

# Accelerating Incremental Checkpointing for Extreme-Scale Computing

Kurt B. Ferreira<sup>a</sup>, Rolf Riesen<sup>b</sup>, Patrick Bridges<sup>c</sup>, Dorian Arnold<sup>c</sup>, Ron Brightwell<sup>b</sup>

<sup>a</sup>*Scalable System Software Department  
Sandia National Laboratories  
Albuquerque, NM 87185-1319*

<sup>b</sup>*IBM Research, Ireland*

<sup>c</sup>*Department of Computer Science  
University of New Mexico  
Albuquerque, NM*

---

## Abstract

Concern is beginning to grow in the high-performance computing (HPC) community regarding the reliability 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 describe `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 and a model outlining the viability and application efficiency increase of this technique, we show that hash-based incremental checkpointing can have significantly lower overheads and increased efficiency than traditional coordinated checkpointing approaches at the scales expected for future extreme-class systems.

*Keywords:* Fault-tolerance, Checkpointing, Incremental checkpointing, Graphics processing units

---

## 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

---

*Email addresses:* [kbferre@sandia.gov](mailto:kbferre@sandia.gov) (Kurt B. Ferreira), [rolf.riesen@ie.ibm.com](mailto:rolf.riesen@ie.ibm.com) (Rolf Riesen), [bridges@cs.unm.edu](mailto:bridges@cs.unm.edu) (Patrick Bridges), [darnold@cs.unm.edu](mailto:darnold@cs.unm.edu) (Dorian Arnold), [rbbrigh@sandia.gov](mailto:rbbrigh@sandia.gov) (Ron Brightwell)

<sup>1</sup>Sandia National Laboratories is a multi-program 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.

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 computation. In the event of a failure, the stored checkpoints are read from stable storage to return the application to a previous known-good state.

Checkpoint performance impacts scalability of large-scale applications to such a degree that many capability applications have their own optimized *application-specific* checkpoint mechanism to minimize the saved checkpoint state and therefore the time to write the checkpoint to stable storage (this time is also referred to as the 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 [1, 2, 3] attempts to reduce the size of a checkpoint, and therefore the checkpoint commit time, by saving only differences (or deltas) in state from the last checkpoint. The underlying assumption being that the mechanism used to determine the differences in state has significantly lower overhead than the time to save the additional data to stable storage.

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 [2]. Relying solely on page-based mechanisms forces such an approach to work at a granularity of the operating systems page size. Therefore, 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 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 describe a hybrid incremental checkpointing approach that uses page protection mechanisms, a hashing mechanism that can be optionally be offloaded to GPUs if available and idle. GPUs reduce the overhead and power consumption of the hash calculation. Using real HPC workloads, this work compares the performance of this technique against a page protection-based incremental systems and a highly optimized, application-specific checkpoint technique. 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. Our results also show that this technique can significantly improve application efficiency, with the key performance factor being the amount of memory compression from the technique (i.e. the size of the checkpoint file), rather than the speed of the incremental approach.

This paper is organized as follow. First in Section 2, we define a model to illustrate when this hash-based approach will pay off both in comparison to a page-based incremental checkpointing approach as well as a more traditional, disk-based checkpointing approach. In Section 3, we describe the design and implementation of `libhashckpt`, a previously published [4] incremental checkpointing library. We show the resulting checkpoint state

compression from this technique using a number of real-world HPC capability workloads in Section 4. In addition, we compare the compression results against an optimal application-based checkpointing mechanism. In Section 5, using a number of hash algorithms, we show the costs of performing this hashing on a CPU versus the speedup seen using a GPU. Section 6 uses the aforementioned model and measured results to present the viability of this technique using a GPU and CPU for possible systems in the exascale design space thereby defining under which situations we would use this approach. In Section 7 we outline the increase in application efficiency in those scenarios where the approach is viable. Related checkpoint optimization work is discussed in Section 8. We conclude with a discussion of the implications of this work as well as ongoing research in Section 9.

## 2. A Model for the Viability of Hash-Based Incremental Checkpointing

To evaluate the viability of this method we compare the performance of this hash-based mechanism first against that of a strictly page-based approach. This hash-based approach outperforms a 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 the sum of the time to hash modified memory ( $T_{hash}$ ), plus the time to write the application blocks that have been determined changed ( $T_{write\ hash}$ ), is less than the time to write the memory that hash been determined changed using a strictly page-based approach ( $T_{write\ whole}$ ). This model was first introduced by Plank et al in [5]. For clarity we provide it here. In more detail we have:

$$T_{hash} + T_{write\ hash} < T_{write\ whole} \quad (1)$$

$$\left(\frac{|c|}{\beta_{hash}}\right) + \left(\frac{(1 - \alpha) \times |c|}{\beta_{ckpt}}\right) < \frac{|c|}{\beta_{ckpt}} \quad (2)$$

Where:

$|c|$  is the size of page-based checkpoint

$\alpha$  is the percent reduction of hash-based approach in comparison to the page-based method

$\beta_{hash}$  is the per-process hash rate

$\beta_{ckpt}$  is the per-process checkpoint commit rate

This equation can be reduced to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < \alpha \quad (3)$$

The maximum per-process checkpoint commit rate ( $\beta_{ckpt}$ ) is generally known for many HPC platforms. Therefore, we must measure the hashing rate ( $\beta_{hash}$ ), which is specific to both a specific platform and hashing algorithm; and the compression percentage ( $\alpha$ ), which will be specific to a particular application. In the next section, we use the `libhashckpt` library to measure these quantities.

### 2.1. Viability Against Coordinated Checkpointing

In this section we outline the viability of this hash-based technique in comparison to traditional checkpoint restart. Similar to above, this approach is viable when the sum of the time to hash modified memory ( $T_{hash}$ ), plus the time to write the application blocks that have been determined changed ( $T_{write\ hash}$ ), plus the time to mark dirty pages during a checkpoint interval ( $T_{dirty}$ ), is less than the time to write the memory that has been determined changed using a strictly page-based approach ( $T_{write\ whole}$ ). Again we have:

$$T_{hash} + T_{write\ hash} + T_{dirty} < T_{write\ whole} \quad (4)$$

$$\left(\frac{|c|}{\beta_{hash}}\right) + \left(\frac{(1 - \alpha) \times |c|}{\beta_{ckpt}}\right) + T_{dirty} < \frac{|c|}{\beta_{ckpt}} \quad (5)$$

As we will show in later sections, for checkpoint intervals expected on future systems,  $T_{dirty}$  is equal to 0. Therefore, this equation again reduces down to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < \alpha \quad (6)$$

The same as Equation 3. Therefore, the viability of this approach is the same in comparison to both a page-based incremental approach and a traditional checkpoint/restart mechanism.

## 3. Libhashckpt: Hash-based Incremental Checkpointing

### 3.1. Overview

This work uses a previously published incremental checkpoint library called `libhashckpt` [4]. The hash-based incremental checkpointing mechanisms in `libhashckpt` works as follows. While the application is running, the library uses the page-protection mechanism (i.e. `mprotect()`) to mark those virtual memory pages that have been written in the checkpoint interval as potentially dirty. To support MPI applications, the library must also intercept all receive calls, including those implicit to collective operations which use buffers internal to the MPI library. `libhashckpt` marks all receive message buffers as dirty, identifying them as candidates to be checked by the hashing mechanism. These message buffers require marking as changes in memory from high-performance, user-level network hardware, such as those found on leadership-class HPC systems, are not subject to the processor’s page protection mechanisms. Therefore, receive operations which write into these buffers will not be marked by the page based mechanisms

When a checkpoint is requested, the library hashes all blocks corresponding to potentially dirty pages, comparing the key with previously stored key values for the block, 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. As many leadership-class, extreme-scale systems are increasing constructed with on-node graphics processing units (GPUs) that a number of DOE and DOD workloads can not effectively utilize, if the node contains an idle GPU, potentially dirty blocks are optionally copied down to the GPU and the computed keys are

copied up to host memory. For those workloads that can use these GPUs, hash calculations can be done on the CPU similar to manner described in [6]. Finally, once the hash calculation has completed, all blocks that have been marked as changed by the library are saved to stable storage for later retrieval, if needed.

### 3.2. Implementation Details

To evaluate the merit of this hash-based approach, we created the `libhashckpt` hash-based, hybrid incremental checkpointing library. `libhashckpt` is based on the `libckpt` library [3], now referred to as `clubs` [7]. `Clubs` is a transparent, user-level, checkpoint library for Unix based systems. It contains a number of checkpointing 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, including those in MPI collective operation, using the MPI profiling layer found in most modern MPI libraries. Also, we added an engine for offloading this hash calculation to graphics processing units, if any are present.

### 3.3. Hash/Checksum Algorithms

In this section we briefly describe each of the checksum and hash algorithms used in this work. These algorithms vary greatly in both their collision resistance and their computational complexity, from the relatively simple `XOR` and `CRC32` checksums to the complex, collision resistant, and cryptographically secure `MD5` and `SHA256`. In later sections we compare the execution performance of these algorithms using CPUs and GPUs.

