On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance

Dewan Ibtesham[1], Dorian Arnold[1],
Kurt B. Ferreira[2,⋆], and Patrick G. Bridges[1]

[1] University of New Mexico, Albuquerque, NM, USA
{dewan,darnold,bridges}@cs.unm.edu
[2] Sandia National Laboratories, Albuquerque, NM
kbferre@sandia.gov

Abstract. The increasing size and complexity of high performance computing 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. In this work, we explore the feasibility of checkpoint data compression to reduce checkpoint commit latency and storage overheads. Leveraging a simple model for *checkpoint compression viability*, we conclude that checkpoint data compression should be considered as a part of a scalable checkpoint/restart solution and discuss additional scenarios and improvements that may make checkpoint data compression even more viable.

Keywords: Checkpoint data compression, extreme scale fault-tolerance, checkpoint/restart.

1 Introduction

Over the past few decades, high-performance computing (HPC) systems have increased dramatically in size, and these trends are expected to continue. On the most recent Top 500 list [27], 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 [14].

It also is expected that future high-end systems will increase in complexity; for example, heterogeneous systems like CPU/GPU-based systems are expected to become much more prominent. Increased complexity generally suggests that

⋆ 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.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 302–311, 2012.

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 also conclude that if current HPC system growth trend continues, expected system MTBF for the biggest systems on the Top 500 lists will fall below 10 minutes in the next few years [10,26]

Checkpoint/restart [5] 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. Rollback recovery is a well studied, general fault tolerance mechanism. However, recent studies [7,10] predict poor utilizations (approaching 0%) for applications running on imminent systems and the need for resources dedicated to reliability.

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* [17,18], *diskless checkpointing* [22] and checkpointing filesystems [3]. The second set of strategies reduce commit latencies by reducing checkpoint sizes. These strategies include *memory exclusion* [23], *incremental checkpointing* [6] and *multi-level checkpointing* [19].

This work falls under the second set of strategies. We focus on reducing the amount of checkpoint data, particularly via checkpoint compression. We have one fundamental goal: to understand the viability of checkpoint compression for the types of scientific applications expected to run at large scale on future generation HPC systems. Using several *mini-applications* or *mini apps* from the Mantevo Project [12] and the Berkeley Lab Checkpoint/Restart (BLCR) framework [11], 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 the next section, we present a general background of checkpoint/restart methods, after which we describe previous work in checkpoint compression and our checkpoint compression viability model. In Section 3, 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 compression research.

2 Checkpoint 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.

2.1 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 [24]. Fundamentally, checkpoint compression is viable when:

$$\begin{array}{ccccc} \textit{compression latency} & + & \begin{array}{c}\textit{time to commit}\\ \textit{compressed checkpoint}\end{array} & < & \begin{array}{c}\textit{time to commit}\\ \textit{uncompressed checkpoint}\end{array} \end{array}$$

or

$$\frac{|\textit{checkpoint}|}{\textit{compression-speed}} + \frac{(1 - \textit{compression-factor}) \times |\textit{checkpoint}|}{\textit{commit-speed}} < \frac{|\textit{checkpoint}|}{\textit{commit-speed}}$$

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

$$\frac{\textit{commit-speed}}{\textit{compression-speed}} < \textit{compression-factor} \tag{1}$$

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 4, we discuss the implications of this assumption.

2.2 Previous Work

Li and Fuchs implemented a compiler-based checkpointing approach, which exploited compile time information to compress checkpoints [16]. 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 [20]. Finally, in a related but different context, Lee et al study compression for data migration in scientific applications [15].

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.

3 Evaluating Checkpoint Compression

The goal of this work is to evaluate the use of state-of-the-field algorithms for compressing checkpoint data from applications that are representative of those expected to run at large scale on current and future generation HPC systems.

3.1 The Applications

We chose four *mini-applications* or *mini apps*[1] from the Mantevo Project [12], 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.

In general, we chose problem sizes that would allow each application to run long enough so that we can take at least 5 different checkpoints. Additionally, at this preliminary stage we were not overly concerned with scaling to large numbers of MPI processes. Primarily, we wish to observe 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 5x6x5.

