Compression in Checkpointing and Fault Tolerance Systems

Saurabh Kadekodi, Northwestern University

Considering the pace at which HPC systems are growing today, reliability, frequency and efficiency of fault tolerance mechanisms have become one of their most important problems. As a part of this study, we understand the many faces of checkpoint / restore, the de-facto fault tolerance method in practice. We then shift our focus to the various ways in which optimizations are performed while checkpointing and focus on optimizations due to size reduction. After a survey of a few important optimization ideas, we narrow down upon a void that can be filled with regards to compiler-assisted checkpoint compression and hope that this the path that will keep fault tolerance abreast with tomorrows HPC systems.

Categories and Subject Descriptors: C.2.2 [**Computer-HPC**]: Fault Tolerance

General Terms: Checkpoint, Compression

Additional Key Words and Phrases: Checkpoint, Compression, Recovery, Optimization

ACM Reference Format:
Saurabh Kadekodi, 2013. Compression in Checkpointing and Fault Tolerance Systems. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January YYYY), 8 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Fault tolerance is probably one of the greatest challenges that the high performance computing (HPC) community is facing today. With exascale computing and petascale machinery, the mean time between failure of the these humongous system is in mere minutes [Schroeder and Gibson 2007]. To be able to withstand this kind of failure rate, measures are being taken to develop fault tolerant mechanisms that save the state of these machines at sub-second intervals. The cost of these mechanisms is so high that a very significant amount of CPU, I/O bandwidth and time is spent in ensuring safety from faults. In a study in 2007 [Schroeder and Gibson 2007], it was predicted that at the then forecasted rate of doubling of cores per chip, by 2013, almost 100% of the time in petascale systems would be spent in checkpointing and recovery from faults. Another study shows that 75-80% of the I/O traffic on current HPC systems is due to checkpointing [Petrini 2002]. 2013 has dawned, and fortunately, we are not spending all our time in recovery from failure, but that situation is very much around the corner. This emphasizes the immediate need to look into fault tolerance and make it withstand the speed at which HPC systems are growing.

We start by defining checkpoint / restore and then classify it according to a couple of dimensions. Following that, we explain the important characteristics that a good checkpointing systems must exhibit in order to keep up with increasing system sizes. Then, we delve into checkpoint optimization and explain some techniques of optimization. A small survey of important checkpoint size reduction ideas follows after which

Author's addresses: Saurabh Kadekodi, Department of Electrical Engineering and Computer, Northwestern University, (Current address) 2145 Sherdian Road, Evanston, Illinois 60201.

DOI:http://dx.doi.org/10.1145/0000000.0000000

we finally conclude with an analysis of what is present and how we should approach the future.

2. CHECKPOINT / RESTORE

Checkpointing is the de-facto fault tolerance mechanism in practice today and has seen decades of research. It involves periodically storing the state of a computer (which primarily consists of memory and the registers) to stable storage such that, in the face of a failure, we can restart the system at the last checkpoint that was stored onto disk. One of the most important benefits of checkpointing over other fault-tolerant techniques is cost-effectiveness. It requires much lesser infrastructure than methods like redundancy, making it an attractive option for HPC systems. On the other hand, it also has disadvantages like complexity. With ever reducing checkpoint intervals, it is becoming a very hard task to provide correct and efficient checkpointing.

3. CLASSIFICATION OF CHECKPOINTING SCHEMES

Checkpointing can be classified along various dimensions. We will start by identifying checkpoints by their scope. These are the typical granularities at which checkpointing schemes are developed, each with their own advantages and disadvantages.

3.1. System-level Checkpoints

These checkpoints are taken at the OS level. The thought behind these checkpoints is to provide fault tolerance to applications that are already running, but do not have any application level fault handling mechanism. We believe these are best suited for machines that either run a bunch of very varied application workloads and its very difficult for all of them to maintain checkpoints independently or have legacy applications running on them that are not equipped with checkpointing. The advantage of these checkpoints are that applications (and thus their developers) need not care about checkpointing / recovery, which can be a daunting task to implement. More often than not, they are not (and dont need to be) aware of the fact that checkpointing is even happening underneath. The downside is that these checkpoints do not really care about the nature of the application that runs on that OS. They measure all the applications by the same yardstick. Some examples of system-level checkpoints are [Pinheiro 1998], [Duell 2003] and [Zhong and Nieh 2001].