#### 3.3.1. Rotating XOR

The rotating `XOR` function, shown in Listing 1, is a simple hash algorithm that repeatably `XOR` input data and folds this input data with individual bytes of the running 32 bit output value. This folding and mixing of the input data gives the rotating hash a much better distribution than a standard `XOR`. The advantage of this method is its simple computation. Though this folding step sufficiently mixes the input data, this algorithm generally is not considered secure enough to be used for cryptographic applications.

Listing 1: Rotating XOR Algorithm

```

1  #include <stdint.h>
2
3  uint32_t
4  rotating_xor( void *addr, int len )
5  {
6      unsigned char *p = addr;
7      uint32_t h = 0;
8      int i;
9
10     for( i = 0 ; i < len ; i++ )
11         h = ( h << 4 ) ^ ( h >> 28 ) ^ addr[ i ];
12
13     return h;
14 }

```

### 3.3.2. ADLER32

Invented by Mark Adler, **ADLER32** is a cyclic redundancy checksum algorithm defined in RFC1950 [8]. This checksum algorithm is part of the widely-used **zlib** compression library as well as the **rsync** data transfer and synchronization utility.

The **ADLER32** checksum is obtained by concatenating two 16-bit checksums A and B into one 32 bit output. In this scheme, A is the sum of all bytes in the block and B is the sum of the individual values of A from each step. The **ADLER32** checksum is considerably faster to compute on most platforms and slightly less collision resistant than a **CRC32**. **ADLER32**'s collision issues occur for very small block sizes, as the sum of A does not have the opportunity to wrap around. Similar to **XOR**, an **ADLER32** checksum can be easily forged and therefore generally not considered appropriate for application requiring strong collision resistance.

### 3.3.3. CRC32

A cyclic redundancy check (**CRC32**) [9] is 32 bit hashing algorithm commonly used for error detection and correction on many storage and network devices such as Ethernet. **CRC32**'s have the advantage of being simple to implement and are well suited in detecting contiguous error symbols. Typically, a **CRC32** can detect a fraction  $(1 - 2^{-n})$  of all burst errors larger than  $n$  bits in length.

A cyclic redundancy check algorithm requires a generator polynomial. This polynomial is the divisor of an operation with the value to be hashed treated as the dividend. The remainder of this polynomial division is the return value, or referred to as the CRC.

Similar to the other methods described thus far in this section, **CRC32**'s are not suitable for cryptographic applications. Most notably, due to the linear nature of a CRC, a message can easily be modified in such a way to leave the CRC output unchanged and therefore is considered vulnerable to collisions.

### 3.3.4. MD5

The fifth Message-Digest Algorithm (**MD5**) [9] is a widely used cryptographic hash function designed by Ron Rivest. Specified in RFC1321 [10], **MD5** produces a 128-bit hash value and is commonly used to check data integrity. Though designed to be collision resistant, a collision attack currently exists for **MD5**. The fact this attack exists has no influence on its choice for an appropriate hashing method, as while collisions can be computed, they rarely occur and are difficult to generate.

| Algorithm | Key Size (bits) | Comparisons Until Collision | Data Checkpointed (bytes) |
|-----------|-----------------|-----------------------------|---------------------------|
| XOR       | 32              | $8.2 \times 10^4$           | $2.1 \times 10^7$         |
| ADLER32   | 32              | $8.2 \times 10^4$           | $2.1 \times 10^7$         |
| CRC32     | 32              | $8.2 \times 10^4$           | $2.1 \times 10^7$         |
| MD5       | 128             | $2.3 \times 10^{19}$        | $5.9 \times 10^{21}$      |
| SHA256    | 256             | $4.2 \times 10^{38}$        | $1.1 \times 10^{41}$      |

Table 1: Average number of hash block comparisons and data checkpointed until a collision is expected to occur assuming a worst case scenario of 256 byte block sizes. These numbers are in comparison to the silent data corruption rate on current leadership-class machines containing hundreds of terabytes of chipkill protected [20] DRAM memory. On these machines, undetected errors are predicted to occur a few times per day [21]. For this worst case block size, hash key sizes greater or equal to 128 bits ensure an undetected bit flip is more likely. For smaller key sizes, the likelihood of collision can be mitigated by increasing hash block size.

### 3.3.5. SHA256

The second Secure Hash Algorithm (SHA256) [11] is one of a family of cryptographic hash functions which includes SHA224, SHA256, SHA384, SHA512, each of which varies by the hash digest size (224, 256, 384, and 512 bits). This set of functions was designed by the National Security Agency in response to a flaw found in the SHA-1 secure hash.

The SHA-2 family of functions are included in a number of widely-used security applications and protocols, including TLS [12] and the Secure Sockets Layer, PGP [13], SSH [14], S/MIME [15], and IPsec [16]. Like all well designed cryptographic hashes, they are highly collision resistant.

### 3.4. The Hash Aliasing Problem

As with any hash-based approach in which the number of hash keys is smaller than the number of original blocks, *aliasing* is a concern. Aliasing, also referred to as hash collision, comes about when modifications to a block are just such that the key values for the two blocks are identical. The danger being that the library will believe a block has not changed, not save the modified data, thereby corrupting the application in the event of a restart.

To determine the expected (or average) number of collisions expected to occur, we can use a well known from probability theory called the *birthday problem* [17, 18]. The birthday problem asks how many people, on average, need to be brought together until there are enough to have a greater than 50% chance that two of these individuals share the same birth month and day (assuming birthdays from the population are from the same distribution and independently distributed). This well studied problem and its surprisingly low result (around 25 people) has been used in many different fields including cryptography and hashing [19].

We use the result of this problem to determine how many hash block comparisons which need to be done until the expected number of collisions is equal to 1. Table 1 has the expected number of key comparisons until a collision, assuming each hash block is equally likely. For each hashing algorithm the important factor is the key size (analogous to number

of days of year for original birthday problem). Also in this table we show the average amount of checkpoint data assuming a worst case block size of 256 bytes. From this table we see the expected result that the larger key sizes have a lower likelihood of aliasing. For key sizes equal to or greater than 128 bits, an undetected errors in DRAM memory is more likely to occur than a hash collision. On these leadership-class machines, which contain hundreds of terabytes of chipkill protected memory [20], these undetected errors are predicted to occur a few times a day [21]. For these smaller 32 bit key sizes, the likelihood collision is much greater than silent data corruption. The chances of these collisions can be mitigated by increasing the hash block size to larger than 256 bytes or ensuring that block modifications do not cause a collision. Previous work, which looked at aliasing [22] showed that the application most similar to many HPC workloads, a matrix multiplication workload, showed no aliasing issues for the XOR and CRC16, using smaller hash key sizes than those studied here.

#### 4. State Compression Measurement

In this section, we present the compression performance of this hash-based approach using the `libhashckpt` library described in the previous section. First, we examine the results of hashing versus page-based protection mechanisms for determining the percentage of application memory that has actually changed. Then, we examine the state compression performance of this library with a number of simulation workloads, comparing this hash-based approach with both standard page protection-based incremental checkpointing and an application’s specific checkpoint mechanism.

##### 4.1. Applications and Platform

To evaluate the compression achieved by hash-based checkpointing, we present results from a number of key HPC applications; CTH [23], LAMMPS [24, 25], SAGE [26], and HPCCG [27]. These application represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads to both the US Department of Defense and Department of Energy. These four applications represent different communication characteristics and compute to communication ratios.

1. CTH [23] is a multi-material, large deformation, strong shock-wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.
2. SAGE, SAIC’s Adaptive Grid Eulerian hydro-code, is a multi-dimensional, multi-material, Eulerian hydrodynamics code with adaptive mesh refinement that uses second-order accurate numerical techniques [26]. It represents a large class of production applications at Los Alamos National Laboratory. It is a large-scale parallel code written

in Fortran 90 and uses MPI for inter-processor communications. It routinely runs on thousands of processors for months at a time.

3. LAMMPS [24] is a classical molecular dynamics code developed at Sandia National Laboratories. For our experiments we use the embedded atom method (EAM) metallic solid input script which is used by the Sequoia benchmark suite. The LAMMPS code and input scripts are provided on the LAMMPS web site [25]. For this experiment we ran LAMMPS in weak-scaling mode.
4. The HPCCG mini-application, part of the Mantevo project [27], is a simple sparse conjugate gradient solver designed to capture an important component of Sandia's production workload. The majority of its runtime is spent performing sparse matrix-vector multiplies, where the sparse matrix is encoded in compressed row storage format. The interprocessor communication is minimal, requiring exchange of nearest neighbor boundary information, in addition to global `MPI_Allreduce()` operations required for the scalar computations in the conjugate gradient algorithm.

CTH, SAGE, LAMMPS each 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 1024, dual-core nodes of the Cray Red Storm system [28] at Sandia National Laboratories. For these application runs, the hashing was performed by a spare on-node CPU core. A checkpoint time of 15 minutes was used for each of these applications. This interval was chosen as, historically, checkpoint commit times on the largest leadership-class machines has remained relatively constant at this level for the past ten years [29]. Checkpoint intervals less than this 15 minutes ensure no progress will be made with the application.

Not included in this section is performance overheads of this library on Red Storm. At the 15 minute checkpoint interval used in this work, no statistically significant slowdown in performance in comparison to traditional checkpointing was observed. This low overhead is due to the fact that each written page is only marked as dirty once in a checkpoint interval. All further accesses proceed without interference. In order to see any slowdown due to the page access tracking, we needed to decrease the checkpoint interval to less than two minutes. As described earlier, this interval is much smaller than could be used on a production HPC system while still ensuring application progress.