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

3.2 The Checkpoint Library

The Berkeley Lab Checkpoint/Restart library (BLCR) [11], a system-level infrastructure for checkpoint/restart, is perhaps the most widely available checkpoint/restart library available and is deployed on several HPC systems. For our experiments, we obtain checkpoints using BLCR. Furthermore, we use the OpenMPI [9] framework which has the capability to leverage BLCR for fault tolerance.

3.3 The Compression Algorithms

For this study, we focused on the popular compression algorithms investigated in Morse's comparison of compression tools [13]. We settled on the following subset, which performed well in preliminary tests[2]:

- **zip**: ZIP is an implementation of Deflate [4], a lossless data compression algorithm that uses the LZ77 [28] 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**[1]: 7zip is based on the Lempel-Ziv-Markov chain algorithm (LZMA) [21]. 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 [8], 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**[8]: pbzip2 is a parallel implementation of bzip2. pbzip2 is multithreaded 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.

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

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.

3.4 The Tests

Each test consists of a mini app, a parameterized compression algorithm[3], and five successive checkpoints. 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. There was no particular logic in varying the checkpoint interval except for making sure to have the checkpoints spread uniformly across the execution time of the application. The BLCR library is used to collect the mini app checkpoints, and then we use the selected algorithms to perform checkpoint compressions. While checkpoints were being compressed, the system was not doing any additional work.

For testing, we used a 64-bit four core Intel Xeon processor with a clock speed of 2.33 GHz and 2 GB of memory running a Linux 2.6.32 kernel. Our software stack consists of OpenMPI-1.4.1 configured with BLCR version 0.8.2. The compression tools used were ZIP 3.0 by Info-ZIP, rzip version 2.1, bzip2 1.0.5, PBZIP2 1.0.5 and p7zip.

3.5 Compression Results

For each application, the average uncompressed checkpoint size ranged from 311 MB to 393 MB. Our first set of results, presented in Figure 1, demonstrate how effective the various algorithms are at compressing checkpoint data. With the exception of the `Rzip(-0)`, all the algorithms achieve a very high *compression factor* of about 70% or higher, 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.

[3] For each algorithm, a different set of parameter values constitute a different test.

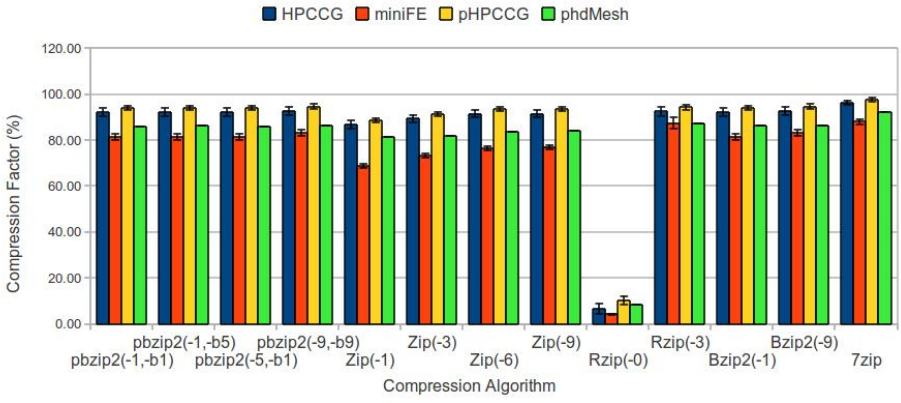

Fig. 1. Checkpoint compression ratios for the various algorithms and applications

4 Discussion

In the previous section, we presented the empirical results of our checkpoint compression. We conclude this paper with a discussion of the implications of these results. We also discuss known limitations and shortcomings of this work that we plan to address as we continue this project.

