SAND2011-4471C

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 87131, 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 (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. 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 the types of improvements necessary to make checkpoint data compression more viable.

Keywords: Fault tolerance, Checkpoint compression

1 Introduction

Over the past few decades, high-performance computing (HPC) systems have increased in size and complexity, and these trends are expected to continue. On the most recent Top 500 list [31], 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. Further, exascale systems are projected to have on the order of tens to hundreds of millions of cores within the current decade [17].

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 individual components are likely to be more failure prone. Furthermore, mean time between failures (MTBF) is inversely proportional to system size, so increased system sizes also will contribute to extremely low system MTBF. In

* 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.

recent studies, Schroeder and Gibson indeed conclude that system failure rates depend mostly on system size, particularly, the number of processor chips in the system. They also conclude that if the current HPC system growth trend continues, expected system MTBF for the biggest machines on the Top 500 lists will fall below 10 minutes in the next few years [13, 28]

Checkpoint/restart [6], is perhaps the most commonly used HPC fault-tolerance mechanism. In checkpoint/restart protocols, during normal operation, process (and communication) state is periodically recorded to persistent storage devices that survive tolerated failures. When a failure occurs, effected processes roll back to the point represented by their most recent stored state – thereby reducing the amount of lost computation. Rollback recovery is a well studied, general fault tolerance mechanism. However, recent studies [8, 13] predict poor utilization 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 problem of *checkpoint commit* or saving an application checkpoint, three general strategies can be considered: (1) reducing checkpoint information, for example, using application-directed checkpoints; (2) reducing checkpoint data, for example, using incremental checkpointing; or (3) reducing the time to commit checkpoint data to stable storage, for example focusing on checkpoint I/O.

This work focuses on the second strategy of 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 [15] and the Berkeley Lab Checkpoint/Restart (BLCR) framework [14], 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 4, 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/Restart Background

In checkpoint-based protocols, checkpoints or snapshots of a process's state are periodically *committed* to stable storage. Process state comprises all the state necessary to run a process correctly including its memory and register states. If

a process failure is detected, the failed process' most recent checkpoint is used to restore (a new incarnation of) the failed process to the intermediate state saved in the checkpoint. Several optimizations have been proposed to improve basic checkpointing including:

- *Incremental checkpointing* [3, 4, 7, 21, 25]: the operating system's memory page protection facilities are used to detect and save only pages that have been updated between consecutive checkpoints. To protect against saving duplicate incremental data, hashing using GPUs has also been explored [11].
- *Forked checkpointing* [7, 10, 18, 20, 21, 23]: the application process forks a *checkpointing process* allowing the original process to continue while the forked process concurrently commits the checkpoint state to stable storage. If the *fork* system call implements *copy-on-write* semantics, both processes efficiently share the same address space until the original process updates a memory segment at which point a copy is made so that the checkpointing process has a copy of the memory state at the time the checkpoint was initiated.
- *Remote checkpointing* [30, 32]: Remote checkpointing leverages network resources to save checkpoints to remote checkpoint servers providing performance gains in environments where I/O bandwidth to the network is more abundant than that to local storage devices. Additionally, remotely stored checkpoints allow systems to survive non-transient node failures.

While checkpoint/restart protocols are among the most well-known fault tolerance techniques, performance drawbacks make them unsuitable for large scale computing environments with hundreds of thousands of components or more. As the number of computational nodes an application uses increase, so does the application's global checkpoint overhead. At the same time, the decreased MTBF that results from the increased number of nodes suggests that an application should take checkpoints more frequently to minimize work loss. Recent studies [8, 29] have suggested that the combination of these factors will result in unacceptably poor application utilization approaching 0% "useful work".

3 Checkpoint Compression

3.1 Previous Work

To the best of our knowledge, there has not been much research towards the goal of reducing checkpoint sizes and commit times that consider data compression. Li and Fuchs implemented a compiler-based checkpointing approach(CATCH), which exploited compile time information to compress checkpoints [19]. The results from their CATCH compiler, which used LZW data compressor, showed that a compression ratio of over 100% was necessary to achieve any significant benefit compared to the time overhead. 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 [26].

Plank et al also proposed *differential compression* to reduce checkpoint sizes for incremental checkpoints [27]. Moshovos and Kostopoulos used hardware-based compressors to improve checkpoint compression ratios [22].

3.2 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. More conceptually, checkpoint compression is viable when: [3]

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

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:

$$\frac{\textit{commit-speed}}{\textit{compression-speed}} < \textit{compression-factor} \quad (1)$$

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. In the rest of this paper, we explore this concept for practical scenarios.

