On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-based Fault Tolerance

Dewan Ibtesham, Dorian Arnold, and Patrick G. Bridges
Department Of Computer Science
The University of New Mexico
Albuquerque, NM 87131
{*dewan,darnold,bridges*}*@cs.unm.edu*

Kurt B. Ferreira and Ron Brightwell
Scalable System Software Department
Sandia National Laboratories
Albuquerque, NM 87185–1319
{*kbferre,rbbrigh*}*@sandia.gov*

***Abstract*—The increasing size and complexity of high performance computing (HPC) systems have lead to major concerns over fault frequencies and the mechanisms necessary to tolerate these faults. Previous studies have shown that state-of-the-field checkpoint/restart mechanisms will not scale sufficiently for future generation systems. Therefore, optimizations that reduce checkpoint overheads are necessary to keep checkpoint/restart mechanisms effective. In this work, we demonstrate that checkpoint data compression is a feasible mechanism for reducing checkpoint commit latency and storage overheads. Leveraging a simple model for *checkpoint compression viability*, we show: (1) checkpoint data compression is feasible for many types of scientific applications expected to run on extreme scale systems; (2) checkpoint compression viability scales with checkpoint size; (3) user-level versus system-level checkpoints bears little impact on checkpoint compression viability; and (4) checkpoint compression viability scales with application process count. Lastly, we describe the impact checkpoint compression might have on projected extreme scale systems.**

***Keywords*-Fault tolerance; Checkpoint Compression;**

I. Introduction

High-performance computing (HPC) systems have been increasing dramatically in size, and this trend is expected to continue. On the current Top 500 list [1], 300 (or 60%) of the 500 entries have greater than 8,192 cores, compared to 17 (or 3.4%) just 5 years ago. Also on this list, four of the systems have more than 200K cores; an additional seven have more than 128K cores, and another 10 have more than 64K cores. This year, the Lawrence Livermore National Laboratory schedules to deploy its 1.6 million core system, Sequoia [2], and future extreme scale systems are projected to have on the order of tens to hundreds of millions of cores by 2020 [3].

Future high-end systems are also expected to increase in complexity; for example, heterogeneous systems like CPU/GPU-based systems are expected to become much more prominent. We expect increased system sizes along with this increased complexity to yield extremely low mean times between failures (MTBF). Recent studies indeed show that system failure rates depend on the numbers of processor chips and that system MTBF for the biggest systems on the Top 500 lists are expected to fall below 10 minutes in the next few years [4]

In HPC, checkpoint/restart is perhaps the most commonly employed fault-tolerance mechanism for applications. Yet, as we describe in Section II, increasing checkpoint/restart overheads coupled with higher failure frequencies threaten to make checkpoint/restart infeasible for future systems. Compressing checkpoint data is, perhaps, an obvious strategy for improving checkpoint/restart efficiency, but compression is not viable if its benefits do not outweigh its costs. Alas, the majority of applications and systems that use checkpoint/restart do not compress checkpoint data. In this work, we demonstrate that given the current and increasing gap between processing and data transfer capabilities, checkpoint compression can be a very effective strategy for improving both the time and space efficiency of checkpoint/restart-based fault tolerance. We use a combination of *mini-applications* or *mini apps* [5] and a representative scientific application, LAMMPS [6] along with the Berkeley Lab Checkpoint/Restart (BLCR) framework [7] and a set of off-the-shelf compression utilities to study the viability of checkpoint compression. In this paper, we present the result of this study and make the following contributions:

- We offer a viability model for checkpoint data compression that accounts for the cost and benefits of compression for both checkpoint commit and recovery operations;
- We show that checkpoint data compression can be a very effective strategy for reducing checkpoint and restart latencies;

- We show what compression algorithms are best suited for checkpoint data compression;
- We show that application scale, in terms of memory footprint size, process counts, or run time can bear little impact on the effectiveness of checkpoint data compression;
- We show that checkpoint data compression can be effective for both application-level and system-level checkpoints;
- We show that checkpoint data compression can improve significantly an application's makespan, the application's time to solution in the presence of failures; and
- We offer a discussion of checkpoint data compression given current high performance processor and I/O technologies and trends.

