SAND2011-7378C

Checkpoint Compression for Extreme Scale 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

Over the past few decades, high-performance computing (HPC) systems have increased dramatically in size, and this trend is expected to continue. On the most recent Top 500 list [1], 223 (or 44.6.%) of the 500 entries have greater than 8,192 cores, compared to 15 (or 3.0%) just 5 years ago. Also from this most recent listing, four of the systems are larger than 200K cores; an additional six are larger than 128K cores, and another six are larger than 64K cores. The Lawrence Livermore National Laboratory is scheduled to receive its 1.6 million core system, Sequoia [2], this year. Furthermore, future extreme systems are projected to have on the order of tens to hundreds of millions of cores by 2020 [3].

With this increased scale, 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. Increased complexity generally suggests that individual components likely will be more failure prone. Increased system sizes also will contribute to extremely low mean times between failures (MTBF), since MTBF is inversely proportional to system size. Recent studies indeed conclude that system failure rates depend mostly on system size, particularly, the number of processor chips in the system. These studies project that system MTBF for the biggest systems on the Top 500 lists will fall below 10 minutes in the next few years [4], [5]

Checkpoint/restart is perhaps the most common application fault-tolerance mechanism in the HPC domain. Yet, as we describe in Section II, increased checkpoint overheads coupled with more frequent failure occurrences threaten to make checkpoint/restart infeasible for future extreme scale systems. In this work, we focus on reducing checkpoint data volumes, particularly via checkpoint compression. The goal of this research is to gain an understanding of the role data compression should have with regard to checkpoint/restart fault-tolerance mechanisms.

Using several *mini-applications* or *mini apps* from the Mantevo Project [6], a real scientific application, LAMMPS [7], the Berkeley Lab Checkpoint/Restart (BLCR) framework [8] and a myriad of compression utilities, we explore the feasibility of state-of-the-field compression techniques for efficiently reducing checkpoint sizes. We use a simple *checkpoint compression viability model* to determine when checkpoint compression is a sensible choice, that is, when the benefits of data reduction outweigh the drawbacks of compression latency. In a preliminary study, we began to investigate the feasibility of data compression for reducing checkpoint commit latencies [9]. The results from this study (presented in Section IV) demonstrated that compression indeed should be considered as a part of the checkpoint/restart solution. In this work, we extend our previous study to address various shortcomings. In total, this paper presents:

- A study of the effectiveness of checkpoint data compression (including the results from our pre-

liminary study);
- A study of the impact of application scale (in memory footprint, time and process counts) on checkpoint data compression;
- A study of the impact of application-level versus system-level checkpoints on checkpoint data compression; and
- A discussion of the position of checkpoint data compression given current high performance processor and I/O technologies and trends.

Why is checkpoint data compression not considered more often?: This study sheds some new light on the huge impact checkpoint data compression can have on fault-tolerance for the types of scientific applications expected to run at large scales on future extreme scale systems. Particularly since data processing continues to become significantly cheaper than data movement, it is our hope that these results can help to make checkpoint data compression a more commonly used solution.

In the next section, we give a background of the checkpoint/restart mechanism and a survey of state-of-the-art checkpoint/restart enhancements. In Section III, we present our checkpoint compression viability model and the related checkpoint compression work. 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

Checkpoint/restart [10] is perhaps the most commonly used HPC fault-tolerance mechanism. During normal operation, checkpoint/restart protocols periodically record process (and communication) state to storage devices that survive tolerated failures. Process state comprises all the state necessary to run a process correctly including its memory and register states. When a process fails, a new incarnation of the failed process is resumed from the intermediate state in 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], [11], [12] predict poor utilizations (approaching 0%) for applications running on imminent systems and the need for dedicated reliability resources.

A. Strategies for Improving Checkpoint Performance

As computing systems increase in scale, the likelihood of a failure impacting the system increases significantly requiring more frequent checkpoints. If checkpoint/restart protocols are to be employed for future extreme scale systems, checkpoint/restart overhead must be reduced. For the *checkpoint commit* problem, saving an application checkpoint to stable storage, we can consider two sets of strategies.

The first set of strategies hide or reduce commit latencies without actually reducing the amount of data to commit. These strategies include *concurrent checkpointing* [13], [14], *diskless checkpointing* [15]–[18], checkpointing filesystems [19], *Remote checkpointing* [20], [21], *Forked checkpointing* [13], [14], [22]–[25].

