SAND2014-15140J

A Study of Checkpoint Compression for High-Performance Computing Systems

Kurt B. Ferreira[a], Dorian Arnold[b], Dewan Ibtesham[b]

[a] *Scalable System Software Department, Sandia National Laboratories* [1] *,Albuquerque, NM 87185–1319*
[b] *Department of Computer Science, University of New Mexico, Albuquerque, NM 87131*

1. Abstract

As high-performance computing systems continue to increase in size and complexity, higher failure rates and increased overheads for checkpoint/restart (CR) protocols have raised concerns about the practical viability of CR protocols for future systems. Previously, compression has proven to be a viable approach for reducing checkpoint data volumes and, thereby, reducing CR protocol overhead leading to improved application performance. In this article, we further explore compression-based CR optimization by exploring its baseline performance and scaling properties, evaluating whether improved compression algorithms might lead to even better application performance and comparing checkpoint compression against and alongside other software and hardware-based optimizations. Our results highlights are that (1) compression is a very viable CR optimization; (2) generic, text-based compression algorithms appear to perform near optimally for checkpoint data compression and faster compression algorithms will not lead to better application performance; (3) compression-based optimizations fare well against and alongside other software-based optimizations; and (4) while hardware-based optimizations outperform software-based ones, they are not as cost effective.

2. Introduction

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) [1]. 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 [1, 2].

CR protocols [3] 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 as well as contentions for network and storage devices. As we describe in Section 3, researchers have proposed several CR protocol performance optimizations to alleviate the data movement challenge, including checkpoint data compression. In this article, 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?

[1] Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

- Might 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?

We explored these questions guided by current system characteristics and an eye toward emerging and new potential technologies. Using a performance model [4] based on Daly's higher order checkpointing model [5], 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 demonstration that checkpoint data compression can improve significantly an application's makespan across a wide range of scenarios;
3. A demonstration that existing, text-based compression algorithms may offer sufficient speeds and checkpoint data compressibility such that enhanced compression algorithms likely will render little application performance improvements;
4. A demonstration that checkpoint data compression can yield application performance improvements when used in conjunction with other software CR protocol optimizations;
5. A demonstration that checkpoint data compression used in conjunction with other software CR protocol optimizations may pose a viable, cost-effective alternative to hardware-based CR solutions.

The rest of this article is organized as follows. First, we provide contextual background by offering a brief overview of CR protocols and proposed software and hardware-based CR optimizations in the next section. Then we describe our evaluation methodology and tool chain in Section 4. We present our results of the performance and scaling features of compression-based checkpoint optimizations in Section 5 followed by a study of the potential benefits of enhanced compression algorithms in Section 6. Our last set of results comprise a comparative studies of checkpoint data compression and other CR optimizations, in Section 7. Finally, we conclude with a summary of our findings and a discussion of the implications of these results for future HPC systems.

3. An Overview of Checkpoint/Restart

During normal operation, CR (or *rollback recovery*) protocols [3] periodically record or *commit* process state to stable storage devices, devices that survive tolerated failures. Process state comprises all the state necessary to run a process correctly including, for example, its address space and register files. When processes fail, new process instances can be *recovered* to the intermediate state encapsulated in the failed processes' most recent checkpoints to avoid lost computations.

To checkpoint and recover processes in distributed applications, checkpoint data must be transferred amongst the local nodes from where the checkpoints originate to storage nodes so that the checkpoints can be available even when their source nodes have failed. 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 as well as contentions for network and storage devices.

3.1. Software-based Checkpoint/Restart Data Movement Optimizations

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

- *diskless and remote checkpointing*: Diskless CR protocols [6] and remote CR protocols [7, 8, 9] leverage the higher bandwidths available to the network or other storage media like RAM to mitigate the performance of slower storage media like magnetic disks. Additionally, remotely stored checkpoints allow systems to survive non-transient node failures.

- *multi-level checkpointing*: Multi-level CR protocols like SCR [10, 11] write checkpoints to RAM, Flash, or local disk on the compute nodes in addition to the parallel file system.

- *checkpointing file systems*: Checkpoint-specific file systems like PLFS [12] leverage the patterns and characteristics specific to checkpoint data to optimize checkpoint data transfers to/from parallel file systems.

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

- *memory exclusion*: CR protocol optimizations based on memory exclusion leverage user-directives or other hints to exclude portions of process address spaces from checkpoints [13].

- *incremental checkpointing*: CR protocols can use the operating system's memory page protection facilities to detect and save only pages that have been updated between consecutive checkpoints [14, 15, 16, 17, 18, 19, 20]. Page hashing techniques can also be used to avoid checkpointing pages that have been updated but not actually changed content-wise [21].

- *checkpoint compression*: Various approaches for compressing checkpoints to improve CR protocol performance have been suggested. Li and Fuchs implemented a compiler-based checkpointing approach, which exploited compile time information to compress checkpoints [22]. Plank and Li proposed in-memory checkpoint compression [23], and in a related vein, Plank et al also proposed *differential compression* to reduce checkpoint sizes for incremental checkpoints [24]. Tanzima et al have shown that similarities amongst checkpoint data from different processes can be exploited to compress and reduce checkpoint data volumes [25].