#### *4.2. Hash-based Dirty Data Detection*

The key feature that `libhashckpt` exploits 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. In this section we show the average percent of memory written using a page protection-based mechanism. In addition we show the average, minimum, and maximum percentage across all nodes of the written memory that is determined changed using a hash-based approach.

Figures 1 – Figure 4 show the percentage of memory that our hash-based mechanism determined changed at each 15 minute checkpoint interval versus the percentage that a page protection mechanism determined were dirtied. For each of these tests, we use a 512 byte block size on an operating system with 4KB pages. Each machine page therefore, contains 8 hash blocks. A small scale sweep of hash block sizes over an order of magnitude smaller and larger show this value to be optimal in terms of write granularity for all the application tested.

In Figure 1, we see that while nearly all of CTH’s 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 for CTH and many similar workloads. The application uses thresholding such that, in a simulation-time interval, if sections of the simulation do not change above a certain threshold deemed to be significant, the values remain the same.

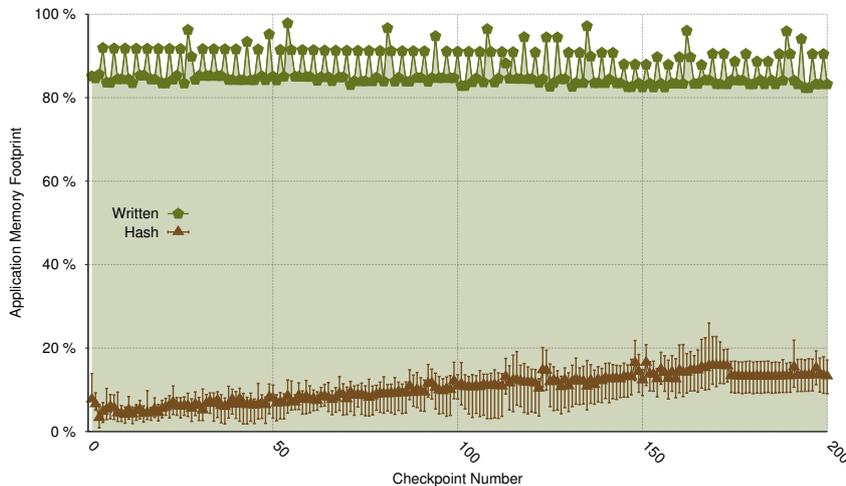


Figure 1: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the CTH exploding pipe problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

In contrast to the CTH results, the amount of data changed for LAMMPS, shown in Figure 2, is nearly identical to the data written. This large change in data is due to the fact that the largest data structure in LAMMPS is the neighbor structure. This structure holds distance information between all atoms and is used for calculating forces. As the simulation progresses, this structure continuously changes as atoms move around.

In Figure 3, we see that the performance of SAGE sits somewhere between that of CTH and LAMMPS. For some nodes in this SAGE problem, much of the node’s data changes in the checkpoint interval. For other nodes, however, the amount of data on a node that changes is much lower than the total amount a page-based mechanism determines changed. The average amount of data changed across all nodes and for all checkpoints is around 55%.

Lastly, Figure 4 shows the results of HPCCG which are similar to LAMMPS, where most of the data written is different than what was there previously. In contrast to LAMMPS, as HPCCG converges an increasingly smaller percentage of the written memory changes.

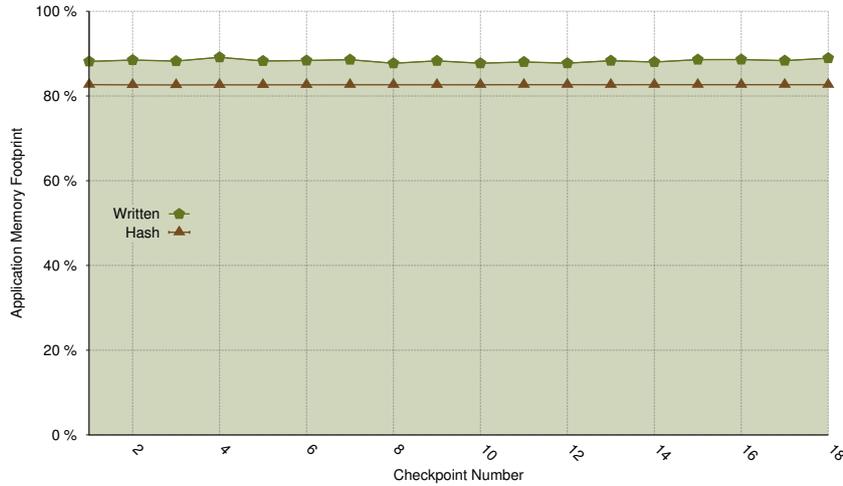


Figure 2: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the LAMMPS EAM problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

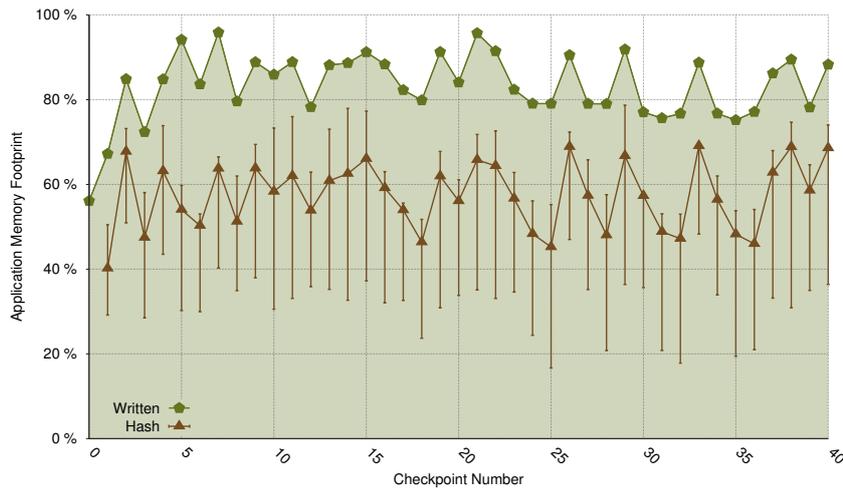


Figure 3: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for SAGE application. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

These results demonstrate the potential accuracy advantage a hash-based incremental checkpointing approach can provide over a purely page protection-based mechanism. On the other hand, these results also show that the potential benefits are also highly application-dependent and in fact may even be dependent to a specific type of problem with a given application.

#### 4.3. Checkpoint File Size Comparison

Based on the results in the previous section, we can now examine the resulting difference in checkpoint sizes between the two incremental checkpointing approaches (pure page

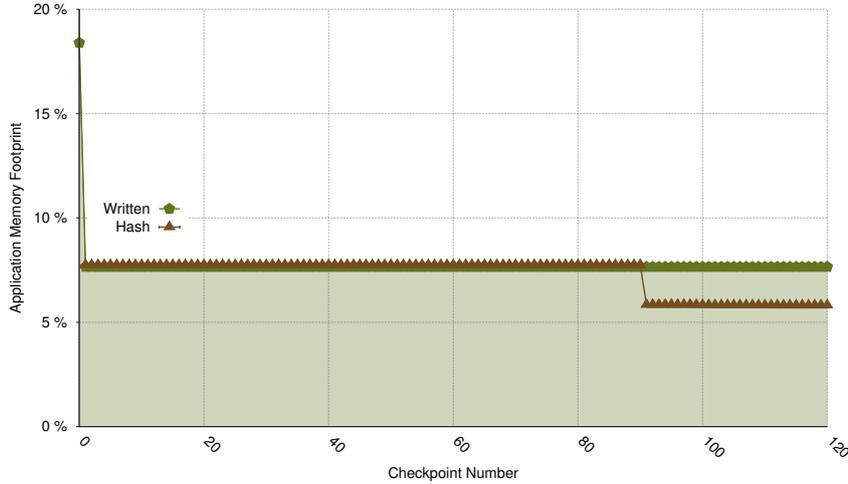


Figure 4: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for HPCCG. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

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

Table 2: 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.

protection vs. `libhashckpt`'s hybrid page protection/hashing scheme) for both LAMMPS and CTH. These two application are chosen due to there highly optimized application-based mechanisms. We also compare the hash and page-based checkpoint sizes 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. The purpose of this comparison is to see how close to this optimal we can get without having any application knowledge beyond what the library can gather from access patterns and hash values.

Table 2 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 internal 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 due to the fact that the application-specific checkpointing mechanism in LAMMPS can completely avoid writing neighbor structures described previously in checkpoints as they can be reconstructed at application restart. LAMMPS only needs to save Atom location and type information. System-based methods, such as ours, do not have the application-specific knowledge required to do this.

## 5. Hashing Costs

In the previous section we used a spare on-node CPU to perform the hashing of modified pages. This hashing can be very expensive on a host CPU. This high cost determines the possible merits of this technique. As we specified in Equation 2, this technique is viable if the hashing costs outweigh the decrease in state compression. Therefore, we are interested in methods to speed up the hashing. The method used to lower the overheads in this work is to offload the hash calculation to GPUs. Therefore, the reasons to use a GPU for this calculation are the higher viability bandwidths (which mean viability in a larger portion of the exascale design space), lower power consumption, or the fact that the CPU is currently busy with other important work.