The second set of strategies reduce commit latencies by reducing checkpoint sizes. These strategies include *memory exclusion* [26], *incremental checkpointing* [14], [16]–[18], [22] and *multi-level checkpointing* [27].

III. Checkpoint Compression

Our strategy for improving checkpoint commit overhead is based on data compression. In this section, we describe the checkpoint compression viability model that we use to determine when checkpoint compression should be considered. We then discuss previous research directly and indirectly related to our checkpoint data compression study.

A. A Checkpoint Compression Viability Model

Intuitively, checkpoint compression is a viable technique when benefits of checkpoint data reduction outweigh the drawbacks of the time it takes to reduce the checkpoint data. Our viability model is very similar to the concept offered by Plank et al [28]. Fundamentally, checkpoint compression is viable when *compression latency*, $t_{compress}$, and the *time to commit the compressed checkpoint*, t_{cc} is less than the *time required to commit the uncompressed checkpoint*,t_{uc}:

$$t_{compress} + t_{cc} < t_{uc}$$

or

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

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

$$\frac{r_{commit}}{r_{compress}} < \alpha \tag{1}$$

or

$$\frac{\textit{commit-speed}}{\textit{compression-speed}} < \textit{compression-factor} \tag{2}$$

In other words, if the ratio of the checkpoint commit speed to checkpoint compression speed is less than the compression factor, checkpoint data compression provides an overall time (and space) performance reduction. Our model assumes 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. In Section V, we discuss the implications of this assumption.

B. Related Compression Research

Li and Fuchs implemented a compiler-based checkpointing approach, which exploited compile time information to compress checkpoints [29]. 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 [28]. In a related vein, Plank et al also proposed *differential compression* to reduce checkpoint sizes for incremental checkpoints [30]. Moshovos and Kostopoulos used hardware-based compressors to improve checkpoint compression ratios [31]. Finally, in a related but different context, Lee et al study compression for data migration in scientific applications [32].

Our work (currently) 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.

IV. An Evaluation of Checkpoint Compression

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

- *Generally, can the compression benefit of reduced checkpoint data sizes outweigh the additional latency overheads necessary to compress checkpoint data?*
- *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?*

Accordingly, our evaluation is devised to answer these questions. 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 results.

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*[1] from the Mantevo Project [6], 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.

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 feel 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: In addition to the *mini apps*, we wanted to evaluate checkpoint compression with a full featured scientific application. We chose LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator). LAMMPS [7] is a classical molecular dynamics code developed at Sandia National Laboratories. For our experiments, we used the embedded atom method (EAM) metallic solid input script which is used by the Sequoia benchmark suite. The LAMMPS code and input scripts are provided on the LAMMPS web site [33].

[1]Mini apps are small, self-contained programs that embody essential performance characteristics of key applications.

3) Compression Utilities: For this study, we focused on the popular compression algorithms investigated in Morse's comparison of compression tools [34]. We settled on the following subset, which performed well in preliminary tests[2]:

- **zip**: `zip` is an implementation of Deflate [35], a lossless data compression algorithm that uses the LZ77 [36] 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.

  We vary `zip`'s parameter that toggles the tradeoff between compression factor and compression latency. This integer parameter ranges from zero to nine, where zero means fastest compression speed and nine means best compression factor. In our charts we use the label `zip(x)`, where `x` is the value of this parameter.
- **7zip** [37]: `7zip` is based on the Lempel-Ziv-Markov chain algorithm (LZMA) [38]. This algorithm uses a dictionary compression scheme similar to LZ77 and has a very high compression ratio.
- **bzip2**: `bzip2` is an implementation of the Burrows-Wheeler transform [39], 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.

  We vary `bzip2`'s compression performance by varying the block size for the Burrows-Wheeler transform. The respective integer parameter ranges in value from zero to nine a smaller value specifies a smaller block size. In our charts, we use the label `bzip2(x)`, where `x` is the value of this parameter.

- **pbzip2** [39]: `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.

  We vary two `pbzip2` 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. This is labeled as `pbzip2(x, y)`, where `x` is the value of the first parameter and `y` is the value of the second parameter.

- **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 use `bzip2` as a backend to compress the encoding.

  We vary `rzip`'s parameter, which toggles the tradeoff between compression factor and compression latency. As was the case for `zip`, this integer parameter ranges from zero to nine, where one means fastest compression speed and nine means best compression factor. In our charts we use the label `rzip(x)`, where `x` is the value of this parameter.