This work seeks to answer the question, *"Should checkpoint compression be considered as a potentially feasible optimization for large scale scientific applications?"* Based on our preliminary experiments, we believe the answer to this question is *"Yes."* Based on Equation 1, 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 as derived from the data in Section 3. Even with many optimizations and high performance parallel file systems that stripe large writes simultaneously across many disks and file servers, it is difficult to achieve disk commit bandwidths on the order of ones of Gigabits/second. Figure 3 shows that we with basic compression tools like pbzip, a file system must achieve per process bandwidth on the order of 14 Gigabits/second and as much as 56 Gigabits/second to compete with checkpoint compression strategy. Furthermore, we can explore additional strategies, like using multicore CPUs or even GPUs, to accelerate compression performance.

4.1 Current Limitations

While the results of this preliminary study are promising, we observe several shortcomings that we plan to address. These shortcomings include:

– **Testing on larger applications:** while the Mantevo mini applications are meant to demonstrate the performance characteristics of their larger

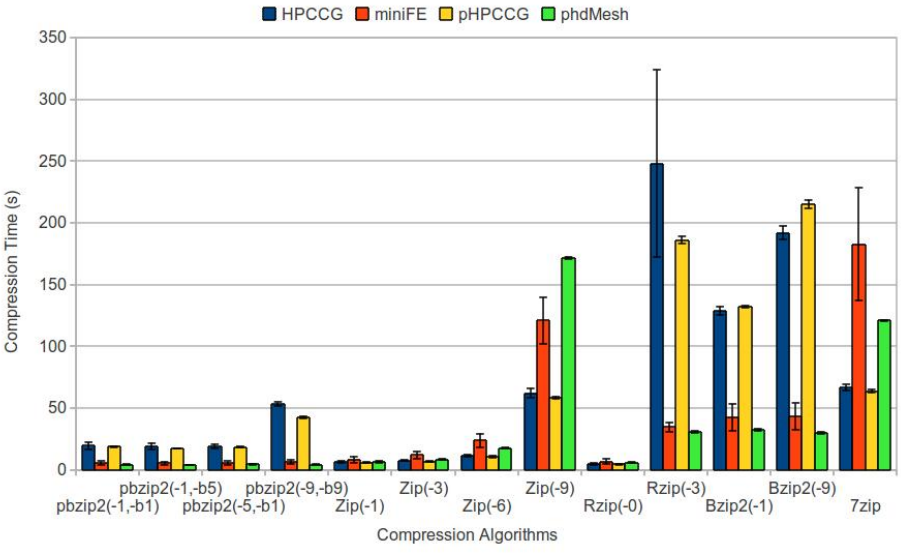

Fig. 2. Checkpoint compression times for the various algorithms and applications

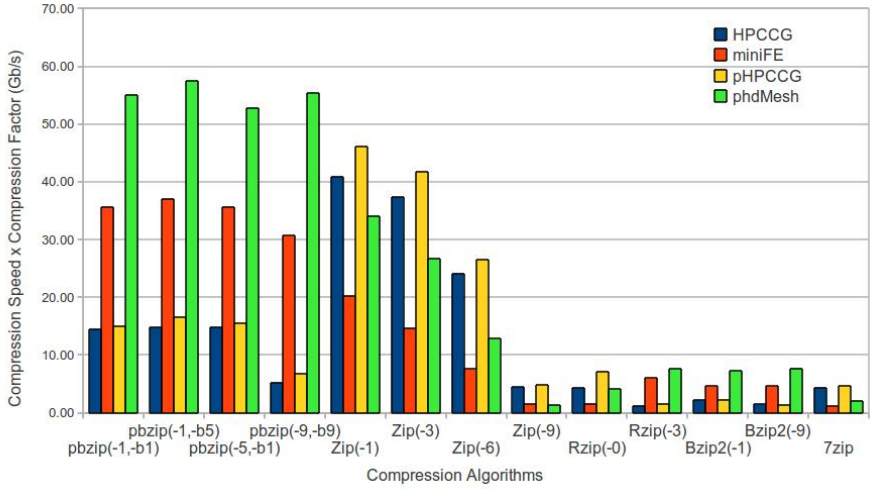

Fig. 3. Checkpoint Compression Viability: Unless, checkpoint commit rate exceeds the compression speed × compression factor product (y-axis), checkpoint compression is a good solution

counterparts, we plan to evaluate the effectiveness of checkpoint compression for these larger applications.