The organization of this paper is as follows: in the next section, we give a background of the checkpoint/restart mechanism and a survey of related checkpoint compression work. In Section III, we present our checkpoint compression viability model and describe how we can use it to model both coordinated and uncoordinated distributed checkpointing protocols. In Section IV, we describe the applications, compression algorithms and the checkpoint library that comprise our evaluation framework as well as our experimental results. We conclude with a discussion of the implications of our experimental results for future checkpoint/restart research, development and deployment.

II. Background and Related Work

During normal operation, *checkpoint/restart* (or *rollback recovery*) protocols [8], periodically record process state to stable storage devices, devices that survive tolerated failures. Process state comprises all the state necessary to run a process correctly including its address space or memory footprint and register states. When a process fails, a new incarnation of the failed process is *recovered* from the intermediate state of the failed process' most recent checkpoint – thereby reducing the amount of lost computation. Checkpoint/restart is a well studied, general fault tolerance mechanism. However, recent studies [4], [9], [10] predict poor utilizations (approaching 0%) for applications running on imminent systems and the need for dedicated reliability resources.

A. Checkpoint Optimizations

Focusing on the checkpoint on the *checkpoint commit* problem, saving a checkpoint to stable storage, we can consider two sets of checkpoint optimization strategies. The first set of strategies hide or reduce (perceived) commit latencies without actually reducing the amount of data to commit. These strategies include *concurrent checkpointing* [11], [12], *diskless* or *multi-level checkpointing* [13]–[15], *remote checkpointing* [16], [17] and checkpointing filesystems [18]. The second set of strategies reduce commit latencies by reducing checkpoint sizes. These strategies include *memory exclusion* [19] and *incremental checkpointing* [20]–[22]. In Section V, we discuss the potential interplay between these optimizations and checkpoint compression.

B. Related Compression Research

Li and Fuchs implemented a compiler-based checkpointing approach, which exploited compile time information to compress checkpoints [23]. They concluded from their results 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 compression and showed that, for their computational platform, compression was beneficial if a compression factor greater than 19.3% could be achieved [24]. In a related vein, Plank et al also proposed *differential compression* to reduce checkpoint sizes for incremental checkpoints [25]. Moshovos and Kostopoulos used hardware-based compressors to improve checkpoint compression ratios [26]. Finally, in a related but different context, Lee et al study compression for data migration in scientific applications [27].

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

III. A Checkpoint Compression Viability Model

Intuitively, checkpoint compression is a viable technique for improving the performance of checkpoint/restart protocols when the benefits of checkpoint data reduction outweigh the costs of reducing the checkpoint data. Our viability model is inspired by the concept offered by Plank et al [24]. Plank et al focused solely on the impact of compression for the checkpoint commit phase. Our model addresses the cost and benefits of compression for both checkpoint and recovery phases. In Section IV-E, we use the results from this model to determine the overall impact of checkpoint compression on application performance.

We assume that individual processes of a distributed application are checkpointed in a coordinated fashion: all processes coordinate at the end of each checkpoint

interval to checkpoint a globally consistent application state comprised of one checkpoint per process. This is the commonly employed strategy in the HPC domain. We also assume that there are an equal number of checkpoint and recovery operations. Our justification for this latter assumption follows: optimally, an application takes a single checkpoint before each failure – upon failure, only the most recent checkpoint is used, therefore, other checkpoints are not useful. The application only needs to recover once per failure. Therefore, in the optimal case, the number of checkpoints equals the number of failures, which equals the number of recoveries. There are various works that define optimal checkpoint intervals [28], [29]. 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:

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

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 $rcommit$ is *commit speed* or the rate of checkpoint commit or reading (including all associated overheads). The last equation can be reduced to:

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

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.