4 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.

4.1 The Applications

We chose four *mini-applications* or *mini apps*[4] from the Mantevo Project [15], namely HPCCG version 0.5, miniFE version 1.0, pHPCCG version 0.4 and phdMesh version 0.1.

[3] Plank et al pose a similar concept [26].
[4] Mini apps are small, self-contained programs that embody essential performance characteristics of key applications.

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.

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.

4.2 The Checkpoint Library

The Berkeley Lab Checkpoint/Restart library (BLCR) [14], 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 [12] framework which has the capability to leverage BLCR for fault tolerance.

4.3 The Compression Algorithms

For this study, we focused on the popular compression algorithms investigated in Morse's comparison of compression tools [16]. We settled on the following subset, which appeared to performed well in preliminary tests:

- **7zip**[1]: 7zip is based on LZMA 7zip uses LZMA(Lempel-Ziv-Markov chain Algorithm) [24] to compress data. The algorithm uses a dictionary compression scheme similar to LZ77 and has a very high compression ratio. Each of these tools have different parameters to achieve faster compression time or better compression ratio.
- **zip**: ZIP is an implementation of Deflate [5], a lossless data compression algorithm that uses 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.
- **bzip2**: BZIP2 is an implementation of the Burrows-Wheeler Transform [9] 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.
- **pbzip2**[9]: A multi-threaded implementation of bzip2 is called parallel bzip2(PBZIP2) uses the same technique as bzip2 but can leverage multi-CPU and multi-core computers giving speed improvement. We have used two parameters to control the compression. The first parameter defines the BWT block size in kB and the second parameter defines the file block size in kB
- **rzip**: Rzip can take advantage of very long distance redundancy as it has a very large buffer. It finds and encodes large chunk of duplicate data and then use bzip2 as backend to compress the encoding.

For this study, we tested other tools that did not exhibit good performance in our case studies, including gzip.

4.4 The Tests

Each test consists of a mini app, a parameterized compression algorithm[5], 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.

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.

4.5 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.

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

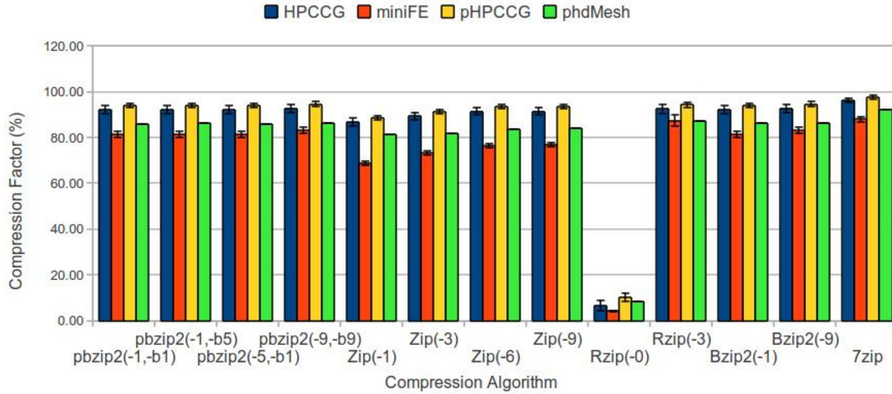

Fig. 1. Checkpoint compression ratios for the various algorithms and applications.

5 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 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 4. 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 in many cases, a file system must achieve at least about 14 Gigabits/second and as much as 56 Gigabits/second to compete with our checkpoint compression strategy. Furthermore, we can explore additional strategies, like using multicore CPUs or even GPUs, to accelerate compression performance.