This work extends our previous study [4] that showed the viability of using standard compression utilities for improved CR protocol performance for extreme scale applications.

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 [26]. More recently, researchers have proposed the use of solid state storage devices (SSDs) for efficient local checkpointing [27] or even in multi-level solutions [10]. 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.

4. Methodology: Data Collection and Performance Models

In this study, we compared checkpoint compression to other CR protocol optimizations. Figure 1 depicts our approach for executing this study and the set of tools that we used. Our general methodology 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 data along with different application workloads and system configurations into validated performance models to observe the resulting application performances. In this section, we offer the comprehensive details of our approach.

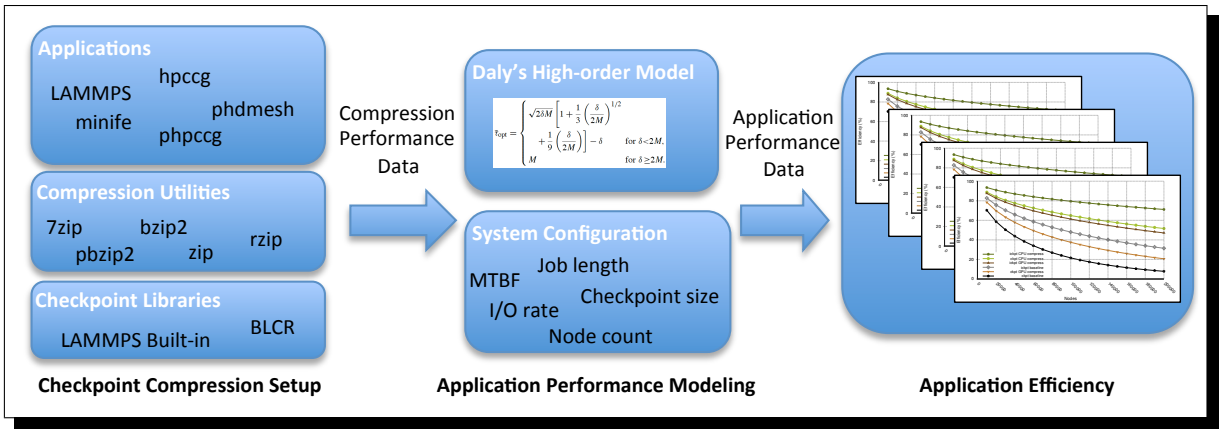


Figure 1: Our Methodology: Empirically collected checkpoint compression data is input to an extension of Daly's Model. The results are used to compute application efficiency.

4.1. Collecting Checkpoint Compression Performance Data

To collect checkpoint compression performance data, we instrumented a set of exascale proxy applications and a full application with CR capabilities. We executed these applications with CR enabled to collect the application checkpoints. Then in an offline manner, we used various compression utilities to measure the extent to which the checkpoint files compress as well as the speed of checkpoint compression and decompression.

4.1.1. 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 [28], namely HPCCG version 0.5, miniFE version 1.0, pHPCCG version 0.4 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. 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.

4.1.2. A Full Application: LAMMPS

We use LAMMPS (the Large-scale Atomic/Molecular Massively Parallel Simulator [29, 30]) to evaluate checkpoint compression on a full-featured scientific application. LAMMPS is a classical molecular dynamics code 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 code. 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.

4.1.3. The Compression Utilities

For this study, we used popular compression algorithms investigated in Morse's comparison of compression tools [31]. Here we present the results from the better-performing algorithms. Additionally, some algorithms can be parameterized to trade off between execution time for compression factor. We only present the parameter sets that represent the best observed trade-offs.

- **zip**: `zip` is an implementation of Deflate [32], a lossless data compression algorithm that uses the LZ77 [33] 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.

- **7zip**[34]: `7zip` is based on the Lempel-Ziv-Markov chain algorithm (LZMA) [35]. It uses a dictionary scheme similar to LZ77.

- **bzip2**: `bzip2` is an implementation of the Burrows-Wheeler transform [36], 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 move to front transform and Huffman coding.

  In bzip2, compression performance varies with block size. `bzip2` takes an integer parameter that ranges from zero to nine, where a smaller value specifies a smaller block size. For our experiments, "`bzip2(1)`" represents the best trade-off.

- **pbzip2**[36]: `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 same block size parameter as in `bzip2`. 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.

- **rzip**: `rzip` uses a very large buffer to take advantage of redundancies that span very long distances. It finds and encodes large chunk of duplicate data and then uses `bzip2` as a back-end to compress the encoding.

  Similar to `zip`, `rzip` 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, "`rzip(3)`" represents the best trade-off.

4.1.4. Checkpoint/Restart Utilities

The Berkeley Lab Checkpoint/Restart library (BLCR) [37] 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 use the OpenMPI [38] framework, which has integrated BLCR support.

