is calculated using bitwise XOR operation, so that in the event of a failure, it can be used to restore the checkpoint. Silva et al. [119] improved this approach by using application processes instead of additional processes to store a neighbor's checkpoint parity data. A similar approach was studied using double checkpointing [29]. They stored checkpoints in neighbor processes called 'buddy processes' based on the fact that in the event of a failure, the probability of the original process and the buddy process to fail simultaneously, is extremely small. However, these approaches created significant memory overhead for storing the checkpoints in memory.

- *multi-level checkpointing*: Multi-level checkpoint/restart protocols like SCR [87, 133] write checkpoints to RAM, flash storage, or a local disk on the compute nodes in addition to the parallel file system to improve checkpoint bandwidth. In terms of the performance and reliability, it was a trade off between getting the fastest access storage and robustness. For example, while local disks provide fastest access to transient storage, they were limited to failures that affects only a small part of the system.

- *checkpointing file systems*: Checkpoint-specific file systems like PLFS [11] leverage the patterns and characteristics specific to checkpoint data to optimize checkpoint data transfers to or from parallel file systems.

The second set of strategies reduce commit latencies by reducing checkpoint sizes. These latter strategies include:

- *memory exclusion*: Checkpoint/restart protocol optimizations based on memory exclusion leverage user-directives or other hints to exclude portions of process address spaces from checkpoints [100].

- *incremental checkpointing*: Checkpoint/restart protocols can use the operating system's memory page protection facilities to detect and save only those pages that have been updated between consecutive checkpoints [18, 19, 34, 79, 102, 98, 3]. Page hashing techniques can also be used to avoid checkpointing pages that have been written to but whose content has not changed [36].

- *checkpoint compression*: Various approaches for compressing checkpoints to improve checkpoint/restart protocol performance have been suggested. Li and Fuchs implemented a compiler-based checkpointing approach, which exploited compile time information to compress checkpoints [78]. They concluded that checkpoint compression is not a viable approach and demonstrated that a compression factor of over 100% was necessary to achieve any significant benefit due to high compression latencies.

  Plank and Li proposed in-memory checkpoints and stored the checkpoints after applying compression [101]. For their computational platform, compression was beneficial if a compression factor greater than 19.3% could be achieved. Because they used memory to compress and store checkpoints, the checkpoint bandwidth was much higher compared to compiler-based checkpoints [78].

  To reduce the incremental checkpoints even further, Plank et al. proposed *Differential compression* and applied compression on incremental checkpoints [103].

  Tanzima et al. have shown that similarities exist among checkpoint data from multiple processes. They leveraged this by concatenating checkpoints from different processes and then applying compression on them to reduce checkpoint data volumes [59]. Their approach called the *MCR Engine* leveraged semantic information from HDF5 data models.

  Nicolae [93] demonstrated that within a checkpoint, duplicate pages exist. He applied deduplication techniques to keep a single copy of duplicate pages in a

checkpoint, thereby reducing checkpoint sizes.

### 2.3.2 Hardware-based checkpoint/restart optimizations

Improved hardware technologies have been suggested as ways to optimize CR protocol performance. Moshovos and Kostopoulos proposed the use of hardware-based compressors for compressing checkpoints [89]. More recently, researchers have proposed the use of solid state storage devices (SSDs) for efficient local checkpointing [63] or even in multi-level solutions [87]. At the cost of greater financial expense and other potential issues like flash wear-out, SSDs provide higher storage bandwidth than traditional stable storage devices such as magnetic disks.

### 2.3.3 My research contribution

This work focuses on the use of software-based compressors for checkpoint compression. Given recent advances in processor technologies, this study demonstrates that, since processing speeds have increased at a faster rate than disk and network bandwidth, data compression enables us to trade faster CPU workloads for slower disk and network bandwidth.

Using a model, this study demonstrates checkpoint compression's viability and provides a viability breakeven point metric, which can be used to decide when to use checkpoint compression. This study also investigated the impact of different compression parameters on application runtime performance and provides a comparison of this approach against other software/hardware-based checkpoint data optimizations. Finally, this study demonstrates that, given the runtime performance improvement, checkpoint compression can improve application energy performance as well.

# Chapter 3

# Intra-process data movement

## 3.1 Introduction

In this chapter, I propose the introduction of paging to the HPC architecture to improve HPC application memory requirements. Specifically, I studied the impact of paging to SSD-based node local storage and also introduced a compressed paging scheme for HPC memory pages. For current HPC systems, where processing nodes still do not have any non-volatile storage attached to them, a compression-based page cache would work as the last level cache for pages.

For this study, I followed a trace driven simulation-based approach. I demonstrate that by using a compression-based in-memory cache for pages, we can increase the perceived memory capacity with minimal runtime overhead and can reduce data-movement that may incur due to paging to node-local storage or moving data for visualization or analytics. In the remainder of the chapter, I first describe the problem in and a more detailed overview of our study in Section 3.2. Next, I describe our testing methodology in Section 3.3. Then I describe how I modeled the different

scenarios into the simulator and the associated parameters values. Following that, I describe the results of our experiments in Section 3.4 and finish the chapter by summarizing our contributions for this study.

## 3.2 Increasing memory capacity by using compressed pages

Recent trends show that the memory capacity per core is decreasing. With the recent advances in many core systems, there will be even less per core memory capacity for future systems [94, 67]. Analysis of the top 500 computers demonstrate that compute capacity growth is 2x faster than memory and memory bandwidth [127]. On most systems, this ratio is currently below 0.1GB per giga FLOP and approaching 0.01; with memory capacity per core expected to drop by 30% every two years [23].

The decrease in memory capacity per core, introduces some key challenges. First of all, for data intensive applications, the decrease in memory capacity is a constraint. These type of applications process large external datasets and often require large working sets that can exceed memory capacity [23]. In addition, more memory capacity enables users to solve problems with higher resolution and improved accuracy. Secondly, simulation-time data analysis and data visualization requires either a large memory footprint (in-situ analysis) or high speed network and I/O (co-analysis). Co-analysis moves intermediate simulation results to a dedicated resource therefore increases data movement and associated costs. Since, the energy cost of processing is getting cheaper than moving data across main-memory, the cost of data movement is a big concern for co-analysis [67]. In-situ analysis doesn't have the data movement cost, but requires large memory capacity.

There are different approaches that can solve this problem. The most straight

forward approach would be to add more hardware to increase the capacity. But that would result in additional procurement costs. Higher capacity memory hardware has also been explored such as, for example, 3D stacked or parallel stacked memory. But that too is expensive. Jayaraj et al [60] have compared the relative cost per bit for different types of memory technologies and showed that these high capacity memory hardware can cost 2-5 times more than the the current DRAM systems. Adding hardware also increases the energy budget which already is very high.

Hybrid software-hardware approaches such as, for example, DI-MMAP[134] provide a runtime that expands an application's address space to locally attached flash storage. Meswani et al.[85] mapped stacked DRAM and conventional DRAM to the same physical address space and developed a hybrid memory management system. But these approaches would still require additional hardware.

It is evident from these recent studies that a hybrid memory architecture is the future solution to solve the capacity problem. So, we seek the answer to the question, what happens if, instead of adding new hardware, we compress and store evicted pages in memory. Basically, we want to answer the following questions-

- **How does a compressed paging mechanism increase HPC per core memory capacity?** We propose a paging mechanism for HPC systems. A common perception for HPC workload is that paging hurts application performance. But due to the decrease in memory capacity, we want to increase the perceived memory capacity by using a compressed page mechanism. Our proposed compressed paging mechanism will keep evicted pages in main memory but within a compressed region. This strategy thus frees up more space for regular pages and effectively increases memory capacity without adding additional hardware.

- **If HPC systems have node local storage for paging how would that**

**impact application performance?** It is projected that future HPC systems will have node local storage for in-situ analysis and can reduce data movement costs[126]. We wish to study the impact of paging if we have on node local NVM storage such as SSD. Our proposed approach is to study the impact of application runtime if we page out to node local NVM storage.

- **Can a compression-based paging mechanism improve the runtime compared to a SSD-based paging mechanism?** Finally, we propose to compare the performance of our compressed paging mechanism to the resulting case when we page to node local SSD storage.

The compressed-memory system presented here follows the design proposed by Douglis et al. [30]. It divides the main memory into an uncompressed region that stores uncompressed pages and a compressed region that holds pages in compressed form. This study differentiates itself from the previous work, due to the nature of the workload being considered. In this work, we studied specifically the impact of paging on the application runtime and the perceived memory capacity benefits for HPC workloads. To the best of our knowledge, this has never been studied in HPC context. On a common system, when the amount of physical memory is less than what an application requires, the OS swaps out some pages to make space for other pages that the application needs. On a compressed-memory system, when an application's working set exceeds the uncompressed region, the OS compresses the pages that haven't been accessed for the longest time and stores them in the compressed region.

## 3.3 Methodology

For this study, I used a trace-driven simulation-based approach. Simulation is a common methodology for conducting design space exploration. It enables researchers to simulate current hardware as well as future hardware which is not yet available. Simulation also often is easier to implement and quicker to deliver results. In HPC, where the majority of computing resources are available only in research labs, it is nearly impossible to make changes to current systems. Simulators provide a safe environment to propose architectural/software/hardware changes as well as the ability to verify our hypothesis. As a result, simulation is a very popular approach for conducting design space exploration.

There are many simulators that enable researchers to investigate the memory system performance of applications [97, 106, 110, 123, 132]. Memory system simulators fall into two main categories: trace-driven or execution-driven. Trace-driven memory simulators use memory address traces as input to simulate application performance but rely on existing tools to collect memory address traces. Execution driven simulators are fed functional/performance models to simulate application performance. Since the address traces are available for reuse, trace driven simulators are popular to simulate different memory system architectures [132].

The methodology for this work is comprised of three principle components:

1. **Memory trace collection:** I instrument application runs and collect traces.
2. **Trace processing for simulation input:** I prepare and format the collected traces according to the specifications of the simulator input.
3. **Simulation-based evaluations of page compression on application runtime:** I configure the simulator for various scenarios and simulate the

traces. After each run, the simulator outputs average read/write latency and execution time for the applications under those simulated environments.

Figure 3.1 provides a high-level view of our approach. What follows here is a detailed description of these components as well as the benchmarks and applications used in this study.

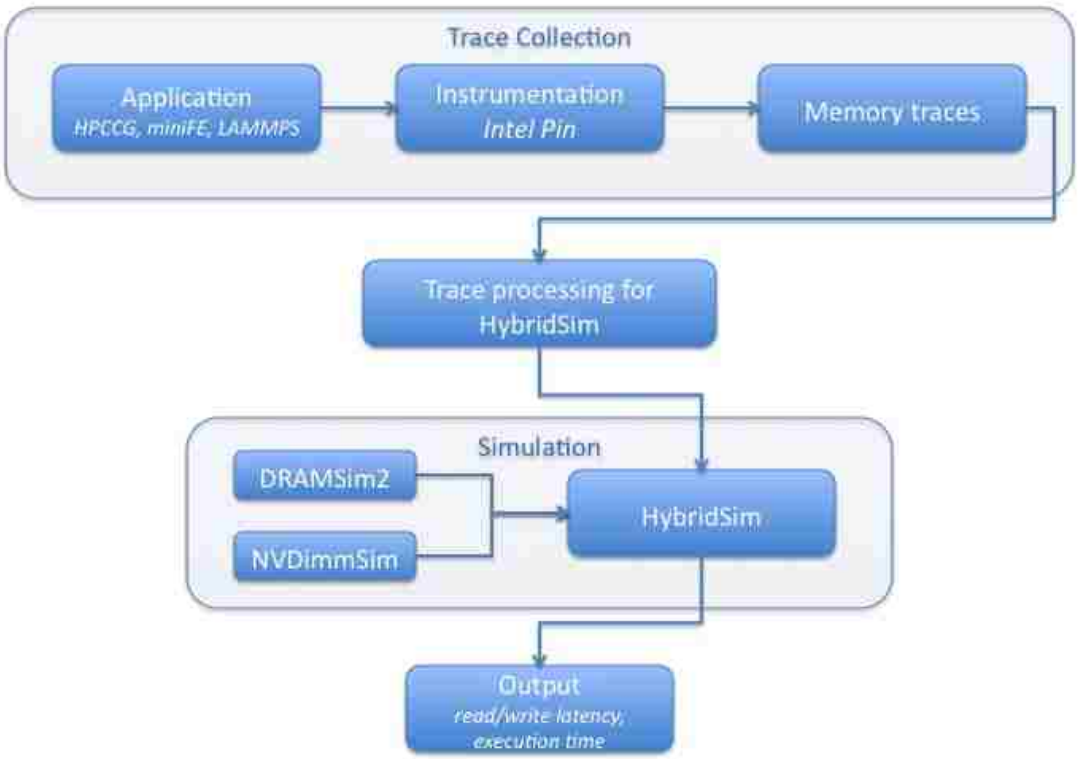



Figure 3.1: Overview of our experimental methodology

### 3.3.1 Memory trace collection

To collect the traces, I used Intel Pin tool [82]. Pin is a dynamic binary instrumentation platform that was developed and maintained by Intel for use on Intel x86 and

x86_64 platforms. Pin uses just in time (JIT) compilation techniques to instrument a running program. It means that the Pin tool generates and introduces instrumentation code into the program immediately before the first execution of that code by inserting extra code into the binary program. Pin is very effective at minimizing instrumentation overhead. For every memory operation traces, I collected three types of information-

- Type of the memory instructions: 1)instructions that have one or more source operands in memory (MEM read), 2)instructions whose destination operand is in memory (MEM write), and 3)instructions whose source and destination operands are in memory (MEM read and write)
- Logical address of the memory instruction.
- Cost to execute the memory instruction in terms of CPU cycles. In other words, how many cycles it would take a CPU to complete that particular memory operation.

**Our Test Applications**

To demonstrate the potential impact of paging and our compressed page cache scheme on HPC applications, we used two mini applications from the Mantevo Project and an actual large-scale application. These benchmarks and applications are described below.

- **The mini applications**

  We used two *mini-applications* or *mini apps* from the Mantevo Project [53], namely HPCCG version 1.0, miniFE version 2.0. These miniapps are implicit finite element mini apps. HPCCG is a conjugate gradient benchmark code

for a 3D chimney domain that can run on an arbitrary number of processors. This code generates a 27-point finite difference matrix with a user-prescribed sub-block size on each processor. miniFE mimics the finite element generation assembly and solution for an unstructured grid problem.

Mini apps are small, self-contained programs that embody essential performance characteristics of key applications. They are intended to mimic real application characteristics but are meant to be lightweight application proxies for the heavy-weight applications. We have compared and observed mini apps' for their performance similarity as compared to the actual application it represents in previous work as well [58].

- **A full application: LAMMPS** We use LAMMPS (the Large-scale Atomic/Molecular Massively Parallel Simulator) to serve as our full featured scientific application. LAMMPS [105, 111] is a classical molecular dynamics code developed at Sandia National Laboratories. LAMMPS also is a key simulation workload for the U.S. Department of Energy and is representative of many other molecular dynamics code. For our experiments, we used the embedded atom method (EAM) metallic solid input script, which is used by the Sequoia benchmark suite.

These applications were selected because they demonstrated a diverse set of memory access patterns based on previous studies. Voskuilen et al. have studied memory access densities of both HPCCG and miniFE and demonstrated their difference in memory access patterns [136]. They identified high density memory regions based on an application's memory accesses. From their study, it was determined that HPCCG has 60% memory access in a small set of high density memory regions (18 regions). On the other hand, miniFE has more uniform memory accesses and only 22% of

| Problem size | Problem dimension | Working set size (Bytes) | | |
|---|---|---|---|---|
| | | HPCCG | miniFE | LAMMPS |
| 8x | 16 x 16 x 16 | 5414912 | 5308416 | 8552448 |
| 4x | 8 x 16 x 16 | 4612096 | 4517888 | 6193152 |
| 2x | 8 x 8 x 16 | 4272128 | 4423680 | 4811776 |
| 1x | 8 x 8 x 8 | 4202496 | 4239360 | 4264960 |

Table 3.1: Problem sizes and their associated working set sizes for our test applications.

memory accesses took place in high density memory regions (5 regions). Jayaraj et al. have demonstrated the difference in memory access patterns between miniFE and LAMMPS and demonstrated LAMMPS' irregular memory access patterns [60]

For our experiments, HPCCG, miniFE and LAMMPS solved a problem with dimension 8x8x8 (1x). We collected traces for each of these applications while doubling the problem size until the problem size was 16x16x16 (8x). Our goal was to make sure that for all the test-cases, the application working set sizes remained between 4MB and 8MB. We believe this was a reasonable workload for our study and similar trace-driven simulation-based studies have also kept their memory limits similar to ours [51].

There were multiple reasons for keeping such small memory footprint. First of all, by keeping even the smallest working set larger than 4MB, we made sure that during our simulations, a 4MB main memory would force applications to page. In Table 3.1, we list the working set sizes for our test cases.

Second and most importantly, a larger memory footprint would create a very large trace and increased simulation time. At the beginning of the application trace collection process in this study, our main challenge was collecting representative

| | 1x | | 2x | | 4x | | 8x | |
|---|---|---|---|---|---|---|---|---|
| | lines | size | lines | size | lines | size | lines | size |
| HPCCG | 9336562 | 234 | 9928388 | 246 | 11160447 | 273 | 13708812 | 328 |
| miniFE | 13288842 | 336 | 17662300 | 448 | 26446484 | 672 | 43896602 | 1116 |
| LAMMPS | 17426219 | 414 | 32237082 | 773 | 61467478 | 1502 | 117814302 | 2791 |

Table 3.2: Trace file sizes in MB and the number of traces for each of our test applications as we increase the problem size

application traces that would finish simulation under a reasonable time frame. For a larger targeted working set size, the trace files began to be as large as tens of gigabytes with tens of billions of memory traces. Not only was this not feasible but also the simulation on these large trace files spanned over multiple days. At this scale, all the experiments required for this study could not be performed in full in a reasonable time frame.

To address this issue, I modified the application code to collect traces of 10 iterations throughout each application. Due to the extremely large size of the traces, this approach was applied before for purposes of memory trace collection [136] and demonstrated that for HPCCG, memory access patterns stayed nearly identical across iteration samples. Table 3.2 presents the the number of traces and the total size for the collected traces. Even with the limited number of the iteration, the traces grew more than billion lines and 2700MB in size. Since, our simulator is cycle accurate and replays the memory operations from the trace files, each simulation run took 3-6 hours.

### 3.3.2 Trace processing

Next, I processed the traces collected by Intel pin tool and converted each memory trace according to the input specification of the memory simulator. The simulator

requires each memory trace to be represented by a line in the trace file. For each line, it requires three values separated by spaces that makes up one memory trace: 1) clock cycle, 2) memory operation type and 3) memory address.

I should mention that, instructions whose source and destination operands are both in memory, my trace collection framework was outputting the same instruction twice in each line. Since the simulator did not support this behavior, I cleaned up the trace files and split each of these instructions into two simultaneous instructions, so that the first one served as a single instruction that was followed by the other. However, the number of such instructions in our collected traces were extremely small, and was under 0.001% of the total traces.

### 3.3.3 Simulation of traces

To simulate application memory performances, I used a trace driven memory simulator, `DRAMSim2` [110]. `DRAMSim2` is configured with two modules to complete our simulation environment- `HybridSim` [123] and `NVDIMMSim` [1]. These two modules enables `DRAMSim2` to simulate a hybrid system with a conventional DRAM memory and a non-volatile memory system. I chose these simulators because they are accurate and easy to use; they provide flexibility offering a large number of options and my desired functionalities and most importantly due to the availability of the source code.

`DRAMSim2`, configured with `HybridSim` and `NVDIMMSim` is becoming popular in similar memory system design exploration research and is being used in recent studies [137, 60, 51, 130] to simulate future hybrid memory systems.

`DRAMSim2` is a cycle accurate model of a DRAM memory controller, the DRAM

modules which comprise system storage, and the bus by which they communicate. Different objects within the memory device such as, for example, the ranks, banks, command queue and the memory controller can be modeled by the simulator. DRAMSim2 timing behavior has been compared and validated against verilog-based device models that have been published by hardware vendors [110].

`HybridSim` provides two ways to integrate nonvolatile memory. The first method uses it as backing storage just like a system that has paging enabled and the second method uses it directly within the memory controller and is considered as part of the main memory. `HybridSim` uses `DRAMSim2` to model DRAM and `NVDIMMSim` to model non-volatile memory. For our experiments, I configured and used `HybridSim` to simulate a system that uses a hybrid memory scheme. Then, I input our proposed compressed cache performance numbers to describe it in `HybridSim` and used the simulator to simulate application performance using the collected traces. I configured the memory capacities and the latency of each memory operation to simulate the behavior of the following three cases

1. The first case is when I have a memory system such that the working set fits entirely into DRAM. So we do not need any paging.

2. Next is the case when I enable paging. I configure the simulator such that I have an SSD as our backing storage for the DRAM and the OS pages to that SSD. Note that in this scenario the size of the SSD doesn't matter since we can assume that we have enough storage to page out to the SSD.

3. The final case is when I allocate a portion of DRAM as our backing storage for paging. In this scenario, I configured `HybridSim` such that the SSD is connected directly to the memory controller. For this case, I next limit the available DRAM capacity and change the SSD read/write latency measure-

ments to simulate the compressed DRAM read/write time. So basically, each read/write latency number would be equal to the latency number for DRAM and the latency number to compress/decompress a 4K page.

**Simulating a compressed page cache**

In order to simulate a compressed page cache system, I add the cost of compression/decompression to the read/write time of the pages, and decreased the size of the available memory as I increase the size of compressed page area. If I consider the average access time of a main memory system without compression $\overline{t_{mem}}$, then

$\overline{t_{mem}} = \frac{s_{page}}{bw_{mem}}$, where $s_{page}$ is the page size and $bw_{mem}$ is the main memory bandwidth.

On the other case when I enable compressed pages, I can measure the average access time with compressed pages $\overline{t_{mem\_compressed}}$ as:

$$\overline{t_{mem\_compressed}} = \frac{s_{page}}{r_{compress}} + \frac{s_{compressed\_page}}{bw_{mem}} \tag{3.1}$$

For our experiments, I configured the simulator to simulate a total memory capacity of 8MB. This is because the working set sizes of our three test applications, solving problems described in Section 3.3.1, were between 7 and 8MB to keep the traces reasonably sized. A larger memory footprint can generate traces exceptionally large and simulation could take days to complete. For this reason, similar studies have also kept memory footprints low[51].

| | test app | read latency | write latency | erase latency |
|---|---|---|---|---|
| SSD-based paging | all apps | 25$\mu$s | 200$\mu$s | 1500$\mu$s |
| compression-based paging | LAMMPS | 28$\mu$s | 42$\mu$s | 1ns |
| | miniFE | 92$\mu$s | 132$\mu$s | 1ns |
| | HPCCG | 100$\mu$s | 117$\mu$s | 1ns |

Table 3.3: Latency numbers as simulator input when simulating a compressed cache

Next, I modified the DRAM capacity to simulate three different cases. For the first case, when there is no paging, the DRAM has all 8MB of capacity. For the other two cases when I enable paging, I change the DRAM capacity to 4MB, 2MB and 1MB respectively. The rest of the memory is configured to simulate either SSD or a compressed memory system. All transfers between the SSD/compressed memory to the main memory is performed at page level granularity. The timing parameters for SSD are based on MLC flash numbers [61, 123] and DRAM timing parameters are based on Micron DRAM data-sheet[121]. All the devices are 8 bits wide.

Generally, SSD devices have three different timing parameters: read time, write time and erase time. On the other hand, DRAMs don't have any erase latency. So when we used SSD device parameters to simulate our compressed cache(which would be stored on DRAM), I changed the erase latency in the simulator to 1ns to make the erase time negligible[1]. In contrast an SSD device would have erase times between 1500-2000$\mu$s. Doing this enables us to simplify the simulation parameters without negatively affecting our simulation results. We present our simulation parameters for our experiments in Table 3.3 respectively. The remainder of our simulator parameters are identical throughout our experiments. The compression numbers we used to calculate Table 3.3 are empirically measured in our previous work [58].

[1]1ns was the minimum erase latency supported by the simulator.

## 3.4 Results

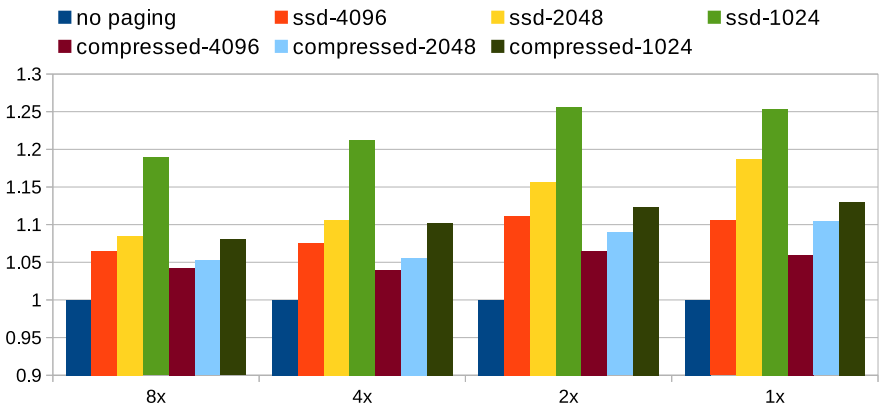
The goal of introducing a compressed-paging-based memory system is to leverage compression to improve the memory capacity with minimal runtime overhead. To demonstrate that process, in this section I first compare the impact of SSD based-paging and our proposed compression based- on different memory performance parameters, namely, average read/write latency. Next, I compare the runtimes when using these two different paging schemes against the case when we don't have paging that is the case when entire working set fits into memory.