In this section we measure and compare the GPU vs CPU performance for a number of hash signature algorithms. For the hashing results in this section, we compare the performance of the Opteron processor on Red Storm [28] against that of a NVIDIA Tesla C1060 and a NVIDIA Tesla D2090 based on the “Fermi” architecture. For each of the tests we did the following. We take one of the checkpoints for the CTH application run described earlier in the chapter. In this checkpoint we send all the written pages to be hashed either by the CPU for the GPU. For the CPU numbers we use the Libgcrypt [30] implementations of XOR, ADLER32, CRC32, MD5, and SHA256 algorithms. All GPU numbers presented in the following section 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 for asynchronous CUDA [31] kernels. 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.

*Rotating XOR.* Figure 5 compares GPU vs. CPU performance of an XOR calculation for varying block sizes. As stated previously, all 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. With a per-process hashing rate between 2800 and 1700 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable

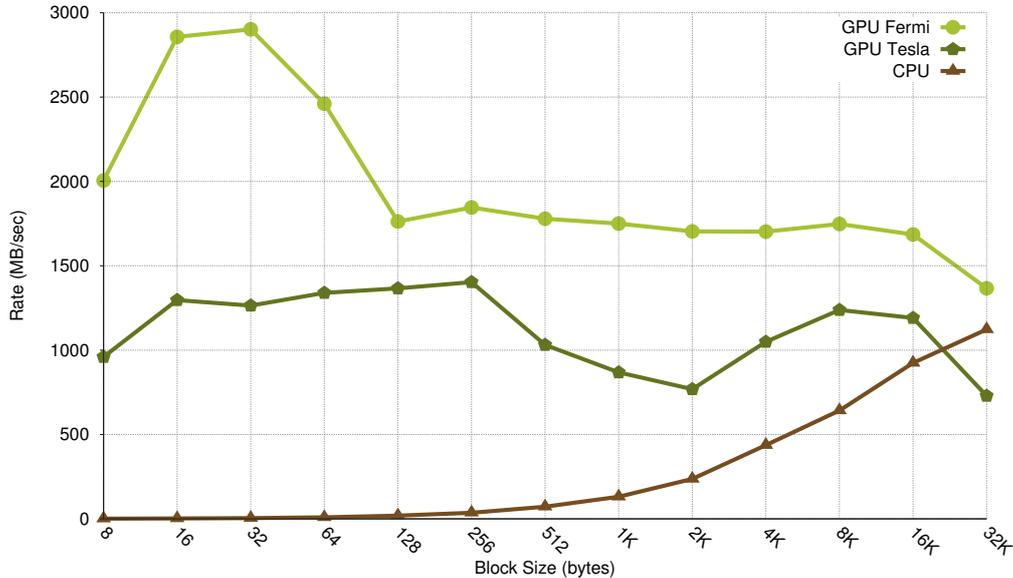


Figure 5: A comparison of rotating XOR hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU test use the XOR algorithm described previously

storage for many large-scale systems. Also, for larger block sizes, including sizes beyond what is shown here, the CPU results exceed that of the GPU cards. For comparison, recent I/O studies [32] place the per-process commit bandwidth to stable storage  $1MB/sec$ .

**CRC32.** Figure 6 compares GPU vs. CPU performance of an CRC32 calculation for varying block sizes. With a per-process hashing rate between 2200 and 700 MB/sec for the GPU cards, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large scale systems. Also, even though the Fermi cards have twice as many resources, for block sizes larger than 64 bytes the performance of the two are nearly the same.

**ADLER32.** Figure 7 compares GPU vs. CPU performance of an ADLER32 calculation for varying block sizes. With a per-process hashing rate between 3200 and 2000 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large-scale systems. Also, for larger block sizes the CPU results exceed that of the Tesla GPU card. For block sizes larger than those show in this figure, the CPU performance exceeds even that of the Fermi card.

**MD5.** Figure 8 compares GPU vs. CPU performance of an MD5 calculation for varying block sizes. With a per-process hashing rate between 600 and 4000 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large-scale systems.

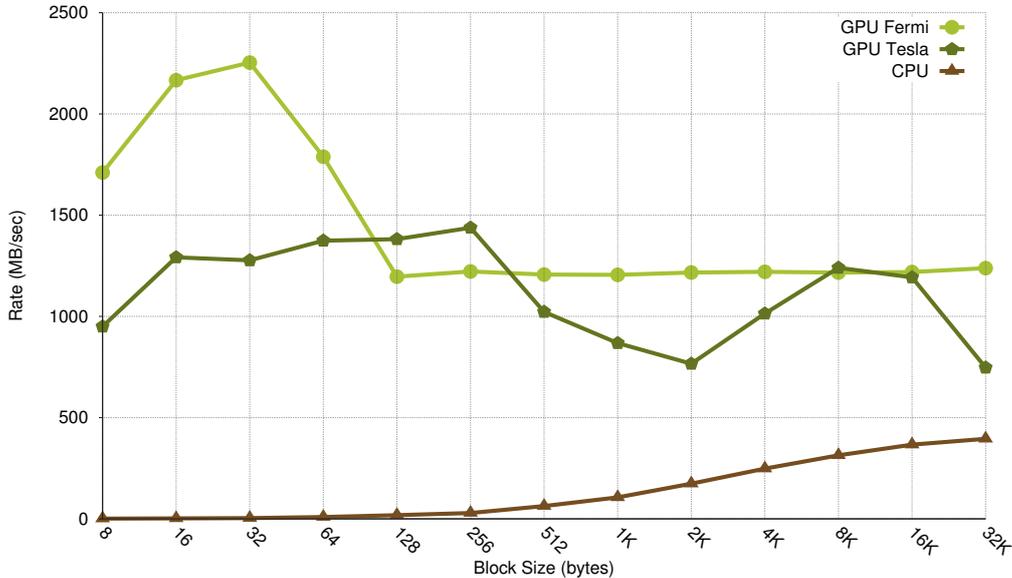


Figure 6: A comparison of CRC32 hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [30] CRC32 hashing algorithm.

*SHA256*. Lastly, Figure 9 compares GPU vs CPU performance of an *SHA256* calculation for varying block sizes. With a per-process hashing rate between 1400 and 2200 MB/sec for the Fermi GPU card, the GPU-based data rates exceed the per-process commit rate to stable storage for many large-scale systems.

## 6. Viability of Hash-Based Incremental Checkpointing

In this section we outline the viability of this hash-based technique for next generation extreme-scale systems. These viability bandwidths provide guidance for when we would want to use this technique. For commit bandwidth greater than these viability bandwidths we would chose traditional checkpoint/restart. For bandwidths less than these viability bandwidths (which we will see to be the majority of the design space), we use this hash-based technique.

Table 3 summarizes the *compression* results shown previously in this paper. For CTH, SAGE, and LAMMPS we use Equation 3, the *compression* values measured in Section 4.2. In addition we use the maximum hash computation rate ( $\beta_{hash}$ ) measured in Section 5. These values are 4.0 GB/sec from an MD5 hash on a Fermi GPU and a value of 500 MB/sec for the CPU hashing algorithm.

Table 3 shows the per-process break-even checkpoint commit bandwidths for the measured applications using the maximum hashing rate and compression percentages. If a

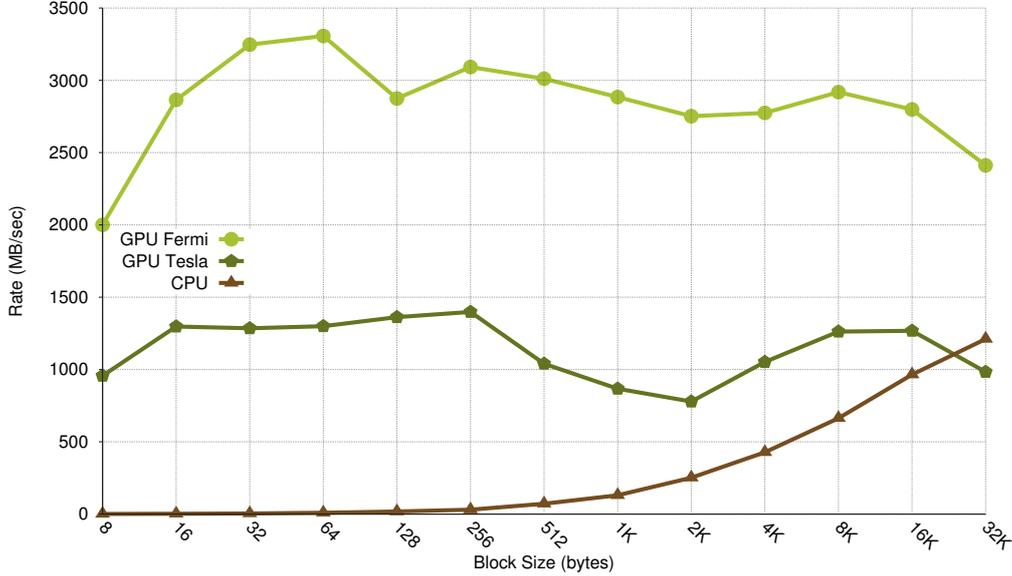


Figure 7: A comparison of ADLER32 hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [30] ADLER32 hashing algorithm.

| Application | $\alpha$ (%) | GPU Break-even $\beta_{ckpt}$ (MB/sec) | CPU Break-even $\beta_{ckpt}$ (MB/sec) |
|-------------|--------------|--|--|
| CTH         | 83           | 3320                                   | 415                                    |
| SAGE        | 35           | 1400                                   | 175                                    |
| LAMMPS      | 2.4          | 92                                     | 12                                     |

Table 3: Per-process checkpoint commit “break-even” bandwidth CPU/GPU comparison calculated using Equation 3 for the applications CTH, SAGE, and LAMMPS. *Compression* values for each of the applications are from Section 4.2 and a  $\beta_{hash}$  value equal to 4.0GB/sec from the GPU MD5 hash as illustrated in Section 5, and a  $\beta_{hash}$  value equal to 500MB/sec from the CPU ADLER32 hash.