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

4.2. Performance Models

4.2.1. 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 Plank et al's [23]. 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 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 to commit a globally consistent application state

comprised of one checkpoint per process[2]. cCR currently dominates CR protocols used in HPC practice. We also assume an equal number of checkpoint and recovery operations. Our justification for this latter assumption follows: optimally, an application averages a single checkpoint before each failure and only needs to recover once per failure. Therefore, in the optimal case, the number of checkpoints equals the number of failures, which also equals the number of recoveries. There are various works that define optimal checkpoint intervals [5, 39]. Finally, we assume that checkpoint commit is synchronous; that is, the primary application process is paused during the commit operation and is not resumed until checkpoint commit is complete.

Checkpoint compression is viable when the time to compress and write or 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 times to read and write are the same (that is, 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:

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

where c is the size of the original checkpoint, *compression factor* α 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:

$$\frac{2\alpha \times r_{comp} \times r_{decomp}}{r_{comp} + r_{decomp}} > r_{commit} \tag{1}$$

Equation 1 defines the minimal ratio between checkpoint commit rate and 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.

4.2.2. 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 [5], 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 the application's execution time in a failure-free environment. We used this model and 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.

[2]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.

Modeling Incremental Checkpointing. We also integrated incremental checkpointing into Daly's performance model. As such, 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 [21].

The second new parameter, the number of incremental checkpoints taken before taking the next full checkpoint, reflects the periodic desire to take full checkpoints. Increased recovery latencies and increased storage costs are two factors that motivate the desire for 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 [40].

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 the full checkpoints. This assumption has been validated using the incremental checkpointing library described in [21].

Other Assumptions. Apart from the emperically observed data we use to parameterize our performance models, we assume each process uses 2 GB of memory (based on observed workloads at Sandia National Laboratories) and checkpoints $\frac{1}{3}$ of that memory [21], a five year node MTBF [41] and a per process I/O rate of 1 MB/s. This latter value was chosen optimistically based on a performance study on Argonne National Laboratory's 557 TFlop Blue Gene/P system (Intrepid) [42].

4.2.3. Modeling System Performance considering Costs

In our comparison of checkpoint data compression optimizations to hardware-based SSD solutions, we consider the relative financial costs of different system configurations. This study is meant to be instructive, not necessarily definitive, allowing us to make simplifying assumptions and the use of a relatively simple cost model. Using system cost factoring in the replacement of worn SSDs and amount of work completed in a fixed time span based on the system's hardare and software configuration, we create 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 before the device wears out. To compute the final procurement cost of an SSD-based system, we compute the number of weeks between SSD replacement based on their lifespan write capability and the average weekly checkpoint data commitment:

$$lifespan_{ssd}(weeks) = \frac{ssd\_lifespan\_write\_capability}{weekly\_checkpoint\_volume} \tag{2}$$

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:

$$tcost_{node} = cost_{node} + \left(cost_{ssd} \times \left\lceil \frac{lifespan_{system}(weeks)}{lifespan_{ssd}(weeks)} \right\rceil \right) \tag{3}$$

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 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 device specified rating [43]. As a result, our our model is optimistic as SSD devices may need to replaced more often.

Also, we only consider procurement costs and ignore ongoing costs to run and maintain the system. Effectively, we are estimating that any differences in expenses for running and maintaining systems of 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 ration 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:

$$Performance\_price = \frac{work \times efficiency}{t_{cost}(node) \times number\ of\ nodes} \tag{4}$$

We assume that the system is fully utilized (100% 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.

5. Checkpoint Compression Performance

5.1. Checkpoint Compression Viability

To test the viability of compression, we only focused 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 given a 100x100x100 problem size. phdMesh and LAMMPS were given a 5x5x5 problem size. Each application was run using 2–3 MPI processes, except for 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, which ranged in size from 311 MB to 393 MB for the mini apps to about 700 MB for LAMMPS.

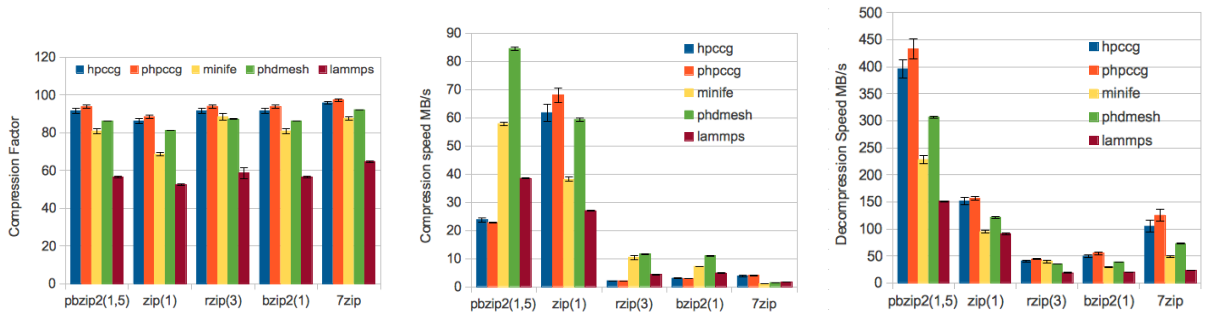
We used compression factor, α as our metric to show how compressible checkpoint data are, where we compute compression factor as: $1 - \frac{compressed\ size}{uncompressed\ size}$

Figure 2(a) shows how effective the various algorithms are at compressing checkpoint data. We can see that all the algorithms achieve a very high *compression factor* of about 70% or higher for the *mini apps* and about 57-65% for LAMMPS. This means, then that the primary distinguishing factor becomes the compression speed, that is, how quickly the algorithms can compress the checkpoint data.