3.2. Application-level Checkpoints

These checkpoints are at the other extreme. They are completely application-based, and are in fact, written by the developers of the application. The notion behind this scheme is that the application itself is the best judge of when and what it needs to save in order to minimize loss in the face of failure. Advantage(s) - In agreement with the notion, the application has precise knowledge of the state it needs to save in order to reconstruct following a failure. There might be large amounts of memory that need not be saved at all (which the system-level checkpointing schemes are not at all aware of), and that could result in very efficient checkpointing. Disadvantage(s) - Neither are all the already running, widespread legacy applications equipped with checkpointing code, nor is it possible to update them to include checkpointing facilities. Moreover, devising algorithms for checkpoint / restore is a complicated task in itself, not everyone's cup of tea. In fact, at a relatively recent survey at the Texas Advanced Computing Center (TACC) [Texas Advanced Computing Center 2009], over 65% respondents said that they would be interested in completely automated checkpoint support.

3.3. Library-level Checkpoints

This checkpointing mechanism is also referred to as compiler-level or runtime-level. In contrast to the two extreme granularities mentioned above, library-level conveniently sits at a spot neither too close to the application, nor too far away from it. In terms of benefits, these checkpointing schemes identify pretty closely with the application that is running and can do a fair bit of analysis to identify important information of an application at compile-time / runtime. While this seems to give the best of both worlds, one of the biggest challenges that this technique faces, is when to take a certain checkpoint. A typical case where timing of the checkpoint can make a huge difference, is when checkpoints are initiated when a process is in a loop, or is performing some temporary computation that is very memory or I/O intensive. If the scheme had been a little more application aware, it would have prevented this by taking a checkpoint at a more appropriate time. Some example of library-level checkpointing are [James Plank et al. 1995], [Michael Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny 1997] and [Yi-min Wang et al. 1995].

Another way of classifying checkpoints is on the amount of data stored at each checkpoint interval.

3.4. Non-Incremental (Whole) Checkpoints

In non-incremental checkpoints, the entire memory (the bulk in any checkpoint) is saved to disk at every interval. This is beneficial when most of the memory is dirtied in every interval. Most workloads show that this is not true because of locality. Another advantage of this technique is that it is only essential that you store the latest checkpoint on disk. In systems where disk space is very limited and / or costly, this technique proves more cost effective than incremental checkpointing. The obvious disadvantage of this scheme is that entire memory needs to be written to disk on every interval, an inherently costly task. For intervals that are very far apart, this scheme might make sense, but for sub-second intervals, it is a near impossible scheme to implement.

3.5. Incremental Checkpoints

As the name suggests, this technique only saves the changed pages of memory (since the last checkpoint) onto disk. Locality aids this technique tremendously. Unlike entire checkpoints, we need to store the whole sequence of checkpoints taken from the start till the latest increment in order to reconstruct the system in case of failure. Due to this, not only is more disk space needed, but reconstruction cost is also higher than non-incremental checkpoints. But, since recovery is not the common case, this technique still fares better than non-incremental checkpoints, especially in read-intensive workloads. Another disadvantage of this technique is that the memory is updated at word-granularity and memory is checkpointed at page-granularity. This incurs substantial cost and interesting schemes have been proposed to tackle this problem, which we will see in later sections.

4. IMPORTANT CHARACTERISTICS OF A GOOD CHECKPOINTING MECHANISM

A good checkpointing technique must exhibit the following characteristics:

4.1. Minimize Size

The smaller the footprint of a checkpoint, the more efficiently and quickly it can be taken. The bottleneck of checkpointing is the disk writes that take place. By minimizing size, we affect the largest overhead in checkpointing, thus guaranteeing speedup.

4.2. Minimize Checkpoint Time

In most cases, application(s) are completely stalled when a checkpoint is taken. This is to prevent inconsistencies that can occur because of changing memory when the checkpoint is in progress. This reduces the effective computation time that the application gets and affects its performance.

4.3. Minimize Resource Usage

4.4. Minimize Power

4.5. Minimize Restore Time

In the face of failure, it is very essential that a system can recover from it as quickly as possible. Downtime is a very costly proposition, especially in high-end systems.

5. OVERVIEW OF CHECKPOINT OPTIMIZATION

As mentioned briefly in the introduction, the trends in HPC systems growth show that the MTBF of the largest systems from the top500.org list are set to fall below 10 minutes in the very immediate future [Bianca Schroeder and Garth Gibson 2005]! With such alarming failure rates, it is essential that massive effort is put into faster, reliable and sub-second interval fault tolerance. There are two main sets in which optimization of checkpoints can be categorized. Both categories aim at reducing the time needed for taking a checkpoint.

