

libhashckpt: Hash-based Incremental Checkpointing Using GPU's

Kurt B. Ferreira^{1,3}, Rolf Riesen², Ron Brighwell¹, Patrick Bridges³, and Dorian Arnold³

¹ Scalable System Software
Sandia National Laboratories*
{kbferre | rbrigh}@sandia.gov

² IBM Research, Ireland
rolf.riesen@ie.ibm.com

³ Department of Computer Science
University of New Mexico
{kurt | bridges | darnold}@cs.unm.edu

Abstract. Concern is beginning to grow in the high-performance computing (HPC) community regarding the reliability guarantees of future large-scale systems. Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in HPC systems for the last 30 years. Checkpoint performance is so fundamental to scalability that nearly all capability applications have custom checkpoint strategies to minimize state and reduce checkpoint time. One well-known optimization to traditional checkpoint/restart is incremental checkpointing, which has a number of known limitations. To address these limitations, we introduce `libhashckpt`; a hybrid incremental checkpointing solution that uses both page protection and hashing on GPUs to determine changes in application data with very low overhead. Using real capability workloads, we show the merit of this technique for a certain class of HPC applications.

1 Introduction

Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing (HPC) systems for at least the last 30 years. In current large distributed-memory HPC systems, this approach generally works as follows: periodically all nodes quiesce activity, write all application and system state to stable storage, and then continue with the computation. In the event of a failure, the stored checkpoints are read from stable storage to return the application to a known-good state.

* Sandia National Laboratories is a multi-program laboratory 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.

Checkpoint performance impacts scalability of large-scale applications to such a degree that many capability applications have their own custom *application-specific* checkpoint mechanism to minimize the saved checkpoint state and therefore the time to checkpoint (this time is also referred to checkpoint commit time). While this approach minimizes the application state that must be written to disk, it requires intimate knowledge of the application’s computation and data structures, and is typically difficult to generalize to other applications.

One well-known and generalized optimization of traditional checkpoint/restart is *incremental checkpointing*. Incremental checkpointing [6, 8, 17] attempts to reduce the size of a checkpoint, and therefore the time to write a checkpoint, by saving only differences in state from the last checkpoint.

Current incremental methods have failed to achieve dramatic decreases in checkpoint size because of a reliance on page protection mechanisms to determine which address ranges have been written, or *dirtied*, during the checkpoint interval [8]. Relying solely on page-based mechanisms forces such an approach to work at a granularity of the operating systems page size. Even if only one byte in a page is written, the entire page is marked as dirty and must be saved. Furthermore, if identical values are written to a location, that page is still marked as dirty. These problems are also compounded by the increasing maximum page sizes of modern processors and the increased performance for HPC applications on these larger page sizes.

To address these limitations, we introduce `libhashckpt`: a hybrid incremental checkpointing approach that uses page protection mechanisms, a hashing mechanism, and MPI hooks to determine the locations within a page that have changed. To reduce the overhead of the hash calculation, `libhashckpt` also uses graphics processing units (GPU) to offload the hash calculation. Using real HPC workloads, we compare the performance of this technique against page protection-based incremental systems and highly optimized, application-specific checkpoint techniques. Our results show that our approach is able to dramatically reduce system checkpoint sizes compared to previous incremental checkpointing systems, in some cases approaching the checkpoint sizes of hand-tuned application-specific checkpointing systems.

2 Approach

2.1 Overview

The hash-based incremental checkpointing mechanism in `libhashckpt` works as follows. While the application is running, the library uses the page-protection mechanism to mark those virtual memory pages that have been written in the checkpoint interval as potentially dirty. To support MPI applications, the library also intercepts receive calls and marks message buffers as dirty, identifying them as candidates to be checked by the hashing mechanism. These message buffers require marking because changes in memory from user-level network hardware is not subject to the processor’s page protection mechanisms.

When a checkpoint is requested, the library hashes all blocks corresponding to potentially dirty pages, comparing the key with previously stored values, if they exist. If no key exists, or if the key has changed, the block is marked to be included in the checkpoint and excluded otherwise. If the node contains a GPU, potentially dirty blocks are copied down to the GPU and the computed keys are copied up to host memory. Finally, once the hash calculation has completed, all blocks that have been marked as changed by the library are then saved to stable storage for later retrieval, if needed.

2.2 Library Implementation Details

`libhashckpt` is based on the `libckpt` library [17], now referred to as `clubs` [2]. `Clubs` is a transparent, user-level, checkpoint library for Unix based systems. It contains a number of optimizations including:

- Virtual memory page-protection based incremental checkpointing;
- Forked checkpointing; and,
- User-directed checkpointing which allows the user to include or exclude portions of the processes address space in the checkpoint.