IV. An Evaluation of Checkpoint Compression

In this study, we seek to answer several fundamental questions regarding checkpoint data compression:

- *Can benefit of compressed checkpoints outweigh the additional latencies necessary to compress and decompress the checkpoint?*
- *Do the time or space scales of an application impact checkpoint data compression?*
- *Does the viability of checkpoint data compression change for application-level versus system-level checkpoints?*; and ultimately
- *What real impact can checkpoint compression have on the execution time of an application?*

We now describe the applications, tools and experiments we use to answer these questions and discuss the conclusions we have made based on our experimental and modeling results.

For all but our scaling experiments, we used a 64-bit, four core Intel Xeon processor with a 2.33 GHz clock cycle rate and 2 GB of memory. For our scaling study, we collected checkpoints from application runs on a Cray XT5 series machine. However, for uniformity and ease of access to the compression utilities, these checkpoints also were compressed/decompressed on our four core workstation.

A. Evaluation Tool Chain

We used a range of applications, libraries and utilities in this study. In this section, we describe these various components.

1) The Mini Applications: We chose four *mini-applications* or *mini apps* from the Mantevo Project [5], 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.

Mini apps are small, self-contained programs that embody essential performance characteristics of key applications. While the Mantevo mini apps are not (yet) as popular as other HPC benchmarks, like the NAS Parallel Benchmarks or the HPC Challenge Benchmark, we believe the mini apps are much better suited for this study. HPC benchmarks generally target the evaluation of computer system performance. On the other hand, the mini apps are meant to be lightweight application proxies for the heavyweight counterparts. In other words, the mini apps are intended to mimic real application characteristics including the memory footprint properties relevant to this checkpoint compression study.

2) A Full Application: LAAMPS: We use LAMMPS (the Large-scale Atomic/Molecular Massively Parallel Simulator) to evaluate checkpoint compression on a full featured scientific application. LAMMPS [6], [30] 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.

3) Compression Utilities: For this study, we focused on the popular compression algorithms investigated in Morse's comparison of compression tools [31]. We do not present results from some algorithms that did not perform well. Additionally, some algorithms can be parameterized to trade between execution time for compression factor. We only present the parameter set that represents the best trade-off.

- **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. The `zip` algorithm treats all types of data as a continuous stream of bytes. Within this stream, duplicate strings are matched and replaced with pointers followed by replacing symbols with new, weighted symbols based on frequency of use.

  `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 backend 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) Checkpoint/Restart Utilities: The Berkeley Lab Checkpoint/Restart library (BLCR) [7], a system-level infrastructure for checkpoint/restart, is an open source checkpoint/restart library and is deployed on several HPC systems. For most of our experiments, excluding some application specific checkpoints taken with LAMMPS, we obtained checkpoints using BLCR. Furthermore, we use the OpenMPI [37] framework, which has integrated BLCR support.

For our scaling study we used a user-level checkpoint 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 applications memory footprint.

B. Evaluating Checkpoint Compression Effectiveness

We compressed and decompressed many checkpoints collected from our application suite using the different compression utilities. For each experiment, we measured the performance metrics the performance metrics necessary to determine checkpoint viability using Equation 1 from Section III, namely compression factor, compression speed and decompression speed.

For our baseline experiments, we were not concerned about scaling along either the time our space dimentions. We chose 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 timestep 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.

Figure 1 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, where compression factor is computed as: $1 - \frac{compressed\ size}{uncompressed\ size}$. 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(a) and 2(b) show compress 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

$$\frac{2\alpha \times r_{comp} \times r_{decomp}}{r_{comp} + r_{decomp}} < r_{commit},$$