4) Checkpoint/Restart Utilities: The Berkeley Lab Checkpoint/Restart library (BLCR) [8], 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 [40] 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

In order to evaluate the effectiveness of different checkpoint compression schemes, we compressed many checkpoints collected from our application and various mini applications with different compression utilities. We measured the performance metrics necessary for us to analyze checkpoint viability using Equation 2 from Section III.

[2]We do not present results for several other algorithms, for example `gzip`, that did not perform well.

For each application, we chose problem sizes that would allow each application to run long enough so that we can take at least 5 different checkpoints. For these experiments, we were not concerned with application scale. Primarily, we observed the compressibility of checkpoints from singleton MPI tasks. For the three implicit finite element mini apps, we chose a problem size of 100x100x100. Both HPCCG and pHPCCG were run with openMPI with 3 processes while miniFE was run with 2 processes. phdMesh was run without MPI support on a problem size of 5x5x5. We ran LAMMPS with openMPI using 2 processors and a problem size of 5x5x5. We pair each of the aforementioned HPC workloads and parameterized compression algorithm. For HPCCG the checkpoint interval was 5 seconds, for miniFE and pHPCCG it was 3 seconds and for phdMesh the 5 checkpoints were taken randomly. We ran LAMMPS on a problem size of 5x5x5 and used an interval of 60s to take the checkpoints. BLCR was used to collect all checkpoints in this set of experiments.

For each of the *mini apps*, the average uncompressed checkpoint size ranged from 311 MB to 393 MB. For LAMMPS the checkpoint size was about 700MB on average. Our first set of results, presented in Figure 1, demonstrate 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.

Figure 2 shows how long the algorithms take to compress the checkpoints. In general, and not surprisingly, the parallel implementation of `bzip2`, `pbzip2`, generally outperforms all the other algorithms.

Based on Equation 2, if the product of checkpoint commit speed (or throughput) is less than the product of compression factor and compression speed, checkpoint compression will provide a time (and space) performance benefit. Figure 3 shows this product derived from our data. For each application in this Figure, the worse case scenario for checkpoint compression viability is the application's maximum compression speed-factor product across all compression algorithms. In the worse case, miniFE, checkpoint compression is viable unless a system can sustain a per process checkpoint commit bandwidth of greater than 2 GB/s. In the best case, phdMesh, the necessary per process checkpoint bandwidth raises to greater than 7 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.

C. Evaluating the Impact of Scale

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

Our first set of scaling experiments were designed for us to evaluate the first two scaling dimensions checkpoint size and time between checkpoints. In the first set of scaling experiments, we progressively increased the LAMMPS problem size so that its memory footprint and, therefore, the checkpoint size also would increase. For each LAMMPS process, five checkpoints were taken uniformly throughout the application run. For a fixed number of processes, larger problem sizes means longer runs, which means the time between checkpoints also increased. For these tests, we fixed the number of LAMMPS processes at two. The checkpoints we collected from these tests averaged about 171MB, 344MB, 482MB, 688MB for the various problem sizes.

Figure 4 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 5 shows once again that application process counts did not bear an impact on checkpoint viability. For clarity of the graph, we only show from 16 to 128 MPI processes. Data from the smaller runs further corroborate our findings. Additionally, we have no reason to believe these results will be different for larger process count runs.

D. Evaluating the Impact of User versus System Level Checkpoints

Lastly in this section, we examine the compression ratios of system-level checkpoints versus that of application specific checkpoints. Again, we use LAMMPS for this testing due to its optimized, application specific checkpointing mechanism described in the previous section. For these tests we compare the compression ratios of the application generated restart files with those generated by BLCR. Both with and without BLCR,