### 3.4.1 Impact on memory read/write latency

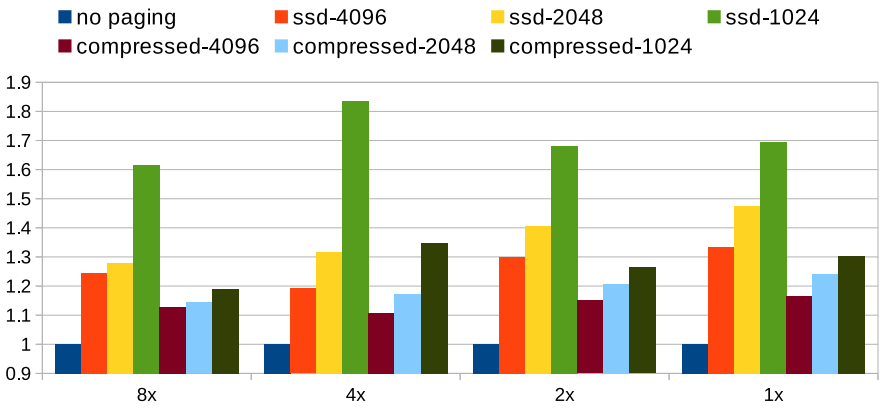
In this section, I compare the average read/write latency for our three test applications and also compare the cases when we used SSD-based paging scheme to our proposed compression-based paging scheme. Then we compare these read/write latencies against the read/write latency of the cases when we don't have any paging, i.e., when the entire working set fits into main memory. Figure 3.2(a)- Figure 3.4(b) presents our read/write latency comparison for our three test applications.

In these graphs, the x-axis represents the problem size where 2x means the problem size is twice as large as the problem size for 1x. Each bar represents the case of a memory subsystem being used for that particular experiment. SSD-2048 presents the case, when we have SSD-based paging with main memory size was limited to 2048MB. Similarly, compressed-2048 means our compression-based paging and the main memory size was limited to 2048KB. By limiting the main memory sizes, we can simulate different paging behavior: the smaller the main memory, the more number of pages were swapped out and was stored to the swapped space (SSD or compressed paging store). For all the test cases, the size of the swapped space (the paging storage, where an evicted page is stored) didn't really impact our

results, due to two reasons - 1) SSD capacity are traditionally much larger than DRAM, so we wouldn't run out of space for paging and 2) for compressed-paging, given the high compression ratio ( 90%) [56], the amount of cache needed is a very small fraction of the actual memory-size.



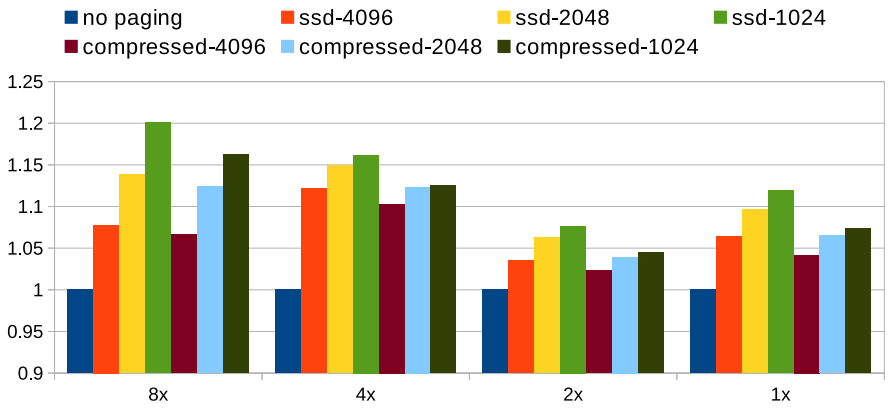

(a) Normalized average read latency for HPCCG



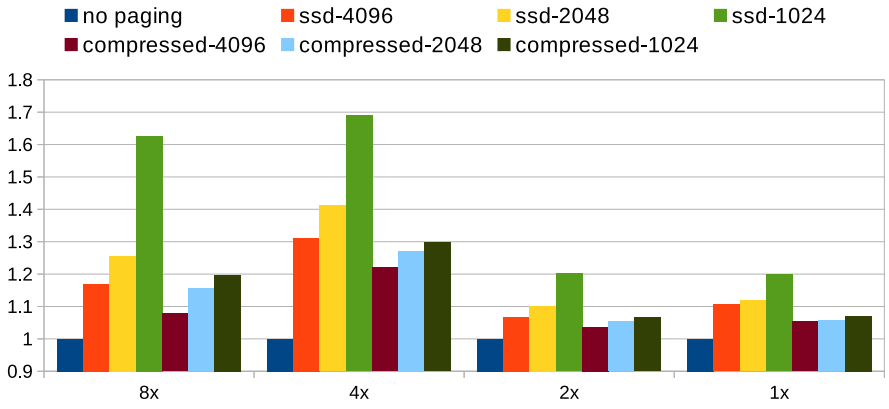

(b) Normalized average write latency for HPCCG

Figure 3.2: Normalized average read/write latency for HPCCG. x-axis represents the problem sizes and y-axis represents latency normalized against the no-paging case. Smaller means better.

From, Figure 3.2(a) through Figure 3.4(b) we can see that for all three test applications, the normalized performance penalty for writes are larger than that of




(a) Normalized average read latency for miniFE

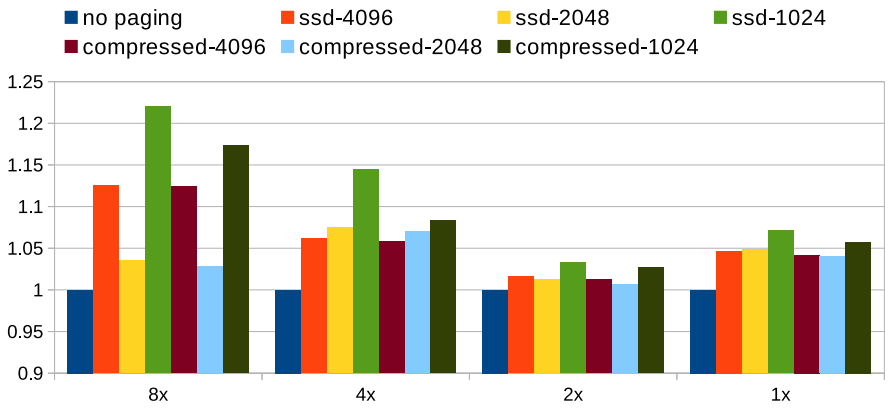



(b) Normalized average write latency for miniFE

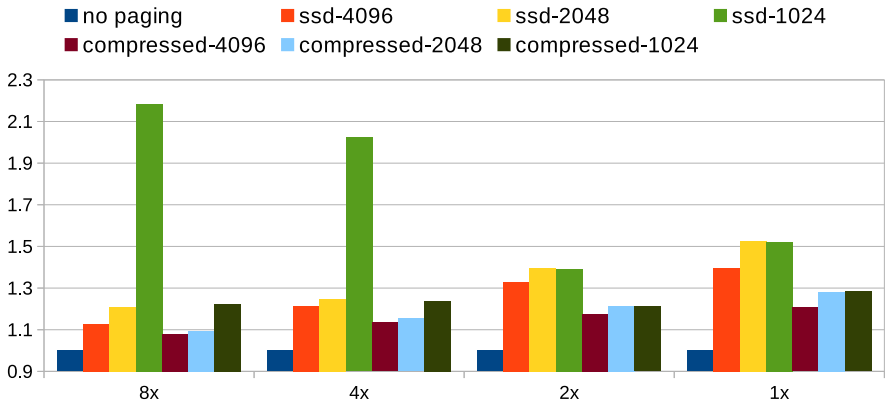
Figure 3.3: Normalized average read/write latency for miniFE. x-axis represents the problem sizes and y-axis represents latency normalized against the no-paging case. Smaller means better.

reads for both the SSD-based and compression-based paging cases. This outcome is expected, since the compression overhead is larger than the decompression overhead [56].

Also as we keep the main memory size smaller, we see the average latency increases as more pages are need to be swapped. Figure 3.2(a) and Figure 3.2(b) also




(a) Normalized average read latency for LAMMPS




(b) Normalized average write latency for LAMMPS

Figure 3.4: Normalized average read/write latency for LAMMPS. x-axis represents the problem sizes and y-axis represents latency normalized against the no-paging case. Smaller means better.

demonstrates HPCCG has much more uniform latency changes (smaller main memory size results in increased latency) compared to the other two test applications. This is due to the fact that, majority of HPCCG memory accesses are in a small number of memory pages [136]. As a result, the number of pages that are swapped out are fairly uniform for the different problem sizes. For LAMMPS and miniFE, we can see that for larger problem size (4x and 8x) when we have smaller main memory (1028KB), the average write latency penalty is much larger than the other cases.

This is due to two facts: 1) the memory accesses for these two applications are more distributed across different regions [136, 60] and 2) due to the main memory size being very small(1028KB), we saw a large number of pages being swapped out which increased the overall average latency due to the compression overhead.

### 3.4.2 Impact on execution time

Next, we compare the execution time of the three test applications. We present the normalized time spent in a memory system following the same type of charts as the previous section. Similar to the previous section, the x-axis represents the problem size for our different memory configurations and y-axis presents the normalized execution time. Figure 3.5- Figure 3.7 shows the normalized execution times for HPCCG, miniFE and LAMMPS respectively.

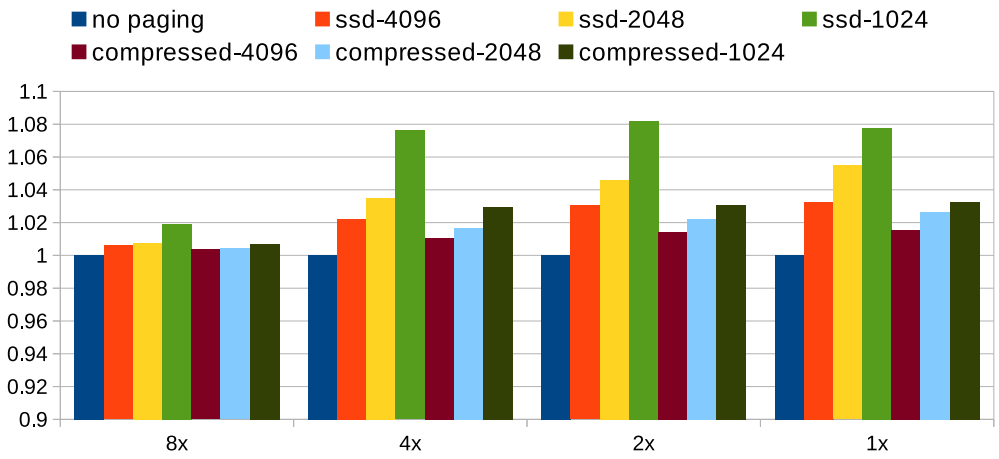



Figure 3.5: Normalized execution time for HPCCG. x-axis represents the problem sizes and y-axis represents execution time normalized against the no-paging case.

From these figures, we make the following observations -

- The overhead for paging in HPC applications is negligible and is under 8% for

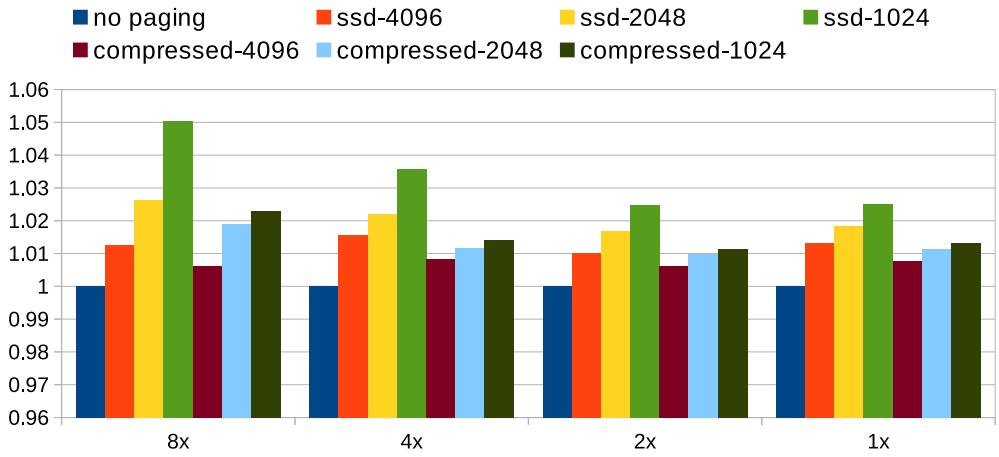



Figure 3.6: Normalized execution time for miniFE. x-axis represents the problem sizes and y-axis represents execution time normalized against the no-paging case.

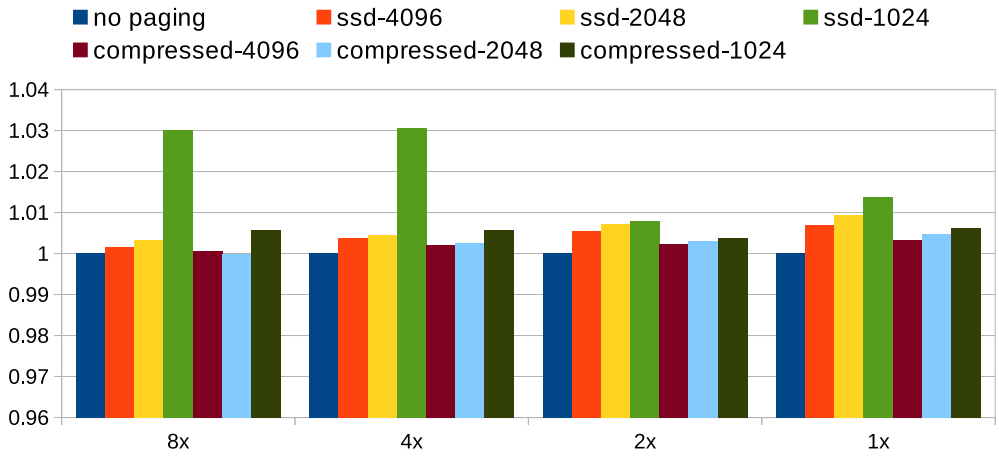



Figure 3.7: Normalized execution time for LAMMPS. x-axis represents the problem sizes and y-axis represents execution time normalized against the no-paging case.

all three of our test applications.

- This overhead can be further reduced by using software-based compression and by using portions of the memory to store pages in compressed form.
- The overhead for compression-based paging for our test applications are under

4%.

### 3.4.3 Impact of compression parameters

Next, we wanted to understand the impact of different compression parameters on our proposed compressed-paging scheme. We do this, by keeping both the memory size and memory trace constant and by changing the compression parameters in Equation 3.1. We looked into two different compression parameters: first the compression factor which is the total space saving due to compression and the other is (de)compression speed which is how quickly we can (de)compress.

#### Compression factor

To understand the impact of the compression factor on the performance of our compressed-paging scheme, we modified the compression factor parameter and experimented with 4 different hypothetical compression factors (90%, 80%, 70% and 60%). A higher compression factor means more data volume reduction. The size of the evicted page (that is swapped to the compressed paging store) is a function of the compression factor based on Equation 3.1 and in turns influences the memory access times for the simulator. Therefore, with decreasing compression factors (larger compressed page sizes), we would see slower memory access times.

For these experiments, the main memory size was set to 4MB. I compared these execution times against the no-paging case and present our results in Figure 3.8.

In this figure, each bar represents the compression factors for our three test applications. All the execution times are normalized against the execution time when we do not have any paging enabled. We can see from Figure 3.8 that changing the compression factor has very limited impact(less than 0.3%) on normalized execution

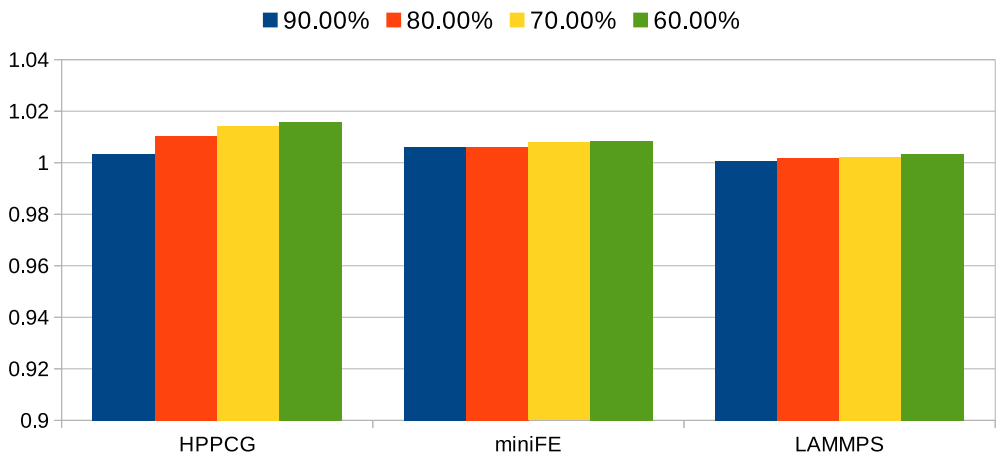



Figure 3.8: Impact of varying compression factor on execution time. y-axis represents normalized execution time compared to the no-paging case. Smaller is better.

time. However, since having a larger compression factor means we can store more pages in memory, the space benefit is larger and the perceived memory capacity is greater with higher compression factors.

### Compression speed

To understand the impact of varying compression speeds on execution time, we varied the (de)compression speeds for our compressed paging scheme and simulated the execution times using two different hypothetical compression algorithms. One is 10x faster than our observed compression algorithm and the other one is 10x slower than our observed compression algorithm. Next, we normalized the two execution times against the no-paging-case execution time and present the results in Figure 3.9. In this figure, we observe that, the compression speed can impact execution more than the compression factor. Our observed compression factor performed very similar to a hypothetical compression algorithm with 10 times faster speed. On the other hand, a slowdown in compression speed negatively impacted the execution time even more.

Hence, we can say that for these three test applications, a slower compression algorithm would impact the execution time more than a faster compression algorithm.

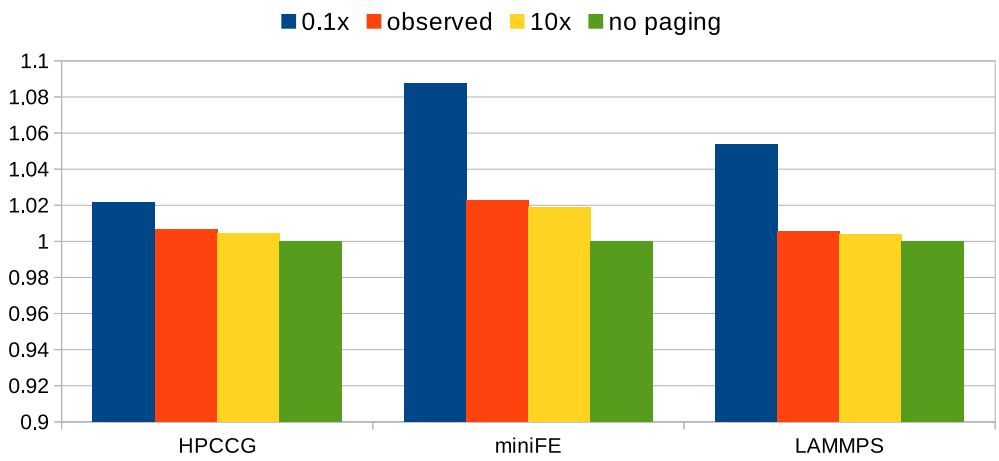



Figure 3.9: Impact of varying compression speed on execution time. y-axis represents execution time normalized against the no-paging case. Smaller is better.

### 3.4.4 Space benefits of compressed-paging

Finally, we compared the perceived memory capacity improvement that was due to our compressed-paging scheme. In 3.4, 3.5 and 3.6, we present the perceived memory capacity improvements for our three test applications. In these tables, memory savings $x\%$ imply that, due to compressed-paging, we could use $x\%$ of the original (non-paging) memory size for other workloads, and that the original workload can be solved using $(1-x)\%$ of the original (non-paging) memory size.

From these three tables, we can see that the larger we allocate memory, the smaller the perceived memory capacity becomes. This is due to the nature of the working set size of our test runs. When allocated memory-sizes are small, we store more pages into a compressed-cache which results in greater memory capacity improvements.

| Problem scale | Allocated DRAM size | Memory savings (%) |
|---|---|---|
| 8x | 4096 | 20.29 |
| 4x | 4096 | 8.15 |
| 2x | 4096 | 1.64 |
| 1x | 4096 | 0.18 |
| 8x | 2048 | 55.14 |
| 4x | 2048 | 49.08 |
| 2x | 2048 | 45.82 |
| 1x | 2048 | 45.09 |
| 8x | 1024 | 72.57 |
| 4x | 1024 | 69.54 |
| 2x | 1024 | 67.91 |
| 1x | 1024 | 67.54 |

Table 3.4: Memory savings due to compressed-paging for HPCCG

To summarize, we can see that for our test applications and test-workloads, we can save up to 78% of the main memory for additional workloads.

| Problem scale | Allocated DRAM size | Memory savings (%) |
|---|---|---|
| 8x | 4096 | 18.89 |
| 4x | 4096 | 6.45 |
| 2x | 4096 | 4.67 |
| 1x | 4096 | 0.96 |
| 8x | 2048 | 54.44 |
| 4x | 2048 | 48.22 |
| 2x | 2048 | 47.33 |
| 1x | 2048 | 45.48 |
| 8x | 1024 | 72.22 |
| 4x | 1024 | 69.11 |
| 2x | 1024 | 68.67 |
| 1x | 1024 | 67.74 |

Table 3.5: Memory savings due to compressed-paging for miniFE

| Problem scale | Allocated DRAM size | Memory savings (%) |
|---|---|---|
| 8x | 4096 | 45.86 |
| 4x | 4096 | 29.05 |
| 2x | 4096 | 11.55 |
| 1x | 4096 | 1.49 |
| 8x | 2048 | 67.93 |
| 4x | 2048 | 59.52 |
| 2x | 2048 | 50.77 |
| 1x | 2048 | 45.75 |
| 8x | 1024 | 78.97 |
| 4x | 1024 | 74.76 |
| 2x | 1024 | 70.39 |
| 1x | 1024 | 67.87 |

Table 3.6: Memory savings due to compressed-paging for LAMMPS

## 3.5 Summary

In this chapter, we have presented our proposed compression-based paging scheme for HPC systems. Using a simulation-based approach, we have compared the runtime performance of our compression-based paging scheme against the current architecture, which does not support paging and against a scenario where we would page to SSD-based node local storage. Our proposed compression-based paging scenario showed minimum runtime overhead (under 4%) for all our test applications and can provide up to 78% of additional memory capacity without having any hardware modifications.

We also compared the impact of two compression parameters on execution time and demonstrated that the currently available compression algorithm can perform fairly well. We demonstrated that a faster compression algorithm would perform similarly even with an order of 10 improvement in compression speed, and that a slowdown in compression speed would penalize the execution time more. On

the other hand, the differences in execution time for varying compression factor is very small although the higher the compression factor the greater the capacity improvements we would see.

One of the reasons for observing little overhead in execution time with the introduction of paging, is found in the memory access patterns of our test applications. Future work should investigate these memory access patterns and how to optimize paging based on how frequently a page is paged out. Based on this information, the application could have variable sizes of compression-based caches, thus further improving the perceived memory capacity.

# Chapter 4

# Inter-process data movement

## 4.1 Introduction

Advances in network technology have not been able to keep pace with the rate of advances in computational capabilities. Therefore, network bandwidth has become a precious commodity and both intra- and inter-application network contention and the potential resulting congestion have become important problems to understand and mitigate [13, 14, 55, 68, 120]. In this chapter, I studied the similarities among inter-process communication data volumes in order to apply compression-based techniques to reduce data movement and improve the perceived network bandwidth. I studied the most commonly used inter-process communication framework, MPI (Message Passing Interface) [138] as a proxy for all other inter-process communication frameworks.

I studied both intra (within a message) and inter-message (among different messages) similarity in the context of MPI-based applications. The goal of the study was to identify message data volume reduction techniques that could reduce network congestion and thereby effectively increase the available bandwidth of existing

network hardware. This study, makes the following contributions:

- I present two approaches that I applied to reduce message volumes at the cost of additional processing: compression and computing message differences or *diffs*.
- In the latter case, I present a novel two-level diff-based approach that reduced message volumes by more than 90% in some cases.
- Finally, I present the application runtime overhead due to the added latency introduced by this approach and discuss how this overhead can be reduced.

.

The rest of this chapter is organized as follows. First, I summarize the problem from previous works which demonstrate network congestion and its negative impacts on performance. Next, I describe my proposed compression-based approach and evaluation methodology by outlining the experimental framework and tool chain used in this work. Then, I present the results and analyses of my approach and conclude with a discussion of the outcomes and future directions for research.

## 4.2 Network congestion due to inter-process communication

Network congestion is a primary cause of performance degradation for communication heavy HPC applications [38, 39, 43, 42, 55, 64]. In this context, network congestion can arise from inter-process interference when an application's own traffic is contending for limited network resources, or inter-application interference when different applications and services contend for network resources.

In many core systems, it has been shown that contention for network interfaces and congestion due to the reduced per-core network bandwidth can degrade application performance significantly [68, 120]. Researchers have attributed the average number of bytes passing through the network to be a major contributor of network congestion [13]. The network congestion is caused by large volumes of data that are produced primarily by the frequent communications among the HPC processes. These processes exchange information using point-to-point and/or group (collective) operations through MPI.

In this work, we studied the message payloads that HPC application processes exchange through MPI and sought to identify the similarities that exist among these messages. Our hypothesis was that if there were similarities among messages, we could leverage them to apply compression and thereby reduce network congestion by reducing the network data volume. The potential benefits of MPI-based inter-process message compression had been demonstrated in various MPI-based studies [38, 39, 42, 43, 64]. But, our work distinguished itself from previous compression-based approaches because we explicitly considered message payload similarity, both within a message and across messages. To the best of our knowledge, no other work has considered a similar approach.

## 4.3 Methodology

The methodology for this work comprise three principle components: (1) message data collection, (2) intra- and inter-message data similarity analyses, and (3) simulation-based evaluations of message compression and diff-based performance impacts. In the following, we illustrate and describe in detail these components as well as the benchmarks and applications used in this study.