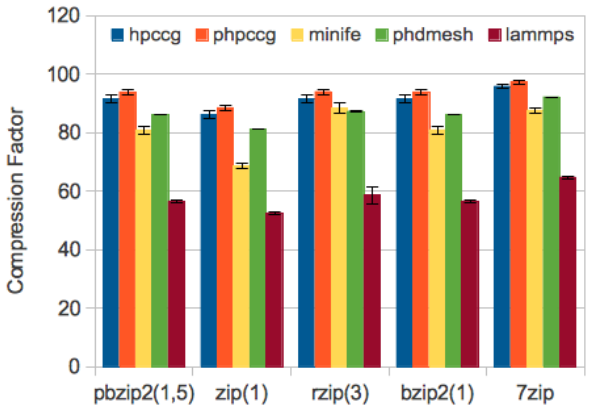


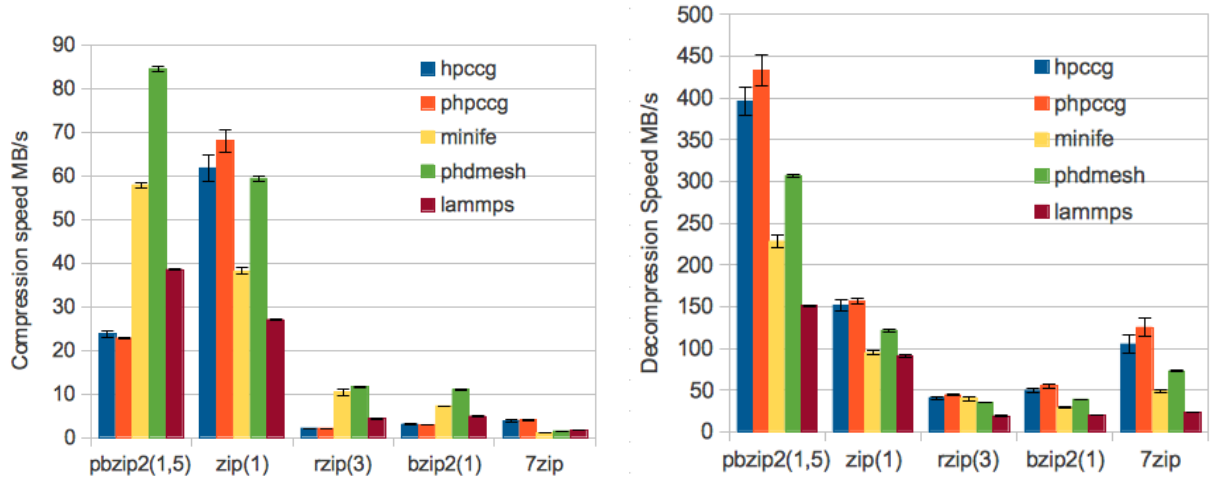
Figure 1. Checkpoint compression factors for the various algorithms and applications. Higher is better: a factor of 90% means that file size was reduced by 90%.

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 three GB/s. In the best case, phdMesh, the necessary per process checkpoint read/write bandwidth raises to greater than 11 GB/s. In Section V, we describe the impact of these results in the context of extreme scale systems. The executive summary is that checkpoint compression is a very viable solution for current and projected HPC systems. (Since pbzip2 and zip performance dominate those of the other compression utilities, for the remainder of this paper, we only present results for these two algorithms.)

C. Evaluating the Impact of User versus System 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. Application specific checkpointing only needs to save the data needed to resume operation. As a



(a) Compression

(b) Decompression

Figure 2. Checkpoint Compression and Decompression Speeds.

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 I, we observe that checkpoint compression is viable for both application specific and system level checkpoints.

There is, however, a qualitatitve 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%). 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).

	pbzip(1,5)	**zip(1)**
System Checkpoint	3.38 GB/s	2.13 GB/s
Application Checkpoint	2.79 GB/s	1.77 GB/s

Table I
COMPRESSION BREAK-EVEN POINTS FOR SYSTEM LEVEL AND APPLICATION SPECIFIC CHECKPOINTS.

