SAND2013-9997C

Checkpoint Compression: Its Limitations and Comparisons with other Optimizations

Dewan Ibtesham and Dorian Arnold
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
{dewan,darnold}@cs.unm.edu

Kurt B. Ferreira
Scalable System Software Department
Sandia National Laboratories
Albuquerque, NM 87185–1319
{kbferre}@sandia.gov

***Abstract*—Identifying checkpoint size as a significant factor in checkpoint/restart performance, in this work we evaluate the limits of a compression-based checkpointing technique and put the performance of this method into context by comparing against other hardware- and software-based techniques. Using a model-based approach, we explore the feasibility and potential impacts of additional compression and speed improvements to compression algorithms and show that such enhancements may not lead to performance improvements. We also compare compression against a number of software and hardware approaches, showing that it can outperform incremental checkpointing but that the greatest checkpoint performance is realized when both techniques are used together. This combination can perform within 20% of an optimal hardware-based approach without requiring additional costly non-volatile local storage and can outperform hardware-based solutions in terms of work done per dollar spent.**

***Keywords*-Fault tolerance; Checkpoint Compression;**

I. Introduction

In high-performance computing (HPC), checkpoint/restart is the prevailing approach for application fault-tolerance. Today, the largest HPC systems comprise of millions of cores and tens of millions components. As we approach the exascale computing era, we expect orders of magnitude increases in core counts. In these extremely large-scale environments, a confluence of issues including decreased mean time between failures (MTBF), increased I/O pressures and increased checkpoint/restart overheads have raised concerns about the continued viability of checkpoint/restart-based fault tolerance [1], [2].

Researchers have shown data compression to be an effective method to decrease checkpoint/restart overheads [3], [4]. In this work, we seek a better understanding of checkpoint compression and the performance impacts of compression factor, compression speed, and I/O commit rate. In addition, we place the performance of this technique in context with other checkpoint optimizations. In particular, we wish to understand: (1) whether checkpoint-data-specific compression algorithms offer better compression than standard text-based utilities, (2) whether faster running compression algorithms offer better overall application performance, (3) how checkpoint compression performs when used in conjunction with other checkpoint optimizations, (4) how checkpoint compression compares against other hardware and software-based optimizations, ,and (5) whether the best performing hardware/software checkpointing methods when considering performance from a time to solution or monetary

We attempt to answer these questions using current system parameters as a guide, while being conscience of how new potential technologies might change the impact of checkpoint compression going forward. In addition, we use information theory along with knowledge from an application-level checkpointing library to evaluate standard compression utilities. Using previously published performance data and a checkpointing model [3] based on Daly's higher order checkpointing model [5], we analyze the impact of compression speeds and compression performance. We then compare the results of this software compression solution with that of a number of state-of-the-art hardware and software checkpointing solutions. Finally, we use these results to evaluate checkpoint compression performance alongside other hardware-based checkpoint/restart optimization. Based on these studies we demonstrate that:

- There is not sufficient improvements possible to motivate checkpoint-specific compression utilities. Current text-based algorithms perform close to a theoretical optimal

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.

- Increasing compression speeds leads to marginal performance improvements on current systems.
- As checkpoint commit bandwidths increase improvement from checkpoint speed increases. However, even at a factor of 4k increase in commit bandwidth and 100 times faster compression speeds results in a less than 15% increase in application efficiency.
- Compression performs better than an optimal incremental checkpointing approach without any knowledge of the application data
- Composition of both compression and incremental checkpointing leads to further performance increases.
- Hardware-based checkpoint optimizations are more efficient that software-based methods.
- However, compression plus incremental checkpointing can perform within 20% of a state-of-the-art hardware/multi-level checkpointing techniques.
- Considering performance from a monetary perspective, compression plus incremental checkpointing is the most cost efficient method and produces the most amount of work per unit price as long as per-node procurement costs are kept low.

The organization of the rest of this paper is as follows: after a discussion of related checkpoint/restart research in the next section, we detail our methodology including our software chain and modeling techniques in Section III. In Section IV and Section V, we present our studies of the impact of compression factor and compression speed respectively. In Section VI, we compare checkpoint compression against other hardware and software based checkpoint optimizations and present the cost efficiency comparison of hardware and software based methods. We conclude in Section VII with a summary of our results and insights.