The first set of ideas aims at reducing the checkpoint time by hiding the latency of committing a checkpoint. Some of the intelligent strategies in this set are:

5.1. Staggered Checkpoints [Alison Nicholas Norman 2010]

This is in contrast with co-ordinated checkpoints. It makes a complete checkpoint of the system possible through multiple checkpoints from multiple processes at different points in time reducing network and disk contention.

5.2. Diskless Checkpoints [James S. Plank et al. 1997]

Since the largest bottleneck are disk writes, this technique eliminates that aspect completely by checkpointing in memory rather than on disk. It has a direct advantage of restoration as it just has to switch between address spaces to make that possible. Of course this has its own repercussions, but it still reduces overall checkpointing time.

5.3. Co-Operative Checkpoints

This technique exploits the memory redundancy across multiple nodes running parallel [Lei Xia and Peter Dinda 2012] applications to distribute the job of taking a checkpoint across multiple nodes. This parallelizes checkpointing and in high-end infrastructures, can prove to be extremely effective in reducing checkpoint time.

The second set aims at reducing the size of the checkpoint in order to reduce the amount of data it needs to write to disk. As this paper concentrates on the reduction of size aspect of optimization, we first explain a couple of important ideas from this approach and then explain a few implementations in more detail.

6. IMPORTANT OPTIMIZATION IDEAS IN CHECKPOINT SIZE REDUCTION

6.1. Deduplication

Recent studies have shown that parallel applications exhibit a very high degree of duplicate memory. This can be of the order of 80% in some cases [Lei Xia and Peter Dinda 2012]. Deduplication aims at saving only one copy per duplicate memory page during a checkpoint. Unique pages along with the information of the various locations of where they were duplicated is good enough to reconstruct the state of the machine following a failure. Deduplication also reduces disk space needed to store checkpoints and makes it a very attractive checkpoint compression method. A challenge in this technique is how to accurately keep track of the duplicates of every page in memory. This metadata can be quite heavy and can itself incur a lot of memory cost. Another method of deduplication within checkpoints is via similarity detection. To understand this better, consider the setup to be an incremental checkpointing scheme. Similarity detection deals with finding similarity between a chunk of memory that is being written in the current checkpoint with chunks in previously written checkpoints. If a match is found, a mere pointer to the checkpoint chunk that contains the same data is enough to ensure correctness of the checkpoint.

6.2. Compression

As the name of the technique indicates, this deals with compressing the data to be checkpointed before writing it to disk. This reduces the size of the checkpoint and thus speeds up the process of taking a checkpoint as writing to disk is the main bottleneck of checkpointing. The downside to this is that compression algorithms are very resource and time intensive. In fact, Plank and Li have derived an equation that suggests that compression can benefit checkpointing only if the rate of compression is higher than the rate of disk writes [James S. Plank et al. 1997]. In a more recent effort [Schroeder and Gibson 2007], Schroeder and Gibson performed a study which explained the nature of failures in petascale systems and future predictions. With the current state-of-the-art systems, they recommended that rather than applications spending time on other optimization techniques, they should consider spending more time and effort compressing checkpoints. As a part of a much more relevant work [Dewan Ibtesham et al. 2011] (and in the very recent past) on the viability of compression for reducing overheads in checkpointing, Ibtesham et. al. concluded that compression was effective not only for storing a checkpoint, but also for recovery from it. In our opinion their work is the only one that concentrates on the effect of compression on checkpointing and they have shown some extremely positive results and comparisons making compression the forerunner in fault tolerance optimization as of today.