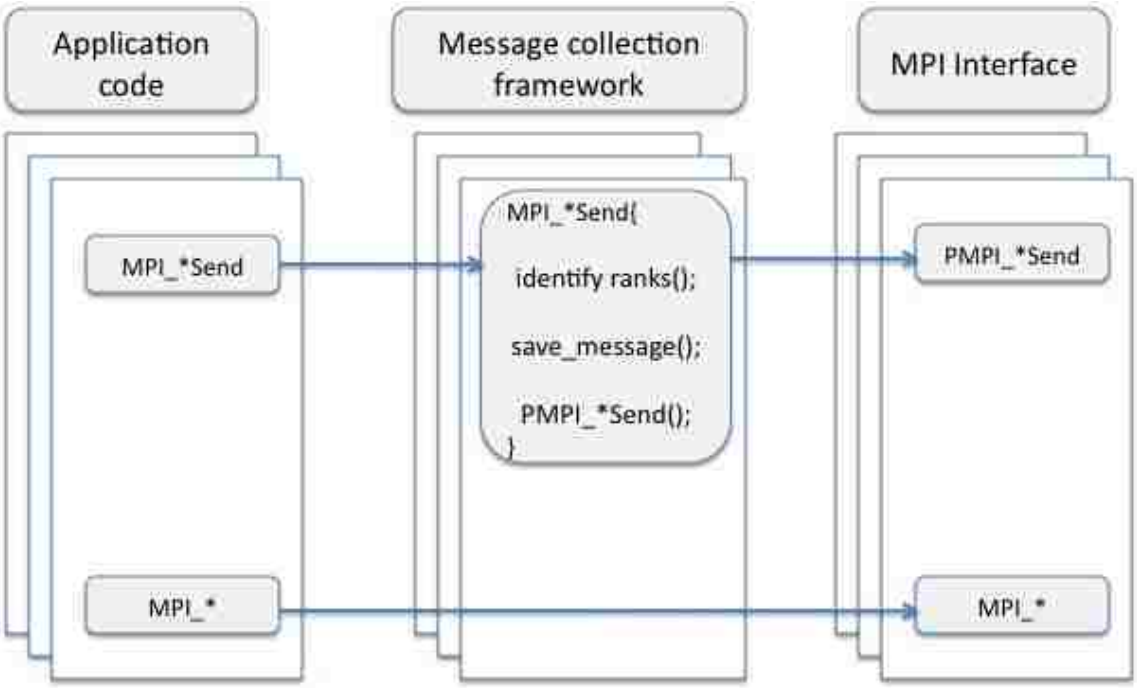



Figure 4.1: Message collection framework through MPI profiling interface.

### 4.3.1 Message Data Collection

Through its *profiling interface*, the MPI standard [138] provides a convenient way to intercept, and prepend and postpend operations to every MPI call. The profiling interface also provides access to message data types. Basically, every MPI function is can be invoked through two distinct names, one prefixed with $MPI$ and the other prefixed with $PMPI$. As demonstrated in Figure 4.1, we can intercept MPI message transmission operations by writing our own $MPI\_*$ routines that call libraries $PMPI\_*$ routines, prepending and postpending our own activity before and after the $PMPI\_*$ routines. For each MPI message transmission operation, we

1. intercept the message using the PMPI profiling layer,

2. identify the process ranks of each message's sender and receiver,

3. log each message's payload, tagged with its source and destination ranks, and finally,

4. send the original message to its destination.

Since, at the message sender process, we capture both sender and receiver ranks with the message payload, sender-based message interception is sufficient. Later, in our offline analysis, we use this information to apply sender- and/or receiver-based message filtering.

### 4.3.2 Intra-message Similarities

An initial question we wished to explore was what level of data similarities can exist within a message, i.e. intra-message similarities. Compression utilities are useful tools to help us answer this question. Compression factor, the extent to which a compression algorithm reduces a data volume, as defined by equation 4.1 below, is directly related to intra-message similarity: the larger the compression factor, the greater the intra-message similarity.

$$\text{compression factor} = 1 - \frac{\text{compressed message size}}{\text{original message size}} \tag{4.1}$$

Compression factor is the percentage reduction due to data compression. So, for example, a 60% compression factor means that the given compression utility can reduce data volume by 60%. Thus the compressed data is 40% of the original data.

### 4.3.3 Inter-message Similarities

A subsequent question we wished to explore was at what level of data similarities among different messages, aka inter-message similarities, exist. We used three differ-

ent methods to answer this question: (1) similarity preserving hashes, (2) a one-level message difference mechanism, and (3) a two-level message difference mechanism.

**Similarity preserving hashes:**

For similarity preserving hashes, we used the Trend Locality Sensitive Hash (TLSH) [96] tool, which compares a series of subsequent messages with a particular base message. TLSH differs from traditional hashes such that a small change in the data to be hashed results in a small change to the hash value. Given two data sets, TLSH works as follows:

1. compute the hash values of the two data sets,
2. compute the distance between the two hashes;
3. map the distance between the two hashes into a quantitative score, where a score of zero means the data sets are completely identical; a low score, for example 50, means they are very similar, and two totally different data sets would render a very high score, for example 1000 or greater.

While TLSH provides a quantitative insight about the difference between two messages, we cannot map this score to the size difference of two messages in bytes. As a result, similarity preserving hashes can lend us insight into message similarities, but we cannot use this approach as a general mechanism to understand potential data volume reductions.

**One-level Diffs**

Difference-based or diff-based tools work by taking two pieces of data and using a `diff()` function to generate a patch based on the incremental difference between the

two data items. Using the base data item and the generated *diff*, a `patch()` function can regenerate the other data item:

$$\texttt{diff}(m_1,\ m_2) \rightarrow d_1$$

$$\texttt{patch}(m_1,\ d_1) \rightarrow m_2$$

Using diff-based approaches, after the base message is transmitted, instead of transmitting full subsequent messages, we can hopefully transmit smaller diffs or patches. We call this first approach a one-level diff because we are computing diffs or patches between subsequent messages. (Later we will see that for two-level diffs, we compute patches of patches.) This way, we can use the difference between a subsequent message and its patch as our difference metric. In this case, we define our diff-based compression factor as a percent data reduction between the patch and the diffed message:

$$\text{compression factor} = 1 - \frac{\text{patch size}}{\text{message size}} \tag{4.2}$$

An example workflow of using one-level patches follows. Imagine a pair of communicating processes, A and B, where A is the sender and B is the receiver. The message sequence from A to B is $m_1, m_2, m_3, m_4, m_5, ....$ Using a diff-based approach, as depicted in Figure 4.2:

1. A sends $m_1$; B receives $m_1$.
2. A calculates $\Delta_{m_2,m_1}$, the diff between $m_1$ and $m_2$ and transmits the diff.
3. B receives $\Delta_{m_2,m_1}$ and regenerates $m_2$ by patching $m_1$ with $\Delta_{m_2,m_1}$;

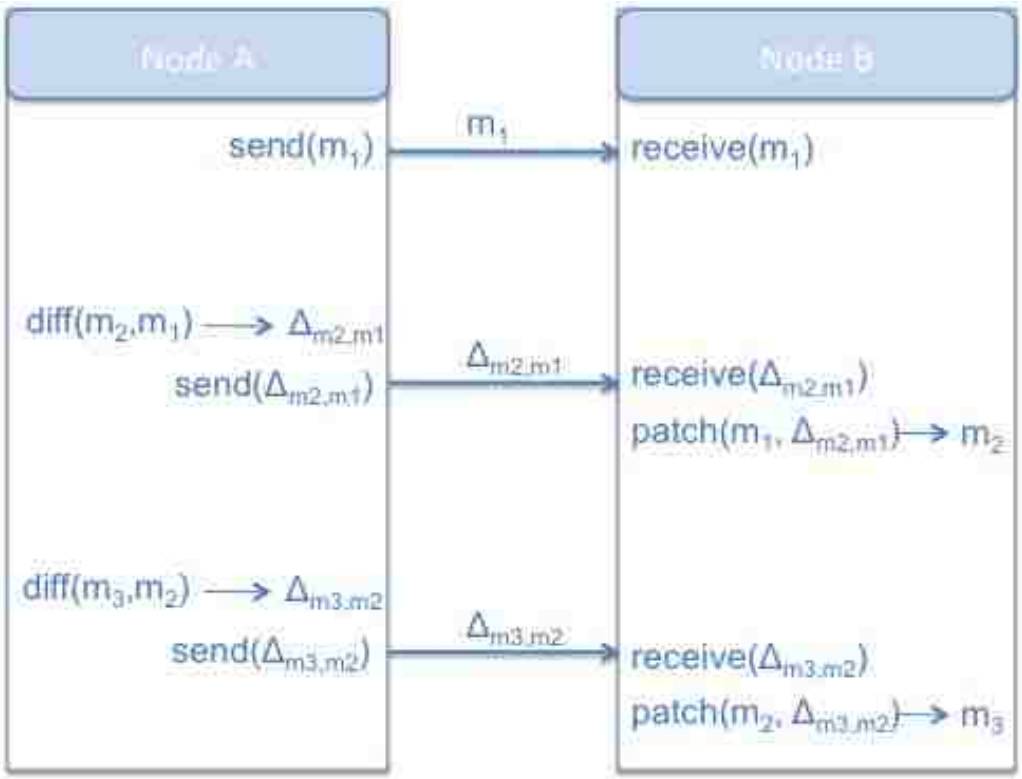



Figure 4.2: An example of one-level diffs

4. A calculates $\Delta_{m_3,m_2}$ and sends it to B.

5. B receives $\Delta_{m_3,m_2}$ and regenerates $m_3$ by patching $m_2$ with $\Delta_{m_3,m_2}$;

6. ... (and so forth)

The above example shows that, by using one-level diffs, at any point in time for each sender-receiver pair, both sender and receiver need to keep only a copy of the last transmitted or received message, respectively.

**Two-level Diffs**

Next, we present a new application of diff, which we call two-level diff. This approach exploits the fact that there often are similarities among the one-level diffs themselves, which recursively leverage diffs of diffs. In our initial set of experiments, we identified a regular pattern in terms of one-level diff sizes. We found that a large number of one-level diffs are very similar in size. Beginning with this insight, we wanted to understand whether there are similarities within these patches, that we can exploit. So, we chose to study two-level diffs.

We now demonstrate two-level diffing using the example from the previous subsection. In this case, depicted in Figure 4.3, we keep our base fixed as m1 and calculate $\Delta_{m2,m1}$ and $\Delta_{m3,m1}$. Our first two-level diff becomes $\Delta_{\Delta_{m2,m1},\Delta_{m3,m1}}$, the difference between the two one-level diffs. The size of two-level diffs depends on the differences between each of the subsequent messages and the base.

The workflow using two-level diffs follows:

1. A sends $m_1$; B receives $m_1$.
2. A computes and sends $\Delta_{m_2,m_1}$ to B; B patches $m_1$ with $\Delta_{m_2,m_1}$ to regenerate $m_2$.
3. A computes $\Delta_{m_3,m_1}$ then $\Delta_{\Delta_{m_2,m_1},\Delta_{m_3,m_1}}$ and sends the latter to B. B patches $\Delta_{m_2,m_1}$ with $\Delta_{\Delta_{m_2,m_1},\Delta_{m_3,m_1}}$ to regenerate $\Delta_{m_3,m_1}$ and then patches $m_1$ with $\Delta_{m_3,m_1}$ to regenerate $m_3$.
4. ...

From the above example, we can see that instead of sending m3, A can send the hopefully smaller $\Delta_{\Delta_{m_2,m_1},\Delta_{m_3,m_1}}$, thus effectively compressing from $|m_3|$ to

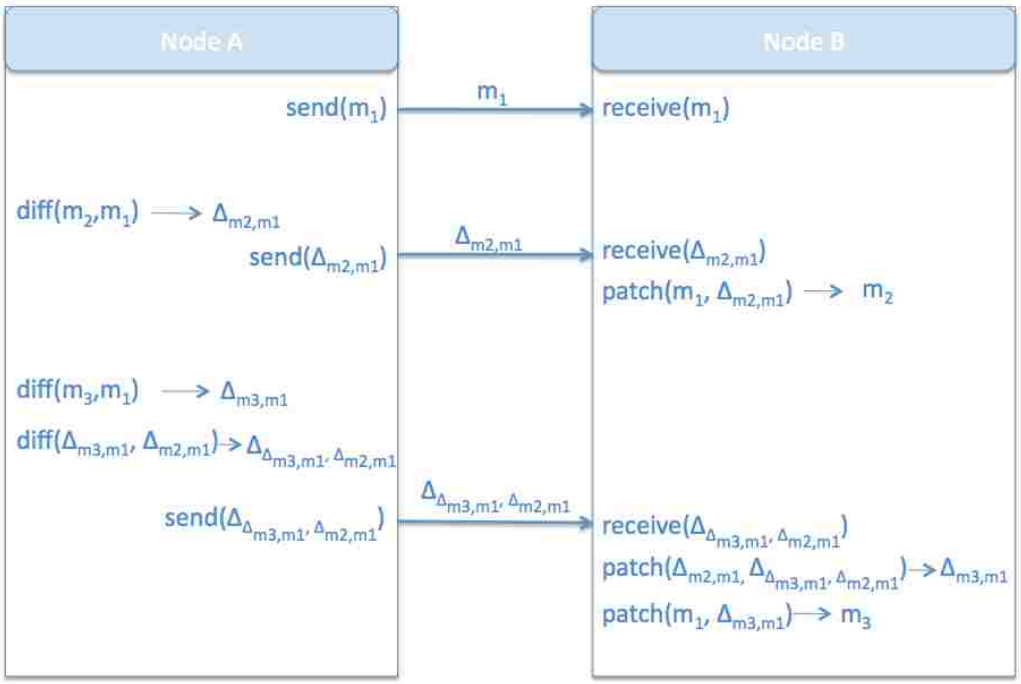



Figure 4.3: An example of two-level diffs

$|\Delta_{\Delta_{m_2,m_1},\Delta_{m_3,m_1}}|$. Thus, the two-level diff compression factor would be

$$\text{compression factor} = 1 - \frac{\text{two-level diff size}}{\text{original message size}} \tag{4.3}$$

Note that for two-level diffs, we can keep a base constant for a certain window of messages. For our experiments, we use a window size of 5 messages. That means, that for every 6th message, we update the base. For our test applications, we found that number to be the threshold at which the two-level diff sizes start to increase. Two-level diffs memory overhead is similar to that of one-level diffs. However, in addition to saving one message per sending and receiving process, we need to store an additional one-level diff for each sender and receiver.

### 4.3.4 Simulating Application Runtimes

We use a simulation-based approach to evaluate the impact of compression on application runtime. We chose simulation, because of its simplicity and availability. Simulation also enables us to run large scale experiments without having to worry about hardware resources.

We used `LogGOPSim` [54], a fast, accurate, freely-available discrete event simulator that supports large scale simulations based on real application traces. This framework is based on an extension of the `LogP` model of parallel computation [5]. The simulator includes a trace collector(`liballprof`) to collect MPI communication traces that serve as input to the simulator. The traces capture and are used to extrapolate (to larger scales) applications' computation and communication characteristics. The extrapolation is done via a schedule generator, `SchedGen`.

#### Trace collection

For our experiment, we first link the provided MPI trace collection library, *liballprof*, to our test applications. Similar to our message similarity experiments, we collect traces for HPCCG and miniFE solving a problem with dimension 100x100x100 and LAMMPS solving a problem with input size 5x5x5. We have used these problem sizes in our previous studies [56, 57, 58] and demonstrated that they are a good representation of actual problem sizes.

Our test applications ran on 4 MPI process, each one running on its own processing core. `LogGOPSim` have demonstrated its ability to extrapolate to large scale traces accurately with under 7% error rate [54]. Hence, we believe our choice of problem size can be accurately extrapolated to our simulated system size.

### Schedule generation and simulation

After the trace collection, we used `SchedGen` to create GOAL (group operation assembly language) schedules from each trace, extrapolated from four to 4096 processes. These extrapolated GOAL schedules eventually serve as our simulator inputs. Additionally, `LogGOPSim` accepts a number of input parameters that describe the characteristics of the target simulated environment. We present in Table 4.1 the `LogGOPSim` input parameters that we used for our experiments. The parameter values are from a previous `LogGOPSim` study [54] which represents a cluster system with 2GHz Opteron Quad core processor nodes connected by Infiniband interfaces. We then modify the values of parameter `P` to the number of processors to be simulated and modify the overhead per byte `O` to simulate the behavior of a system where messages are compressed before transmitting and decompressed before receiving. This is because overhead is the time when CPU is busy performing (de)/compression.

| | | |
|---|---|---|
| L | Maximum latency between any two processors | 5300 ns |
| o | CPU overhead per message | 2300 ns |
| g | Time between two message injections in the network | 2000 ns |
| G | Cost per byte per message | 3 ns |
| S | Message synchronization threshold | 32768 B |
| O | Overhead per byte | compression overhead |
| P | Number of processors | 16 |

Table 4.1: `LogGOPSim` simulator input parameters and their values

## 4.3.5 Our Test Applications

To demonstrate the potential impact of network congestion on performance, we used the Sandia Micro-Benchmark Suite. To characterize application message similarities, we used three mini applications from the Mantevo Project and two actual large scale

applications. We chose these applications because they are open source, available and widely-used in scientific and systems research. The mini apps embody essential performance characteristics of key applications. On the other hand, the two full applications, LAMMPS and MILC, demonstrate two different communication patterns. These benchmarks and applications are described below.

### Sandia micro-benchmark

Sandia Micro-Benchmark Suite (SMB) [10] is a collection of micro-benchmarks developed for testing and evaluating high-performance network interfaces and protocols. From this suite, we used the message rate benchmark, which provides a sustained message throughput in application scenarios. We measured message rates while increasing the message sizes, running our experiments on 16 nodes of Volta, a 24 core/node Cray system with an Aries interconnect from Sandia testbed [70]. We used two different configurations of 24 and 48 processes per node.

### The mini applications

We used three *mini-applications* or *mini apps* from the Mantevo Project [53]. HPC benchmarks generally target the evaluation of computer system performance. On the other hand, the mini apps are intended to mimic real application characteristics. We compared and observed the mini apps' performance similarity against the actual application it represents in previous work as well [58].

The *mini apps* we used for this study are HPCCG version 1.0, miniFE version 2.0 and miniMD version 1.2. The first two miniapps are implicit finite element mini apps. HPCCG is a conjugate gradient benchmark code for a 3D chimney domain that can run on an arbitrary number of processors. This code generates a 27-point

finite difference matrix with a user-prescribed sub-block size on each processor. miniFE mimics the finite element generation assembly and standard solution for an unstructured grid problem. miniMD is the mini application that represents the molecular dynamics application LAMMPS [105, 111]) and contains the performance impacting code from LAMMPS.

**Two full application: LAMMPS and MILC**

We used LAMMPS (the Large-scale Atomic/Molecular Massively Parallel Simulator) to as one of our full featured scientific applications. LAMMPS [105, 111] is a classical molecular dynamics code that was developed at Sandia National Laboratories. LAMMPS is a key simulation workload for the U.S. Department of Energy and is representative of many other molecular dynamics codes. For our experiments, we used the embedded atom method (EAM) metallic solid input script, which is used by the Sequoia benchmark suite.

The MIMD Lattice Computation (MILC) collaboration [21] solves lattice QCD problems and is available as part of the NERSC and NSF benchmarks. The MILC su3_rmd application is used to create sample four dimensional SU lattice gauge configurations for physics simulation. In our experiment, we used the problem ks_imp_dyn from the MILC problem suites. The ks_imp_dyn simulates a full QCD in a staggered fermion scheme.

Each application was run using 4 MPI processes and each process was run on an individual hardware core. In Table 4.2, we present the input parameters and the total number of messages collected from the test applications for this study. These are all of the messages that the processes exchanged throughout the application

| Application | Problem size | Problem | Number of messages |
|---|---|---|---|
| HPCCG | 100x100x100 | Conjugate gradient | 981 |
| miniFE | 100x100x100 | Finite element | 2448 |
| miniMD | 100x100x100 | EAM | 3440 |
| LAMMPS | 5x5x5 | EAM | 6600 |
| MILC | 8x8x8x8 | ks_imp_dyn | 70094 |

Table 4.2: Our test applications, the problems they were solving, and the total number of message collected

runs. We believe that the number of these messages is sufficiently representative to support the findings in our study. The numbers we present are averaged over all of these messages. From the number of messages, it's evident that MILC is extremely communication-heavy as compared to our other test applications. Most of the communications that MILC used were also asynchronous; hence, there was an ample number of overlaps in communications and computations in MILC.

### Compression utilities

For our chosen compression algorithm, we targeted the best (in terms of compression factor), lossless compression algorithm. We required losslessness to ensure that the destination processes can reconstruct precisely the message originally sent by source processes. We wanted the highest compression factor to identify the upper bound of the message similarities. This, in turn, represents the best opportunities for bandwidth savings.

For the above reasons, we used the compression algorithm that performed best in one of our previous studies [56], parallel bzip2 library `pbzip2`. `pbzip2` is a parallel implementation of `bzip2`. `bzip2` is an implementation of the Burrows-Wheeler transform [46], which utilizes a technique called block-sorting to permute the sequence of bytes to an order that is easier to compress. The algorithm converts frequently-

recurring character sequences into strings of identical letters and then applies a move to front transform and Huffman coding. `pbzip2` is lossless and also multi-threaded and, therefore, can leverage multiple processing cores to improve compression latency. The input file to be compressed is partitioned into multiple files that can be compressed concurrently.

For diffing, we used bsdiff [99] to compute the differences between subsequent messages. bsdiff is a binary patching tool and is widely used. bsdiff also leverages bzip2 compression to compute patches. So, the patches computed by bsdiff are already compressed. bsdiff computes the difference between two files by finding a set of identical regions and then extending the regions forward and backwards to allow mismatches. When the patches are reapplied, it can reproduce the original file, thus providing lossless compression. We chose bsdiff because of its ability to generate lossless, platform independent patches from binary data, and for being able to provide the best compression ratio [91, 77]. On average, the patches produced by bsdiff are a factor of five smaller than those produced by any other binary patch tool.

For our runtime overhead comparison, we also looked into the runtime overhead for two faster diff algorithms. The first one is xor+lz4. Levy et. al [75] developed this diff algorithm, which first applies bitwise exclusive or (xor) followed by compressing the xor with the lightweight, lossless data compression algorithm lz4 [22]. The other diff tool is xdelta, a binary diff algorithm based on VCDiff [69]. Both are fast diff algorithms but produces much larger patches compared to bsdiff.

## 4.4 Results

In this section, we seek to answer several important questions:

- How does network congestion impact performance?
- Are duplicate messages exchanged between sender/receiver process pairs?
- How much data similarity exists with a message?
- What is the difference (or delta) between two subsequent messages?
- Do subsequent messages from a particular sender/receiver pair differ over time?
- What is the application overhead of these compression techniques?

The remainder of this section answers these questions.

### 4.4.1 Network congestion negatively impacts performance

In order to address how network congestion impacts performance, we use the message rate benchmark from the Sandia Micro-Benchmark Suite (SMB) [10] With this benchmark, we measured the message rates as we increased the message sizes. We ran our tests on a CRAY XC-30 cluster, Volta, from Sandia National Lab. Each node had 24 Ivy Bridge cores and was connected by Aeries interconnects. We ran the SMB message rate benchmark on 16 nodes and two different configurations: 24 processes per node (ppn) and 48 ppn. We varied the message sizes starting from 4KB up to 64KB, each time doubling the message size and measuring the message rates for two different communication patterns:

- **single-peer, unidirectional:** A process communicates with exactly one other process and it either sends or receives one message with that peer at a time.
- **multi-peer, bidirectional:** A process communicates with multiple process peers and it can both send and receive messages from its peer.

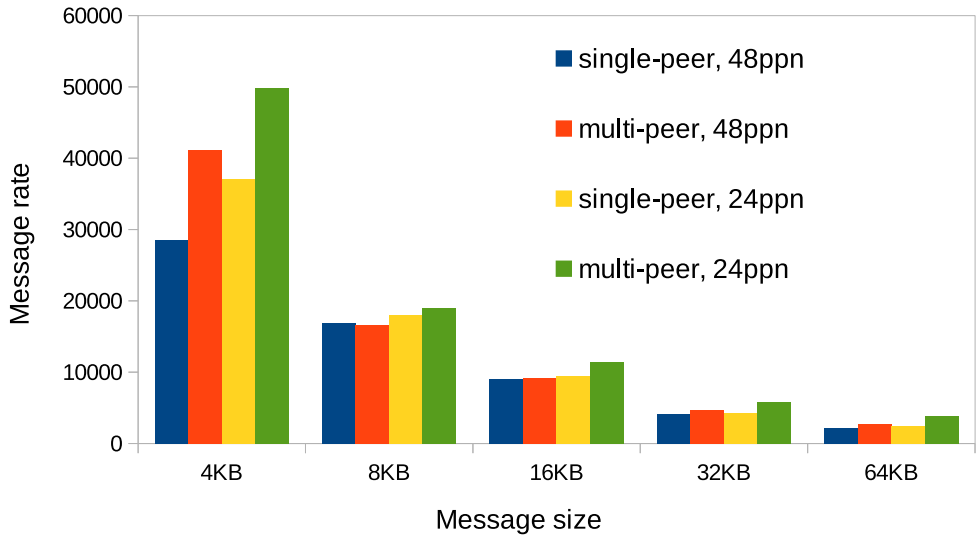



Figure 4.4: Message rate drops as we saturate the network leading to congestion. x-axis represents message size and y-axis represents message rates (higher means better). Each bar represents results from a single run with either 24 or 48 processes per node from different communication patterns.

Figure 4.4 shows our experimental results. From the figure, we can see that, as we increase the message size, we are saturating the network with more data and as a result the message rate is decreasing. We should note that the benchmark includes both intra-node and inter-node message exchanges. As a result, we see slight increase in message rates at higher message sizes when we use 48 processes per node compared to 24 processes per node.

### 4.4.2 Identifying duplicate messages

In studying message similarity, the first question we addressed is duplicate messages. If we can identify a large percentage of duplicate messages, we can store them at the receiver and can avoid sending the entire message.

In order to answer this question, we analyzed the message contents for all of our applications. First, we grouped all messages within an application by the sender ranks and then by the receiver ranks. Next, we compared all messages against all other messages for a particular process pair.

| | HPCCG | miniFE | LAMMPS | miniMD | MILC |
|---|---|---|---|---|---|
| sender | 0.175% | 0.001% | 0.022% | 0.0259% | 0.003% |
| receiver | 0.174% | 0.002% | 0.022% | 0.0259% | 0.003% |

Table 4.3: Only a very small percentage of messages are identical within a process pair.