We added the following functionality to this library. Firstly, we added a framework for calculating and storing hash keys of arbitrary block size. The block size can be adjusted to be larger or smaller than the native page size. We also modified the library to intercept MPI receive calls using the MPI profiling layer found in most modern MPI libraries. Finally, we added an engine for offloading this hash calculation to graphics processing units, if any are present.

2.3 Applications and Platform

To evaluate the merit of our hash-based checkpointing library, we present results from two key HPC applications; `CTH` [9] and `LAMMPS` [18, 19]. These applications represent important HPC modeling and simulation workloads. They use different computational techniques, are frequently run at very large scale for weeks at a time, and are key simulation applications for the US Department of Energy. Also, each of these applications contain highly-optimized application-specific checkpoint mechanisms that will be used for comparison with the methods outlined in this paper.

These application tests were conducted on the Cray Red Storm system at Sandia National Laboratories. For these application runs, the hashing was performed by a spare on-node CPU core as Red Storm system does not contain GPUs. For the GPU results in this paper, we compare the performance of the Opteron processor on Red Storm [5] against that of a NVIDIA Tesla C1060 GPU.

3 Results

In this section, we outline the performance of `libhashckpt`. First, we examine the results of hashing versus page-based protection mechanisms for determining the percentage of application memory that has actually changed. Following this, we examine the performance of this library with the two aforementioned simulation workloads, comparing this hash-based approach with both standard page protection-based incremental checkpointing and each application’s specific checkpoint mechanism. Finally, we examine the performance advantage of computing the MD5 [12] hash used by `libhashckpt` using a GPU versus a CPU and use a simple model to outline the viability of this method.

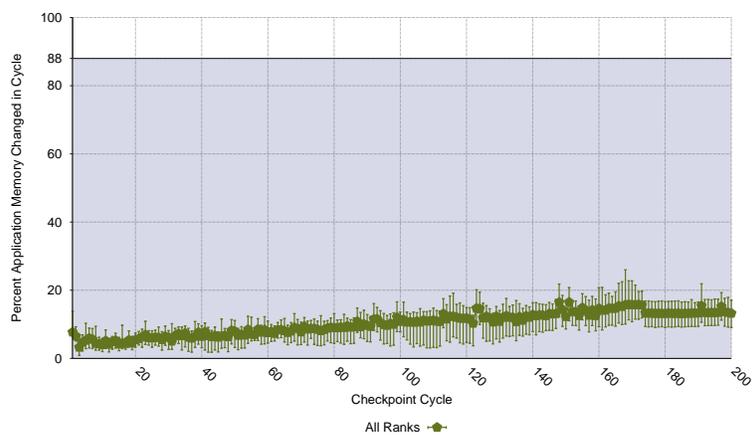
With this hash-based approach *aliasing* is a concern. Aliasing, also referred to as collisions, comes about when modifications to a block are just such that the key values are identical. The danger with aliasing is the library will not save modified application data, thereby corrupting the application in the event of a restart. Previous studies have shown the likelihood of aliasing to be higher in practice than expected theoretically for a number of hash functions. Specifically, with the hash signature functions CRC32 and XOR, the probability of collision has been shown to be too high to be considered safe [7]. Secure hash signatures like MD5 and SHA1, however, have been shown to behave in practice as expected theoretically, and therefore reliable enough to be used in a hash-based approach [13].

3.1 Hash-based Dirty Data Detection

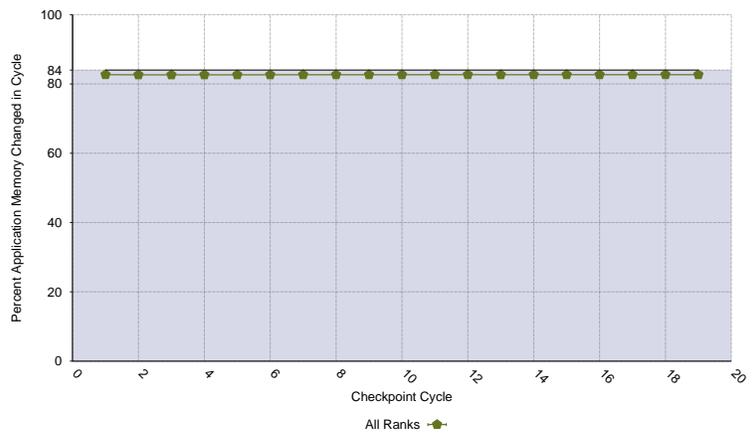
The key feature that `libhashckpt` hopes to exploit is finer-grained detection of dirtied blocks than is currently possible using mechanisms based solely on page protection mechanisms. To examine the overall potential of such a hash-based approach, we first used `libhashckpt` to examine what portion of an application’s memory actually changed (using fine-grained hashing) versus the percentage that a pure page protection-based mechanism would indicate was changed.