7. A SMALL SURVEY OF OPTIMIZATION IDEAS RELATED TO CHECKPOINT SIZE REDUCTION

7.1. Compiler Assisted Techniques for Checkpointing (CATCH) [Micah Beck et al. 1994]

This was one of the first works in compiler-assisted checkpointing. It analyzes compile time program information to inject code, which at runtime checkpoints a process subject to a certain checkpoint interval that needs to be maintained. Its analysis of the program to identify the places that it needs to checkpoint the process is called the potential checkpoint selection problem. Since potential checkpoint selection is NP-Complete, in this technique, the authors propose a suboptimal solution, achieved by heuristics. At compile-time this technique injects a function call _checkpoint() into the program to checkpoint the process at an appropriate time. A big challenge in doing this is to find when a checkpoint can and should be taken. For this purpose, they inject another function call called _potenial() at several places in the code. The job of this function is twofold:

— To honor the checkpoint interval time (i.e. not exceed the checkpoint interval, or it might result in loss of more computation)
— To minimize resources (i.e. space and time for checkpointing)

Once it finds a sweet spot that obeys the above rules (and minimizes the second rule), it calls the _checkpoint() routine and establishes a checkpoint. Through potential checkpoint measurement, it succeeds on the idea that small variation in checkpoint interval timings may save substantial system resources. The authors mention that compression could work really well in assisting them to further reduce the checkpoint size. In fact, their algorithm even has an LZW compressor algorithm within in, but it really slows down checkpointing because of its resource hogging.

7.2. Data Aware Aggregation and Compression [Tanzima Zerin Islam et al. 2012]

This is a very recent study that explores the effects of data-aware aggregation coupled with data-aware compression in order to improve compressibility of checkpoints. Data-aware aggregation deals with aggregating across process checkpoints such that semantically similar data stays close together. The motivation behind this is that dictionary based compression algorithms work best when they have similar data that is close together. This is because almost all compression utilities have a finite window in which they search for similarities that they can compress. Based on the nature of the data, the implementation dynamically selects the compression algorithm that is best suited for that data. The semantics of data are captured through I/O libraries like HDF5 and netCDF. The authors claim a substantial increase in compressibility and significantly reduced checkpoint and restart overheads over simplistic compression.

7.3. Compiler assisted memory exclusion (CAME) [James S. Plank et al. 1996]

This technique relies on memory exclusion, i.e. it identifies regions of memory of a process that need not be stored in a checkpoint at all, before checkpointing, and thus reduces the size of a checkpoint. The authors define unneeded memory pages as either read-only (meaning that they have not changed in the current checkpoint interval) or as dead (meaning that they are not required for the successful completion of the process). It performs this by combining user directives with static, data flow analysis to achieve the necessary optimization to checkpointing.

7.4. Adaptive block size variation [Saurabh Agarwal et al. 2004]

This is a paper that has a relatively fresh take on optimizing incremental checkpointing to reduce checkpoint size. The authors propose a self-optimizing algorithm that adaptively calculates the optimal block-boundaries of memory to be checkpointed based on historical checkpoints. With each checkpoint they narrow in on the near-exact memory granularity that guarantees minimum wastage in terms of writing memory to disk. An interesting utility that the authors have developed is called merge. This stand-alone utility does the job of merging incremental checkpoint files into one non-incremental checkpoint file. Merge is expected to be executed by the system administrator at regular intervals to reduce disk space required by incremental checkpoints and also to assist the speed of recovery from failure.

7.5. Compressed Differences [James S. Plank et al. 1995]

This technique proposes to reduce incremental checkpoint size through the combination of tracking word-level changes to memory (as against page level), buffering and fast compression. It directly acts upon the disadvantage of incremental checkpointing. The compression technique used in this paper is delta compression. The authors mention the reason of why they chose delta compression over LZW and Burrows-Wheeler transform and attribute its inspiration to a technique mentioned in Diskless Checkpointing. The reason for choosing this was that according to analysis by Plank and Li [James S. Plank et al. 1997], compression only aids checkpointing if rate of compression is higher than the rate of disk writes. This is a crucial insight into how the specificity of compression algorithms affects checkpoint optimization substantially. This is one of the only paper among the ones we referenced where we found some reason about why a particular compression algorithm was chosen (or for that matter rejected).

8. TAKEAWAY AND FUTURE DIRECTION

An important point to note among all the above ideas and the general consensus of researchers in fault tolerance is that compressing checkpoints is very essential, and in fact it is looked at to be the next-big-thing as far as optimizing checkpoints is concerned. Having said this, there is a large void that surrounds this insight. The general trend in checkpointing has been to firstly, try and reduce the need to checkpoint at all. Now that we all are at a stage where checkpointing is a must, the focus has been on when to checkpoint. This does remarkably affect checkpoint size as seen in the results of techniques like CATCH [Micah Beck et al. 1994].