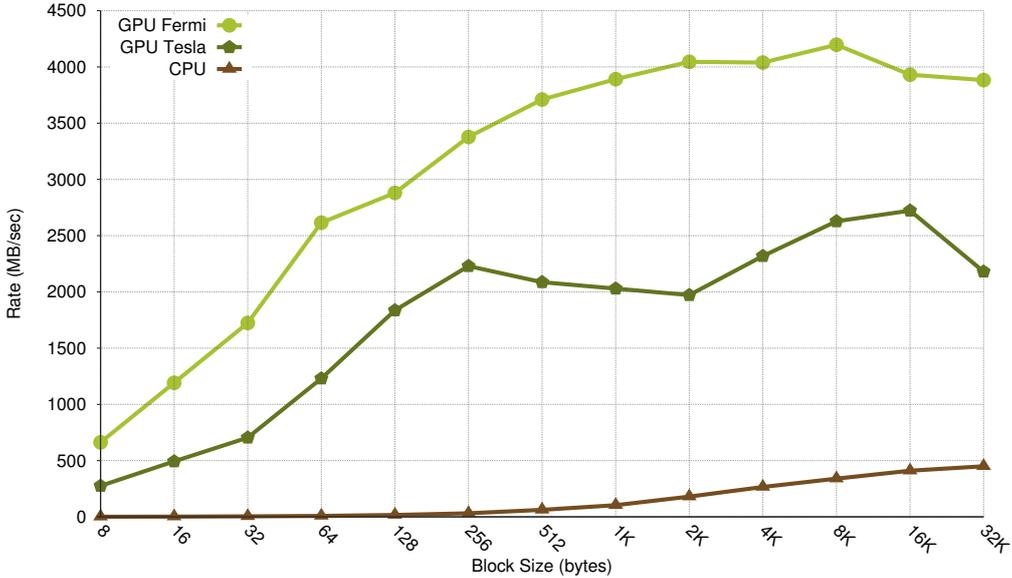


Figure 8: A comparison of MD5 hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [30] MD5 hashing algorithm.

proposed exascale-class machine has a per-process checkpoint commit speed is less than this break-even value, then the hash-based approach has a lower overhead than a strictly page-based approach.

The per-process break-even CPU results in this table vary from 415 to 12 MB/sec, with the greatest per-process bandwidth being for CTH which has the greatest compression and lowest for LAMMPS. For the GPU, if a machine has a per-process checkpoint commit speed is less than 3.32 GB/sec 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 future large-scale systems as these values for the GPU are larger than what we even see today.

For illustration, in Figure 10 we relate these results to a platform's aggregate checkpoint break-even bandwidth for a range of socket counts and measured compression factors measured with the library. The shaded region in this figure corresponds to possible socket count for an exascale class machine [33]. In this figure we use the optimal GPU and CPU hash bandwidths measured in this work (4GB/sec and 500MB/sec respectively). This aggregate break-even bandwidth is derived using Equation 3 and set  $\beta_{ckpt} = \frac{\beta_{agg}}{N}$ , where  $\beta_{agg}$  is the aggregate bandwidth of the system and  $N$  is the number of sockets. Putting these two equations together we get the following equation for the aggregate checkpoint break-even bandwidth.

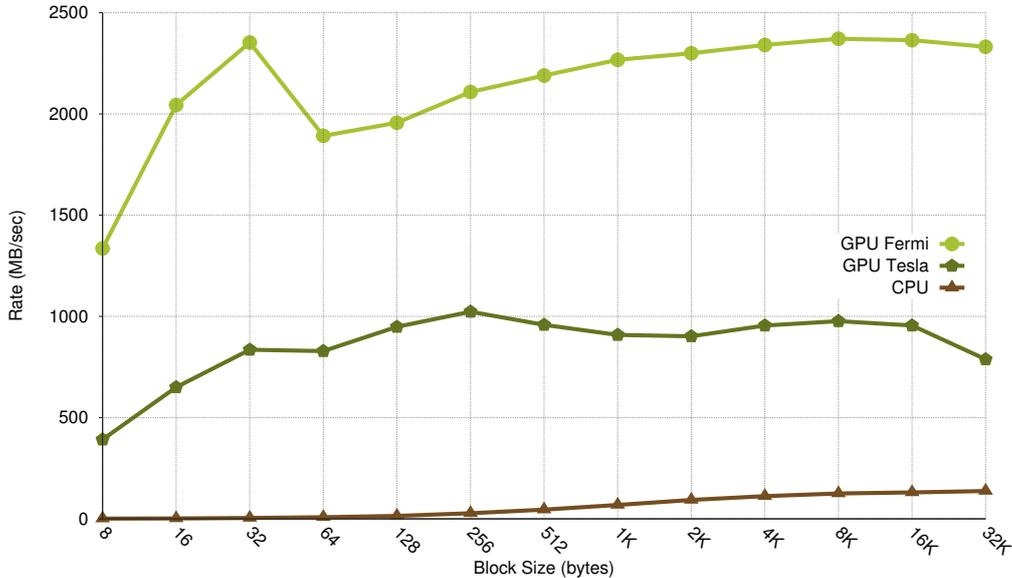


Figure 9: A comparison of SHA256 hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [30] SHA256 hashing algorithm.

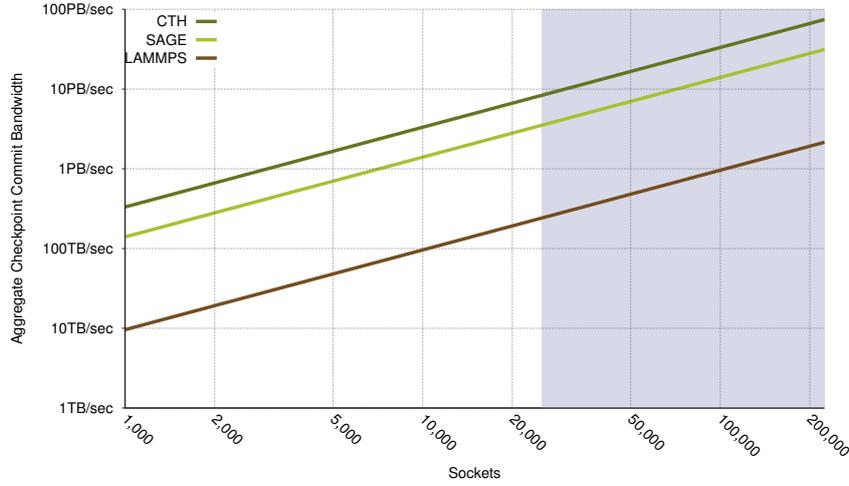
$$\beta_{agg} < n \cdot \alpha \cdot \beta_{hash} \quad (7)$$

From the figure we see that a platforms aggregate checkpoint commit rate less than the value this break-even values means the hash based approach outlined in the paper will use less resources than that of a traditional coordinated checkpointing scheme.

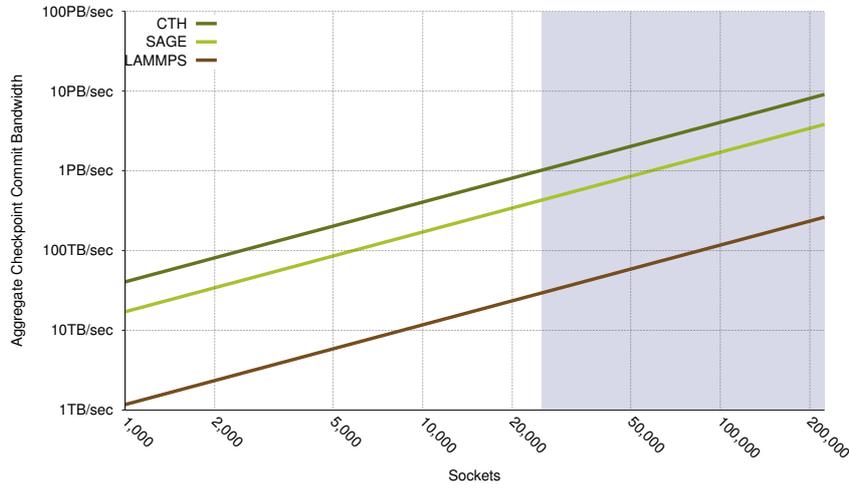
From Figure 10(a), the GPU results show that a systems checkpoint commit rate must exceed 1PB/sec ( $10^{15}$  bytes) for the majority of the design space for traditional checkpointing to perform better than this hash-based approach. The CPU results in Figure 10(b) show a slightly different story. Due to the significantly lower hashing rates of the CPU, the breakeven bandwidth is an order of magnitude lower than the GPU results and well within the reach in the case for LAMMPS is local non-volatile storage is used.