Figures 2(b) and 2(c) show our empirically 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, since 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.

Based on the above results and Equation 1, which represents our viability model, Figure 3 demonstrates the checkpoint read/write bandwidths that make compression viable. For each application, the highest bar of all the compression algorithms represents its worse case scenario. For the worse case application, LAMMPS, checkpoint compression is viable unless a system can sustain a per process checkpoint read/write bandwidth of greater than about 34.6 MB/s. In the best case, phdMesh, the necessary per process checkpoint read/write bandwidth raises to greater than about 114.2 MB/s.

The relationship between compression performance (compression factor and compression and decompression speeds) and checkpoint I/O bandwidth is the key factor of the viability of checkpoint compression. As Figure 3 shows, for our worse case application, LAMMPS with pbzip2 compression, compression is viable if



(a) Compression Factor (b) Compression Speed (c) Decompression Speed

Figure 2: Checkpoint Compression Factors and Compression/Decompression Speeds. For compression factors, higher is better: a factor of 90% means that file size was reduced by 90%.

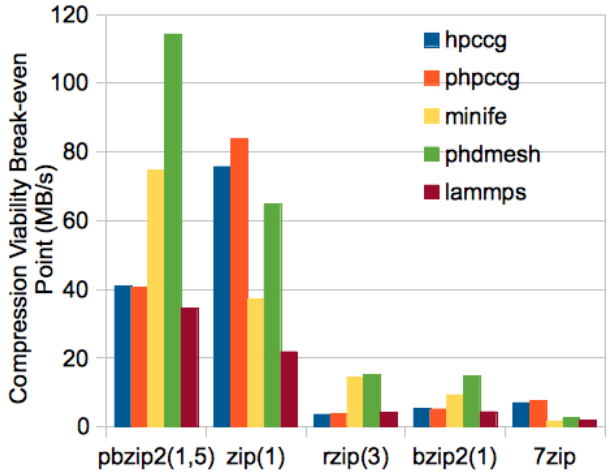


Figure 3: Checkpoint Compression Viability: Unless, checkpoint read/write bandwidth exceeds our viability factor (y-axis), checkpoint compression should be used.

per-process checkpoint bandwidths are less than 34.6 MB/s. In the best case, phdMesh with pbzip2 compression, per process checkpoint bandwidths must exceed 114.2 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) [42]. 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 1 MB/s for both reading and writing. This performance scales to about 32,768 processes and then 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 needed. Similarly, the Lawrence Livermore Dawn IBM BG/P system has a peak per-node checkpoint bandwidth of about 2 MB/s [3] As a result, aggressive use of checkpoint compression appears to be viable and indeed desirable on current large-scale platforms.

[3]Oak Ridge's Spider Lustre-based file system provides 240 GB/s of aggregate bandwidth[44], while Dawn's Lustre file system is listed as providing 70 GB/sec of peak bandwidth on LLNL reference pages [45].

5.2. Compressing System-level versus Application-level Checkpoints

Next, we examine the compression effectiveness of system-level checkpoints versus that of application specific checkpoints. 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 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. So it not only captures the application specific data but also saves shared library states etc. On the other hand application specific checkpointing only needs to save the data needed to resume 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 170MB in size compared to about 700MB BLCR generated checkpoints. However, based on our results in Table 1, 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 56.5% compared to 43.3%). This is because application level checkpoints are optimized so that data that can be reconstructed on an application restart are omitted from the checkpoints. This reduces the compressibility of the user level checkpoints. For the same reason, we observed the differences in sizes for these two types of checkpoints. Additionally, the average compression and decompression speeds were higher for system level checkpoints than for user level checkpoints (again for pbzip2, 94.8 MB/s compared to 87 MB/s).

	Compression Factor %		Compression Speed MB/s		Decompression Speed MB/s		Compression Viability Break-even point MB/s	
	pbzip(1,5)	**zip(1)**	**pbzip(1,5)**	**zip(1)**	**pbzip(1,5)**	**zip(1)**	**pbzip(1,5)**	**zip(1)**
System	56.46	52.49	38.46	27	150.8	90.52	34.6	21.84
Application	43.28	41.47	44.15	27.6	129.9	105.2	28.52	18.14