We present the results of our duplicate message study in Table 4.3. We see that for all five test applications, the number of duplicate messages sent from a particular sender or received at a particular receiver is very small, less than 0.2%. This result suggests that is not a viable message compression technique for these workloads.

### 4.4.3 Intra-message similarity

In the previous subsection, we demonstrated that most of the messages sent/received within an application are unique. Next, we wanted to investigate the similarity within a message to see if that might be exploited. In order to study the similarity within a message, we applied the parallel bzip utility on all the messages individually and then used the compressibility of the messages as a quantitative indicator of the similarity within the message.

After compressing the messages, we calculated the compression factor using Equation 4.1. Next, we created a histogram based on the observed compression factors and counted the percent of messages with that compression factor.

We present our compression results in figure 4.5. In this figure, the higher the compression factors the better. As we can see from the figure, more than half of the

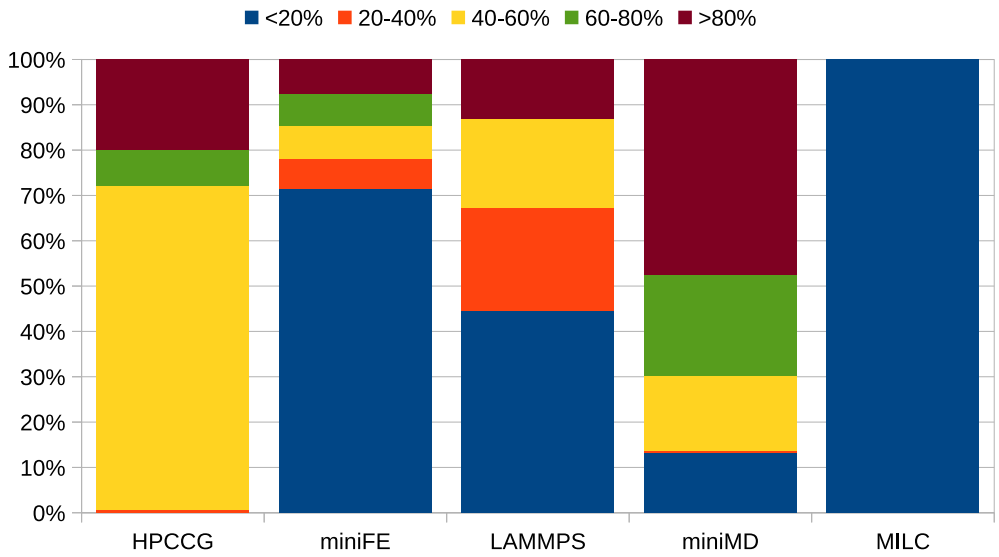



Figure 4.5: Compressibility of messages for our test applications using parallel bzip. Higher compression factors are better.

messages (almost 70%) for miniFE and LAMMPS, the compression factor is under 40%. For HPCCG, more than 70% of the messages have less than a 60% compression factor. miniMD messages demonstrated the best compression performance and MILC messages demonstrated the worst. Almost all messages of MILC showed very low levels of compression factor, while more than 60% of miniMD messages have 60% or more compression factor. This demonstrated that for all of our applications, except for MILC, some intra-message similarity existed: at least 25% of the messages can be reduced by 20% or more. But this technique had limited usefulness in reducing message volumes since only 30% or fewer messages have more than a 60% compression factor, except for miniMD.

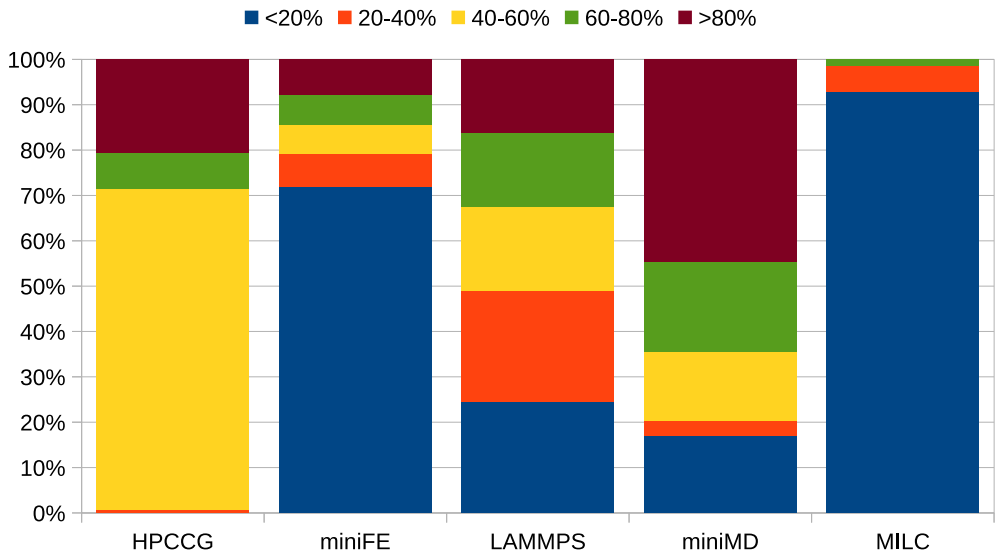



Figure 4.6: Compression factor for one-level diffs of the messages from our test applications. Higher is better.

### 4.4.4 Inter-message similarity

Thus far, we have focused on identifying similarities within a message or finding duplicate messages over the course of an application run. In this section, we studied whether there are similarities among subsequent messages to exploit the possibility of reducing message volumes.

First, we investigated the effectiveness of a one-level diff for our five test applications. We calculated the compression factor for these messages using equation 4.2 as described in section 4.3.3. Similar to the previous subsection, we created a histogram of the one-level diff compression factors, and present our results in figure 4.6. The y-axis represents the percentage of messages that fall within the compression factor range. Higher compression factors are better as they mean a smaller size for the sent messages.

From the figure, we can see that the one-level diffs perform similarly to our

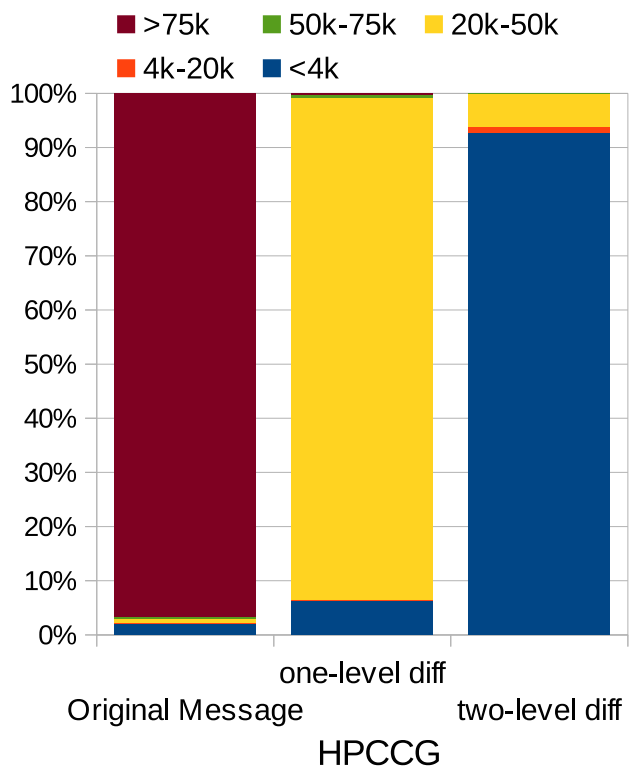



Figure 4.7: A comparison of the sizes of the original messages, the one-level diffs and the two-level diffs for HPCCG. Smaller is better.

compression numbers from the previous section both in terms of compression factor and the percentage of messages that achieve that compression factor. For HPCCG, miniFE and miniMD, the compression factor numbers are very similar. However, for LAMMPS we see that the one-level diffs perform slightly better than parallel bzip as more than 30% messages now have compression factors of 60% or greater. We also begin to see some improved compression factors for MILC, although the majority of MILC messages still have not demonstrated any improvement in compression factor.

Next, we present the results for our two-level diff experiments described in section 4.3.3. In Figures 4.7- 4.11, we depict a comparison of the original sizes of the messages, their one-level diff sizes and their two-level diff sizes, as a histogram

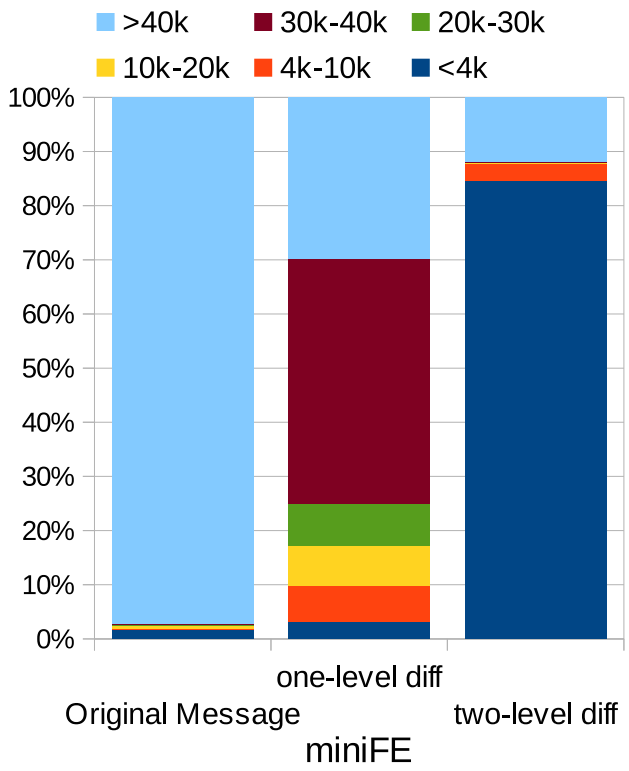



Figure 4.8: A comparison of the sizes of the original messages, the one-level diffs and the two-level diffs for HPCCG. Smaller is better.

for our test applications. Note that the bin sizes for the histograms presented in these figures are all different. This is due to the fact that the distribution of message sizes and their diff sizes (both one- and two-level) for our test applications were different from one another.

In these figures, the y-axis represents the percent of messages that fall within the message size bins. As we can see from the figure, while the one-level diffs can reduce the network data volume to some extent, we can substantially reduce the network data volume by applying the two-level diffs. The two-level diff can reduce the network traffic by more than 90% since more than 90% of the two-level diffs are

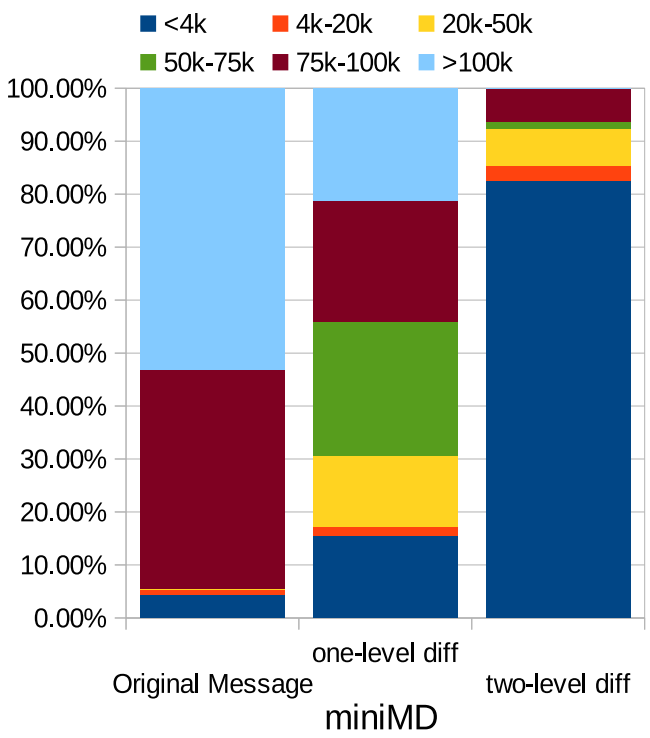



Figure 4.9: A comparison of the sizes of the original messages, the one-level diffs and the two-level diffs for miniMD. Smaller is better.

under 4K bytes in size. This shows that two-level diff can be extremely useful to reduce network congestion in communication-heavy MPI applications.

Now we ask, what does it mean to have such a small size of our two-level diff? While the one-level diffs consist of the difference between two messages, the two-level diffs are the difference between two one-level diffs. So, if we think of our messages as an array of bytes, the one-level diffs represent the bytes where the array values are changed and the two-level diff represents the changes within those byte changes. Therefore, having such small values for two-level diffs implies that the changes in messages are regular and extremely similar. As a result, when we compare these changes, we get these small two-level diff sizes.

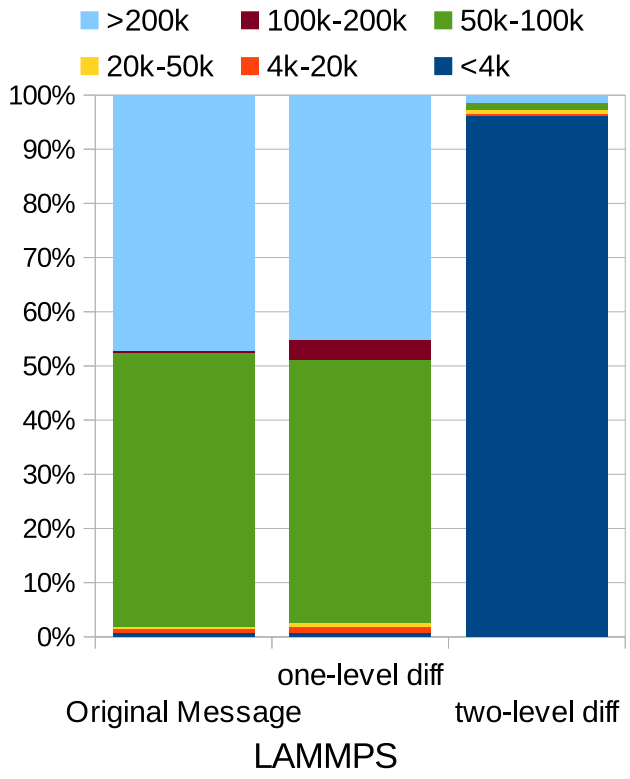



Figure 4.10: A comparison of the sizes of the original messages, the one-level diffs and the two-level diffs for LAMMPS. Smaller is better.

### 4.4.5 Impact on application runtime

The results from the previous subsections show that there are similarities among messages that we can exploit to reduce the network data volume by trading off computation for this reduced volume. This compression, although useful, introduces additional latency to message transmission. In the remainder of this section, we attempt to understand how this latency impacts the runtime of our test applications.

To address this performance question, we use the `LogGOPSim` validated simulator framework [124]. `LogGOPSim` allows us to simulate application behavior while taking into account the overhead of compression. From our experiments, we have measured

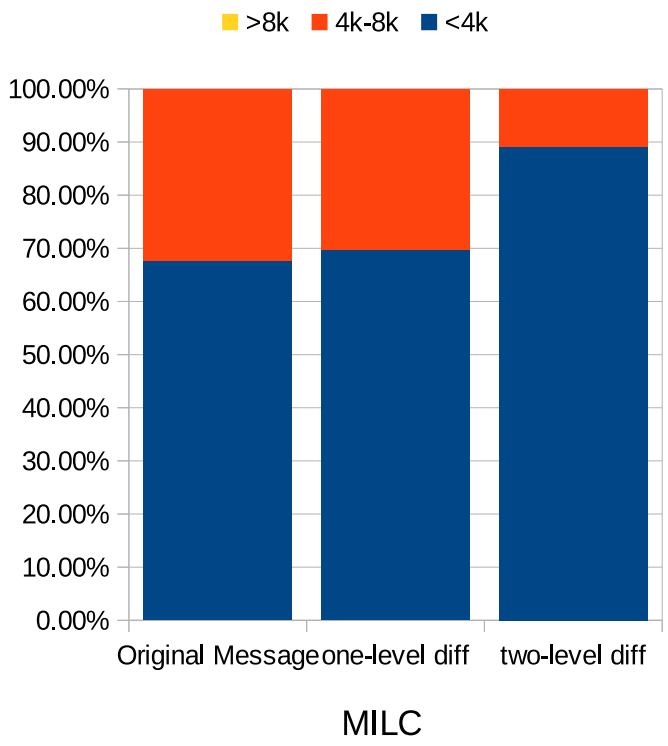



Figure 4.11: A comparison of the sizes of the original messages, the one-level diffs and the two-level diffs for MILC. Smaller is better.

the compression rates for our compression and diff tool. We took the reciprocal of the compression rate and used that as the compression overhead per byte (O) parameter of LogGOPS model.

Before studying the overhead in application runtime performance, we should also mention that, at the time of the study, the current implementation of `LogGOPSim` does not account for network congestion. The simulator assumes that when a message is being sent, it has a dedicated link between the communicating nodes for that message. As a result, we could not leverage the improved bandwidth due to the smaller size of the compressed message or the patch. But since the simulator is the only available simulator that works with application traces, we believe that it can

give us an upper bound of the performance penalty that our test applications can experience due to the added compression latency.

We ran our traces in two scenarios: no overhead and compression overhead. We then compared these two simulated runtimes and present the runtime overhead for 4096 processes in Table 4.4.

| diff algorithms | Overhead per byte | HPCCG | miniFE | miniMD | LAMMPS | MILC |
|---|---|---|---|---|---|---|
| xor+lz4 | 4ns | 0.27% | 0.09% | 0.06% | 0.58% | 0.07% |
| xdelta | 37ns | 6.77% | 2.72% | 1.59% | 14.16% | 1.55% |
| bsdiff | 250ns | 50.63% | 21.63% | 16.59% | 101.83% | 9.33% |
| bzip2 | 387ns | 84.10% | 34% | 26.92% | 156.74% | 17.2% |

Table 4.4: Normalized runtime for our test applications against the case when we do not have the overhead due to our diff-based approach. Smaller means better. Note: these numbers do not include the benefits of reduced congestion because of the limitation of our simulator.

In Table 4.4 we see that for our two compression tools, due to the large overhead per bytes (O), there are large slowdowns in application runtime for all five of our applications. These slowdowns are upper bounds in overhead, since they do not take into account the positive effect of reduced congestion.

Since the overheads are quite large, we wanted to see how a faster compression tool might mitigate this overhead. So, we used two faster diff algorithms from the literature [75] (xor+lz4 and xdelta) and simulated our collected traces against these two diff algorithms.

As we can see from Table 4.4 the faster diff algorithms, due to their smaller overhead per byte (O) value, can reduce the runtime overhead to under 14%. However, these two diff algorithms reduce the patch creation time overhead by trading off the size of the patches [75]. As a result, we would not observe the same

level of compression factor that we observed with bsdiff in Figure 4.7- Figure 4.10 and the bandwidth advantage would be limited.

**Does the runtime overhead change with scale?**

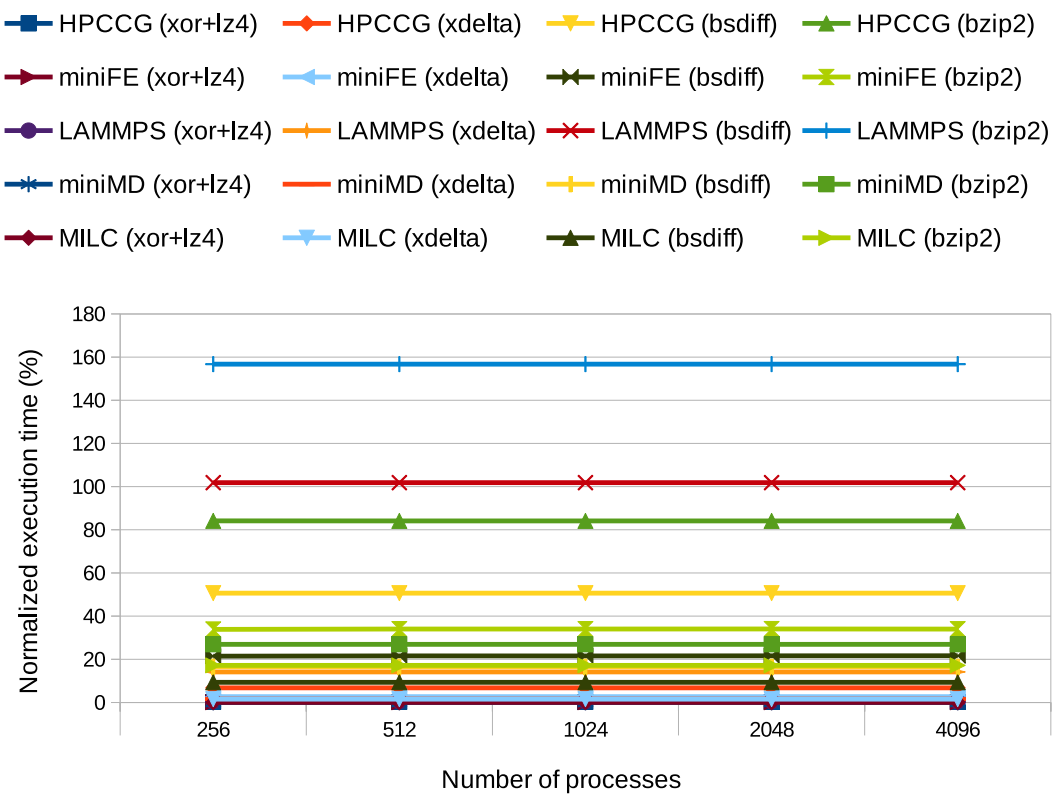



Figure 4.12: As we increased the process count, the runtime overhead remained nearly the same.

Finally, we would like to verify the scalability of our compression based approach over application runtime. In other words, we wanted to answer the question whether the upper bound of overhead described in subsection 4.4.5 changes as we increase process count. In order to do that, we performed a scaling experiment for all of

our test applications and measured their runtime overheads due to compression. We first ran simulations for 256 processes and then doubled the process count until we reached 4096 processes. Next, we normalized the runtimes reported by the simulator against the case when we do not have compression overhead.

We present our results in Figure 4.12. The y-axis represents the runtime overhead for compression for the four different compression approaches mentioned in our study. The lower the overhead the better. And in the x-axis we increased the process count. From this figure, we can see that the runtime overhead does not have any effect with changes in application scale.

The scalability results demonstrate that, if we could simulate congestion, we would see a reduction in overhead as we scaled up. The reason is that as we increase the number of processes, we increase network data volume and thus increase congestion. But, since our runtime overhead remains roughly the same, we would see less runtime overhead for our applications.

### 4.4.6 Memory overhead due to the diff-based approach

In Section 4.3.3, we explained how the diff-based approach works and the requirement of storing few messages and diffs locally to be able to apply the patch and reconstruct subsequent messages. In this section, I will analyze how that may impact the memory.

For a system using one-level diffs, we claimed that, at any point in time for each sender-receiver pair, both sender and receiver need to save only a copy of the last transmitted or received message, respectively. So, for a process that is sending messages to $n$ processes and receiving messages from $m$ processes, it needs to save at most $O(n + m)$ messages at any time. To illustrate this, we can use the same example that we used in Section 4.3.3 to describe the one-level diff process. For

a particular receiver, process `A` stores $m_1$ until the diff, $\Delta_{m_2,m_1}$, is transmitted. In which case, `A` can discard $m_1$ and store $m_2$ since $m_2$ would be required to calculate the next message $\Delta_{m_3,m_2}$. Similarly, after transmitting $\Delta_{m_3,m_2}$, message $m_2$ can be replaced by $m_3$. As a result, for each receiver of `A`, as a sender, process `A` needs to save only 1 message per receiver. Similarly, as a receiver, `A` needs to save only the last reconstructed message (calculated from the diff by applying the last transmitted patch). Therefore, a process needs to store $O(n + m)$ messages at any time.

For a system, that uses the two-level diff-based approach, memory overhead is similar to that of one-level diffs. However, in addition to saving one message per sending and receiving process, we need to store an additional one-level diff for each sender and receiver. The sender would use the one-level diff to create the two-level diff and the receiver would use the one-level diff and the transmitted two-level diff to apply the patch and reconstruct the original message. Therefore, for two-level diffs, a process communicating with $n$ senders and $m$ receivers would need to save $O(n+m)$ messages as well as $O(n+m)$ one-level diffs at any time.

Given the bounds in number of messages that are required to be stored in each process, the memory overhead can grow really high if a process is communicating with a large number of other processes. To keep the design scalable, processes can keep a threshold, such that the maximum number of messages stored in memory cannot exceed that threshold. RDMA-based MPI implementations use a similar approach in order to limit the memory overhead caused by the number of mailboxes in memory.

Another solution to this problem would be to maintain an automatic timeout such that, if a process has not communicated with another process for a certain amount of time, it can discard the locally stored messages. However, studies have

observed that, in many MPI applications, a process only communicates with a subset of all other process [80, 135]. Liu et al. tested 5 HPC applications and benchmarks and observed that, for a world of 1024 processes, each process communicated with on average between 1 to 11 other processes [80]. The maximum average of 11 was demonstrated by CG, which is a NAS parallel benchmark [8] that mimics the behavior of a communication-heavy application.

The memory overhead is also a function of the size of the messages. However, studies have demonstrated that messages sizes become smaller as we increase the number of nodes and stays within 10s to 100s KB [65, 108]. Given these message sizes and the average number of communication peers, we believe that the memory overhead would be minimal.

### 4.4.7 Impact of bandwidth improvement

Finally, we wanted to explore the impact of perceived bandwidth improvement and how that impacted the message rates. We used the SMB message rate benchmark to run these tests and varied the message sizes based on different compression factors. We ran our tests on 8 nodes and 48 process per nodes using the Sandia Volta machine. We first calculated the message rates for a message size of 256KB. Next, we compared that against the cases where the message size is reduced by different compression factors. We achieved this by changing the message sizes to simulate the bandwidth improvement. However, we did not modify any latency numbers in this experiment. Hence the test only considers message rate improvement due to the reduced message sizes. We ran these tests for two different communication patterns: single-peer and multi-peer.

| | Compression factors | | | | |
|---|---|---|---|---|---|
| | 99.00% | 95.00% | 90.00% | 80.00% | 70.00% |
| single-peer | 1.75 | 1.37 | 1.41 | 1.25 | 1.19 |
| multi-peer | 1.70 | 1.34 | 1.35 | 1.20 | 1.13 |

Table 4.5: Message rate improvements due to the improved network bandwidth for different compression factors. These numbers are normalized against the case when there is no compression. Higher means better.