Figure 1 shows the percentage of memory that our hash-based mechanism indicates actually changed at each 15 minute checkpoint interval versus the percentage that a page protection mechanism indicates may have changed. For each of these tests, we use a 512 byte block size on an operating system with 4KB pages. Therefore each machine page contains 8 hash blocks. In Figure 1(a), we see that, while nearly all the allocated memory is written in a checkpoint interval, a very small percentage of that memory actually changes. This small percentage of change is an artifact of the simulation problem. The application uses thresholding such that, in a small simulation-time interval, sections of the simulation do not change. In contrast, for LAMMPS in Figure 1(b), the amount of data changed is nearly identical to the data written. This is because the largest data structure in LAMMPS is the neighbor structure, which continuously changes as atoms move around.

These results demonstrate the potential accuracy advantage a hash-based incremental checkpointing approach can provide over a purely page protection-



(a) CTH



(b) LAMMPS

Fig. 1. Average percent of allocated memory changed detected using a hash-based incremental checkpointing mechanism for the CTH and LAMMPS. The shaded region represents the average percent of memory written to using a page-protection based mechanisms. Errorbars are shown for CTH but omitted for LAMMPS as the per-process variation is $\pm 0.5\%$

based mechanism. On the other hand, these results also show that the potential benefits are also highly application-dependent.

3.2 Checkpoint File Size Comparison

Based on the results in the previous section, we then examined the resulting difference in checkpoint sizes between the two incremental checkpointing approaches (pure page protection vs. `libhashckpt`'s hybrid page protection/hashing scheme). We also compared the size of these checkpoints with those generated by the application-specific mechanisms. These application specific methods are highly optimized, and, for the purpose of this work, we view these checkpoint sizes as a file size optimum.

Application	VM CKPT (MB)	Hash CKPT (MB)	App CKPT (MB)
CTH	513	35 (93%)	26 (95%)
LAMMPS	2735	2670 (2.3%)	608 (78%)

Table 1. Per-process checkpoint size for CTH and LAMMPS. This table contains the size of the checkpoint using standard page protection-based system-level incremental checkpointing (VM CKPT), `libhashckpt`'s hybrid approach, and an application-specific checkpointing approach (App CKPT). For the latter two columns the number in parenthesis is the percent reduction in size when compared to a system-based incremental checkpoint. The VM CKPT and Hash CKPT checkpoints contains data from both the application as well as other libraries linked with the application, for example MPI library data and its associated buffers.

Table 1 shows a comparison in per-process checkpoint sizes for our two applications. We see that for CTH, `libhashckpt`'s hash-based method dramatically reduces the size of system-based incremental checkpoints based solely on a page protection mechanism. Custom application-specific checkpointing mechanism does better still, but our hybrid scheme results in checkpoints that are only 35% larger than this highly-optimized approach. One reason our hash-based library is larger than the application-specific method has to do with the fact that the application checkpoint contains *only* application data, while the other methods shown save state from the application as well as the libraries linked with the application, most notably the MPI library and its associated data and buffers.

In contrast to CTH, the hash- and page-based schemes are nearly identical in size for LAMMPS, with application-specific checkpointing routines offering a 75% reduction in checkpoint sizes. This is because the application-specific checkpointing mechanism in LAMMPS can completely avoid writing neighbor structures to checkpoints because they can be reconstructed at application restart, while system-based methods do not have the application-specific knowledge needed to do this.

3.3 GPU Performance

Figure 2 compares GPU vs CPU performance of an MD5 calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation. Also, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed keys to host memory. The CPU numbers use the Libgcrypt MD5 implementation. From this figure, we see that the GPU greatly outperforms the CPU implementation.