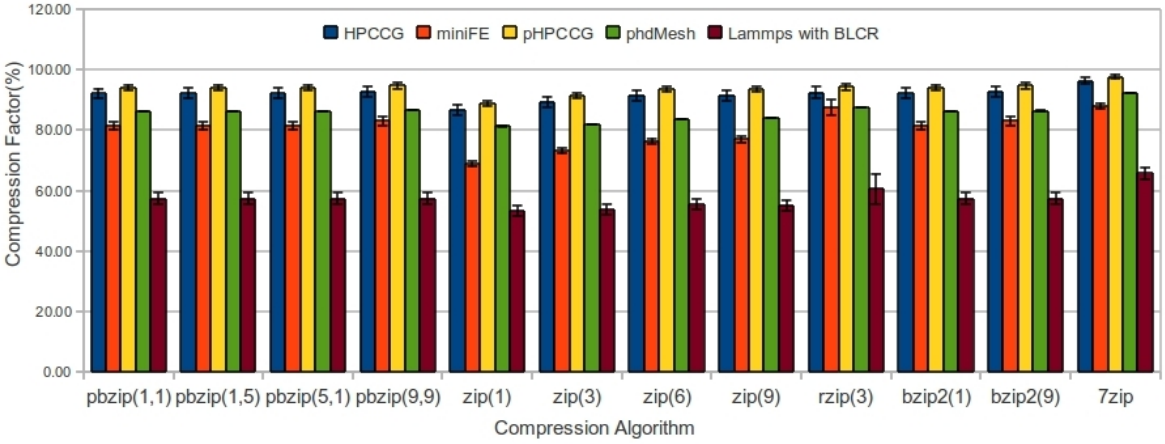

Figure 1. Checkpoint compression ratios for the various algorithms and applications.

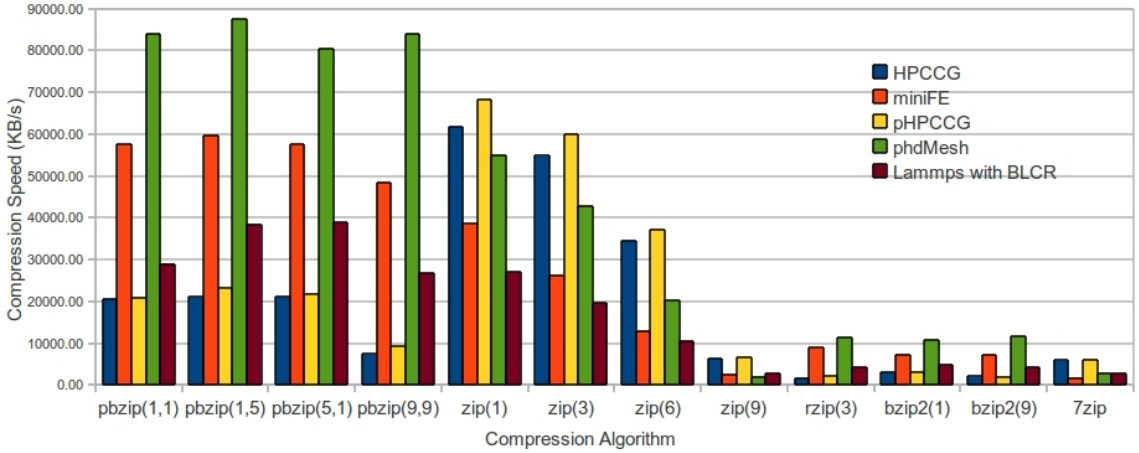

Figure 2. Checkpoint compression times for the various algorithms and applications.

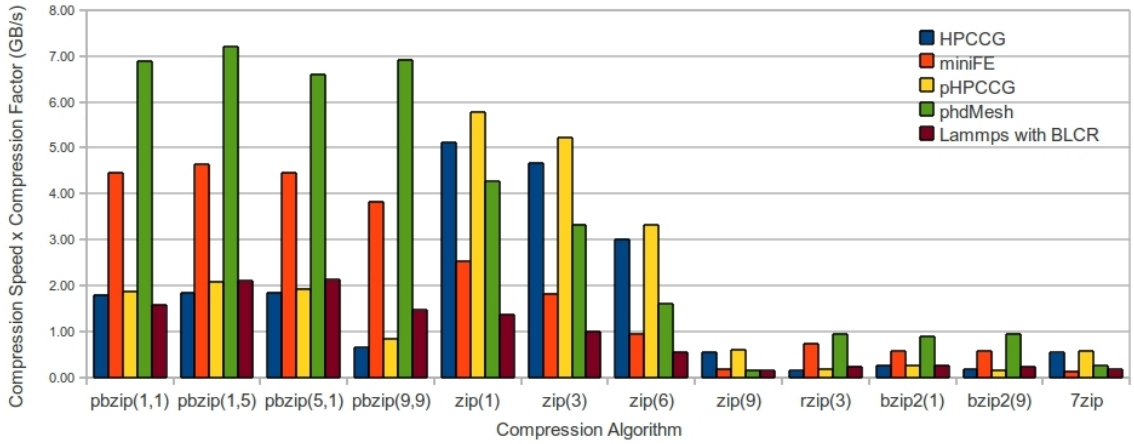

Figure 3. Checkpoint Compression Viability: Unless, checkpoint commit rate exceeds the compression speed $\times$ compression factor product (y-axis), checkpoint compression is a good solution.