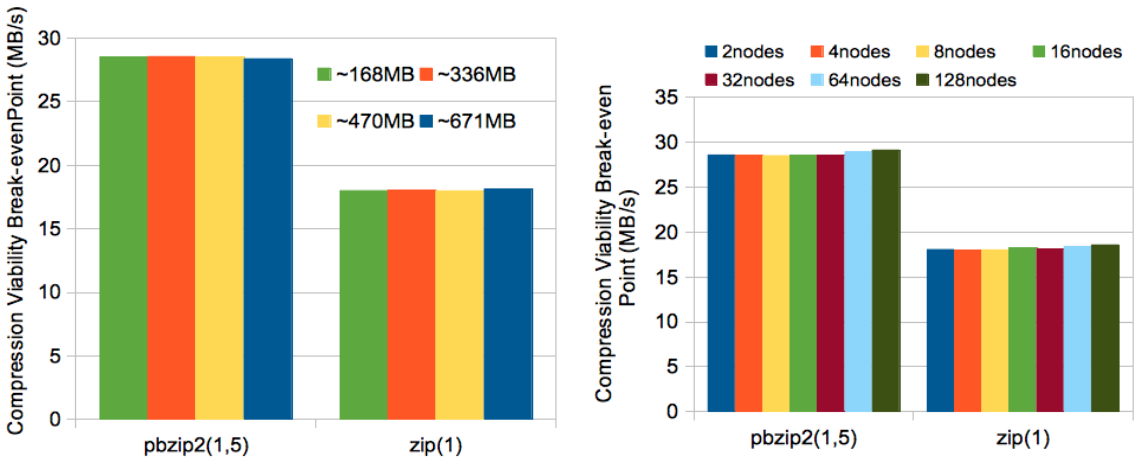
Table 1: Compression Break-even Points for System Level and Application Specific Checkpoints.

5.3. Checkpoint Compression Performance and Application Scale

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

In our first set of scaling experiments, we evaluate 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 increase. 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 4(a) shows the viability results from these experiments. We readily observe 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 compare the compression ratios for a weak scaling LAMMPS EAM simulation for between 2 and 128 MPI processes. In each test, the per-process restart file size is over 170 MB. In these runs we take 5 equally spaced checkpoints. Figure 4(b) shows once again that application process counts did not bear an impact on checkpoint compression viability. We have no reason to believe these results will be different for larger process count runs.



(a) Scaling Checkpoint Sizes and Application Runtime.

(b) Scaling Process Counts.

Figure 4: Results from our Scaling Experiments.

6. Understanding Checkpoint Compression Performance

Given the viability results from the previous section that show 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 speeds.

6.1. The Impact of Compression Factor

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

6.1.1. An Application-specific Case Study

Based on the compression performance results from Section 5.1, we focus 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 use knowledge of the LAMMPS on-disk checkpoint format to translate application-specific checkpoint data into its composite data elements. Using this, we compute the entropy of LAMMPS checkpoints using Shannon's information theory [46].

Shannon's theorem tells 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, as 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 2 shows the results of this minimal checkpoint encoding. This checkpoint contained about 3.5 million total symbols of which about 1 million were unique, 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

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

Table 2: Comparing a theoretical minimal encoding with bzip2.

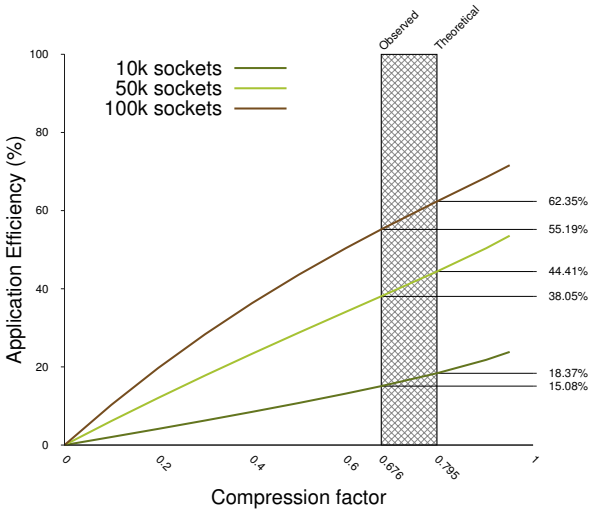


Figure 5: Varying compression factor

(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%.

Next, we use this LAMMPS checkpoint compression comparison data to model how LAMMPS performance would improve with this optimal algorithm that could better compress its checkpoints. Optimistically, we keep compression speed constant, assuming that the optimal algorithm would take no longer to run than bzip2. We look at three different scenarios, systems with 10K, 50K and 100K total sockets. Figure 5 shows the impact on application efficiency as compression factor varies, highlighting our observed compression factor and our theoretic maximum compression factor. For each of the three scenarios, we observe 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 standard text-based compression algorithms. In fact, with the expected growth of I/O on future systems, these differences in efficiencies will further decrease, supporting our position that current compression algorithms are sufficient for future systems as well.

6.2. The Impact of Compression Speed

While compression factor likely is the biggest determinant of the performance impact of checkpoint compression, we must also understand the importance of compression speed. We evaluate the potential benefits of accelerating our top performing (in terms of compression factor) algorithm, either using algorithmic enhancements or hardware technologies, for example, GPUs. Using the compression performance exhibited by pbzip2 on phpccg checkpoints (our top performer for compression factor) as a baseline, we varied compression and decompression rates in a range from a slow-down of 100 to a speed-up of 10,000.