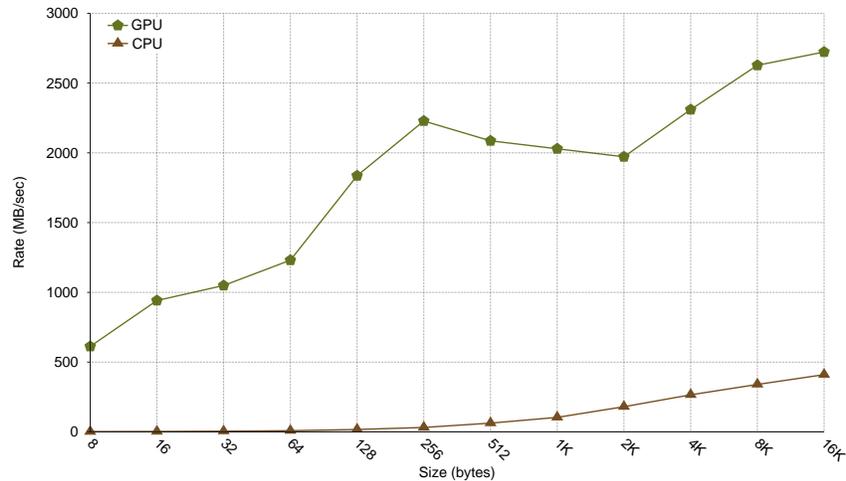


Fig. 2. A comparison of MD5 hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [1] MD5 hashing algorithm.

In addition, with a per-process rate between 600 and 2600 MB/sec, the GPU-based data rates greatly exceed the per-process rate to stable storage for many large scale systems. In the next section we construct a simple model to further illustrate the viability of this approach.

3.4 Viability of Hash-Based Incremental Checkpointing

To evaluate the viability of this method we will compare the performance of this hash-based mechanism with that of a strictly page-based approach. This hash-based approach will outperform the page-based approach when the reduction in

the checkpoint size for the hash method outweighs the cost of computing the hashes of the modified pages. More specifically, this approach is viable when: ⁴

$$\frac{|checkpoint|}{\beta_{hash}} + \frac{(1 - compression) \times |checkpoint|}{\beta_{ckpt}} < \frac{|checkpoint|}{\beta_{ckpt}} \quad (1)$$

where $|checkpoint|$ is the size of page-based checkpoint, $compression$ is the percent reduction of hash-based approach in comparison to the page-based method, β_{hash} is the hashing rate, and β_{ckpt} is the rate of checkpoint commit. This equation can be reduced to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < compression \quad (2)$$

Using the CTH data presented previously in this paper, $compression$ is 83% and the β_{hash} mean is around 2.0GB/s. Therefore, if a machine has a per-process checkpoint commit speed is less than 1.66GB/s then the hash-based approach will have a lower overhead than the strictly page-based approach. 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 per-process disk commit bandwidth of this magnitude for many large scale systems. A per-process commit rate greater than this 1.66GB/sec value and the page based approach will have lower overheads. For LAMMPS, the compression is 2.4%, therefore the per-process checkpoint commit breakpoint speed is much lower at 48MB/sec; a value more easily reached by current parallel I/O systems.

4 Related Works

Checkpoint/restart is a well-known method for application fault-tolerance for large-scale distributed and parallel systems that has been studied extensively for over thirty years [8]. A number of optimizations has been suggested including; forked or copy-on-write checkpointing [10], checkpointing to remote nodes [20], communication-induced checkpointing [15], compiler-assisted checkpointing [4], incremental checkpointing [6, 11], and probabilistic or hash-based checkpointing [14, 3]. However, none of these methods have yet matched the performance of application-specific methods and are therefore not widely accepted by most capability workloads.

Most closely related to this work, Agarwal et al. [3] investigated the performance characteristics of a hash-based adaptive incremental checkpointing library. Similar to this work, the authors use an MD5 hash to determine the portions of an application address space that have changed in a checkpoint interval. In contrast to this work, we evaluate the merit of this hash-based technique on actual HPC capability workloads. In addition, we show how GPUs can be used

⁴ Plank et al pose a similar concept [16]

to significantly reduce the overhead of the hash computations. This overhead is important as the computation overhead must be kept significantly lower than the rate to save to stable storage. Also, we compare the merit of this technique with an optimal application-specific checkpoint mechanism. Finally, our work varies from this previous work as we show that, while this technique may be appropriate for some applications, there are classes of HPC applications for which this method is clearly not appropriate.