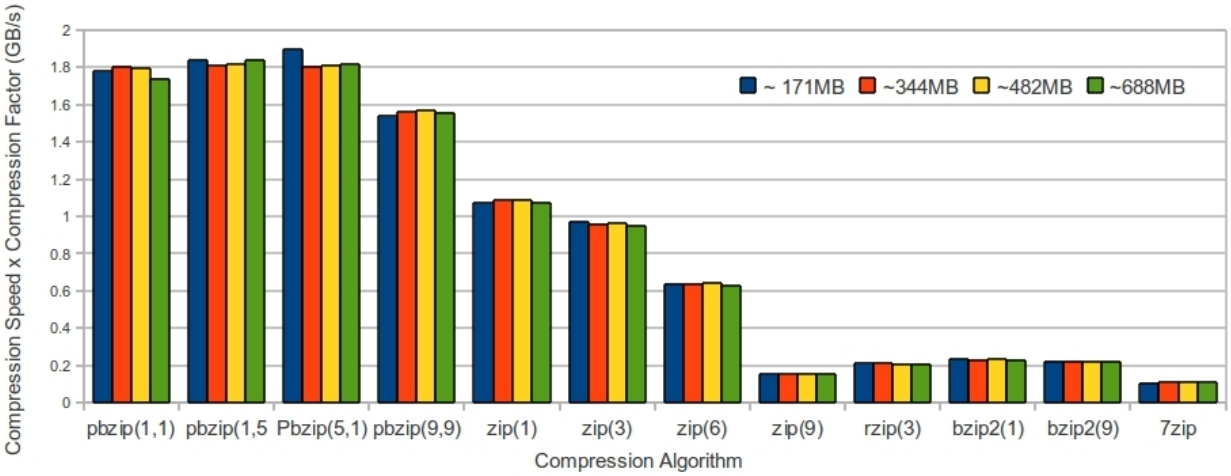

Figure 4. Scaling Checkpoint Sizes: Comparison of compression viability at different scales.

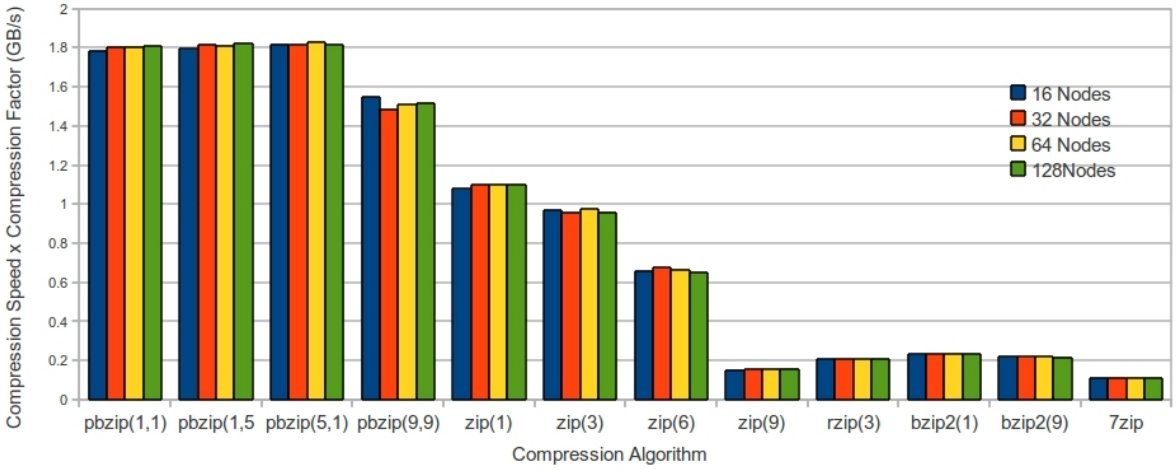

Figure 5. Scaling Process Counts: Comparison of compression viability at different scales.

we take 5 checkpoints equally spaced throughout the application run, with the averages displayed.