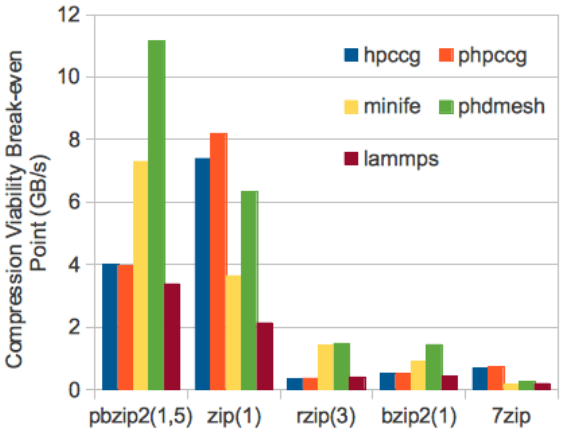
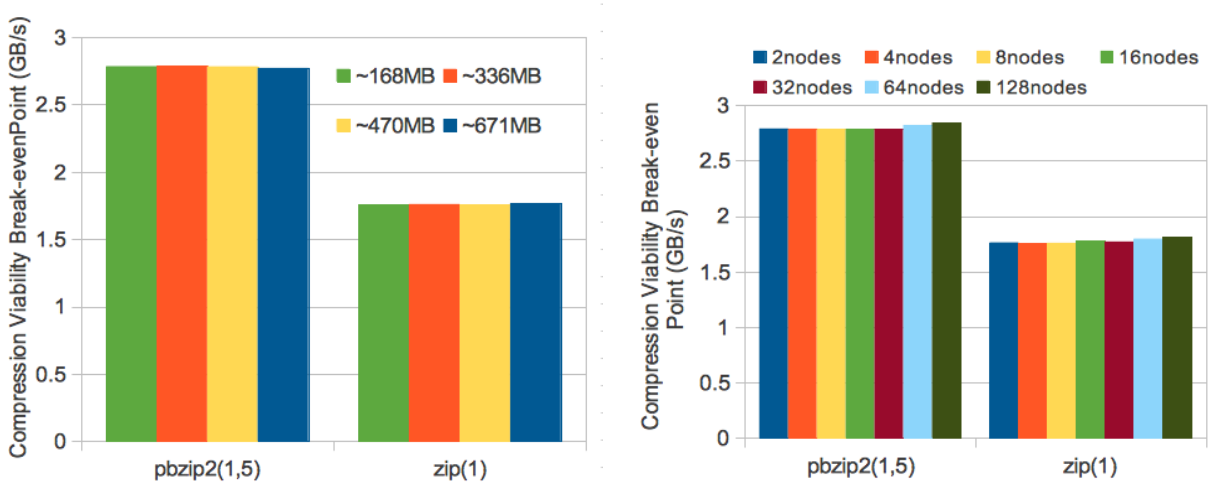


Figure 3. Checkpoint Compression Viability: Unless, checkpoint read/write bandwidth exceeds our viability factor (y-axis), checkpoint compression should be used.

D. Evaluating the Impact of Scale

For our scaling experiments, we use the LAMMPS and its 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 applicaiton processes fixed at two. In this manner, memory footprint and checkpoint sizes increases. 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.



(a) Scaling Checkpoint Sizes and Application Runtime.

(b) Scaling Process Counts.

Figure 4. Results from our Scaling Experiments.

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.

E. Performance Impact of Compression

To outline the impact of checkpoint compression on application time to solution, we created a performance model for expected time to solution for an application with checkpoint/restart. This model is based on Daly's higher order model [28], which assumes node failures are independent and exponentially distributed. The model takes as input the mean time between failures (MTBF) for the system, the checkpoint commit time, the checkpoint restart time, the number of nodes used in the application and the time the application would take to complete in a failure-free environment.

We modified this model to integrate checkpoint compression and decompression. For the checkpoint commit time we included the time to compress the checkpoint image as well as the time to write this compressed image to stable storage on the parallel system. Similarly, on restart we included the time to read the compressed checkpoint image and perform the decompression step.

In Figure 5, we show the result of this model. In this figures we show the efficiency of an application computation. This efficiency metric is defined as the time to solution in the failure environment divided by the time to solution in a failure-free environment. For this figure we use the best compression ratio and rates for each application described previously in the paper. In addition, this model assumes each node uses 2GB of memory and that $\frac{1}{3}$ of that memory is written on each checkpoint. These values are representative of what we have observed at the Sandia National Laboratory for our capability workloads. Finally, we assume a five year node MTBF as has been measured in current studies [38].