5 Conclusions and Future Work

In this paper, we introduced `libhashckpt`, an incremental checkpointing library that uses hashing to save only the changed state of an application in a checkpoint interval. To significantly decrease the overhead of the hash calculation, `libhashckpt` can utilize GPUs. Using this library, we compare the checkpoint file sizes of this hash-based method with that of a standard page-protection mechanism and a highly optimized application-specific mechanism. Using real capability HPC workloads we show that, for a certain class of applications, this hash-based method can reduce the checkpoint file size to be around 15% of that of a page-based approach. In addition, this method can create checkpoint files which are only 35% larger than that of a manually-coded, application-specific method. Finally, we introduced a simple model to illustrate this proposed techniques viability for real-world HPC workloads.

There are several avenues of future work related to this research. First, we would like to analyze more applications in order to evaluate the merit of this technique to a broader set of large-scale applications. In addition, we would like to investigate other hash and checksum algorithms. For this study we used a cryptographically secure hash (MD5), but this algorithm may be overkill for determining block changes and other collision resistant, yet less computationally intense, hash signatures may have lower overheads. Lastly, we need to compare this method with other checkpoint optimization techniques, such as compiler-assisted incremental checkpoint methods.

References

1. libgrypt web page (Jul 2010), <http://directory.fsf.org/project/libgrypt/>
2. Libckpt web page (2011), <http://web.eecs.utk.edu/~plank/plank/www/libckpt.html>
3. Agarwal, S., Garg, R., Gupta, M.S., Moreira, J.E.: Adaptive incremental checkpointing for massively parallel systems. In: Proceedings of the 2004 International Conference on Supercomputing. St. Malo, France (2004)
4. Bronevetsky, G., Marques, D., Pingali, K., McKee, S.A., Rugina, R.: Compiler-enhanced incremental checkpointing for openmp applications. In: IPDPS. pp. 1–12. IEEE (2009)
5. Camp, W.J., Tomkins, J.L.: Thor’s hammer: The first version of the Red Storm MPP architecture. In: In Proceedings of the SC 2002 Conference on High Performance Networking and Computing. Baltimore, MD (November 2002)

6. Chen, Y., Plank, J.S., Li, K.: CLIP: a checkpointing tool for message-passing parallel programs. In: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM). pp. 1–11. Supercomputing '97, ACM, New York, NY, USA (1997), <http://doi.acm.org/10.1145/509593.509626>
7. Elnozahy, E.N.: How safe is probabilistic checkpointing? In: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing. pp. 358–. FTCS '98, IEEE Computer Society, Washington, DC, USA (1998), <http://portal.acm.org/citation.cfm?id=795671.796882>
8. 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)
9. E.S. Hertel, J., Bell, R., Elrick, M., Farnsworth, A., Kerley, G., McGlaun, J., Petney, S., Silling, S., Taylor, P., Yarrington, L.: CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In: Proceedings of the 19th International Symposium on Shock Waves, held at Marseille, France. pp. 377–382 (July 1993)
10. Feldman, S.I., Brown, C.B.: Igor: a system for program debugging via reversible execution. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging. pp. 112–123. PADD '88, ACM, New York, NY, USA (1988), <http://doi.acm.org/10.1145/68210.69226>
11. Gioiosa, R., Sancho, J.C., Jiang, S., Petrini, F.: Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In: Proceedings of the 2005 ACM/IEEE Conference on High-Performance Computing and Networking. Seattle, WA, USA (2005)
12. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. (1996)
13. chang Nam, H., Kim, J., Hong, S.J., Lee, S.: A secure checkpointing system. In: Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on. pp. 49 –56 (2001)
14. Nam, H.C., Kim, J., Hong, S., Lee, S.: Probabilistic checkpointing. In: Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on. pp. 48 –57 (jun 1997)
15. Netzer, R.H.B., Xu, J.: Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.* 6, 165–169 (February 1995), <http://dx.doi.org/10.1109/71.342127>
16. Plank, J.S., Li, K.: ickp: A consistent checkpointer for multicomputers. *Parallel & Distributed Technology: Systems & Applications*, *IEEE* 2(2), 62–67 (1994)
17. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: transparent checkpointing under unix. In: Proceedings of the USENIX 1995 Technical Conference Proceedings. pp. 18–18. TCON'95, USENIX Association, Berkeley, CA, USA (1995), <http://portal.acm.org/citation.cfm?id=1267411.1267429>
18. Plimpton, S.J.: Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics* 117, 1–19 (1995)
19. Sandia National Laboratory: LAMMPS molecular dynamics simulator. <http://lammmps.sandia.gov> (Apr 10 2010)
20. Zandy, V.C., Miller, B.P., Livny, M.: Process hijacking. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing. pp. 32–. HPDC '99, IEEE Computer Society, Washington, DC, USA (1999), <http://portal.acm.org/citation.cfm?id=822084.823234>