Table 4.5 lists the message rate improvements for different compression scenarios. We can see that, the message rate improvements began to diminish with decreases in the compression factor. Given the large amount of message volume reduction, we demonstrated that, due to our two-level diff-based approach, the perceived network bandwidth improvement can improve message rates by up to 75% in the best case (HPCCG).

## 4.5 Discussion and future work

In this work, we demonstrated that both intra- and inter-message similarities can exist among HPC application messages and that these similarities can be exploited to reduce network data volumes. For our application set, we characterized and quantified these similarities and presented a new, two-level, diff-based algorithm to further exploit the similarities. Our new approach can reduce network data volume by more than 90%. We also simulated a compression based MPI framework and used it to explore an upper bound of the performance penalty due to the added compression latency. We demonstrated that the bandwidth savings can come at a significant runtime penalty. In order to reduce compression overhead, we explored faster diff-based algorithms and demonstrated that, for faster diff based algorithms, the performance penalty falls under 14%. But the faster diff algorithms also trade off compression savings.

For future work, we would like to explore diff algorithms that offer both high compressibility and fast performance. Below, we summarize some future opportunities in this area:

- **Demonstrating the benefit of reduced congestion.** We would like to investigate the performance impact on application runtime due to reduced congestion. A straight forward approach would be to implement our two-level diff-based approach as an extended MPI runtime and compare application runtime in scenarios where application performances are constrained by network bandwidth.

- **Faster diff algorithms.** One possibility for improving algorithm speed is via the use of hardware acceleration, for example GPU-based algorithms. GPUs have many fast, parallel processing units, which can reduce the overhead and thus the performance penalty. However, parallelizing the diff algorithm may not be straight forward, and we could not find any general implementations of parallel diff algorithms.

  One possibility would be to partition messages into blocks of uniform lengths and calculate the diffs on each block. But, as we reduce block sizes to take advantage of many parallel cores, we may limit the scope of identifying similarities across larger window sizes, thus potentially reducing the compression factor. Another approach might be to leverage the capacity of many core processors, by having dedicated cores for de/compression. That way, compression would be offloaded from main application processors potentially reducing the performance impact of the de/compression overhead.

- **Leveraging application specific knowledge.** Inspired by the results in this study, application developers can consider computing message diffs

at the application level and sending incremental differences only. While this would create an additional complexity for application developers, they may be able to leverage domain specific knowledge. Such a concept has been successfully exploited within the context of faster, domain specific diff algorithms [90, 129, 2].

# Chapter 5

# Inter-application data movement reduction

In this chapter, I study inter-application data movement reduction using a checkpoint/restart-based(CR) fault tolerance protocol as a case study. I demonstrate that, by reducing the inter-application data volume, we can reduce the overheads that improve the efficiency of the overall application runtime.

For this study, a model-based approach is used employing empirically measured performance data as model parameters. I propose this model to explore the viability of compression for the checkpoint operation. I further explore compression-based CR optimization in three respects: (1) by exploring its baseline performance and scaling properties, (2) by evaluating whether improved compression algorithms might lead to even better application performance, and (3) by comparing checkpoint compression against and alongside other software- and hardware-based optimizations. Finally, I study the energy impact of compression-based CR optimization.

## 5.1 Data movement in checkpoint/restart-based fault tolerance

Fault-tolerance (also termed reliability or resilience) is a major concern for current, large-scale high-performance computing (HPC) systems. This concern grows for future, extreme-scale systems for which increased node counts, more complex nodes and changes in chip manufacturing processes are projected to lead to low component mean times between failures (MTBFs) [113]. In these environments, decreased MTBFs and a confluence of other issues, including increased I/O pressures and increased overheads of traditional fault-tolerance approaches, have motivated new research endeavors to understand and improve the viability of fault-tolerance mechanisms like checkpoint/restart (CR) protocols. In particular, several studies have raised concerns about the continued viability of checkpoint/restart-based fault tolerance [113, 35].

CR protocols [33] periodically save process state to stable storage devices. For large-scale applications comprised of many thousands or even millions of processes, checkpoint data movement can lead to performance bottlenecks due to excessive data volumes and contention for network and storage devices. As we described in Section 2.3, researchers have proposed several CR protocol performance optimizations to alleviate this data movement challenge, including checkpoint data compression. In this chapter, we focus on the checkpoint compression optimization and reveal several insights regarding its impacts on the performance of large-scale applications.

This work aims to answer several broad questions:

- What is the general viability of checkpoint compression CR optimizations?
- Would better or faster compression algorithms render better overall application performance?

- How do checkpoint compression optimizations compare against other hardware and software-based optimizations?
- How do checkpoint compression optimizations perform in conjunction with other CR optimizations?
- What is the energy impact of checkpoint compression optimization?

We explored these questions guided by current system characteristics and with an eye toward emerging and new potential technologies. Using a performance model [56] based on Daly's higher order checkpointing model [25], we analyzed the impact of compression speeds and compression performance. We compared these results against a number of state-of-the-art software and hardware CR optimizations. In addition, we used information theory along with knowledge from an application-level checkpointing library to evaluate the efficacy of standard compression utilities. Based on these studies, this work offers the following contributions:

1. A viability model for checkpoint data compression that accounts for the cost and benefits of compression for checkpoint commit and recovery operations.
2. A validated coarse-grained energy model that accounts for the total energy spent by an application using checkpoint-compression.
3. A demonstration that checkpoint data compression can significantly improve an application's runtime across a wide range of scenarios.
4. A demonstration that existing, text-based compression algorithms may offer sufficient speed and checkpoint data compressibility such that enhanced compression algorithms likely will render little application performance improvement.

5. A demonstration that checkpoint data compression can yield application performance improvements when used in conjunction with other software CR protocol optimizations.

6. A demonstration that checkpoint data compression used in conjunction with other software CR protocol optimizations can be a viable, cost-effective alternative to hardware-based CR solutions.

7. A demonstration that checkpoint data compression can lead to up to 90% energy savings.

The rest of this chapter is organized as follows: First, in Section 5.2 we describe our evaluation methodology and tool chain. In Section 5.3, we present our results of the performance and scaling features of compression-based checkpoint optimizations, followed in Section 5.4 by a study of the potential benefits of enhanced compression algorithms. Our last set of results, presented in Section 5.5, comprise a comparative study of checkpoint data compression and other CR optimizations, . Finally, we conclude with a summary of our findings and a discussion of the implications of these results for future HPC systems.

## 5.2 Methodology: Data Collection and Performance Models

In this study, we compared checkpoint compression to other CR protocol optimizations. Figure 5.1 depicts our approach for executing this study and the set of tools that we used. Our general method was to (1) collect empirical data for the functional (amount of compression) and performance (compression/decompression speeds) behavior of different compression algorithms on real checkpoint data; and (2) feed this

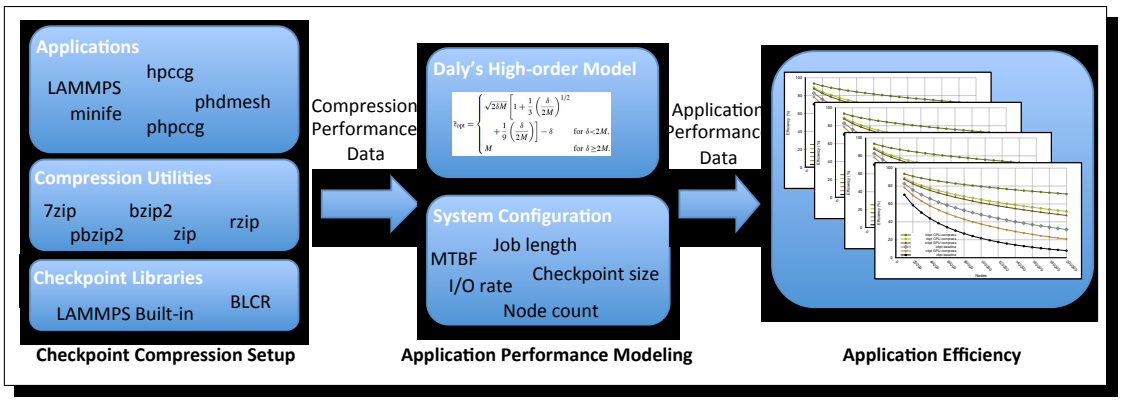



Figure 5.1: Our Method: Empirically collected checkpoint compression data is input to an extension of Daly's Model. The results are used to compute application efficiency.

data along with different application workloads and system configurations into validated performance models to observe the resulting application performance. In this section, we offer the comprehensive details of our approach.

### 5.2.1 Collecting Checkpoint Compression Performance Data

To collect checkpoint compression performance data, we instrumented a set of exascale proxy applications and two large-scale production applications with CR capabilities. We executed these applications with CR enabled to collect the application checkpoints. Then, we used various compression utilities offline to measure the extent to which the checkpoint files can be compressed as well as the speed of checkpoint compression and decompression.

#### The Proxy Applications

*Proxy applications* (or *mini-applications* or *mini apps*) are small, self-contained programs that embody essential performance characteristics of key applications. We

chose four mini apps from the Mantevo Project [53], namely HPCCG version 0.5, miniFE version 1.0, pHPCCG version 0.4, miniMD 1.2 and phdMesh version 0.1. The first three are implicit finite element mini apps and phdMesh is an explicit finite element mini app. HPCCG is a conjugate gradient benchmark code for a 3D chimney domain that can run on an arbitrary number of processors. This code generates a 27-point finite difference matrix with a user-prescribed sub-block size on each processor. miniFE mimics the finite element generation assembly and solution for an unstructured grid problem. pHPCCG is related to HPCCG, but has features for arbitrary scalar and integer data types, as well as different sparse matrix data structures. MiniMD is the mini application that represents the molecular dynamics application LAMMPS [105, 111]) and contains the performance impacting code from LAMMPS. PhdMesh is a full-featured, parallel, heterogeneous, dynamic, unstructured mesh library for evaluating the performance of operations like dynamic load balancing, geometric proximity search or parallel synchronization for element-by-element operations.

**The Production Applications**

We evaluated checkpoint compression performance on two full-featured scientific applications - LAMMPS (the Large-scale Atomic/Molecular Massively Parallel Simulator [105, 111]) and CTH [31].

- LAMMPS is a classical molecular dynamics codes developed at Sandia National Laboratories (SNL). LAMMPS is a key simulation workload for the U.S. Department of Energy (US DOE) and is representative of many other molecular dynamics codes. In addition, LAMMPS has built-in checkpointing support that allows us to compare generic, system-based mechanisms with an application specific mechanism. For our experiments, we used the embedded

atom method (EAM) metallic solid input script, which is used by the Sequoia benchmark suite.

- CTH is a production shock physics application from SNL that scales from tens to hundreds of thousands of processes. The input description used is the "fragmenting pipe" problem, a multi-material enclosed tube with a rapid shock expansion on one end. This is an unclassified input that is similar to classified CTH workloads run at SNL.

### The Compression Utilities

For this study, we used popular compression algorithms that were investigated in Morse's comparison of compression tools [88]. In previous work, we showed the results from a wide variety of algorithms [56]. Here we present the results from the algorithms that yielded better compression factors, namely parallel bzip and zip. Additionally, some algorithms can be parameterized to trade off between execution time and compression factors; for example, getting better compression factors but with slower compression decompression rates or getting worse compression factor but with faster compression decompression rates. But in some cases, we observed that small improvements in one parameter would not necessarily improve the viability bandwidth (the I/O commit bandwidth threshold for compression to remain viable) due to the poor performance of the other parameter. Although we calculated the viability bandwidths associated with different parameters, we only present here the parameter sets that yielded the best trade-offs, in terms of compression/decompression speeds versus compression factor.

- **zip**: `zip` is an implementation of Deflate [27], a lossless data compression algorithm that uses the LZ77 [143] compression algorithm and Huffman coding. It is highly optimized in terms of both speed and compression efficiency.

`zip` takes an integer parameter that ranges from zero to nine, where zero means fastest compression speed and nine means best compression factor. For our experiments, `zip(1)` represents the best trade-off.

- **pbzip2**[46]: `pbzip2` is a parallel implementation of `bzip2`. `pbzip2` is multi-threaded and, therefore, can leverage multiple processing cores to improve compression latency. The input file to be compressed is partitioned into multiple files that can be compressed concurrently.

  `pbzip2` takes two parameters. The first parameter is the block size, an integer that ranges from zero to nine, where a smaller value specifies a smaller block size. The second parameter defines the file block size into which the original input file is partitioned. For our experiments, `pbzip2(1,5)` represents the best trade-off.

### Checkpoint/Restart Utilities

The Berkeley Lab Checkpoint/Restart library (BLCR) [52] is an open-source, system-level CR library available on several HPC systems. For all of our experiments, excluding the ones that required application-specific checkpoints, we obtained checkpoints using BLCR. Furthermore, we used the Open MPI [41] framework, which has integrated BLCR support.

For our studies of application-specific and user-level checkpointing, we used the CR library built into LAMMPS. LAMMPS can use application-specific mechanisms to save the minimal state needed to restart its computation. Specifically, it saves each atom location and speed. The largest data structure in the application, the neighbor structure that is used to calculate forces, is not saved in the checkpoint but is recalculated upon restart. This scheme reduces per-process checkpoint files to about one eighth of the application's memory footprint.

**Power Instrumentation and Measurements**

We used the PowerInsight energy measurement devices designed and developed jointly by Sandia National Laboratories and Penguin Computing [72]. This instrument enables component-based power and energy measurements by using individual hall-effect sensors on each of the power rails leading to the CPU, memory and other devices. PowerInsight is completely separate from the system under test; it is electrically separated and uses a separate computing device for performance data collection and aggregation, thereby allowing for high sampling rates with no perturbation of the system under test. In this work, we used a sampling frequency of 10 Hz per power rail, which is adequate for observing the phases within the experiment that occur at a frequency of two orders of magnitude greater.

The test system is an AMD A10-5800K APU 100W CPU containing four general purpose (x86) cores. The power per core is a factor or two less energy efficient than other intel and AMD architectures. Experimentation with a less energy-efficient processor provides an upper bound for the average Joules per operation. Other architectures should yield even better results as the core count increases without a similar increase in total socket power consumption.

With this setup, we sampled instantaneous power during application execution, checkpoint commit, restarts, compression and decompression. We observed measurements from a single node, running on all four cores, since prior work had shown that generally energy consumption stays uniform across an application's nodes [32]. In a different work [86], Mills et al. observed that the energy and power performance of the network and other devices do not vary significantly from run to run. Furthermore, the processor and memory constitutes the majority of the energy consumption. Therefore, we only accounted for the power performance of the processor and memory subsystem to calculate the total energy consumption of each operation.

### 5.2.2 Performance Models

#### Checkpoint Compression Viability Model

Checkpoint data compression is a viable approach when its benefits outweigh its costs. Our checkpoint compression viability model is inspired by the one presented in Plank et al.[101]. Plank et al. focused solely on the impact of compression for the checkpoint commit phase. Our model accounts for the cost and benefits of compression for both the checkpoint and recovery phases.

We assume coordinated CR (cCR) in which all processes of a distributed application explicitly or implicitly coordinate at the beginning of each checkpoint interval, so as to commit a globally consistent application state composed of one checkpoint per process[1]. cCR currently dominates CR protocols used in HPC practice.

We also assumed an equal number of checkpoint and recovery operations. Our justification for this assumption follows: Ideally, an application will only take a checkpoint, immediately before an imminent failure so that both the overhead of checkpoint/restart and the amount of lost work is minimal. Therefore, the optimal checkpointing protocol will average a single checkpoint before each failure and only needs to recover once per failure. In fact, Daly demonstrates that when, $\delta$, the time to take a checkpoint is small compared to, $M$, the mean time between failures, $M$ is a very good approximation for $\tau_{opt}$, the optimal checkpoint interval [25]. Hence, in the optimal case, the number of checkpoints equals the number of failures, which also equals the number of recoveries. Several works define optimal checkpoint intervals [25, 17]. Finally, we assumed that checkpoint commit is synchronous; that is, that the primary application process is paused during the commit operation and is

[1]We can coarsely approximate the performance of uncoordinated CR by adjusting our model parameters to reflect different commit and recovery costs due to independent local checkpoints and local recovery protocols.

not resumed until the checkpoint commit is complete.

Checkpoint compression is viable when the time to compress and commit a checkpoint and the time to read and decompress that checkpoint is less than the time to commit and read the uncompressed checkpoint. Assuming the read and write transfer rates are equal:

$$t_{comp} + 2t_{cc} + t_{decomp} < 2t_{uc}$$

where $t_{comp}$ is *compression latency*, $t_{decomp}$ is *decompression latency*, $t_{cc}$ is the *time to read or write the compressed checkpoint* and $t_{uc}$ is the *time to read or write the uncompressed checkpoint*. This expression can be rewritten as:

$$\frac{c}{r_{comp}} + \left(2 \times \frac{(1-\alpha) \times c}{r_{commit}}\right) + \frac{c}{r_{decomp}} < 2 \times \frac{c}{r_{commit}}$$

where $c$ is the size of the original checkpoint, *compression factor* $\alpha$ is the percentage reduction due to data compression, $r_{comp}$ is *compression speed* or the rate of data compression, $r_{decomp}$ is *decompression speed*, and $r_{commit}$ is *commit speed* or the rate of checkpoint commit or reading (including all associated overheads). The last equation can be reduced to:

$$\frac{2\alpha \times r_{comp} \times r_{decomp}}{r_{comp} + r_{decomp}} > r_{commit} \tag{5.1}$$

Equation 5.1 defines the minimal ratio between checkpoint commit rate, compression rate, decompression rate and compression factor in order for the overall time savings of checkpoint compression to outweigh its costs. Of course, checkpoint compression has the additional benefit of saving storage space, but we do not factor that into our model.

### Application Efficiency Performance Model

*Application efficiency* is the ratio of an application's time to solution when the application is using some fault-tolerance mechanism to recover from failures as they occur to the application's time to solution assuming perfect conditions, that is, no failures and, therefore, no need to employ any fault-tolerance mechanisms. In the context of CR protocols, the higher an application's efficiency, the greater the time spent executing the application's intended computation and the less the time spent taking checkpoints, recovering from failures or re-doing computations lost due to failures.

**Modeling Checkpoint Compression** Daly's higher order model [25], which assumes node failures are independent and exponentially distributed, takes as input the system MTBF, the checkpoint commit time, the checkpoint restart time, the number of nodes used in the application and the time for the application's execution time in a failure-free environment. We used this model with integrated checkpoint compression and decompression. Checkpoint commit times include the time to compress the checkpoint data and the time to write this compressed data to stable storage. Similarly, restart times include the time to read the compressed checkpoint data from stable storage and perform the decompression step.

**Modeling Incremental Checkpointing** We also added support for incremental checkpointing into Daly's performance model. The model takes two additional parameters. The first new parameter specifies the size ratio of an incremental checkpoint to a full checkpoint. We assume that approximately the same fraction of the address space changes between each checkpoint. This assumption is based on the results of a previous incremental checkpointing study [37].

The second new parameter, the number of incremental checkpoints taken before taking the next full checkpoint, reflects the periodic need to take full checkpoints.

Increased recovery latencies and increased storage costs are two factors that motivate periodic full checkpoints. If an application fails and is recovered from the $i^{th}$ incremental checkpoint after a full checkpoint, additional overhead is required to either coalesce the full checkpoint and the $i$ increments, or to recover the full checkpoint and iteratively recover the state in each increment. Incremental checkpointing necessarily increases storage costs since it requires maintaining a full checkpoint as well as subsequent increments. If each increment is on average $1/s$ the size of the full checkpoint, after $s$ increments, storage costs would have doubled. We use Naksinehaboon et al.'s derivation of the optimal number of increments $n$ between two full checkpoints as: $n = \left\lceil 4c/5r_{commit} - 1 \right\rceil$, where $c$ is the size of a full checkpoint and $r_{commit}$ is the rate a file can be committed to stable storage [92].

For simplicity, we assume that taking incremental checkpoints and reconstructing a checkpoint from the increments do not incur additional costs. There are a number of techniques, such as concurrent coalescing, that make this assumption reasonable. Additionally, we assume that checkpoint increments have similar compression ratios as that of the full checkpoints. This assumption has been validated using the incremental checkpointing library described in [37].

**Other Assumptions** Apart from the empirically observed data we used to parameterize our performance models, we assumed that each process uses 2 GB of memory (based on observed workloads at Sandia National Laboratories) and checkpoints $\frac{1}{3}$ of that memory [37], a five year node MTBF [114] and a per process I/O rate of 1 MB/s. This latter value was optimistically chosen, based on a performance study on Argonne National Laboratory's 557 TFlop Blue Gene/P system (Intrepid) [71].

### Modeling System Performance considering Costs

In our comparison of checkpoint data compression optimizations to hardware-based SSD solutions, we considered the relative financial costs of different system configurations. This study was meant to be instructive, not necessarily definitive, thereby allowing us to make simplifying assumptions and to use a relatively simple cost model. Using system cost factoring in the replacement of worn SSDs and the amount of work completed in a fixed time span, based on the system's hardware and software configuration, we created a performance-price model.

**System Cost Model** Unlike traditional storage technologies, SSDs suffer a wear or *endurance* problem: SSDs have an *endurance number* that specifies the number of write/erase cycles can occur before the device wears out. To compute the final procurement cost of an SSD-based system, we computed the number of weeks between SSD replacement based on their lifespan write capability and the average weekly checkpoint data commitment:

$$lifespan_{ssd}(weeks) = \frac{ssd\_lifespan\_write\_capability}{weekly\_checkpoint\_volume} \tag{5.2}$$

where

$$ssd\_lifespan\_write\_capability = SSD\ capacity \times SSD\ endurance\ number$$

and

$$weekly\_checkpoint\_volume = number\ of\ weekly\ checkpoints \times checkpoint\ size.$$

We can now compute $tcost_{node}$, the total per node procurement cost, as:

$$tcost_{node} = cost_{node} + \left( cost_{ssd} \times \left\lceil \frac{lifespan_{system}(weeks)}{lifespan_{ssd}(weeks)} \right\rceil \right) \tag{5.3}$$

where $cost_{node}$ is the cost of a node without SSD devices, $cost_{ssd}$ is the per node cost of new SSDs, and $lifespan_{system}(weeks)$ is the overall lifespan of the system

in weeks. We assume that only checkpoint data is written to the SSD devices and that they wear uniformly and according to their specifications. Several studies have shown that these devices can wear out as much as ten to 30 times faster than the specified rating of the device [125]. As a result, our model is optimistic since SSD devices may need to be replaced more often.

Also, we only considered procurement costs and ignored ongoing costs to run and maintain the system. Effectively, we are assuming that any differences in expenses for running and maintaining systems with different configurations are negligible.

**A Performance-price Model** For a given system lifespan and different system configurations, our performance-price model calculates the work per dollar ratio using the amount of application work completed given the application's efficiency based on the effectiveness of its fault-tolerance mechanisms and the system's cost:

$$Performance\_price = \frac{work \times efficiency}{t_{cost}(node) \times number\ of\ nodes} \tag{5.4}$$

We assume 100% system utilization throughout its lifetime. Application efficiency under various fault-tolerance configurations (including optimizations) will determine how much useful work is achieved within the five-year period.

### A Coarse-grained Checkpoint Compression Energy Model

To study the energy impact of our compression-based CR optimization, we used a coarse-grained energy measurement approach. We decompose a CR-based application into three phases: running the application, taking a checkpoint, and restarting from a checkpoint. (We do not distinguish executing application code during normal operation from re-executing application code after rollback due to a failure.) Therefore, the energy, $E$, consumed during an application's execution can be modeled

simply as:

$$E = E_{app} + E_{ckpt} + E_{rst}, \tag{5.5}$$

where $E_{app}$ is the energy expended on running the application's code, $E_{ckpt}$ is the energy expended from taking checkpoints, and $E_{rst}$ is the energy expended during restarts. We consider each phase as a blackbox unit and don't address finer details. For example, when taking a checkpoint, we consider the energy consumption due to a checkpoint as a single cost. In contrast, a finer grained approach may further decompose the checkpoint phase into finer sub-tasks, like inter-process coordination, calculating the portion of the address space to checkpoint and committing the checkpoint to stable storage. We hypothesize that this coarse-grained approach is sufficiently accurate for modeling an application's energy consumptions and that we do not need such finer grained details.

Our coarse-grained approach assumes for each phase that we can use a simple computation based on average power and time to estimate the energy consumed by that phase:

$$E_{app} = T_{app} \times \bar{P}_{app}, \tag{5.6}$$

where $T_{app}$ is the time spent executing the application, including normal execution and rework, and $\bar{P}_{app}$ is the average power during application execution.

$$E_{ckpt} = T_{ckpt} \times \bar{P}_{ckpt}, \tag{5.7}$$

where $T_{ckpt}$ is the time spent checkpointing, and $\bar{P}_{ckpt}$ is the average power during the checkpoint phase.

$$E_{rst} = T_{rst} \times \bar{P}_{rst}, \tag{5.8}$$

where $T_{rst}$ is the time spent restarting from a failure, and $\bar{P}_{rst}$ is the average power during a restart. Furthermore, $T_{ckpt}$ is the number of checkpoints times the per checkpoint latency, and $T_{rst}$ is the number of failures times the per restart latency.

We empirically measured the values of $\bar{P}_{app}$, $\bar{P}_{ckpt}$ and $\bar{P}_{rst}$. Using Daly's equation [25], we determined the number of checkpoints taken, the number of occurring failures and the amount of rework time. Daly's equation assumes that node failures are independent and exponentially distributed, and calculates time as optimal checkpoint intervals. In general, HPC users are more concerned about application runtime and finishing applications faster rather than about their energy costs. Hence, we didn't use the energy optimal checkpoint interval as proposed by Meneses et al [84]. To add the checkpoint compression optimization to the model, we empirically measured the average power for checkpoint compression and decompression and multiply those values by the total time spent compressing and decompressing checkpoints, respectively, and to add those additional energy costs to the others.

Using this approach, we can model any application workload, failure rate, checkpoint commit rate, compression performance, etc. and thereby estimate the total energy costs for that application run. Similarly, we can profile application energy consumption with other CR optimizations – only changes in the costs of checkpoint and restarts must be accounted for. We validated our model, as shown in Section 5.6.1.