– **Testing at larger scales:** Our current tests are limited to very small scale applications. We plan to extend this study to applications running at much larger scales, on the order of tens or even hundreds of thousands of tasks.

Qualitatively, we expect similar results since compression effectiveness is primarily a function of the compression performance for individual process checkpoints.

- **Checkpoint intervals:** For these tests, in order to keep run times managable, we used a relatively small checkpoint intervals. We plan to evaluate whether compression effectiveness changes as applications execute for longer times. We have no reason to expect significant qualitative differences.

Acknowledgments. This work was supported in part by Sandia National Laboratories subcontract 438290. A part of this work was performed at the Sandia National Laboratories, a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000. The authors are grateful to the member 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. 7zip project official home page, `http://www.7-zip.org`
2. ASC Sequoia, `https://asc.llnl.gov/computing_resources/sequoia` (visited May 2011)
3. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: Plfs: a checkpoint filesystem for parallel applications. In: Conference on High Performance Computing Networking, Storage and Analysis (SC 2009), pp. 21:1–21:12. ACM, New York (2009)
4. Deutsch, P.: Deflate compressed data format specification
5. Elnozahy, E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
6. Elnozahy, E.N., Johnson, D.B., Zwaenpoel, W.: The performance of consistent checkpointing. In: 11th IEEE Symposium on Reliable Distributed Systems, Houston, TX (1992)
7. Elnozahy, E.N., Plank, J.S.: Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. IEEE Transactions on Dependable and Secure Computing 1(2), 97–108 (2004)
8. Elytra, J.G.: Parallel data compression with bzip2
9. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004), doi:10.1007/978-3-540-30218-6_19
10. Gibson, G., Schroeder, B., Digney, J.: Failure tolerance in petascale computers. CTWatch Quarterly 3(4) (November 2007)

11. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (blcr) for linux clusters. Journal of Physics: Conference Series 46(1) (2006)
12. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratory (2009)
13. Morse Jr., K.G.: Compression tools compared (137) (September 2005)
14. Kogge, P.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO) (September 2008)
15. Lee, J., Winslett, M., Ma, X., Yu, S.: Enhancing data migration performance via parallel data compression. In: Proceedings International on Parallel and Distributed Processing Symposium, IPDPS 2002, Abstracts and CD-ROM, pp. 444–451 (2002)
16. Li, C.-C., Fuchs, W.: Catch-compiler-assisted techniques for checkpointing. In: 20th International Symposium on Fault-Tolerant Computing, FTCS-20, Digest of Papers, pp. 74–81 (June 1990)
17. Li, K., Naughton, J.F., Plank, J.S.: Real-time, concurrent checkpoint for parallel programs. In: 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 1990), pp. 79–88. ACM, Seattle (1990)
18. Li, K., Naughton, J.F., Plank, J.S.: Low-latency, concurrent checkpointing for parallel programs. IEEE Transactions on Parallel and Distributed Systems 5(8), 874–879 (1994)
19. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: 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, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
20. Moshovos, A., Kostopoulos, A.: Cost-effective, high-performance giga-scale checkpoint/restore. Technical report, University of Toronto (November 2004)
21. Pavlov, I.: Lzma sdk (software development kit) (2007)
22. Plank, J., Li, K., Puening, M.: Diskless checkpointing. IEEE Transactions on Parallel and Distributed Systems 9(10), 972–986 (1998)
23. Plank, J.S., Chen, Y., Li, K., Beck, M., Kingsley, G.: Memory exclusion: Optimizing the performance of checkpointing systems. Software – Practice & Experience 29(2), 125–142 (1999)
24. Plank, J.S., Li, K.: ickp: A consistent checkpointer for multicomputers. IEEE Parallel & Distributed Technology: Systems & Applications 2(2), 62–67 (1994)
25. Plank, J.S., Xu, J., Netzer, R.H.B.: Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee (August 1995)
26. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: Dependable Systems and Networks (DSN 2006), Philadelphia, PA (June 2006)
27. Top 500 Supercomputer Sites, `http://www.top500.org/` (visited September 2011)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)