Regarding file I/O rates to stable storage, we use a report based on a study of I/O performance on Argonne National Laboratories 557 TFlop Blue Gene/P system (Intrepid) [39] to select I/O rates for our model. This work executes an I/O scaling study majoring 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. At any rate, for our study, we optimistically choose the best observed per process I/O bandwidth of 1 MB/s.

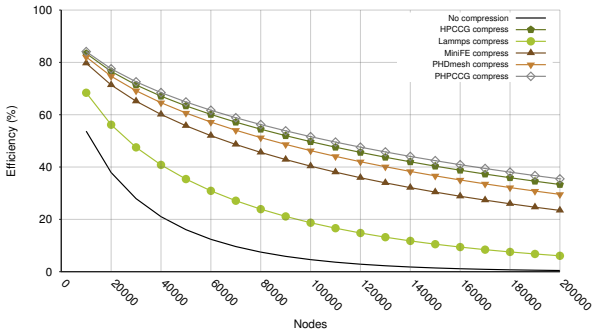


Figure 5. Impact of Checkpoint Compression on Application Efficiency.

From this figure we see that each of the compression rates measured in this work have a dramatic and positive influence on the performance of traditional checkpoint/restart at scales seen in today's systems as well as the expected scales of future systems.

V. Discussion

A. Compression versus Checkpoint I/O Bandwidth

The relationship between compression performance (compression factor and compression and decompression speeds) 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 per-process checkpoint bandwidths are less than 3 GB/s. In the best case, phdMesh with pbzip2 compression, per process checkpoint bandwidths must exceed 11 GB/s. In Section IV-E, we described the best observed per process I/O bandwidths that we found in the literature, 1 MB/s. For comparison, 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 [1] As a result, aggressive use of checkpoint compression appears to be viable and indeed desirable on current large-scale platforms.

The performance impact of checkpoint compression on future systems depends highly on future computer storage architecture developments. One report suggested using multiple disks per processor to provide sufficient storage bandwidth for high-speed checkpointing (5 GB/sec per process) [3]. Other researchers have suggested using similar approaches that combine non-volatile memories with spinning storage to reduce the potential costs of the checkpoint file systems [42], though anticipated storage system costs are still $60M. These aggressive bandwidths are at the boundary of where checkpoint compression is viable, and so it is unclear whether or not checkpoint compression would be useful if such high-bandwidth (and expensive) I/O systems were adopted.

Checkpoint compression also reduces the bandwidth and storage pressures on checkpointing file systems. If energy consumption and cost of such storage systems are important design limiters compared to the CPU power and costs, as is expected [3], checkpoint compression could be an important technology in reducing the demands on exascale storage systems.

B. Compression versus Other Checkpoint Optimizations

We believe that checkpoint compression is completely complementary to other known checkpoint optimizations such as the ones listed in Section II. Latency saving techniques diskless, multi-level and remote checkpointing still require the transfer of data from one node to another. As such, checkpoint compression can still significantly decrease the time it takes to transfer a checkpoint. Additionally, techniques that store checkpoints in remote DRAM memories to avoid disk latencies benefit from reduced checkpoint sizes. Particularly for capability class applications, which are resource intensive and often memory-bound, checkpoint compression would reduce storage pressures on the memory system.

Optimization strategies that also aim at reducing checkpoint sizes, like memory exclusion and incremental checkpointing, can benefit further from data compression. For example, if applications employing these optimizations have similar "memory footprint features" as the applications in this study, the portions of the application processes' address space that still need to be checkpointed would demonstrate the same compression viability features as in this study.

C. Compression Viability Model Assumptions

In this study, we assumed coordinated distributed checkpoints, In actuality, for our viability model, it does not really matter how many processes are checkpointing simultaneously. Our model specifies the minimum per process (or per checkpoint) bandwidth render compression useless. This is the case whether all processes are actively checkpointing simultaneously or some subset thereof (including singleton sets).

[1] Oak Ridge's Spider Lustre-based file system provides 240 GB/sec of aggregate bandwidth [40], while Dawn's Lustre file system is listed as providing 70 GB/sec of peak bandwidth on LLNLL reference pages [41].