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

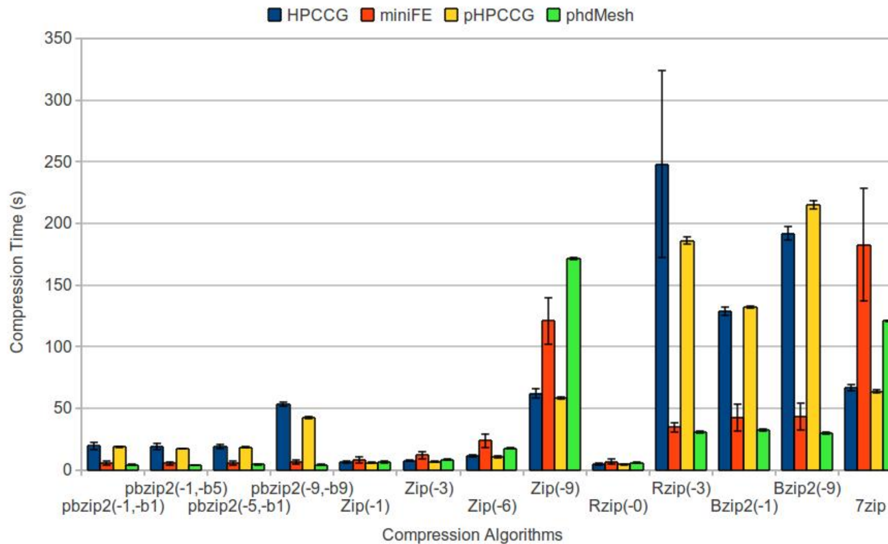

Fig. 2. Checkpoint compression times for the various algorithms and applications.

- **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.

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. 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*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.
4. Y. Chen, K. Li, and J. S. Plank. Clip: A checkpointing tool for message-passing parallel programs. In *SuperComputing '97*, San Jose, CA, 1997.
5. P. Deutsch. Deflate compressed data format specification.
6. 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*, 34(3):375–408, 2002.

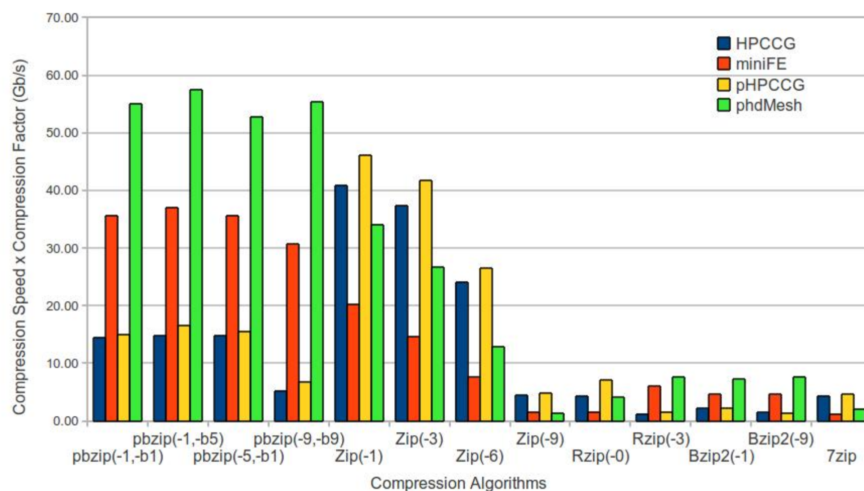

Fig. 3. Checkpoint Compression Viability: Unless, checkpoint commit rate exceeds the compression speed × compression factor product (y-axis), checkpoint compression is a good solution.

7. 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.
8. 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*, 1(2):97–108, April-June 2004.
9. J. G. Elytra. Parallel data compression with bzip2.
10. 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)*, pages 112–123, New York, NY, 1988. ACM Press.
11. K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold. Libhashckpt: Hash-based incremental checkpointing using GPUs. In *Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011 [to appear].
12. 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 D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 353–377. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30218-6_19.
13. G. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. *CTWatch Quarterly*, 3(4), November 2007.
14. P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1), 2006.
15. 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. Technical Report SAND2009-5574, Sandia National Laboratory, 2009.
16. K. G. M. Jr. Compression tools compared. (137), September 2005.
17. P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), September 2008.
18. J. Leon, A. L. Fisher, and P. Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, Pittsburgh, PA, February 1993.
19. 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*, pages 74–81, jun 1990.
20. 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)*, pages 79–88, Seattle, Washington, 1990. ACM.
21. K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.
22. A. Moshovos and A. Kostopoulos. Cost-effective, high-performance giga-scale checkpoint/restore. Technical report, University of Toronto, November 2004.
23. 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)*, pages 124–129, Madison, WI, 1988. ACM Press.
24. I. Pavlov. Lzma sdk (software development kit), 2007.
25. J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *USENIX Winter 1995 Technical Conference*, pages 213–224, New Orleans, LA, January 1995.
26. J. S. Plank and K. Li. ickp: A consistent checkpointer for multicomputers. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 2(2):62–67, 1994.
27. J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.
28. 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.
29. B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics Conference Series*, 78(1), 2007.
30. G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society.
31. Top 500 Supercomputer Sites. `http://www.top500.org/` (visited May 2011).
32. V. C. Zandy, B. P. Miller, and M. Livny. Process hijacking. In *8th International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 177–184, Redondo Beach, CA, August 1999.
33. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.