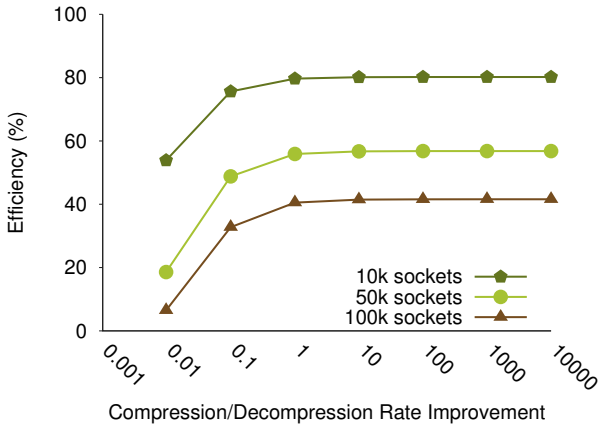


Figure 6: Varying compression/decompression speed

The results, shown in Figure 6, show that a four orders of magnitude improvement in speed would yield an insignificant improvement in application efficiency on current systems. While this is an important result, it is not so surprising: given current checkpoint commit rates (based on available per process I/O bandwidth to checkpoint storage), the time spent compressing a checkpoint is insignificant to the time spent committing the checkpoint to stable storage. What is unclear is the impact of compression speed increases with the expected I/O bandwidth increases expected in future systems.

These results suggest that attempting to improve compression rates is not worthwhile exploration 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.

Figure 7 shows the increase in application efficiency as a function of the per-node checkpoint commit bandwidth. Similar to previous work in this paper, we assume a 5 year socket MTBF and use optimal compression factors. The Y-axis in this is the difference in application efficiency in the accelerated and non-accelerated case. For the accelerated case, we assume a hypothetical compression of 100 times the CPU compression speeds. These optimal speedups have been observed with carefully crafted codes and workloads with GPUs [47]. We model these overheads for a number of node counts between 10k and 200k. From this figure, we see that a two order magnitude increase in compression/decompression speeds lead 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.

7. Checkpoint Compression and Other Optimizations

Finally, we put the performance of checkpoint compression in context by comparing against a number of popular software, hardware, and mixed hardware/software solutions. Also, we investigate 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 performance of the combination of both incremental checkpointing with compression. We then compare 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) [10].

7.1. Compression and Increment-based Optimizations

Figure 8 shows the comparison results of compression-based and increment-based scenarios alongside the standard cCR performance and make make several observations:

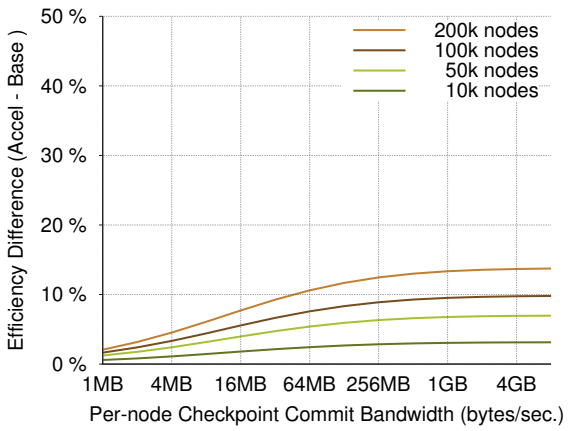


Figure 7: Efficiency increase for a number of node counts as a function per-node checkpoint commit speeds assuming a compression/decompression speed a factor of 100 greater than what we see on current systems. The efficiency difference is defined as the accelerated efficiency minus the efficiency using current speeds

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 checkpoint (labeled "ickpt"). This result is more notable than it may first appear: 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 unknown, checkpoint compression is a simple solution that can achieve better performance and with no programmer burden.
3. The combination of compression-based and increment-based optimizations yields the best performance of these software-only methods.

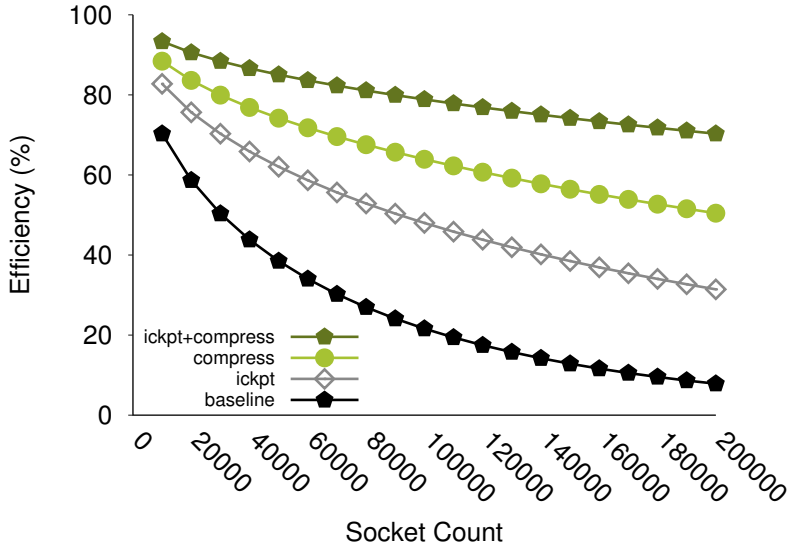


Figure 8: Impact of the software-only optimizations checkpoint compression and incremental Checkpointing on application efficiency.

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.