## 5.3 Checkpoint Compression Performance

### 5.3.1 Checkpoint Compression Viability

To test the viability of compression, we only focused on problem sizes that allowed each application to run long enough to generate 5 checkpoints. The three implicit finite element mini apps, HPCCG, pHPCCG and miniFE were each given a 100x100x100 problem size. Alternatively, phdMesh and LAMMPS were each given a 5x5x5 problem size. Each application was run using 2–4 MPI processes, except for

| Parameter | Description |
|---|---|
| $E$ | Total application energy consumed |
| $E_{app}$ | Energy consumption for running the application code. |
| $E_{ckpt}$ | Energy expenditure due to checkpoint operation. |
| $E_{rst}$ | Energy expenditure due to restarts. |
| $T_{app}$ | Time spent execute application code. |
| $T_{ckpt}$ | Time spent taking checkpoints. |
| $T_{rst}$ | Time spent restarting from failures. |
| $\bar{P}_{app}$ | Average power for executing application code. |
| $\bar{P}_{ckpt}$ | Average power during checkpoint operation. |
| $\bar{P}_{rst}$ | Average power for restart. |

Table 5.1: A summary of our coarse-grained energy model parameters.

phdMesh, which was run without MPI support. Checkpoint intervals for miniFE, pHPCCG, HPCCG and LAMMPS were 3, 3, 5 and 60 seconds, respectively. For phdMesh the 5 checkpoints were taken at simulation time step boundaries. BLCR was used to collect all checkpoints. The per-process checkpoint sizes were approximately 93MB for miniMD, and 311 MB to 393 MB for the rest of the mini apps. For the two production applications, checkpoint sizes were approximately 80 MB for LAMMPS and 25MB to 30MB for CTH.

We used compression factor, $\alpha$, as our metric to show how compressible checkpoint data are, where we compute compression factor as: $1 - \frac{compressed\ size}{uncompressed\ size}$.

Figure 5.3.1 shows how effective the various algorithms are at compressing check-

point data. We can see that both the algorithms achieve a very high *compression factor* of about 80% or higher for all our test apps and *mini apps* except for miniFE which was between 68-80%. This means that the primary distinguishing factor becomes the compression speed, that is, how quickly the algorithms can compress the checkpoint data.

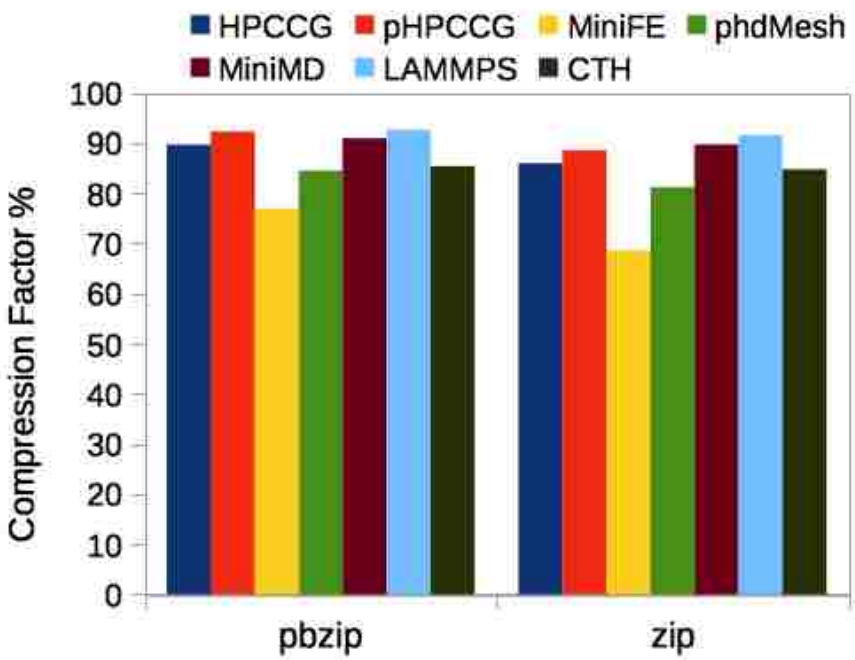
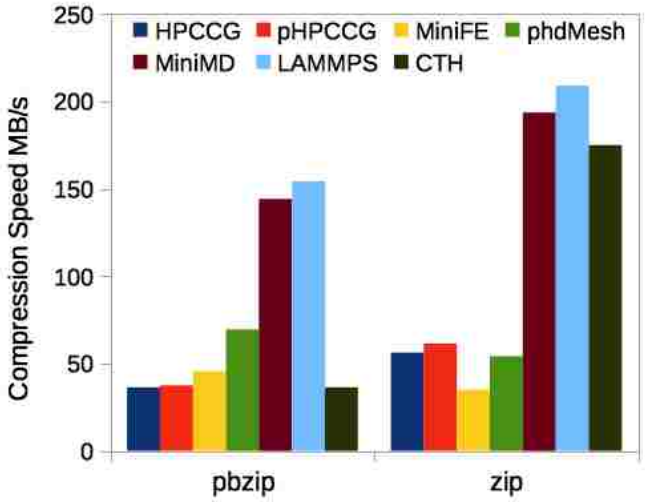
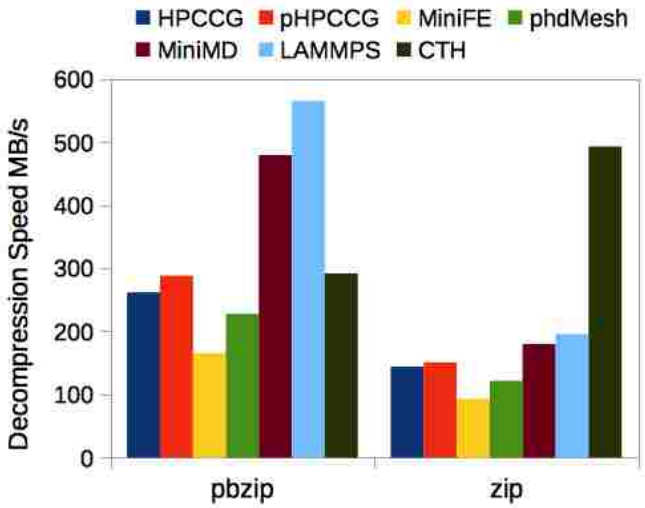



Figure 5.2: Checkpoint Compression Factors, higher is better: a factor of 90% means that the file size was reduced by 90%.

Figures 5.3(a) and 5.3(b) show our observed compression and decompression speeds, respectively. In general, and not surprisingly, the parallel implementation of `bzip2`, `pbzip2`, generally outperforms all the other algorithms. Decompression is a much faster operation than compression, because during the compression phase we must search for compression opportunities, while during decompression, we simply are using a dictionary or lookup table to expand compressed items.




(a) Compression Speed




(b) Decompression Speed

Figure 5.3: Checkpoint Compression/Decompression Speeds. Higher is better

Based on the above results and Equation 5.1, which represents our viability model, Figure 5.4 demonstrates the checkpoint read/write bandwidths that make compression viable. For each application, the lowest bar of the two compression algorithms represents its worst case scenario. For the worst-case application, miniFE, checkpoint compression applying zip is viable unless a system can sustain a per process

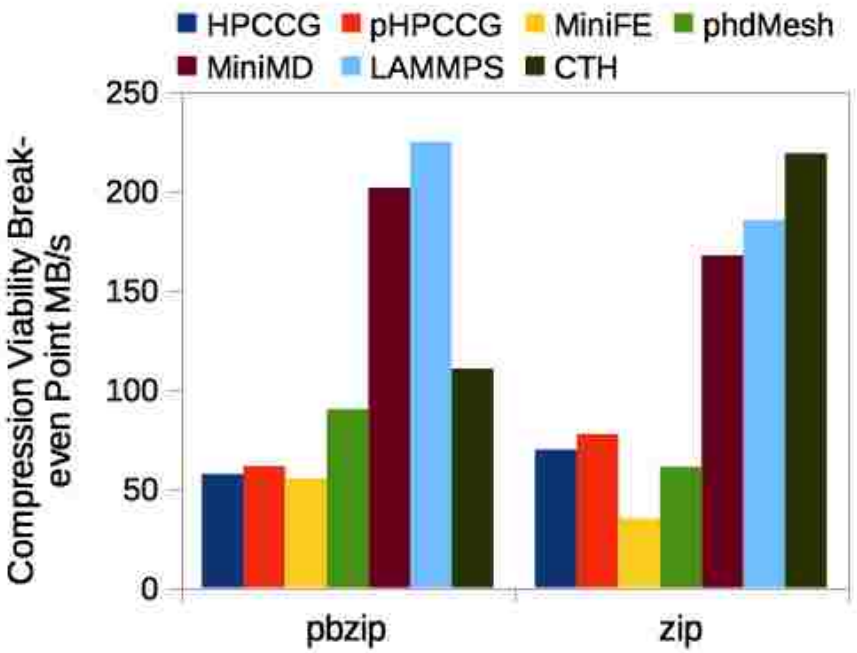



Figure 5.4: Checkpoint Compression Viability: Unless, checkpoint read/write bandwidth exceeds our viability factor (y-axis), checkpoint compression should be used.

checkpoint read/write bandwidth of greater than about 34.8 MB/s. In the best case, LAMMPS, the necessary per process checkpoint read/write bandwidth increases to greater than about 224.5 MB/s.

The relationship between compression performance (compression factor and compression and decompression speeds) and checkpoint I/O bandwidth is the key factor in the viability of checkpoint compression. As Figure 5.4 shows, for our worst-case application, miniFE with zip compression, compression is viable if per-process checkpoint bandwidths are less than 34.8 MB/s. In the best case, LAMMPS with pbzip2 compression, per process checkpoint bandwidths must exceed 224.5 MB/s. To compare this against real world systems, we use a report based on a study of I/O performance on Argonne National Laboratory's 557 TFlop Blue Gene/P system (Intrepid) [71]. This work executes an I/O scaling study measuring maximum

achieved throughput for carefully selected read and write patterns. From this report, the best observable per process I/O bandwidths were 1 MB/s for both reading and writing. This performance scales to about 32,768 processes and then it decreases. For example, at 131,072 processes, per process read bandwidth is 385 KB/s and per process write bandwidth is 328 KB/s. The Oak Ridge Cray XT5 Jaguar petascale system has peak per-node and per-core checkpoint bandwidths of 5.3 MB/s and 1 MB/s, respectively, three orders of magnitude less than what is needed. Similarly, the Lawrence Livermore Dawn IBM BG/P system has a peak per-node checkpoint bandwidth of about 2 MB/s [2] As a result, aggressive use of checkpoint compression appears to be viable and indeed desirable on current large-scale platforms.

**Checkpoint Compression Performance for Mini versus Real Apps** It is legitimate to wonder whether for this study the mini apps serve as reasonable proxies of their full application counterparts. To help address this concern, we observed that the mini app miniMD is meant to correspond to the full application LAMMPS. The results from Figures 5.3 and 5.4 are tabulated in Table 5.2. From this data, we see that the compression performance and viability points for miniMD and LAMMPS are very comparable: the compression viability bandwidth for miniMD is within 10% of that of LAMMPS. While this single result is not conclusive for all the mini apps, it suggests that it is not unreasonable to consider the mini apps to be suitable stand-ins for full applications for this study.

[2]Oak Ridge's Spider Lustre-based file system provides 240 GB/s of aggregate bandwidth[118], while Dawn's Lustre file system is listed as providing 70 GB/sec of peak bandwidth on LLNL reference pages [9].

| | Compression Factor % | | Compression Speed MB/s | | Decompression Speed MB/s | | Compression Viability Break-even point MB/s | |
|---|---|---|---|---|---|---|---|---|
| | pbzip | zip | pbzip | zip | pbzip | zip | pbzip | zip |
| LAMMPS | 92.7 | 91.6 | 154.2 | 209.1 | 564.9 | 195.5 | 224.6 | 185.2 |
| miniMD | 91 | 89.7 | 144 | 193.6 | 478.8 | 179.9 | 201.5 | 167.4 |

Table 5.2: Checkpoint compression performance similarity for miniMD and LAMMPS, solving same problem. The parameters for pbzip are (1,5) and for zip it is (1).

### 5.3.2 Compressing System-level versus Application-level Checkpoints

Next, we examined the compression effectiveness of system-level checkpoints compared to that of application specific checkpoints. A number of scientific applications provide their own mechanisms to save their checkpoints and can restart the problem from those saved states. We would like to verify whether checkpoint compression is effective for application generated checkpoints as well. We use LAMMPS for this testing due to its optimized, application-specific checkpointing mechanism described in the previous section. For these tests we compare application generated restart files with those generated by BLCR. In each case, we take 5 checkpoints that are equally spaced throughout the application run.

System-level checkpointing saves a snapshot of the application context such that it can be restarted where it left off by capturing application specific data, shared library states etc. On the other hand, application specific checkpointing only needs to save the data that is necessary to resume the operation. As a result, for a fixed problem, system level checkpoints are typically much larger in size. In our tests, LAMMPS' application specific checkpoints were 12MB in size compared to about 80MB BLCR generated checkpoints for the same problem. However, based on our

results in Table 5.3, we observe that checkpoint compression is viable for both application specific and system-level checkpoints.

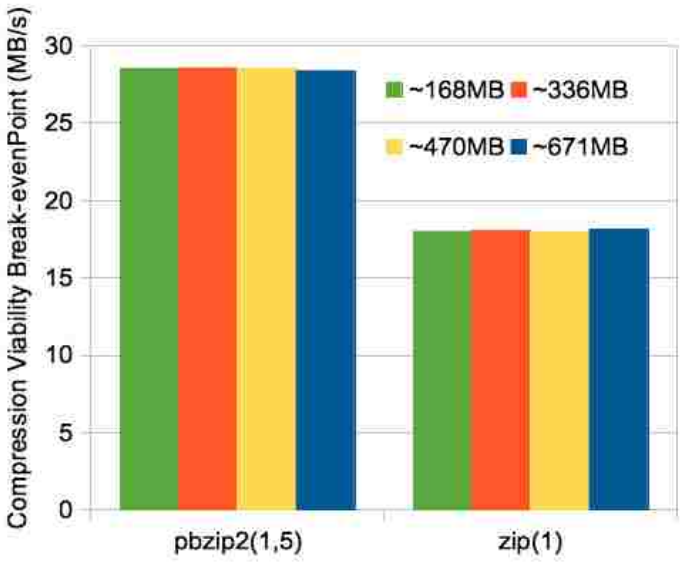
There is, however, a qualitative difference in the break-even points for checkpoint compression. Our data reveals that the major reason is that, system-level checkpoints compressed better than user-level checkpoints (for example, pbzip2 compression factors are 92.7% compared to 43%). This is because application-level checkpoints are optimized by omitting data that can be reconstructed when an application restarts. This reduces the compressibility of the application level checkpoints. For the same reason, we observed the differences in sizes for these two types of checkpoints. Moreover, the average compression and decompression speeds were higher for system-level checkpoints than for user-level checkpoints (again for pbzip2, 154.2 MB/s compared to 37 MB/s).

| | Compression Factor % | | Compression Speed MB/s | | Decompression Speed MB/s | | Compression Viability Break-even point MB/s | |
|---|---|---|---|---|---|---|---|---|
| | pbzip | zip | pbzip | zip | pbzip | zip | pbzip | zip |
| **System** | 92.7 | 91.6 | 154.2 | 209.1 | 564.9 | 195.5 | 224.6 | 185.2 |
| **Application** | 43 | 41.9 | 37.2 | 26.7 | 104.5 | 97.1 | 23.6 | 17.6 |

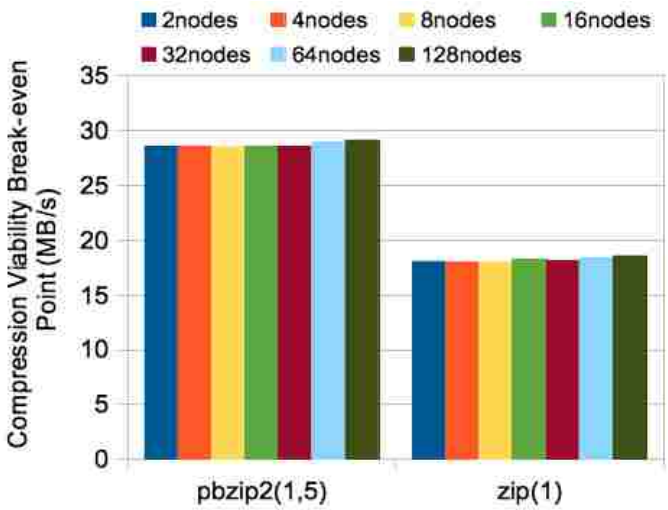
Table 5.3: Compression Break-even Points for system-level and Application Specific Checkpoints. The parameters for pbzip are (1,5) and for zip it is (1)

### 5.3.3 Checkpoint Compression Performance and Application Scale

For our scaling experiments, we use LAMMPS with its built-in checkpoint mechanism. We observe how checkpoint viability scales with (1) memory size; (2) time (between checkpoints); and (3) process counts.




(a) Scaling Checkpoint Sizes and Application Runtime.




(b) Scaling Process Counts.

Figure 5.5: Results from our Scaling Experiments.

In our first set of scaling experiments, we evaluated the first two scaling dimensions, checkpoint size and time between checkpoints. We progressively increased the LAMMPS problem size while keeping the number of application processes fixed at two. In this manner, memory footprint and checkpoint sizes increased. This also means that the application runs for a longer time, since the per process workload has been increased. For each LAMMPS process, five checkpoints were taken uniformly throughout the application run. The checkpoints we collected from these tests averaged about 168MB, 336MB, 470MB and 671MB for the various problem sizes. Figure 5.5(a) shows the viability results from these experiments which demonstrates that in no case did checkpoint size show any impact on the viability of checkpoint compression for LAMMPS.

For the study of scaling in terms of process count, we compared the compression ratios when weak scaling LAMMPS EAM simulations from 2 to 128 MPI processes. In each test, the per-process restart file size was over 170 MB taken using LAMMPS' built in application checkpointing mechanism. In these runs we took 5 equally spaced checkpoints.

Figure 5.5(b) shows once again that application process counts did not impact checkpoint compression viability. Since compression and decompression (in the case of failures) are performed on the checkpointing or recovery node, increasing scale does not increase compression/decompression and application resource contention. Additionally, we have no reason to believe that compression performance results will be different for larger process count runs. Our rationale is as follows: checkpoint compression performance (compression speed, decompression speed and compression factor) is a function of the "data features" of the memory footprint, such as, for example, data types and values. For typical scientific computing applications, we do not expect these data features to vary among a given application's processes even as

scale grows.

## 5.4 Understanding Checkpoint Compression Performance

Given the viability results from the previous section which show that checkpoint data compression can yield significant improvements in application performance, a natural question is whether further improvements to checkpoint compression can render even more benefits. We answered these questions by performing studies that allow us to evaluate the performance impact of compression factor and compression speed.

### 5.4.1 The Impact of Compression Factor

Checkpoint data volume reduction is arguably the most significant user-controllable factor that impacts checkpoint-restart performance. Therefore, it is important to examine the limits of checkpoint data volume reduction through compression. A related question is whether it is worth considering compression algorithms that specifically target checkpoint data. We provided novel insights into these questions by using information theory to explore the compression performance of off-the-shelf utilities and to evaluate the additional impact of a hypothetical, custom algorithm that achieves optimal compression. For this discussion, we use the metric *compression factor* which, as we previously defined, is the inverse of the compression ratio; therefore higher compression factors are better.

### An Application-specific Case Study

Based on the compression performance results from Section 5.3.1, we focused on checkpoint/restart for the LAMMPS application. LAMMPS exhibits the poorest checkpoint compressibility and, hypothetically, the greatest opportunity for improvement for all the applications tested. We used knowledge of the LAMMPS on-disk checkpoint format to translate application-specific checkpoint data into its composite data elements. Using this, we computed the entropy of LAMMPS checkpoints using Shannon's entropy calculation [117].

Shannon's theorem gives us the minimal number of bits needed to represent a certain amount of information. Using our understanding of the LAMMPS' checkpoint format, we calculated a frequency distribution for the values in the checkpoint file. We calculated this distribution in a representation independent way; for example, the double 0.0 is interpreted to be the same value as the integer 0, because they contain the same information. Using this frequency distribution, we then calculated the entropy of this newly-created "checkpoint language" for LAMMPS' checkpoints. This entropy calculation gives us a minimal encoding.

Table 5.4.1 shows the results of this minimal checkpoint encoding. This checkpoint contained about 3.5 million total symbols of which about 1 million were unique, thereby resulting in an entropy of 10.59 or a theoretically maximal compression factor of 79.5%. Comparatively, our bzip2-encoded strings for the same checkpoint (excluding the bzip2 dictionary and headers, as we do not include this information in the entropy calculation above) had a compression factor of 67.6%, a significant difference in compression performance. Therefore, a hypothetical optimal checkpoint compression algorithm tailored specifically for the information contained within it will compress the checkpoint to 20% of its original size, in comparison to bzip2, which compressed the checkpoint to 32%.

| Total Symbols | Unique Symbols | Entropy | Optimal Compression Factor | Bzip Compression Factor |
|---|---|---|---|---|
| 3,584,043 | 1,023,367 | 10.59 | 79.5% | 67.6% |

Table 5.4: Comparing a theoretical minimal encoding with bzip2.

Next, we used this LAMMPS checkpoint compression comparison data to model how LAMMPS performance would improve with this optimal algorithm that could better compress its checkpoints. Based on gathered experimental data used to determine the better algorithms and algorithm parameterization, we observed that an improved compression factor generally results in slower compression/decompression rates. For this study, we optimistically assume that the hypothetical optimal algorithm would not experience such slowdowns. Furthermore, we optimistically assumed that the optimal algorithm could run as quickly as the parallel bzip2 algorithm, the algorithm with the best observed compression/decompression rates across all of our experiments.

We looked at three different scenarios, systems with 10K, 50K and 100K total sockets. Figure 5.6 shows the impact on application efficiency as compression factors varied, highlighting our observed compression factor and our theoretic maximum compression factors. For each of the three scenarios, we observed that optimal compression would yield a relatively small increases in application efficiency, the largest being an additional 7.2% of efficiency in the 100K socket scenario. Therefore, we conclude that exploring checkpoint-specific compression algorithms is unlikely to yield significant improvement over the standard text-based compression algorithms. In fact, with the expected growth of I/O bandwidth on future systems, these differences in efficiencies will continue to decrease, thus supporting our position that current compression algorithms are sufficient for future systems as well.

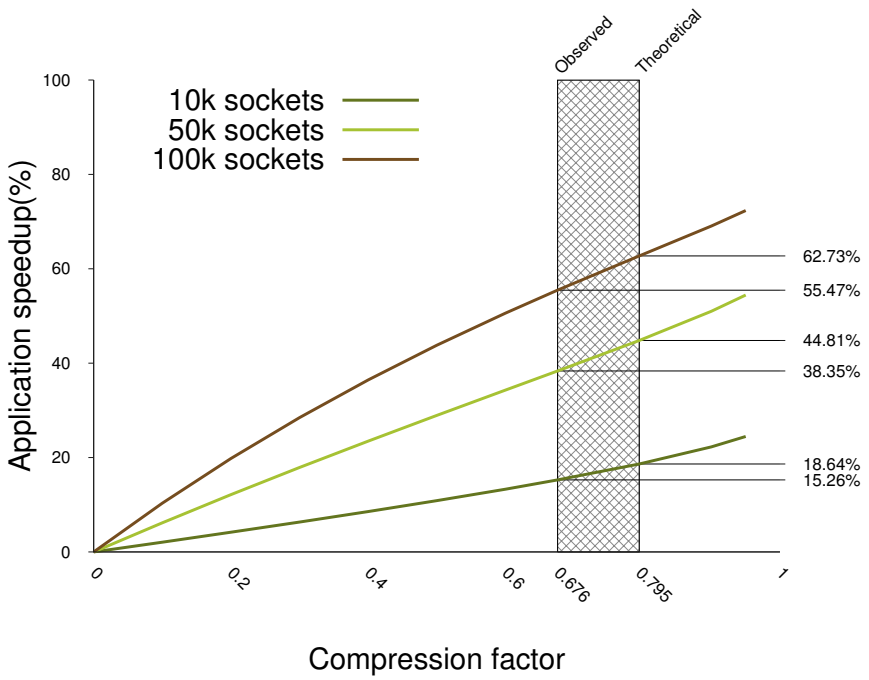



Figure 5.6: Varying compression factor

### 5.4.2 The Impact of Compression Speed

While compression factor is probably the biggest determinant of the performance impact of checkpoint compression, it is also necessary to understand the importance of compression speed. We evaluated the potential benefits of accelerating our top performing algorithm (in terms of compression factors), for example, by using algorithmic enhancements or hardware technologies like GPUs. Using the compression performance exhibited by pbzip2 on phpccg checkpoints (our top performer for compression factor) as a baseline and the application efficiency performance model from Section 5.2.2, we varied compression and decompression rates from a slow-down of 100 to a speed-up of 10,000. This allows us to explore how application efficiency varies with compression/decompression rates.