## 7. Impact on Application Efficiency

In the previous section we show where in the extreme-scale design space this hash-based technique is a viable approach (i.e. what aggregate checkpoint bandwidth). This however is not the whole story for this method. While the viability model tells you when this approach should be used it does not, however, tell you the impact this method will have on application runtime. To evaluate the impact on performance we will use the difference in applications efficiency as our metric. This efficiency metric is defined as the time to solution in the failure



(a) GPU



(b) CPU

Figure 10: Aggregate commit breakeven bandwidths using Equation 7 for a number of possible socket counts. In the figure we use the optimal GPU hashing rate (4GB/sec) and optimal CPU rates (500MB/sec) and the compression factors measured with the library. The shaded regions corresponds to the possible socket counts range expected for an exascale class machine [33].

environment divided by the time to solution in a failure-free environment. This difference in efficiency metric we use for the remainder of this section is defined as the efficiency with this hash-based approach minus the efficiency of a traditional rollback/recovery to stable storage.

To outline application efficiency, we created a performance model for expected time to solution for an application using checkpoint/restart. This model is based on Daly's higher order model [34], which assumes node failures are independent and exponentially distributed. The model takes as input the mean time between failures (MTBF) for the system, the checkpoint commit time, the checkpoint restart time, the number of nodes used

in the application and the time the application would take to complete in a failure-free environment.

We modified this model to integrate this hash-based incremental checkpointing approach. For the checkpoint commit time we included the time to hash the checkpoint image as well as the time to write the changed blocks from that image to stable storage. Regarding file I/O rates to stable storage, we use a report based on a study of I/O performance on Argonne National Laboratory’s 557 TFlop Blue Gene/P system (Intrepid) [32] to select I/O rates for our model. This work executes an I/O scaling study measuring maximum achieved throughput for carefully selected read and write patterns. From this report, the best observable per process I/O bandwidths 1 MB/s for both reading and writing. This performance scales to about 32,768 processes and then decreases. For example, at 131,072 processes, per process read bandwidth is 385 KB/s and per process write bandwidth is 328 KB/s. At any rate, for our study, we optimistically choose the best observed per process I/O bandwidth of 1 MB/s.

In Figure 11, we show the result of this model for the values measured previously in this work for LAMMPS, SAGE, and CTH. Again in these figures we show the increase in application efficiency. Therefore a efficiency difference of 40 means that, with this method, the efficiency increases 40 percentage points (for example, from 20% to 60%). In each of these figure we use the hash rates and percentage changed memory measured earlier in this work. In addition, this model assumes a five year socket MTBF as has been measured in current studies [35]. Finally, the shaded region in these figures corresponds to the possible socket count range expected for an exascale class machine [33].

From these figures we note two things; one, the efficiency of this method is independent of whether the hashing is done on the CPU or GPU, and, two, this efficiency is dependant only on the amount of data that has changed, a property that varies greatly across applications. Therefore, the decision whether to perform hashing on CPU or GPU may be based on a power consumption argument or based on which component is free to perform the operation, rather than a performance-based argument. The fact that the hash speed has no impact on efficiency is due to the fact that at these scales, performance is dominated by the commit and recovery times and not the hash times. Therefore, for applications like LAMMPS, where much of the memory written to changes in an interval, this method has little to no performance impact. CTH, on the other shows a dramatic and positive influence on performance.

### 7.1. Impact of Changed Memory

In the previous section we showed that the hash speed is not critical to application performance. We showed that what is key to performance is the amount of changed memory. Therefore, to conclude this section, we model the performance impact of this changed memory on application performance. Similar to the previous section, we use the same efficiency difference defined in the previous section. Additionally, we use the same application model described previously.

In Figure 12, we show the result of this model varying the amount of changed memory. The X-axis in this figure is the amount of memory unchanged in the checkpoint interval.

Therefore, 0% of memory unchanged means that all memory has changed. For this model, we again assume a 5 year socket MTBF, the hashing viability bandwidth measured with CTH (although this speed does not matter), and a socket count of 100,000 – a number we expect to see of future extreme-scale systems.

From this figure we can more clearly see that the increase in unchanged memory can have a dramatic and positive influence on the performance of this approach in comparison to a traditional disk-based rollback/recovery approach.

## 8. Related Work

Checkpoint optimizations generally fall within one of two strategies; the first hide or reduce the perceived checkpoint commit latencies without actually reducing the data that is committed. These strategies include *concurrent checkpointing* [36, 37], *diskless* or *multi-level checkpointing* [38, 39, 40, 41, 42, 43, 44, 6], *remote checkpointing* [45, 46, 44] and checkpointing filesystems [47]. The second set of strategies reduce commit latencies by reducing checkpoint sizes. These strategies include *memory exclusion* [48] and *incremental checkpointing* [2, 49, 50, 51, 52, 1, 53, 3, 54, 55, 4]. In this work we focus on the most closely related work, incremental checkpointing and multi-level checkpointing using dedicated resources for fault-tolerance resources. See [2] for a more complete study of methods outlined above.

Incremental checkpointing decreases the overhead of taking a checkpoint by reducing the amount of application state or data saved to stable storage at each checkpoint. Incremental checkpointing reduces the amount of state saved by only saving that state which has changed since that last checkpoint has been written. A variety of methods have been used to determine which state has changed, from compiler based [49] to techniques based on saving dirty virtual memory pages [52, 51].

Hash-based Incremental Checkpointing, sometimes referred to as *probabilistic checkpointing* [55], is a system-based checkpoint method that attempts to minimize the state saved in a checkpoint and therefore optimize checkpoint commit times. This technique uses computational hash algorithms to determine the portions of a process' address space that has changed in a checkpoint interval, rather than the dirtied pages used in standard incremental checkpointing. Another key feature of this method is the ability to allow finer-grained detection of dirtied blocks than is currently possible using mechanisms based solely on page protection mechanisms. This approach has previously been dismissed as being too computationally expensive [22, 56] to reap the meager benefits in state compression.

With a probabilistic 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 [22]. Secure hash

signatures like MD5 and SHA256, however, have been shown to behave in practice as expected theoretically, and are therefore reliable enough to be used in a hash-based approach [56].

Recently, Agarwal et al. [50] investigated the performance characteristics of a hash-based adaptive incremental checkpointing library. The authors use an MD5 hash to determine the portions of an application address space that have changed in a checkpoint interval. This work failed to evaluate the merit of this hash-based technique on actual HPC capability workloads, instead using micro-benchmarks. In addition, the authors failed to evaluate the merit of this technique compared to application-specific checkpoint mechanisms that exist in many capability workloads.

Multi-level checkpointing, such as SCR, [44] are library-based approaches for controlling checkpointing to multiple storage targets, including memory-based checkpoints, checkpointing to local storage, and remote checkpoints, into a single system. Because of this, they share some of the advantages and disadvantages of memory-based checkpointing and local storage techniques. Unlike these techniques, however, multi-level checkpointing has the flexibility to choose between multiple levels of storage based on system design parameters, making it a promising technique for exascale systems.

Hashing and various erasure codes, for example Reed-Solomon encoding, can be used at various levels in these multi-level schemes. Recently, a low-overhead multi-level checkpointing technique has been investigated for hybrid systems, such as those with GPUs [6]. This technique is essentially identical to SCR except for a Reed-Solomon encoding being used rather than XOR, thereby increasing the number of simultaneous failures that can be tolerated. In contrast to work described in this paper, the authors perform all application processing on the GPUs and have the CPUs perform only communication for the application and encoding of checkpoints. As we show in this paper, the GPUs can generally perform this work much faster than the CPU, even including the time to copy data between the separate memory space. In addition, not all extreme-scale HPC applications are well suited to computing only on the GPUs, therefore the costs will be much higher when the application computation and checkpointing mechanisms must share the CPU. For those applications which can effectively utilize on-node GPUs, a technique similar to [6] can be used – where the application runs on the GPU and the hashing mechanism can be run on CPU.

## 9. Conclusions

In this paper, we use a series of simple models to illustrate the viability of hash-based incremental checkpointing and its increase in application efficiency. In addition, we use a previously published library `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 and therefore increase the number of platforms this technique will be viable, `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 ap-

proach. In addition, this method can create checkpoint files which are only 35% larger than that of a manually-coded, application-specific method. We use the model and results from real applications to outline the viability of this technique for next-generation exascale systems, comparing against both a traditional and strictly page-based approach. Additionally we show the viability of this hash-based incremental checkpointing using both the GPU and CPU to compute the hashes. More specifically, we show that at GPU hashing speeds this technique has significantly lower overheads and significant increase in application efficiency in much of the exascale design space than comparative approaches.