7.2. Compression and Other Optimizations

Next, we compare our checkpoint compression technique against the performance of two hardware-based checkpoint optimizations. More specifically, we compare against a local SSD checkpointing solution [27] and a multi-level solution(SCR) that uses local and remote memory, SSDs, a parallel file system, and a software RAID to ensure reliability [10]. It is important to note that these hardware checkpointing solutions are considerably more expensive than a software only solution such as incremental and compression-based checkpointing. In fact, the device reliability required for the SSD only solution maybe 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 [10] 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; with the dramatic core count increases, we are moving from a compute-scare environment to one where we have an abundance of compute cycles but a scarcity of memory.

Again, we assume each process uses 2GB of memory and checkpoints $\frac{1}{3}$ of that memory. We also assume 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 assume a 2GB/s checkpoint commit rate and a 8GB/sec checkpoint read rate. Lastly, for SCR, we assume a per-process mean checkpoint commit rate of 211MB/s for both read and write. This mean commit rate is calculated from [48], where the authors presented a user-space file system, CRUISE, which dramatically improve 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 9 shows a comparison of compression with the hardware-based techniques outlined in this section. For comparison we also include the efficiency of standard rollback/recovery to the parallel filesystem shown previously. From this figure we make the following observations:

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 count 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 two orders magnitude slower commit speeds, 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 get within 20% of a costly hardware solution. Compression based approach can be made to be readily available to existing systems while for the hardware based solutions we need to make changes to existing systems and install hardware to support it.

7.3. A Performance/Prive Evaluation of SSD-based Systems

In this section, we examine the cost efficiency of these hardware-based, software-based, and hybrid methods CR optimization strategies. For this study, we compute and compare the performance-price for a hypothetical cluster under different configurations that map to hardware-based CR optimizations, namely SSD-enhanced, and software-based CR optimizations, namely compression and incremental checkpointing. Recall our performance-price model from Section 4.2.3:

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

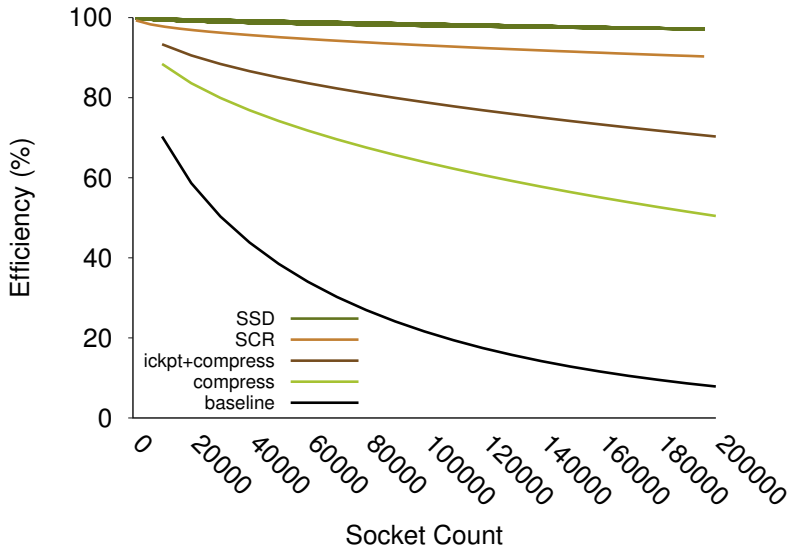


Figure 9: Comparison of hardware/multi-level checkpointing techniques with pure software techniques like compression and incremental checkpointing

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 assume a system lifespan of 260 weeks (five years) and our workload comprises one process per core and executes for the entire 260 weeks. We use application efficiencies obtained from the results in the previous section: 90.94% efficiency for the SSD-based optimizations and 71.025% efficiency for the software-based optimizations.

We compute $ssd\_lifespan\_write\_capability$ 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 shown in Table 3.

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

Table 3: Endurance ratings and price for various SSDs

We compute the last column of Table 3, $lifespan_{ssd}(weeks)$, 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 using Daly's model to calculate the number of checkpoint commits to each SSD per week.

Using the above method, Figure 10 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 increases, SSD cost overheads are amortized such that hardware-based become almost as cost-efficient as the software based one.

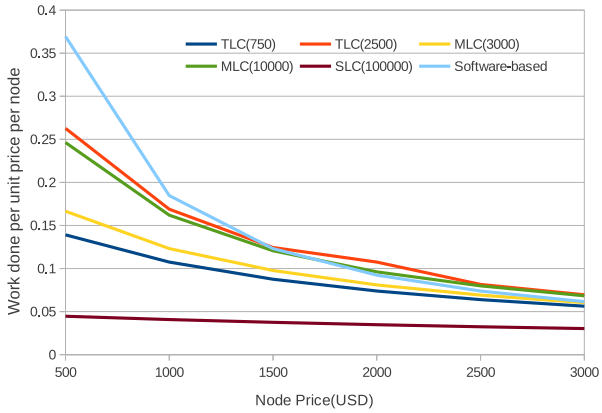


Figure 10: Comparison of work done per unit price per node for a system with different types of SSD device compared against software-based solution. (higher is better).

8. Conclusion

In this article, we showed checkpoint data compression to be 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 theoretical optimal 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 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. Finally, we 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.

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 toward efficient application 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. A remaining open question we are currently investigating is the power/energy considerations of checkpoint data compression and their impact on large-scale systems.