We also assumed an equal number of checkpoints and recoveries. The equations in Section III can easily be modified to accommodate a disproportionate number of checkpoints to recoveries. Lastly, we assumed a synchronous checkpoint commit mechanism in which the target process is preempted until the checkpoint has been written to stable storage. As such, we can safely assume that the application is interupted completely for the entire checkpointing operation. If checkpoints could be committed asynchronously, our model would have to account for the reduced **perceived** latency and, therefore, reduced impact of checkpoint commit on application performance.

D. Future Enhancements

Our results show that different compression algorithms exhibit different performance on different application checkpoints. We would like to understand what aspects and features of the checkpoint data impact compression algorithm performance. This would help us predict the optimal compression algorithm for a particular application as well as give us insights into how we might improve an algorithms performance.

The positive results from standard, off-the-shelf compression utilities suggests that we might be able to yield even better results with some customizations. As suggested above, a detailed study what makes a good or bad compression algorithm for checkpoint compression could lead to optimization opportunies. Another promising avenue, which we are now exploring, is the use of GPUs for accelerating compression speeds.

Acknowledgments

This work was supported in part by Sandia National Laboratories subcontract 438290. Sandia National Laboratories is a multiprogram 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. The authors are grateful to the members of the Scalable Systems Laboratory at the University of New Mexico and the Scalable System Software Group at the Sandia National Laboratory for helpful feedback on portions of this study.

REFERENCES

[1] "Top 500 Supercomputer Sites," http://www.top500.org/ (visited March 2012). [Online]. Available: http://www.top500.org/

[2] "ASC Sequoia," https://asc.llnl.gov/computing_resources/sequoia (visited May 2011). [Online]. Available: https://asc.llnl.gov/computing_resources/sequoia/

[3] K. Bergman *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep., September 2008.

[4] 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.

[5] 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.

[6] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal Computation Physics*, vol. 117, pp. 1–19, 1995.

[7] 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.

[8] 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.

[9] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, April-June 2004.

[10] 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*, S. Lathrop, J. Costa, and W. Kramer, Eds. ACM, Nov. 2011.

[11] D. Z. Pan and M. A. Linton, "Supporting reverse execution for parallel programs," in *1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*. Madison, WI: ACM Press, 1988, pp. 124–129.

[12] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpoint for parallel programs," in *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*. Seattle, Washington: ACM, 1990, pp. 79–88.

[13] 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

[14] 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.

[15] 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

[16] G. Stellner, "Cocheck: Checkpointing and process migration for MPI," in *International Parallel Processing Symposium*. Honolulu, HI: IEEE Computer Society, April 1996, pp. 526–531.

[17] 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.

[18] 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

[19] 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.

[20] 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: citeseer.ist.psu.edu/elnozahy92performance.html

[21] 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

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] A. Moshovos and A. Kostopoulos, "Cost-effective, high-performance giga-scale checkpoint/restore," University of Toronto, Tech. Rep., November 2004.

[27] J. Lee, M. Winslett, X. Ma, and S. Yu, "Enhancing data migration performance via parallel data compression," in *International Parallel and Distributed Processing Symposium*, 2002, pp. 444–451.

[28] 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.

[29] 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.

[30] Sandia National Laboratories. (2010, April) The LAMMPS molecular dynamics simulator. [Online]. Available: http://lammps.sandia.gov

[31] K. G. M. 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," 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] 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

[38] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics Conference Series*, vol. 78, no. 1, 2007.

[39] 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

[40] 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.

[41] B. Barney. (2011, August) Introduction to livermore computing resources. [Online]. Available: http://computing.llnl.gov/tutorials/lc_resources

[42] G. Grider, "Exa-scale FSIO: Can we get there? can we afford to?" in *Proceedings of the 7th IEEE Workshop on Storage Network Architecture and Parallel I/Os*, 2011.

[43] S. Lathrop, J. Costa, and W. Kramer, Eds., *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. ACM, 2011, 2011.