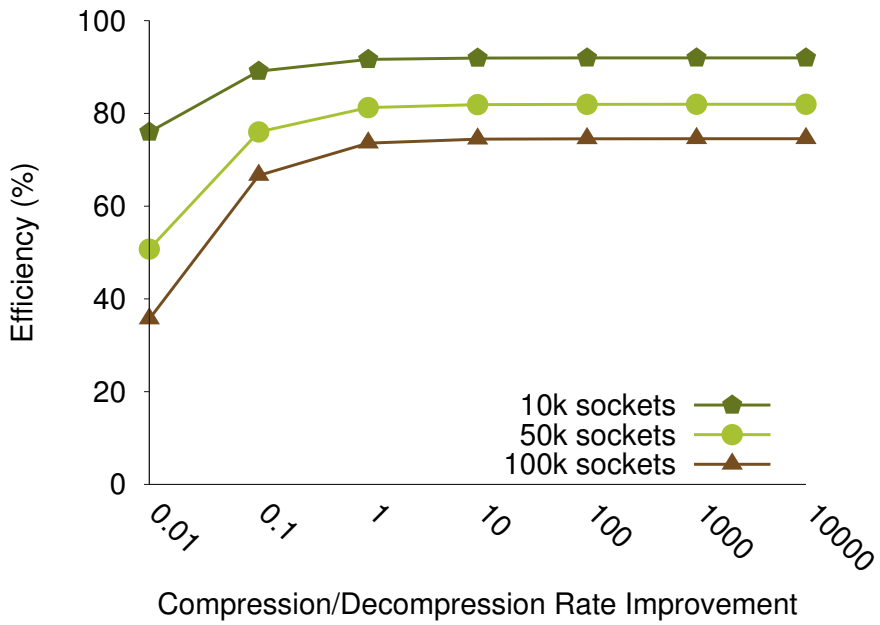



Figure 5.7: Varying compression/decompression speed

The results, depicted in Figure 5.7, show that four orders of magnitude improvement in speed would yield only an insignificant improvement in application efficiency on current systems. While this is an important result, it is not so surprising: Given the current checkpoint commit rates (based on available per process I/O bandwidth to checkpoint storage), the time spent compressing a checkpoint is insignificant when compared to the time spent committing the checkpoint to stable storage. These results suggest that attempting to improve compression rates is not worthwhile as long as our platforms checkpoint commit bandwidths remain less than the CPU viability bandwidths from the previous section. For the vast majority of current leadership-class capability machines, the CPU viability bandwidth is dramatically higher than that of the per-process checkpoint commit bandwidth.

What remains unclear is the impact of increasing compression speed as the I/O bandwidth increases, which is expected in future systems. Figure 5.8 shows the

increase in application efficiency as a function of the per-node checkpoint commit bandwidth. Once again, we used the application efficiency performance model from Section 5.2.2. Similar to previous work in this manuscript, we assumed a 5 year socket MTBF and use optimal compression factors. The Y-axis in this figure is the difference in application efficiency between the accelerated and non-accelerated case. For the accelerated case, we assumed a hypothetical compression of 100 times the CPU compression speeds. These optimal speedups have been observed with carefully crafted codes and workloads with GPUs [20]. We modeled these overheads for a number of node counts between 10k and 200k. From this figure, we can see that a two order magnitude increase in compression/decompression speeds leads to only marginal increases in application efficiency. This result suggests that the effort involved in accelerating compression/decompression speeds may not be worth the performance return.

## 5.5 Checkpoint Compression and Other Optimizations

Finally, we put the performance of checkpoint compression in context by comparing it against a number of popular software, hardware, and mixed hardware/software solutions. Also, we investigated the performance of scenarios where checkpoint compression can be combined with these techniques. We compare checkpoint compression performance against a software-only, incremental checkpointing solution, showing the performance of the combination of incremental checkpointing with compression. We used the best-observed checkpoint compression performance, specifically, pbzip compression with LAMMPS checkpoints, which yielded about 92.7% compression factor, 154.2 MB/s compression rate and 564.9 MB/s decompression rate. For incremental checkpointing, we used an 80% compression factor, the optimal incremental

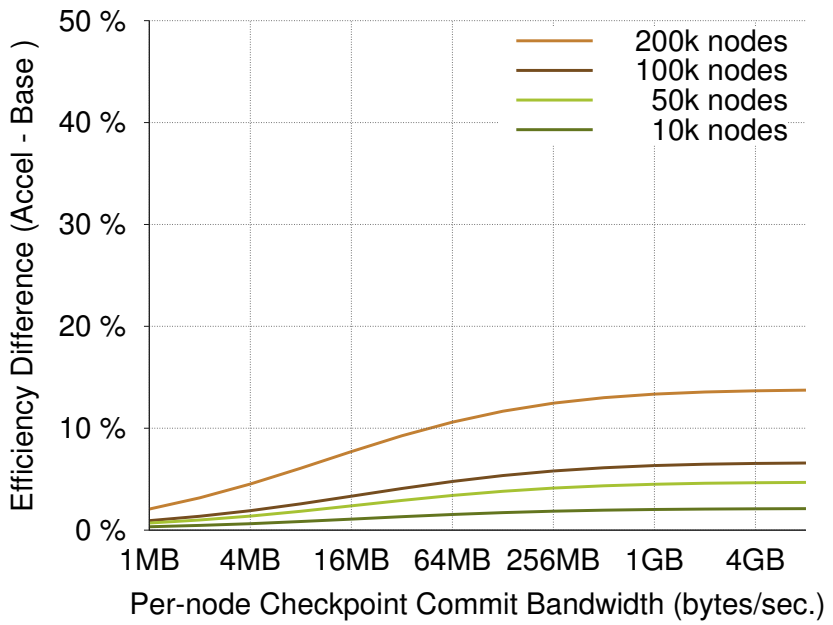



Figure 5.8: Efficiency increases for a number of node counts as a function of per-node checkpoint commit speeds assuming that a compression/decompression speed is a factor of 100 greater than what we see on current systems. The efficiency difference is defined as the accelerated efficiency minus the efficiency using current speeds

checkpointing compression found in [36]. As before, we assumed that incremental checkpoints have similar compression ratios as the standard full checkpoints. This assumption has been validated previously in [36]. We then compared these software-only checkpointing solutions against state-of-the-art and considerably more costly hardware-based solutions: checkpointing to SSDs (solid-state device) and the multi-level checkpointing solution Scalable Checkpoint Restart (SCR) [87].

### 5.5.1 Compression and Increment-based Optimizations

Figure 5.9 shows the comparison results of compression-based and increment-based scenarios alongside the standard cCR performance. We make several observations:

1. Unsurprisingly, all combinations of compression-based and increment-based optimizations outperform standard coordinated checkpointing (labeled "baseline" in the figure).

2. Compression yields greater application efficiency than pure, optimal incremental checkpointing (labeled "ickpt"). This result is notable in that our model does not include the potentially high-overhead of the mechanisms used in incremental checkpoints to detect updated memory regions or introspective application knowledge. So in environments where this overhead is prohibitively excessive or application characteristics are unknown, checkpoint compression is a simple solution that can achieve better performance with no application programmer burden.

3. The combination of compression-based and increment-based optimizations yields the best performance of these software-only methods.

From these results, we conclude that checkpoint compression can lead to significant performance improvements for large-scale applications. Most importantly, this method can be combined with other checkpoint optimizations to further improve application efficiency.

### 5.5.2 Compression and Other Optimizations

Next, we compared our checkpoint compression technique against the performance of two hardware-based checkpoint optimizations. Specifically, we compared it against a local SSD checkpointing solution [63] and a multi-level solution(SCR) that uses local and remote memory, SSDs, a parallel file system, and a software RAID to ensure reliability [87]. It is important to note that these hardware checkpointing solutions are considerably more expensive than a software-only solution such as incremental and

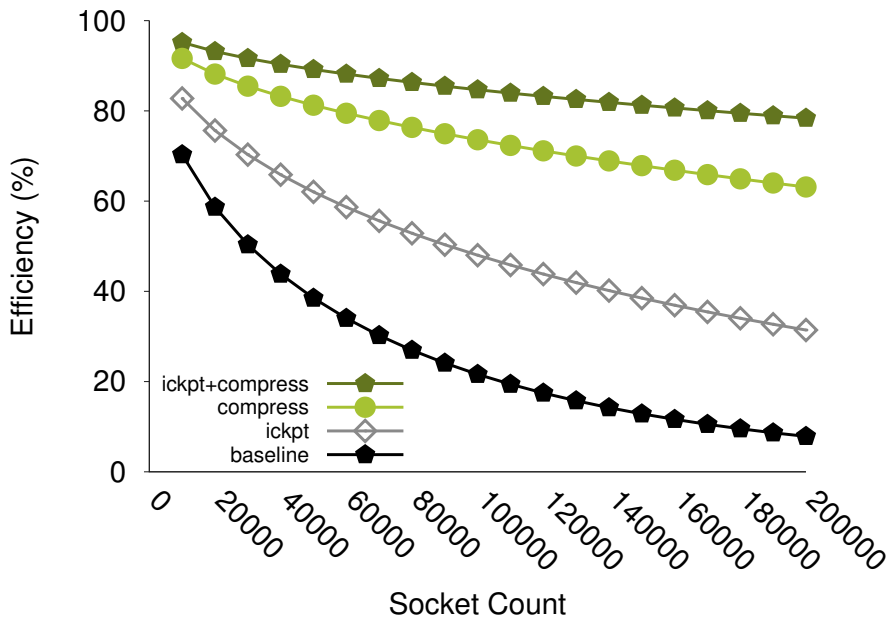



Figure 5.9: Impact of the software-only optimizations checkpoint compression and incremental checkpointing on application efficiency.

compression-based checkpointing. In fact, the device reliability required for the SSD only solution may be prohibitively expensive even at smaller scale as recent studies have shown that in 15% of failures, the checkpoint cannot be recovered from current SSD technology [87] and may require a highly reliable backing store like a parallel file system. Also, the SCR approach, in addition to using additional hardware, uses a portion of on-node memory to store checkpoints. This point is especially important for future extreme-scale systems. As core counts increase dramatically, we are moving from a compute-scarce environment to one where we have an abundance of compute cycles but a scarcity of memory.

Again, we assumed each process uses 2GB of memory and checkpoints $\frac{1}{3}$ of that memory. We also assumed a 5 year MTBF and a per-process I/O rate of 1MB/s for the compression and incremental checkpointing case. For the SSD-only case, we assumed a 2GB/s checkpoint commit rate and a 8GB/sec checkpoint read rate.

Lastly, for SCR, we assumed a per-process mean checkpoint commit rate of 211MB/s for both read and write. This mean commit rate is calculated from that used in another study [107], where the authors presented a user-space file system, CRUISE, which dramatically improved the performance of SCR. The take-away here is that the per-process checkpoint commit rates of these hardware-based solutions are several orders of magnitude larger than the software solutions.

Figure 5.10 shows a comparison of compression with the hardware-based techniques outlined in this section. For comparison, we also included the efficiency of standard rollback/recovery to the parallel file system shown previously. From this figure, we observe that:

1. Perhaps as expected, the hardware-based solutions perform significantly better than the software solutions
2. The SSD-only solution has nearly 100% efficiency through the socket counts tested, though as pointed out previously, recent work suggests this solution may not be achievable.
3. The multi-level checkpointing approach, which uses multiple levels of the system storage and can recover from all observed failures, performs similarly to an SSD-only approach.
4. The optimal software-only approach (ickpt+compress), though having a commit speed slower by two orders of magnitude, only performs 20% worse than the other approaches.

This set of results shows the benefit of this compression approach. With no application-specific knowledge, no additional hardware, minimal memory overhead, using standard and freely available compression algorithms, and using checkpoint commit bandwidths observed on today's systems, we can obtain efficiency within 20%

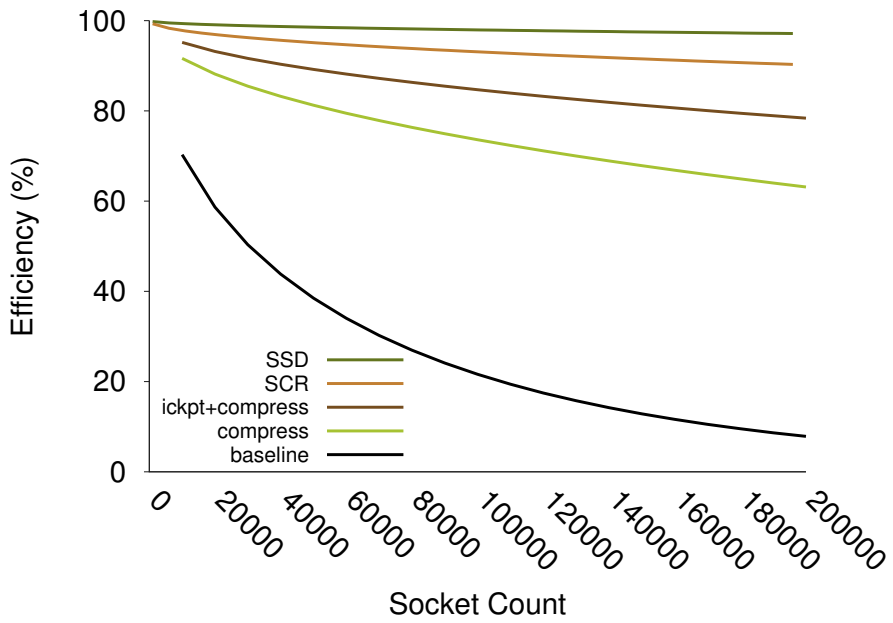



Figure 5.10: Comparison of hardware/multi-level checkpointing techniques with pure software techniques like compression and incremental checkpointing

of the costly hardware solution. Compression-based approaches can be made readily available to existing systems while hardware-based solutions require installation and other disruptive changes.

### 5.5.3 A Performance/Price Evaluation of SSD-based Systems

In this section, we examined the cost efficiency of these hardware-based, software-based, and hybrid CR optimization strategies. For this study, we computed and compared the performance-price ratio for a hypothetical cluster under different configurations that map to hardware-based CR optimizations, SSD-enhanced, and software-based CR optimizations, namely compression and incremental checkpointing. Recall

our performance-price model from Section 5.2.2:

$$Performance\_price = \frac{workload \times efficiency}{t_{cost}(node) \times number\ of\ nodes}$$

where

$$tcost_{node} = cost_{node} + \left( cost_{ssd} \times \left\lceil \frac{lifespan_{system}(weeks)}{lifespan_{ssd}(weeks)} \right\rceil \right)$$

and

$$lifespan_{ssd}(weeks) = \frac{ssd\_lifespan\_write\_capability}{weekly\_checkpoint\_volume}$$

and

$$ssd\_lifespan\_write\_capability = SSD\ capacity \times SSD\ endurance\ number$$

Our hypothetical cluster has 12,250 nodes, two sockets per node and eight cores per socket for a total of 16 cores per node. We assumed a system lifespan of 260 weeks (five years) and our workload comprises one process per core and executes for the entire 260 weeks. We used application efficiencies obtained from the results in the previous section: 90.94% efficiency for the SSD-based optimizations and 78.92% efficiency for the software-based optimizations.

We computed $ssd\_lifespan\_write\_capability$ for different SSD technologies, namely single layer cell (SLC), multi-level cell (MLC), and three-level cell (TLC), assuming 256 GB SSDs and the write endurance for specific device instances as shown in Table 5.5.

We computed the last column of Table 5.5, $lifespan_{ssd}(weeks)$, assuming that there is one 256 GB SSD per socket (per eight cores), that each process running on a core has 2 GB of memory available and each checkpoint is one-third of 2 GB, and used Daly's model to calculate the number of checkpoint commits to each SSD per week.

| Type | Name | Price(USD) | $E_{rating}$ | $E_{max}$ | lifespan(weeks) |
|---|---|---|---|---|---|
| TLC | Samsung 840Pro | $200 | 750 | 2,500 | 47.4 |
| MLC | OCZ Revo drive 3 3 | $460 | 3,000 | 10,000 | 189.5 |
| SLC | OCZ Z drive R2 | $4800 | 100,000 | 100,000 | 6,315 |

Table 5.5: Endurance ratings($E_{rating}$) and price for various SSDs. Also $E_{max}$ in this table represents maximum endurance.

Using the above method, Figure 5.11 shows the performance-price comparisons of the various hardware-based and software-based CR optimizations for a range of baseline node costs from $500 to $3000. We see that for lower baseline per-node costs a software based approach produces significantly more units of work per dollar. However, as the node prices increase, SSD cost overheads are amortized such that the hardware-based solutions become almost as cost-efficient as the software based one.

## 5.6 Energy impacts of checkpoint compression

In this section, we answer two primary questions: (1) *What is the accuracy of a coarse-grained approach to measuring and modeling the energy performance of CR protocols?* and (2) *Does checkpoint compression lead to an increased or decreased energy expenditure?* We now address these questions, first presenting the validation of our coarse-grained model and then using the model to predict the performance of CR using compression.

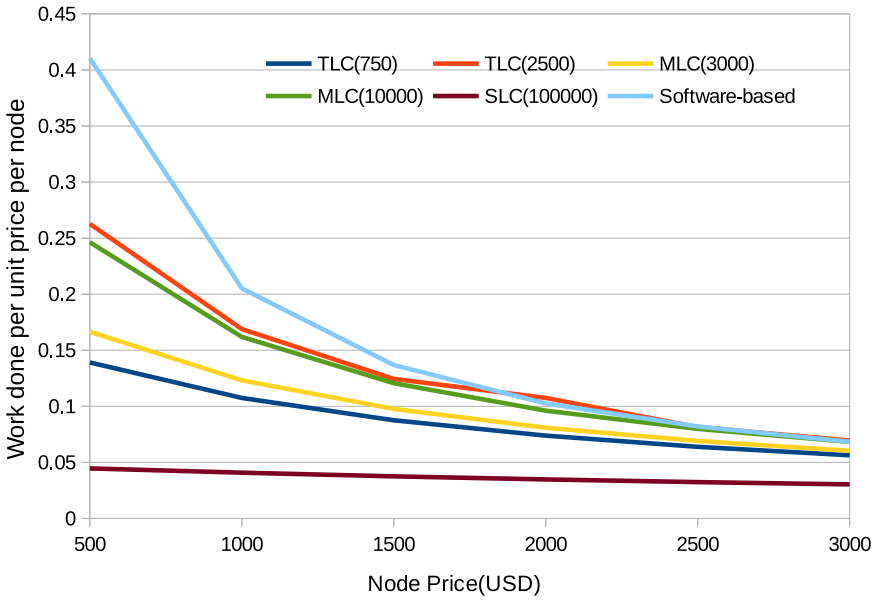



Figure 5.11: Comparison of work done per unit price per node for a system with different types of SSD device compared against software-based solution. (higher is better).

### 5.6.1 Validating our Energy-performance model

We validated our coarse-grained checkpoint-compression energy model described in Section 5.2.2 using LAMMPS and BLCR. Our validation approach was as follows: we ran LAMMPS for a period sufficiently long enough that allowed us to take many checkpoints. In our *measurement phase*, we collected coarse-grained power measurements obtained from the execution period up to and including the first checkpoint. Then we input these measurements into our model to predict the energy footprint of the entire remainder of the application's execution. (Of course, we continued the measurement collection throughout the application's execution to compare our model's predicted values to those actually observed.) We repeat this process three times, and the results we present here are the average of these three runs. Each LAMMPS run included taking 50 checkpoints at a fixed 10-second intervals.

During the measurement phase, we sampled power to obtain $\bar{P}_{app}$, the average power while executing the application code, and $\bar{P}_{ckpt}$, average power for taking a checkpoint. For the entire application's execution, we also measured $T_{app}$, the total time spent executing the application's code and $T_{ckpt}$, the total time spent taking checkpoints. We input these parameters into our model to predict the energy expenditure, as the application execution increases, in five checkpoint increments. That is, the first prediction predicts what the application's energy consumption will be after taking five checkpoints; the second prediction predicts through ten checkpoints and so on up to all 50 checkpoints.

Figure 5.12 shows that **our coarse-grained model can predict the energy consumption of a CR-based application very accurately with prediction accuracy ranging from 94% to 99% for our experiments.** For this validation, we did not have a failure injection framework and hence did not include restart or rework. However, we see no reason to believe that our coarse-grained model will not work just as effectively for restart and rework.

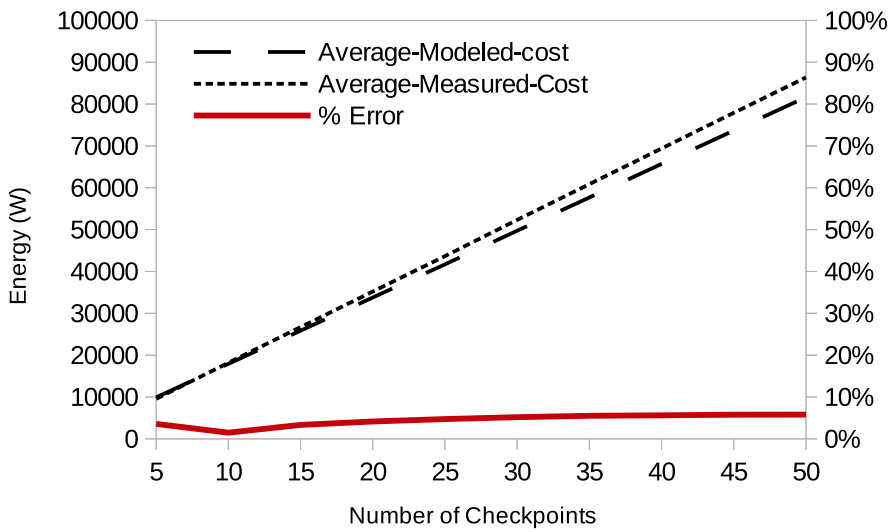



Figure 5.12: Model predicted energy costs are very accurate and within 94-99% of the average measured costs.

| | $\bar{P}_{app}$(W) | $\bar{P}_{ckpt}$(W) | $\bar{P}_{rst}$(W) |
|---|---|---|---|
| HPCCG | 70.66 | 58.79 | 40.87 |
| LAMMPS | 79.88 | 53.63 | 91.15 |
| pHPCCG | 60.51 | 54.74 | 53.43 |
| MiniFE | 59.81 | 23.80 | 50.03 |

Table 5.6: Measured average power costs of application run, checkpoint and restart for different applications

### 5.6.2 Checkpoint compression energy performance

Using our validated model, we predict the energy performance for LAMMPS and three *mini apps* from the Mantevo Project [53], namely HPCCG, pHPCCG and MiniFE. As previously described, we obtained the checkpoint sizes and compression/decompression performance statistics for these applications from a previous project [56]. We also used our application efficiency model from that project to calculate the time spent for (1) checkpointing, (2) restarting from failures and (3) executing rework after restarts.

Our efficiency model used Daly's optimal checkpoint interval calculation [25], as described in Section 5.2.2. In Table 5.6, we list our empirically-measured average power costs of (1) unit time application run, (2) checkpoint operation and (3) restart operation that we input to our model. We also empirically measured the average power consumption of checkpoint compression and decompression, and incorporated those metrics into the model to account for the energy costs of checkpoint compression optimization.

Figure 5.13 shows the overall energy savings using CR with checkpoint compression versus regular CR. We make two observations:

1. **the CR compression optimization always provides a reduction in**

**overall application energy consumption**; and

2. **the energy savings yielded by the CR compression optimization increases with application scale.** In our study, the energy savings increased from 10% at a socket count of 10,000 to almost 90% at a socket count of 90,000.

The reduction in the number of checkpoints taken offsets the extra per checkpoint energy consumed due to checkpoint compression. This reduction is seen in Figure 5.14, which compares the total number of checkpoints taken for uncompressed and compressed cases for the same workload. Due to the increased application efficiency for checkpoint compression, even though checkpoint frequency increases, the decreased application execution time leads to fewer overall checkpoints being taken.

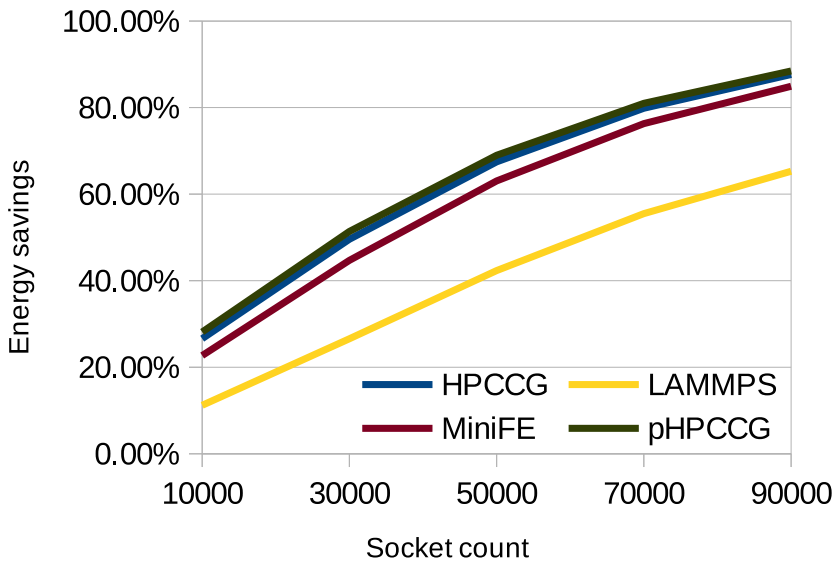



Figure 5.13: Total energy savings compared to regular checkpoint/restart for different applications

The energy savings increase with scale because as an application's scale increases, the application becomes increasingly inefficient using normal CR (falling below 10%

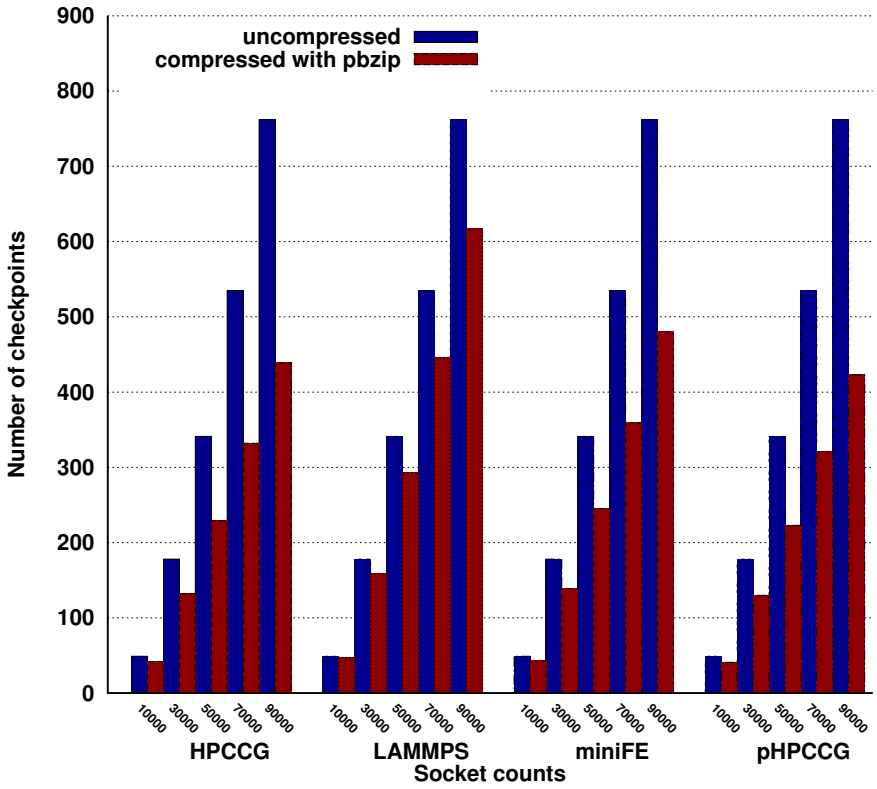



Figure 5.14: Comparison of the number of checkpoints taken with or without checkpoint compression

[56]) and the impact of CR optimizations like compression becomes greater. Again, as shown in Figure 5.14, as scale increases the difference in the number of checkpoints taken with and without checkpoint compression increases.