[1] B. Schroeder and G. A. Gibson, "A Large-scale Study of Failures in High-performance Computing Systems," in *Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006.

[2] K. Ferreira, R. Riesen, P. Bridges, D. Arnold, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, and R. Brightwell, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *SC*. ACM, Nov 2011.

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

[4] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," *2012 41st International Conference on Parallel Processing*, vol. 0, pp. 148–157, 2012.

[5] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[6] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 10, pp. 972–986, oct 1998.

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

[8] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *International Parallel Processing Symposium*. Honolulu, HI: IEEE Computer Society, April 1996, pp. 526–531.

[9] V. C. Zandy, B. P. Miller, and M. Livny, "Process Hijacking," in *8th International Symposium on High Performance Distributed Computing (HPDC '99)*, Redondo Beach, CA, August 1999, pp. 177–184.

[10] A. Moody, G. Bronevetsky, K. Mohror, and B. 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)*, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[11] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '95/PERFORMANCE '95. New York, NY, USA: ACM, 1995, pp. 64–73. [Online]. Available: http://doi.acm.org/10.1145/223587.223596

[12] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[13] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software – Practice & Experience*, vol. 29, no. 2, pp. 125–142, 1999.

[14] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *IEEE International Symposium on Parallel&Distributed Processing*, 2009, pp. 1–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1586640.1587642

[15] Y. Chen, K. Li, and J. S. Plank, "CLIP: A Checkpointing Tool for Message-passing Parallel Programs," in *SuperComputing '97*, San Jose, CA, 1997. [Online]. Available: http://citeseer.ist.psu.edu/chen97clip.html

[16] E. N. Elnozahy, D. B. Johnson, and W. Zwaenpoel, "The Performance of Consistent Checkpointing," in *11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, 1992. [Online]. Available: http://citeseer.ist.psu.edu/elnozahy92performance.html

[17] K. Li, J. F. Naughton, and J. S. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, August 1994.

[18] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995, pp. 213–224.

[19] M. Paun, N. Naksinehaboon, R. Nassar, C. Leangsuksun, S. L. Scott, and N. Taerat, "Incremental Checkpoint Schemes for Weibull Failure Distribution," *International Journal of Computer Science*, vol. 21, no. 3, pp. 329–344, 2010.

[20] S. Al-Kiswany, M. Ripeanu, 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*, june 2008, pp. 613–624.

[21] K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold, "Libhashckpt: Hash-based Incremental Checkpointing Using GPUs," in *Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[22] C.-C. Li and W. Fuchs, "CATCH-compiler-assisted techniques for checkpointing," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, jun 1990, pp. 74–81.

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

[24] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," University of Tennessee, Tech. Rep. CS-95-302, August 1995. [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-95-302.html

[25] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. De Supinski, and R. 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.

[26] A. Moshovos and A. Kostopoulos, "Cost-Effective, High-Performance Giga-Scale Checkpoint/Restore," University of Toronto, Tech. Rep., November 2004.

[27] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Proceedings of the nternational Parallel and Distributed Processing Symposium*, ser. IPDPS '13. New York, NY, USA: ACM, 2013.

[28] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratory, Tech. Rep. SAND2009-5574, 2009.

[29] S. J. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal Computation Physics*, vol. 117, pp. 1–19, 1995.

[30] Sandia National Laboratories. (2010, April) The LAMMPS Molecular Dynamics Simulator. [Online]. Available: http://lammps.sandia.gov

[31] K. G. Morse Jr., "Compression Tools Compared," *Linux Journal*, no. 137, September 2005.

[32] P. Deutsch, "Deflate Compressed Data Format Specification." [Online]. Available: ftp://ftp.uu.net/pub/archiving/zip/doc

[33] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, May 1977.

[34] "7Zip Project Official Home Page." [Online]. Available: http://www.7-zip.org

[35] I. Pavlov, "LZMA SDK (Software Development Kit)," 2007. [Online]. Available: http://www.7-zip.org/sdk.html

[36] J. G. Elytra, "Parallel Data Compression With Bzip2."

[37] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006.

[38] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2004, vol. 3241, pp. 353–377, 10.1007/978-3-540-30218-6_19. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30218-6_19

[39] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *SC*, S. Lathrop, J. Costa, and W. Kramer, Eds. ACM, 2011, p. 33.

[40] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 783–788. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2008.109

[41] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *Journal of Physics Conference Series*, vol. 78, no. 1, 2007.

[42] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654100

[43] R. Templeman and A. Kapadia, "Gangrene: Exploring the mortality of flash memory," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, ser. HotSec'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372387.2372388

[44] G. Shipman, D. Dillow, S. Oral, and F. 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.

[45] B. Barney. (2011, August) Introduction to Livermore Computing Resources. [Online]. Available: http://computing.llnl.gov/tutorials/lc_resources

[46] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 623–, july, october 1948.

[47] A. Colic, H. Kalva, and B. Furht, "Exploring nvidia-cuda for video coding," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, ser. MMSys '10. New York, NY, USA: ACM, 2010, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/1730836.1730839

[48] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. 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*, 2013, pp. 143–154.