We believe that now, assuming compression as a mandatory requirement for efficient checkpointing, we need to focus on what exactly is the content being checkpointed and exploit the knowledge of its structure and nature to achieve overall efficiency through focused, type-specific compression. The motivation for this research lies in locality and the fact that no compression algorithm is best for all types of data [David Salomon 2004]. Consider for example scientific workloads that are predominant on high-end machines. They tend to have very focussed applications running on them for very large amounts of time. Weather predictors, matrix multipliers, earthquake analyzers are examples of workloads that are seen on HPC systems. Their data consists of known type of, but very large set of similar type objects. Temporal / spatial locality and sub-second checkpoint intervals result in the presence of extremely similar data which needs to be checkpointed in every interval.

This can undoubtedly be looked at best from the application specific level, but substantial analysis can also be performed at the compiler level. Compilers can provide immense insight into the structure of data of a program. For languages like C, if we can make a note of the various structures corresponding to the variables at compile-time and identify their occurrence at runtime, we will essentially know which datatype we are dealing with. For more high-level languages, the runtime itself exposes information of the datatype of the variables.

We could leverage this information to dynamically choose from a set of known datatype-specific compression algorithms. We hope that this would result in a much better compression ratio, while at the same time also provide the necessary compression speed making it a win-win situation, especially for HPC systems.

9. CONCLUSION

In this report, we have shown checkpoint / restore mechanisms from a variety of perspectives. We classified checkpoints and gave several examples of currently existing techniques. Along with that, we also covered some interesting optimization techniques that show a very small subset of the various approaches that researchers explore for making checkpoints more efficient. This study throws light on an important void that remains frugally answered - What compression algorithm should be used for checkpoint compression? In our opinion, the answer lies in compiler-level checkpoint techniques that can leverage the data being checkpointed at runtime to dynamically choose the best compression algorithm.

ACKNOWLEDGMENTS

The author would like to thank Prof. Peter Dinda for his immense patience and assistance thorughout this study.

REFERENCES

Bianca Schroeder and Garth Gibson. 2007. Understanding Failures in Petascale Computers, (2007).

Fabrizio Petrini. 2002. Scaling to Thousands of Processors with Buffered Coscheduling, (2002).

Eduardo Pinheiro. 1998. Truly-Transparent Checkpointing of Parallel Applications, (1998).

Jason Duell. 2003. The design and implementation of Berkeley Labs linux Checkpoint/Restart, (2003).

Hua Zhong and Jason Nieh. 2003. CRAK: Linux Checkpoint/Restart As a Kernel Module, (2001).

James Plank and Micah Beck and Gerry Kingsley and Kai Li. 1995. Libckpt: Transparent Checkpointing under Unix, (1995).

Michael Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny. 1997. Checkpointing and migration in the Condor Distributed Processing System, (1997).

Yi-min Wang, Yennun Huang, Kiem-phong Vo, Pi-yu Chung and Ra Kintala. 1995. Checkpointing and Its Applications, (1995).

Bianca Schroeder and Garth Gibson. 2005. A Large-Scale Study of Failures in High-Performance Computing, (2005).

Texas Advanced Computing Center. 2009. Texas Advanced Computing Center, (2009).

Alison Nicholas Norman. 2010. Compiler-Assisted Staggered Checkpointing, (2010).

James S. Plank, Kai Li and Michael A. Puening. 1997. Diskless Checkpointing, (1997).

James S. Plank, Micah Beck and Gerry Kingsley. 1996. Compiler-Assisted Memory Exclusion for Fast Checkpointing, (1996).

Lei Xia and Peter Dinda. 2012. A Case for Tracking and Exploiting Inter-node and Intra-node Memory Content Sharing in Virtualized Large-Scale Parallel Systems, (2012).

Dewan Ibtesham, Dorian Arnold, Kurt B. Ferreira and Patrick G. Bridges. 2011. On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance, (2011).

Micah Beck, James S. Plank and Gerry Kingsley. 1994. Compiler-Assisted Checkpointing, (1994).

Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, Rudolf Eigenmann and Gerry Kingsley. 2012. McrEngine - A Scalable Checkpointing System Using Data-Aware Aggregation and Compression, (2012).

Saurabh Agarwal, Rahul Garg and Meeta S. Gupta. 2004. Adaptive incremental checkpointing for massively parallel systems, (2004).

James S. Plank, Jian Xu and Robert H.B. Netzer. 1995. Compressed Differences: An Algorithm for Fast Incremental Checkpointing, (1995).

David Salomon. 2004. Data Compression: The Complete Reference, version 2. (2004).