Finally, Figure 5.15 isolates the energy savings just for CR operations yielded by compression. This figure shows energy savings from 45% to 96%. Referring again to Figure 5.14, compression does not reduce the number of checkpoints taken by LAMMPS as much as it does for the other applications. This results in lower energy savings for LAMMPS overall, as shown in Figure 5.13, and when considering only CR operation, as in Figure 5.15.

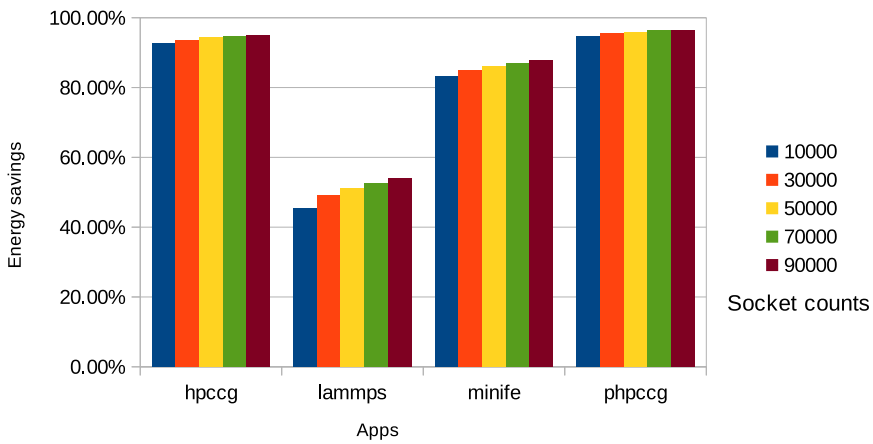



Figure 5.15: Energy savings for checkpoint/restart operations only.

## 5.7 Summary

In this chapter, we investigated checkpoint data compression as a proxy for inter-application data movement reduction. We demonstrated that checkpoint data compression is a very viable approach for CR protocol optimization. We then studied the performance limits of checkpoint compression and put the results of this technique in the context of the current state-of-the-art in checkpointing. Specifically, we used information theory to show that current compression techniques are close enough to a theoretically optimal solution so that improved algorithms likely will render little to no difference in overall application performance. We also showed that checkpoint compression outperforms another popular software-based checkpoint optimization, incremental checkpointing, and that a combination of both leads to further performance improvements. Together, compression and increment-based optimizations can yield performance to within 20% of current state-of-the-art hardware-based solutions. We also showed that our software-based checkpoint/restart optimization produces more work per unit cost than the hardware-based approaches as long as

per-node procurement costs are kept low. Finally, we showed that due to the increased runtime efficiency, compression-based checkpoint optimization leads to up to 90% savings in energy as well.

We believe that this work reveals many fundamental insights about the role checkpoint data compression can and should have as a part of the solution space of efficient application-independent fault-tolerance strategies. Perhaps the greatest outcome is the insight that this simple, application-agnostic approach can render significant performance improvements when used in isolation or in combination with other software and hardware-based optimizations.

# Chapter 6

# Conclusion and Future Work

HPC systems have observed tremendous improvements in processing performance as compared to other areas of the systems. Therefore, data movement has resulted in the bottleneck, rather than computation, as the most significant cost. The goal of this work was to study different areas of data movement in HPC systems and to demonstrate that compression-based approaches can be effectively applied to reduce them. I identified and studied three different areas in HPC where data movement was becoming bottlenecks and applied compression-based techniques to reduce data movement volumes and to improve HPC application's runtime performance. In this chapter, I summarize my findings and explore future research directions.

## 6.1 Contributions

In this dissertation, I demonstrated viable approaches of applying software-based compression to reduce data movement in HPC, using a combination of modeling, simulation and implementation. The major contributions of this work are:

- **Intra-process data movement reduction**

  In Chapter 3, I studied intra-process data movement reduction by improving the perceived per-core memory capacity by using compression-based paging. Improved memory capacity enables in-situ data analysis and visualization and does not require moving data to a different node, as well as enabling larger problem-solving capacity. I used a simulation-based approach to introduce paging for HPC workloads and to study the runtime overheads due to the introduction of paging. I modeled compression performance numbers to memory performance numbers as simulator inputs and proposed a compression-based memory paging solution. I demonstrated that my proposed compression-based paging scheme can improve perceived memory capacity by 78% with minimal runtime overhead (under 4%).

- **Inter-process data movement reduction**

  In Chapter 4, I studied different ways to reduce inter-process data movement. Here I identified the similarities that exist among inter-process messages using a diff-based approach and used trace-based simulation to study the overheads of my approach to reduce network congestion. I proposed a novel two-level diff-based approach that can reduce inter-process data movement by up to 99% although with potentially large runtime overhead.

- **Inter-application data movement reduction**

  In Chapter 5, I studied inter-application data movement reduction by focusing on checkpoint/restart (CR) optimization as a proxy for all inter-application data movement. I studied the viability of checkpoint compression and demonstrated its application to improve overall application runtime performance. I developed a viability model for checkpoint data compression that accounts for the cost and benefits of compression for checkpoint commit and recovery operations. Using an extension of Daly's model, I demonstrated

that compression-based checkpoint data movement reduction can significantly improve an applications runtime (by more than 50%) across a wide range of scenarios.

I studied compression algorithm parameters and their impact on checkpoint compression and demonstrated that text-based compression algorithms may offer sufficient speed and checkpoint data compressibility such that enhanced compression algorithms likely will render little application performance improvement.

I compared my compression-based checkpoint data movement reduction optimization against other software- hardware-based checkpoint optimizations and demonstrated that checkpoint data compression used in conjunction with other software CR protocol optimizations can be a viable, cost-effective alternative to hardware-based CR solutions. Finally, I developed and validated a checkpoint-compression energy performance model and demonstrated that compression-based checkpoint optimization would also improve an application's energy performance (by up to 90%) due to the application's efficiency improvement.

As HPC systems approach exascale performance, we will continue to see massive growth in data movement. This work demonstrated that similarity exists in HPC data and can be exploited to reduce the data movement problem in various levels of the data movement hierarchy. Scientists and application developers can improve the application performance by leveraging the data similarity. Other areas in HPC, that I have not studied can also benefit from this research. For example, HPC storage can readily reduce the data volume to be committed and improve effective throughput by compression. Data processing and analysis that transfers data outside the node

can also improve by leveraging data similarity. Outside of the HPC domain, cloud-computing is also experiencing the 'big data' problem due to the the rise of user generated data through social media and internet of things. I believe the insights from this study can be studied in cloud-computing contexts and can achieve similar positive results.

## 6.2 Future Work

In this dissertation, I have studied compression-based data movement reduction optimization for different areas in HPC systems. However, a number of potential research inquiries remain.

My research on intra-process data movement reduction, described in Chapter 3, demonstrated that the memory access patterns for our test applications were an important indicator for the perceived memory capacity improvement. Future work, should investigate the impact of different page-replacement policies based on applications' memory access patterns. The insight from these studies could motivate variable-sized compressed-cache, thus further improving the perceived memory capacity. In addition, a demonstration of leveraging the additional memory capacity that was available due to compressed-paging would make for an interesting study as well.

The two-level diff-based approach that I introduced in Chapter 4 demonstrated that there exists great opportunity to reduce MPI-based inter-process data movement but at a high runtime overhead. However, the simulator used in that study, `LogGOPSim`, did not have support for modeling network congestion. Overcoming this limitation is a challenge, but it would allow us to clearly understand the trade-off

of using our two-level diff-based approach by accounting for both the overhead and the benefit of reduced congestion. Inspired by the results in this study, application developers can consider computing message diffs at the application level and sending incremental differences only. While this would create an additional complexity for application developers, they may be able to leverage domain specific knowledge and implement faster diff algorithms.

Finally, in Chapter 5, I demonstrated the runtime and energy benefits of inter-application data movement reduction by using checkpoint-compression as a proxy of application services. The next step is to extend this study to other types of checkpoints (for example: uncoordinated checkpoints) and see whether we can get similar benefits. A streaming compression-based approach would be another avenue of work that can be easily extended from my study, as a way to reduce checkpointing time even further by overlapping compression with checkpoint commits. The final step would be to implement compression into a checkpointing library and study the actual impact on an application's runtime and energy performance.

# References


[1] FlashDIMMSim: A reasonably accurate flash DIMM simulator. *URL:https://github.com/jimstevens2001/NVDIMMSim*, 2009.

[2] K. Agarwal. Wire-speed differential soap encoding. In *2014 IEEE International Conference on Web Services*, pages 217–224, June 2014.

[3] S. Al-Kiswany, M. Ripeanu, S.S. Vazhkudai, and A. Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 613–624, june 2008.

[4] Alaa R Alameldeen and David A Wood. Adaptive cache compression for high-performance processors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 212–223. IEEE, 2004.

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[6] ASC Sequoia. URL:https://asc.llnl.gov/computing_resources/sequoia (visited May 2011).

[7] Steve Ashby, P Beckman, J Chen, P Colella, B Collins, D Crawford, J Dongarra, D Kothe, R Lusk, P Messina, et al. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee (November 2010)*, 2010.

[8] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[9] Blaise Barney. Introduction to livermore computing resources. *URL:http://computing.llnl.gov/tutorials/lc_resources*, August 2011.

[10] B. W. Barrett and K. S. Hemmert. An application-based MPI message throughput benchmark. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, Aug 2009.

[11] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pages 21:1–21:12, 2009.

[12] K Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), September 2008.

[13] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P. T. Bremer, M. Schulz, and L. V. Kale. Identifying the culprits behind network congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 113–122, May 2015.

[14] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 41:1–41:12, New York, NY, USA, 2013. ACM.

[15] Susmit Biswas, Bronis R de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T Chong. Exploiting data similarity to reduce memory footprints. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 152–163. IEEE, 2011.

[16] Susmit Biswas, Diana Franklin, Timothy Sherwood, Frederic T Chong, Bronis R de Supinski, and Martin Schulz. PSMalloc: content based memory management for MPI applications. In *Proceedings of the 10th workshop on Memory performance: Dealing with Applications, systems and architecture*, pages 43–48. ACM, 2009.

[17] Marin Bougeret, Henri Casanova, Mikaël Rabie, Yves Robert, and Frédéric Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 33:1–33:11, New York, NY, USA, 2011. ACM.

[18] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Sally McKee, and Radu Rugina. Compiler-enhanced incremental checkpointing for OpenMP applications. In *IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, 2009.

[19] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In *SuperComputing '97*, San Jose, CA, 1997.

[20] Aleksandar Colic, Hari Kalva, and Borko Furht. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 13–22, New York, NY, USA, 2010. ACM.

[21] Collaboration, M and others. MIMD lattice computation (MILC) collaboration home page. *Information available at http://physics.indiana.edu/sg/milc.html*.

[22] Yann Collet. Lz4: Extremely fast compression algorithm. *URL:http://lz4.github.io/lz4/*, 2013. visited May, 2016.

[23] William D Collins, Hans Johansen, Katherine J Evans, Carol S Woodward, and Peter M Caldwell. Progress in fast, accurate multi-scale climate simulations. *Procedia Computer Science*, 51:2006–2015, 2015.

[24] J. Cornwell and A. Kongmunvattana. Efficient System-Level Remote Checkpointing Technique for BLCR. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1002–1007, april 2011.

[25] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer System*, 22(3):303–312, 2006.

[26] Rodrigo S De Castro, APD Lago, and M Silva. Adaptive compressed caching: Design and implementation. In *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pages 10–18. IEEE, 2003.

[27] P Deutsch. Deflate compressed data format specification. *ftp://ftp.uu.net/pub/archiving/zip/doc*.

[28] Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, and Ravishankar Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 25–36. IEEE, 2015.

[29] Jack Dongarra, Thomas Herault, and Yves Robert. Revisiting the double checkpointing algorithm. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 706–715. IEEE, 2013.

[30] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Winter*, pages 519–529. Citeseer, 1993.

[31] Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th Intl. Symp. on Shock Waves*, pages 377–382, July 1993.

[32] M. El Mehdi Diouri, O. Gluck, L. Lefevre, and F. Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.

[33] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[34] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The performance of consistent checkpointing. In *11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, 1992.

[35] Kurt Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *SC*. ACM, Nov 2011.

[36] Kurt B. Ferreira, Rolf Riesen, Ron Brightwell, Patrick G. Bridges, and Dorian Arnold. Libhashckpt: Hash-based incremental checkpointing using GPUs. In *Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[37] KurtB. Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: Hash-based incremental checkpointing using gpus. In Yiannis Cotronis, Anthony Danalis, DimitriosS. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 272–281. Springer Berlin Heidelberg, 2011.

[38] Rosa Filgueira, David E Singh, Alejandro Calderón, and Jesús Carretero. CoMPI: Enhancing MPI based applications performance and scalability using run-time compression. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 207–218. Springer, 2009.

[39] Rosa Filgueira, David E Singh, Jesús Carretero, Alejandro Calderón, and Félix García. Adaptive-CoMPI: Enhancing MPI-based applications performance and scalability by using adaptive compression. *International Journal of High Performance Computing Applications*, 25(1):93–114, 2011.

[40] Peter A Franaszek and John T Robinson. On internal organization in compressed random-access memories. *IBM Journal of Research and Development*, 45(2):259–270, 2001.

[41] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 353–377. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30218-6_19.

[42] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. In *European Parallel Virtual Machine / Message Passing Interface Users Group Meeting*. Springer, 1998.

[43] Félix García, Alejandro Calderón, and Jesús Carretero. MiMPI: A multithread-safe implementation of MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 207–214. Springer, 1999.

[44] Pedro J Garcıa, J Flich, J Duato, I Johnson, Francisco J Quiles, and F Naven. Dynamic evolution of congestion trees: Analysis and impact on switch architecture. *Lecture Notes in Computer Science (HiPEAC 2005)*, 3793:266–285, 2005.

[45] Garth Gibson, Bianca Schroeder, and Joan Digney. Failure tolerance in petascale computers. *CTWatch Quarterly*, 3(4), November 2007.

[46] Jeff Gilchrist. Parallel data compression with Bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, volume 16, pages 559–564, 2004.

[47] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor Mudge, and David Blaauw. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. ACM, 2013.

[48] Ernst Gunnar Gran and Sven-Arne Reinemo. Infiniband congestion control: modelling and validation. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 390–397. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[49] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[50] Erik G Hallnor and Steven K Reinhardt. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 201–212. IEEE, 2005.

[51] Simon D Hammond, Arun F Rodrigues, and Gwendolyn R Voskuilen. Multi-level memory policies: What you add is more important than what you take out. In *Proceedings of the Second International Symposium on Memory Systems*, pages 88–93. ACM, 2016.

[52] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 46(1), 2006.

[53] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratory, 2009.

[54] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, 2010.

[55] Craig Hyatt and Dharma P. Agrawal. Congestion control in the wormhole-routed torus with clustering and delayed deflection. In Sudhakar Yalamanchili

and José Duato, editors, *Parallel Computer Routing and Communication: Second International Workshop, PCRCW'97 Atlanta, Georgia, USA, June 26–27, 1997 Proceedings*, pages 33–38. Springer Berlin Heidelberg, 1998.

[56] Dewan Ibtesham, Dorian Arnold, Patrick G. Bridges, Kurt B. Ferreira, and Ron Brightwell. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. *2012 41st International Conference on Parallel Processing*, 0:148–157, 2012.

[57] Dewan Ibtesham, David Debonis, Ferreira Kurt, and Dorian. Arnold. Coarse-grained energy modeling of rollback/recovery mechanisms. In *Proceedings of the 4th Workshop on Fault-tolerance for HPC at Extreme Scale*, FTXS '14, 2014.

[58] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. A checkpoint compression study for high-performance computing systems. *The International Journal of High Performance Computing Applications*, 29(4):387–402, 2015.

[59] Tanzima Zerin Islam, K. Mohror, Saurabh Bagchi, A. Moody, B.R. De Supinski, and Rudolf Eigenmann. MCRENGINE: A scalable checkpointing system using data-aware aggregation and compression. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012.

[60] Jagan Jayaraj, Arun F Rodrigues, Simon D Hammond, and Gwendolyn R Voskuilen. The potential and perils of multi-level memory. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 191–196. ACM, 2015.

[61] Myoungsoo Jung, Ellis Herbert Wilson, David Donofrio, John Shalf, and Mahmut Taylan Kandemir. NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[62] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon EC2. In *2009 5th IEEE International Conference on E-Science Workshops*, pages 59–66, Dec 2009.

[63] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. Optimizing checkpoints using NVM as virtual memory. In *Proceedings of the nternational Parallel and Distributed Processing Symposium*, IPDPS '13, New York, NY, USA, 2013. ACM.

[64] Jian Ke, Martin Burtscher, and Evan Speight. Runtime compression of MPI messanes to improve the performance and scalability of parallel applications. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 59. IEEE Computer Society, 2004.

[65] JunSeong Kim and David J Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 202–216. Springer, 1998.

[66] Morten Kjelsø, Mark Gooch, and Simon Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of Systems Architecture*, 45(8):571–590, 1999.

[67] Peter Kogge and John Shalf. Exascale computing trends: Adjusting to the" new normal" in computer architecture. In *2013 Third Berkeley Symposium on Energy Efficient Electronic Systems (E3S)*. IEEE, 2013.

[68] M. J. Koop, M. Luo, and D. K. Panda. Reducing network contention with mixed workloads on modern multicore clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[69] David Korn, J MacDonald, J Mogul, and K Vo. The vcdiff generic differencing and compression data format. Technical report, 2002.

[70] Sandia National Laboratories. Advance systems technology test beds. *URL:http://www.sandia.gov/asc/computational_systems/HAAPS.html*, October 2013.

[71] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pages 40:1–40:12, 2009.

[72] James H. Laros, David DeBonis, and Phi Pokorny. *PowerInsight - A Commodity Power Measurement Capability*. Apr 2013.

[73] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 184–191. IEEE, 1999.

[74] Jonghyun Lee, M. Winslett, Xiaosong Ma, and Shengke Yu. Enhancing data migration performance via parallel data compression. In *International Parallel and Distributed Processing Symposium*, pages 444–451, 2002.

[75] Scott Levy, Kurt B Ferreira, and Patrick G Bridges. Improving application resilience to memory errors with lightweight compression. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 323–334. IEEE, 2016.

[76] Scott Levy, Kurt B Ferreira, Patrick G Bridges, Aidan P Thompson, and Christian Trott. An examination of content similarity within the memory of HPC applications. *Sandia National Laboratory, Tech. Rep. SAND2013-0055*, 2013.

[77] Scott N Levy, Kurt Brian Ferreira, and Patrick G Bridges. Similarity engine: Using content similarity to improve memory resilience. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2016.

[78] C.-C.J. Li and W.K. Fuchs. CATCH-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81, jun 1990.

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

[80] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[81] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: reading patterns for extreme scale science IO. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[82] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[83] Philip J Maechling, Fabio Silva, Scott Callaghan, and Thomas H Jordan. SCEC broadband platform: System architecture and software implementation. *Seismological Research Letters*, 2014.

[84] Esteban Meneses, Osman Sarood, and Laxmikant V. Kale. Assessing energy efficiency of fault tolerance protocols for HPC systems. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High*

*Performance Computing*, SBAC-PAD '12, pages 35–42, Washington, DC, USA, 2012. IEEE Computer Society.

[85] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136. IEEE, 2015.

[86] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[87] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–11, 2010.

[88] Kingsley G. Morse Jr. Compression tools compared. *Linux Journal*, (137), September 2005.

[89] Andreas Moshovos and Alexandros Kostopoulos. Cost-effective, high-performance giga-scale checkpoint/restore. Technical report, University of Toronto, November 2004. URL:http://www.eecg.toronto.edu/~moshovos/research/gigacr.pdf.

[90] G. Motta, J. Gustafson, and S. Chen. Differential compression of executable code. In *Data Compression Conference, 2007. DCC '07*, pages 103–112, March 2007.

[91] Giovanni Motta, James Gustafson, and Samson Chen. Differential compression of executable code. In *Data Compression Conference, 2007. DCC'07*, pages 103–112. IEEE, 2007.

[92] Nichamon Naksinehaboon, Yudan Liu, Chokchai (Box) Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott. Reliability-aware approach: An incremental checkpoint/restart model in HPC environments. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 783–788, Washington, DC, USA, 2008. IEEE Computer Society.

[93] Bogdan Nicolae. Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal. In *Parallel & Distributed Processing*

*(IPDPS), 2013 IEEE 27th International Symposium on*, pages 19–28. IEEE, 2013.

[94] DOE(Department of Energy). Scientific discovery through advanced computing (scidac); exascale challenges. *http://science.energy.gov/ascr/research/scidac/exascale-challenges*.

[95] Ron A Oldfield, Sarala Arunagiri, Patricia J Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C Roth. Modeling the impact of checkpoints on next-generation systems. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 30–46. IEEE, 2007.

[96] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH–a locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*, pages 7–13. IEEE, 2013.

[97] Vijay S Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, volume 178. Citeseer, 1997.

[98] Mihaela Paun, Nichamon Naksinehaboon, Raja Nassar, Chokchai Leangsuksun, Stephen L. Scott, and Narate Taerat. Incremental checkpoint schemes for weibull failure distribution. *International Journal of Computer Science*, 21(3):329–344, 2010.

[99] Colin Percival. Binary diff/patch utility. *URL:http:// www.daemonology.net/bsdiff*, 2003.

[100] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software – Practice & Experience*, 29(2):125–142, 1999.

[101] J. S. Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 2(2):62–67, 1994.

[102] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *USENIX Winter 1995 Technical Conference*, pages 213–224, New Orleans, LA, January 1995.

[103] James S. Plank, Jian Xu, and Robert H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[104] J.S. Plank, Kai Li, and M.A. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, oct 1998.

[105] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[106] Matt Poremba and Yuan Xie. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pages 392–397. IEEE, 2012.

[107] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. (DK) Panda. A 1 pb/s file system to checkpoint three million MPI tasks. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 143–154, 2013.

[108] R. Riesen. Communication patterns (message-passing patterns). In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, April 2006.

[109] Rolf Riesen, Kurt Ferreira, Jon Stearley, Ron Oldfield, James H Laros III, Kevin Pedretti, Ron Brightwell, et al. Redundant computing for exascale systems. *Technical report SAND2010–8709, Sandia National Laboratories, Tech. Rep.*, 2010.

[110] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[111] Sandia National Laboratories. The LAMMPS molecular dynamics simulator, April 2010. URL:http://lammps.sandia.gov.

[112] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 19. IEEE Computer Society Press, 2012.

[113] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006.

[114] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics Conference Series*, 78(1), 2007.

[115] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. Memzip: Exploring unconventional benefits from memory compression. In *HPCA*, 2014.

[116] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[117] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–, july, october 1948.

[118] Galen Shipman, Dave Dillow, Sarp Oral, and Feiyi Wang. The Spider center wide file system: From concept to reality. In *Proceedings of the 2009 Cray User Group (CUG) Conference*, Atlanta, GA, May 2009.

[119] Luís Moura Silva and Joao Gabriel Silva. An experimental study about diskless checkpointing. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 395–402. IEEE, 1998.

[120] Mohsen Soryani, Morteza Analoui, and Ghobad Zarrinchian. Improving inter-node communications in multi-core clusters using a contention-free process mapping algorithm. *J. Supercomput.*, 66(1):488–513, October 2013.

[121] JEDEC Memory Standards. DDR3 SDRAM standard. *JESD79-3, Joint Electron Device Engineering Council (JEDEC)*, 2012.

[122] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society.

[123] Jim Stevens. HybridSim memory simulator. *URl:https://github.com/jimstevens2001/HybridSim*, 2014.

[124] T. Hoefler. LogGOPSim - a LogGOPS (LogP, LogGP, LogGPS) simulator and simulation framework. URL:http://www.unixer.de/research/LogGOPSim/, Apr. 10 2013.

[125] Robert Templeman and Apu Kapadia. GANGRENE: Exploring the mortality of flash memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[126] Devesh Tiwari, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter Desnoyers. Reducing data movement costs using energy-efficient, active computation on SSD. In *HotPower*, 2012.

[127] Top 500 supercomputer sites. URL:http://www.top500.org/ (visited March 2012).

[128] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. IBM memory expansion technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, 2001.

[129] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM.

[130] Paul Tschirhart, Jim Stevens, Zeshan Chishti, and Bruce Jacob. The case for associative DRAM caches. In *Proceedings of the Second International Symposium on Memory Systems*, pages 211–219. ACM, 2016.

[131] Irina Chihaia Tuduce and Thomas R Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250, 2005.

[132] Richard A Uhlig and Trevor N Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[133] Nitin H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 64–73, New York, NY, USA, 1995. ACM.

[134] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. DI-MMAP: a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[135] Jeffrey S Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, 2003.

[136] Gwendolyn Voskuilen, Arun F. Rodrigues, and Simon D. Hammond. Analyzing allocation behavior for multi-level memory. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 204–207, New York, NY, USA, 2016. ACM.

[137] Gwendolyn Renae Voskuilen, Simon David Hammond, Arun F Rodrigues, Branden J Moore, and Karl Scott Hemmert. Structural simulation toolkit. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[138] David W Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[139] Paul R Wilson. Operating system support for small objects. In *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, pages 80–86. IEEE, 1991.

[140] Paul R Wilson, Scott F Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference, General Track*, pages 101–116, 1999.

[141] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265. ACM, 2000.

[142] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *8th International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 177–184, Redondo Beach, CA, August 1999.

[143] J Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.