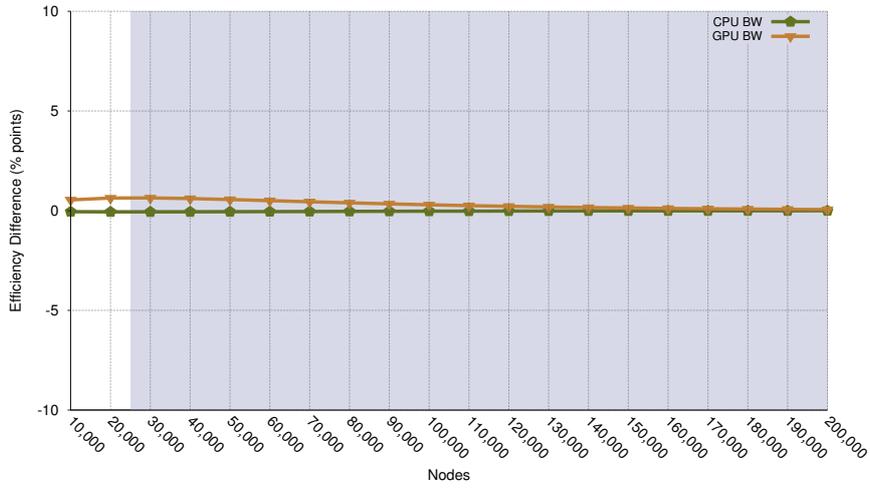
## References

- [1] 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). Supercomputing '97; New York, NY, USA: ACM. ISBN 0-89791-985-8; 1997, p. 1–11. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/509593.509626>}. URL <http://doi.acm.org/10.1145/509593.509626>.
- [2] Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput Surv* 2002;34(3):375–408. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/568522.568525>}.
- [3] Plank, J.S., Beck, M., Kingsley, G., Li, K.. Libckpt: transparent checkpointing under unix. In: Proceedings of the USENIX 1995 Technical Conference Proceedings. TCON'95; Berkeley, CA, USA: USENIX Association; 1995, p. 18–18. URL <http://portal.acm.org/citation.cfm?id=1267411.1267429>.
- [4] Ferreira, K.B., Riesen, R., Brightwell, R., Bridges, P.G., Arnold, D.. Libhashckpt: Hash-based incremental checkpointing using GPUs. In: Proceedings of the 18th EuroMPI Conference. Santorini, Greece; 2011.
- [5] Plank, J.S., Li, K.. ickp: A consistent checkpointer for multicomputers. *Parallel & Distributed Technology: Systems & Applications*, IEEE 1994;2(2):62–67.
- [6] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.. FTI: high performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC'11; New York, NY, USA: ACM. ISBN 978-1-4503-0771-0; 2011, p. 32:1–32:32. doi:\bibinfo{doi}{[10.1145/2063384.2063427](http://doi.acm.org/10.1145/2063384.2063427)}. URL <http://doi.acm.org/10.1145/2063384.2063427>.
- [7] Libckpt web page. 2011. URL <http://web.eecs.utk.edu/~plank/plank/www/libckpt.html>.
- [8] Deutsch, P., Gailly, J.L.. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational); 1996. URL <http://www.ietf.org/rfc/rfc1950.txt>.
- [9] Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.. Handbook of Applied Cryptography. Boca Raton, FL, USA: CRC Press, Inc.; 1st ed.; 1996. ISBN 0849385237.
- [10] Rivest, R.. The MD5 Message-Digest Algorithm. RFC 1321 (Informational); 1992. Updated by RFC 6151; URL <http://www.ietf.org/rfc/rfc1321.txt>.
- [11] Housley, R.. A 224-bit One-way Hash Function: SHA-224. RFC 3874 (Informational); 2004. URL <http://www.ietf.org/rfc/rfc3874.txt>.
- [12] Dierks, T., Rescorla, E.. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard); 2008. Updated by RFCs 5746, 5878, 6176; URL <http://www.ietf.org/rfc/rfc5246.txt>.
- [13] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.. OpenPGP Message Format. RFC 4880 (Proposed Standard); 2007. Updated by RFC 5581; URL <http://www.ietf.org/rfc/rfc4880.txt>.
- [14] Ylonen, T., Lonvick, C.. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard); 2006. URL <http://www.ietf.org/rfc/rfc4253.txt>.
- [15] Ramsdell, B.. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specifi-

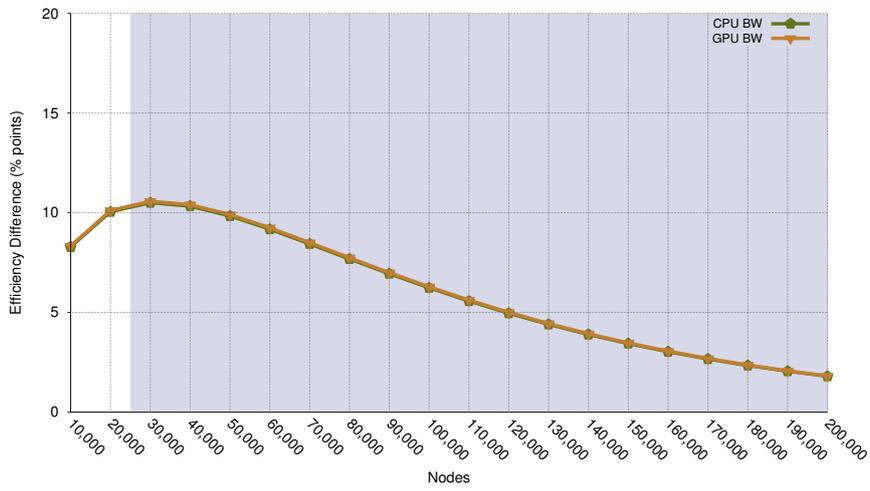
- cation. RFC 3851 (Proposed Standard); 2004. Obsoleted by RFC 5751; URL <http://www.ietf.org/rfc/rfc3851.txt>.
- [16] Manral, V.. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4835 (Proposed Standard); 2007. URL <http://www.ietf.org/rfc/rfc4835.txt>.
- [17] Mathis, F.H.. A generalized birthday problem. *SIAM Review* 1991;33(2):265–270.
- [18] Holst, L.. The general birthday problem. In: *Random Graphs 93: Proceedings of the sixth international seminar on Random graphs and probabilistic methods in combinatorics and computer science*. New York, NY, USA: John Wiley & Sons, Inc.; 1995, p. 201–208.
- [19] Knuth, D.E.. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.; 1998. ISBN 0-201-89685-0.
- [20] Dell, T.J.. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division; 1997.
- [21] Geist, A.. What is the monster in the closet?; 2011. Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in Our Thinking.
- [22] Elnozahy, E.N.. How safe is probabilistic checkpointing? In: *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing. FTCS '98*; Washington, DC, USA: IEEE Computer Society. ISBN 0-8186-8470-4; 1998, p. 358–. URL <http://portal.acm.org/citation.cfm?id=795671.796882>.
- [23] E. S. Hertel, J., Bell, R.L., Elrick, M.G., Farnsworth, A.V., Kerley, G.I., McGlaun, J.M., et al. CTH: A software family for multi-dimensional shock physics analysis. In: *Proceedings of the 19th International Symposium on Shock Waves*. 1993, p. 377–382.
- [24] Plimpton, S.J.. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys* 1995;117(1):1–19.
- [25] Sandia National Laboratories, . The LAMMPS molecular dynamics simulator. 2010. URL <http://lammmps.sandia.gov>.
- [26] Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.. Predictive performance and scalability modeling of a large-scale application. In: *Proceedings of the ACM/IEEE conference on Supercomputing*. ISBN 1-58113-293-X; 2001, p. 37–48. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/582034.582071>}.
- [27] Sandia National Laboratories, . The mantevo project home page. 2010. URL <https://software.sandia.gov/mantevo>.
- [28] 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; 2002,.
- [29] Ferreira, K.B.. Keeping checkpoint/restart viable for exascale systems. Ph.D. thesis; University of New Mexico, Department of Computer Science; 2011.
- [30] libgcrypt web page. 2010. URL <http://directory.fsf.org/project/libgcrypt/>.
- [31] Nickolls, J., Buck, I., Garland, M., Skadron, K.. Scalable parallel programming with CUDA. *Queue* 2008;6:40–53. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/1365490.1365500>}. URL <http://doi.acm.org/10.1145/1365490.1365500>.
- [32] Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.. I/O performance challenges at leadership scale. In: *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ISBN 978-1-60558-744-8; 2009, p. 40:1–40:12. doi:\bibinfo{doi}{[10.1145/1654059.1654100](http://doi.acm.org/10.1145/1654059.1654100)}. URL <http://doi.acm.org/10.1145/1654059.1654100>.
- [33] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., et al. Exascale computing study: Technology challenges in achieving exascale systems. [http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware\(2008\).pdf](http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf); 2008.
- [34] Daly, J.T.. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener Comput Syst* 2006;22(3):303–312. doi:\bibinfo{doi}{<http://dx.doi.org/10.1016/j.future.2004.11.016>}.
- [35] Schroeder, B., Gibson, G.A.. Understanding failures in petascale computers. *Journal of Physics:*