System-level checkpointing saves a snapshot of the application context such that it can be restarted where it left off. Application specific checkpointing , on the other hand, 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. This size difference was also observed in our testing. For example, LAMMPS built-in checkpointing scheme generated checkpoints of about 170MB, in comparison to BLCR generated checkpoints which were 700MB on average. Regardless of the two types of checkpoints, our checkpoint viability model shows consistent results. In Figure 6, we observe that no matter what the approach, system versus application-specific, the viability of checkpoint compression remains the same for LAMMPS. This makes sense for many HPC workloads as the the address space is comprised mostly of application data. Therefore, though user level checkpoints are smaller in size, application data is expected to have same *"fingerprint"* and therefore exhibit similar compression properties.

V. Discussion

A. Compression versus Checkpoint I/O Bandwidth

As mentioned earlier, the relationship between compression speed and checkpoint I/O bandwidth is the key factor of the viability of checkpoint compression. As Figure 3 shows, checkpoint compression is viable

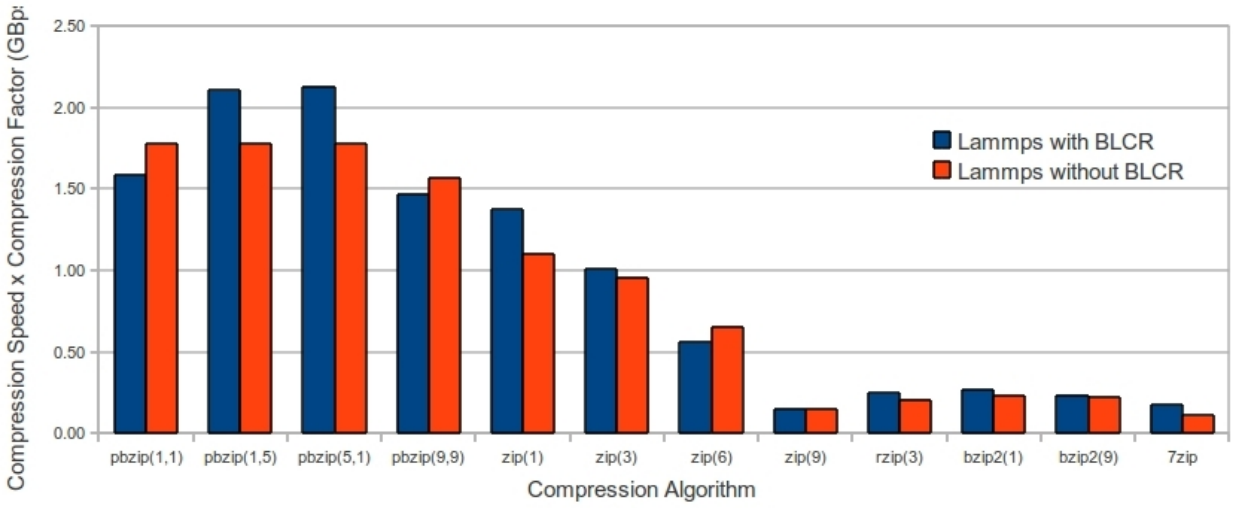

Figure 6. Comparison of system level Checkpoint and application specific checkpoint in terms of compression Viability for the various algorithms

with current compression algorithms at per-process checkpoint bandwidths less than 2 and 5 GB/sec. For comparison, the Oak Ridge Cray XT5 Jaguar petascale system has per-node and per-core checkpoint bandwidths of 5.3 MB/s and 1 MB/s, respectively, a factor of 1000 difference in performance. Similarly, the Lawrence Livermore Dawn IBM BG/P system has per-node checkpoint bandwidths of approximately 2 MB/s [3] As a result, aggressive use of checkpoint compression appears to be viable and indeed desirable on current large-scale platforms.

The viability of checkpoint compression on future systems depends highly on future computer storage architecture developments. One report suggested using more than one disk per processor to provide sufficient storage bandwidth for high-speed checkpointing (5 GB/sec per process) [43]. Other researchers have suggested using similar approaches that combine non-volatile memories with spinning storage to somewhat reduce the potential costs of the checkpoint file system [44], though still anticipates spending over $60M on the storage system. These 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.

Importantly, checkpoint compression also reduces the bandwidth pressure on checkpointing file systems. If power and cost of such storage systems are important design limiters compared to the CPU power and costs, as is expected [43], checkpoint compression could be an important technology in reducing the demands on exascale storage systems. Additional work is therefore needed to examine the impact of checkpoint compression on file system design, especially given the current uncertainty in terms of CPU architectures, levels of parallelism, and memory sizes in these systems.

[3]Oak Ridge's Spider Lustre-based file system provides 240 GB/sec of aggregate bandwidth [41], while Dawn's Lustre file system is listed as providing 70 GB/sec of peak bandwidth on LLNLL reference pages [42].

B. Future Enhancements

Our results show that different compression algorithms perform differently in terms of compression factor or compression speed while compressing checkpoints from different applications. We would like to understand what affects the performances of these algorithms and based on this predict the optimal compression algorithm for a particular application. This will improve application based checkpointing performance as the application developers can provide compression while taking checkpoints as well as checkpoint infrastructures can provide in memory compression. In our tests, for simplicity we assumed the *compression time* as the *time to read the uncompressed file + time to compress the file + time to write back the compressed file into disk*. Using in memory compression we can eliminate the read and write time, thus reducing the compression time even further giving us better speedup. We leave this as future work.

The positive impact of compression performance on

checkpoints opens the door to a number of research possibilities that is not addressed in this work. Graphics processing units (GPUs) typically have many more processing cores and wider memory buses than conventional CPUs. Given their high processing power and memory throughput a lot of work is going on among the GPU communities to leverage these benefits for data compression [45]–[48]. O'Neil and Burtscher's GPU based compression library GFC [47] achieved a rate of 75 Gb/s which emphasizes the effectiveness of GPU based compression. Exploring our checkpoint compression performance using GPUs is ongoing.

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. We also acknowledge our reviewers for comments and suggestions for improving this paper.

REFERENCES

[1] "Top 500 Supercomputer Sites," http://www.top500.org/ (visited September 2011). [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] P. Kogge, "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] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Quarterly*, vol. 3, no. 4, November 2007. [Online]. Available: http://www.ctwatch.org/quarterly/articles/2007/11/failure-tolerance-in-petascale-computers/

[5] 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.

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

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

[8] 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.

[9] D. Ibtesham, D. Arnold, K. Ferreira, and P. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Lecture Notes in Computer Science: Proceedings of the* 17^{th} *European Conference on Parallel and Distributed Computing (Euro-Par) 2011:* 4^{th} *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids*. Bordeaux, France: Springer Verlag, Berlin, Germany, Aug. 29 - Sep. 2, 2011.

[10] 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.

[11] 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.

[12] 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 *ACM/IEEE Conference on Supercomputing (SC'11)*, Nov. 2011 [to appear].

[13] 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.

[14] ——, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, August 1994.

[15] 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.

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

[17] 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: citeseer.ist.psu.edu/chen97clip.html

[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] 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)*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

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

[21] 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.

[22] 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

[23] S. I. Feldman and C. B. Brown, "Igor: A system for program debugging via reversible execution," in *1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*. New York, NY: ACM Press, 1988, pp. 112–123.

[24] J. Leon, A. L. Fisher, and P. Steenkiste, "Fail-safe pvm: A portable package for distributed programming with transparent recovery," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-93-124, February 1993.

[25] 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.

[26] 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.

[27] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[28] 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.

[29] 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.

[30] 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

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

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

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

[34] K. G. M. Jr., "Compression tools compared," no. 137, September 2005.

[35] P. Deutsch, "Deflate compressed data format specification." [Online]. Available: ftp://ftp.uu.net/pub/archiving/zip/doc

[36] 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.

[37] "7zip project official home page," http://www.7-zip.org.

[38] I. Pavlov, "Lzma sdk (software development kit)," 2007. [Online]. Available: http://www.7-zip.org/sdk.html

[39] J. G. Elytra, "Parallel data compression with bzip2."

[40] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "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

[41] 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.

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

[43] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf, Sep. 2008.

[44] 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.

[45] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical visualization and compression of large volume datasets using GPU clusters," in *In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04) (2004*, 2004, pp. 41–48.

[46] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors."

[47] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a GPU," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 7:1–7:7. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964189

[48] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore, "Accelerating lossless data compression with GPUs," *CoRR*, vol. abs/1107.1525, 2011.