- Conference Series 2007;78(1):012022.
- [36] Pan, D.Z., Linton, M.A.. Supporting reverse execution for parallel programs. In: 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88). Madison, WI: ACM Press. ISBN 0-89791-296-9; 1988, p. 124–129. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/68210.69227>}.
  - [37] Li, K., Naughton, J.F., Plank, J.S.. Real-time concurrent checkpoint for parallel programs. In: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, PPOPP '90; New York, NY, USA: ACM. ISBN 0-89791-350-7; 1990, p. 79–88. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/99163.99173>}. URL <http://doi.acm.org/10.1145/99163.99173>.
  - [38] Plank, J.S., Kim, Y.B., Dongarra, J.J.. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers. Pasadena, CA, USA: Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1995; 1995, p. 351–360.
  - [39] Plank, J.S., Kim, Y., Dongarra, J.J.. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. J Parallel Distrib Comput 1997;43:125–138. doi:\bibinfo{doi}{<http://dx.doi.org/10.1006/jpdc.1997.1336>}. URL <http://dx.doi.org/10.1006/jpdc.1997.1336>.
  - [40] Silva, L.M., Silva, J.G.. An experimental study about diskless checkpointing. In: 24th EUROMICRO Conference. Vasteras, Sweden: IEEE Computer Society Press; 1998, p. 395 – 402.
  - [41] Engelmann, C., Geist, A.. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform. In: Proceedings of the 1st International Workshop on Challenges of Large Applications in Distributed Environments. CLADE '03; Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-1984-9; 2003, p. 47–. URL <http://portal.acm.org/citation.cfm?id=792760.793177>.
  - [42] Dong, X., Muralimanohar, N., Jouppi, N., Kaufmann, R., Xie, Y.. Leveraging 3d pcam technologies to reduce checkpoint overhead for future exascale systems. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. New York, NY, USA: ACM. ISBN 978-1-60558-744-8; 2009, p. 1–12. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/1654059.1654117>}.
  - [43] Vaidya, N.H.. A case for two-level distributed recovery schemes. In: ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '95/PERFORMANCE '95; New York, NY, USA: ACM. ISBN 0-89791-695-6; 1995, p. 64–73. doi:\bibinfo{doi}{[10.1145/223587.223596](http://doi.acm.org/10.1145/223587.223596)}. URL <http://doi.acm.org/10.1145/223587.223596>.
  - [44] Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.. 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'10; Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4244-7559-9; 2010, p. 1–11. doi:\bibinfo{doi}{<http://dx.doi.org/10.1109/SC.2010.18>}. URL <http://dx.doi.org/10.1109/SC.2010.18>.
  - [45] Stellner, G.. CoCheck: Checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium. IPPS '96; Washington, DC, USA: IEEE Computer Society. ISBN 0-8186-7255-2; 1996, p. 526–531. URL <http://portal.acm.org/citation.cfm?id=645606.660853>.
  - [46] Zandy, V.C., Miller, B.P., Livny, M.. Process hijacking. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing. HPDC '99; Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-0287-3; 1999, p. 32–. URL <http://portal.acm.org/citation.cfm?id=822084.823234>.
  - [47] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., et al. PLFS: a checkpoint filesystem for parallel applications. In: Conference on High Performance Computing Networking, Storage and Analysis (SC '09). ISBN 978-1-60558-744-8; 2009, p. 21:1–21:12. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/1654059.1654081>}. URL <http://doi.acm.org/10.1145/1654059.1654081>.
  - [48] Plank, J.S., Chen, Y., Li, K., Beck, M., Kingsley, G.. Memory exclusion: optimizing the performance of checkpointing systems. Softw Pract Exper 1999;29:125–142. doi:\bibinfo{doi}{[10.1002/\(SICI\)1097-024X\(199902\)29:2<125::AID-SPE224>3.0.CO;2-7](http://portal.acm.org/citation.cfm?id=309087.309095)}. URL <http://portal.acm.org/citation.cfm?id=309087.309095>.

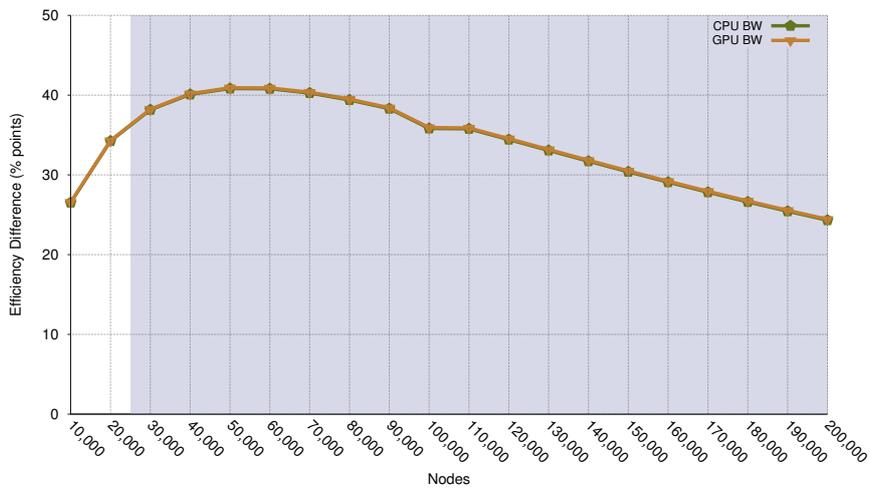
- [49] Bronevetsky, G., Marques, D.J., Pingali, K.K., Rugina, R., McKee, S.A.. Compiler-enhanced incremental checkpointing for openmp applications. In: PPOPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. New York, NY, USA: ACM. ISBN 978-1-59593-795-7; 2008, p. 275–276. doi:\bibinfo{doi}{<http://doi.acm.org/10.1145/1345206.1345253>}.
- [50] 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,.
- [51] 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 Supercomputing. SC '05; Washington, DC, USA: IEEE Computer Society. ISBN 1-59593-061-2; 2005, p. 9–. doi:\bibinfo{doi}{<http://dx.doi.org/10.1109/SC.2005.76>}. URL <http://dx.doi.org/10.1109/SC.2005.76>.
- [52] Sancho, J.C., Petrini, F., Johnson, G., Fernandez, J., Frachtenberg, E.. On the feasibility of incremental checkpointing for scientific computing. Parallel and Distributed Processing Symposium, International 2004;1:58b. doi:\bibinfo{doi}{<http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1302982>}.
- [53] Li, K., Naughton, J.F., Plank, J.S.. Low-latency, concurrent checkpointing for parallel programs. IEEE Trans Parallel Distrib Syst 1994;5:874–879. doi:\bibinfo{doi}{<http://dx.doi.org/10.1109/71.298215>}. URL <http://dx.doi.org/10.1109/71.298215>.
- [54] Elnozahy, E.N., Johnson, D.B., Zwaenepoel, W.. The performance of consistent checkpointing. In: 11th Symposium on Reliable Distributed Systems. Houston, TX, USA: IEEE Computer Society Press; 1992, p. 39–47.
- [55] 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. 1997, p. 48–57. doi:\bibinfo{doi}{10.1109/FTCS.1997.614077}.
- [56] 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. 2001, p. 49–56. doi:\bibinfo{doi}{10.1109/PRDC.2001.992679}.



(a) LAMMPS



(b) SAGE



(c) CTH

Figure 11: Efficiency increase for LAMMPS, SAGE, and CTH. These figures use a variant of Daly's equation [34] and the associated maximum viability bandwidths, amount of memory changed, and a socket MTBF of 5 years [35] for each of the applications. The efficiency difference used in this figure is defined as the efficiency of the hash-based approach (efficiency defined as the percentage of useful work) minus the efficiency of strictly traditional checkpoint/restart. The shaded regions corresponds to the possible socket counts range expected for an exascale class machine [33].

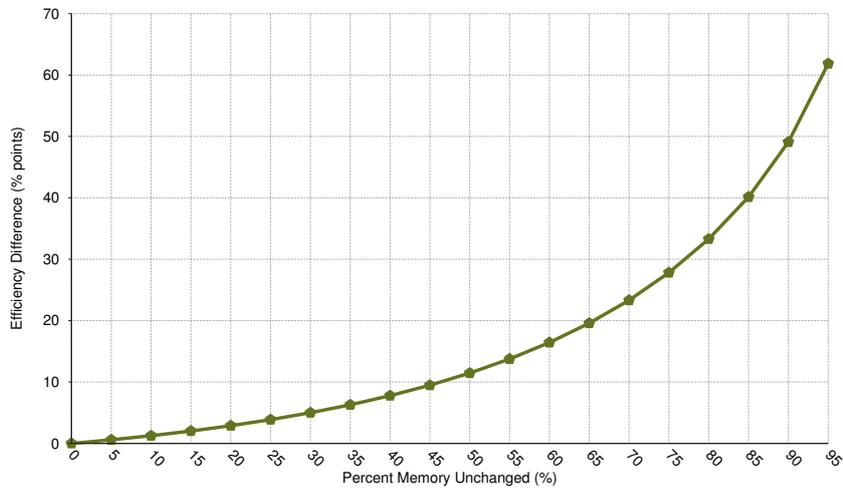


Figure 12: Efficiency increase for a 100,000 socket application based on the percentage of application memory that remains unchanged. This figure uses a variant of Daly’s equation [34], the maximum viability bandwidth from testing with CTH, and a socket MTBF of 5 years [35]. The efficiency difference used in this figure is defined as the efficiency of the hash-based approach minus the efficiency of the strictly traditional checkpoint/restart approach. The shaded region of this figure corresponds to the possible socket count range expected for an exascale class system [33]