II. Background and Related Research

Checkpoint/restart protocols [6] periodically *commit* process state to stable storage devices. When failures occur, a new process can be *recovered* to the intermediate state captured in the failed process' most recent checkpoint – thereby reducing lost computation. Checkpoint optimizations can be classified broadly into two categories. The first category includes protocols that hide or reduce (perceived) checkpoint commit latencies without actually reducing the amount of data to commit. These strategies include *concurrent checkpointing* [7], [8], *disk less* or *multi-level checkpointing* [9]–[11], *remote checkpointing* [12], [13] and checkpointing filesystems [14].

The second strategy focuses on reducing the volume of checkpoint data. These strategies include *memory exclusion* [15], *incremental checkpointing* [16]–[21] and *checkpoint compression* [3], [4], [22]–[25]

Many recent studies, including this one, have focused on the later–volume reduction strategies for two main reasons. First, latency hiding strategies do not alleviate resource contentions. For I/O intensive applications, generally increasing pressures on I/O substrates make this a major issue. In fact, such contentions can make it hard to hide the perturbation that checkpoint commits can have on application performance. Furthermore, techniques like concurrent checkpointing are seldom used in practice due to the complexity and required applications modifications.

The second reason follows from the observation that checkpoint data volume appears to be one of the few relevant factors over which users have direct control. A first order approximation of the optimal checkpoint interval is:

$$\sqrt{2 * MTBF_{sys} * lat_{commit}},$$

where $MTBF_{sys}$ is the system mean time between failures, and lat_{commit} is the time to commit a checkpoint [26]. Therefore, $\sqrt{\frac{lat_{commit}}{2*MTBF_{sys}}}$ approximates the fraction of system time lost due to checkpoints and restarts. $lat_{commit} = \frac{||ckpt||}{\beta_{ckpt}}$ [27] where $||ckpt||$ is size of checkpoint and β_{ckpt} is the checkpoint commit bandwidth. Generally, $MTBF_{sys}$ and β_{ckpt} are fixed and based on hardware technologies, leaving $||ckpt||$ to be our focal point.

III. Methodology and Tool Chain

To have a better understanding of the impacts of the different parameters of checkpoint compression namely compression factor, compression speed and I/O commit rate, we modeled checkpoint compression performance with hypothetical improvements to these parameters. Figure 1 depicts our approach for executing this study as well as the set of tools that we use. First, we collect checkpoint compression performance data using a set of applications, checkpoint libraries and compression utilities. Second, we use an extension of Daly's model for checkpoint/restart performance that integrates checkpoint compression to compute application performance based on our observed compression data. Last, we use the output from this performance model to compute the overall efficiency and answer the following scenarios

- The maximum possible improvement in application efficiency due to compression factor. Discussed in detail in Section IV.
- Impact on application efficiency due to faster or slower compression algorithm. Section V has the

details of our finding.

- Comparison of application efficiency and cost efficiency between our approach and other software-hardware based approach. We present our approach and results in Section VI.

In the rest of this section we describe our tool chain and the two models that we used to perform our study in detail.

A. Compression Performance Data Collection

The majority of our checkpoint compression performance data collection was performed as a part of a previous study [3] and used the following setup:

- **Applications**: We performed our experiments with a set of mini applications from the Mantevo Project [28] namely HPCCG, pHPCCG, phdMesh and miniFE along with LAMMPS [29], a key simulation workload for Department of Energy.
- **Checkpoint Libraries**: We used BLCR [30] as our system level checkpoint library to generate checkpoints at a small interval uniformly distributed over application runs. We also used LAMMPS' capability of generating checkpoints and generated checkpoints using this application-specific checkpointing method.
- **Compression Utilities**: We chose popular text compression tools from linux's software stack, for example: parallel bzip [31], bzip, zip [32], rzip [33], 7zip [34], etc.

The compression performance results we previously collected [3] are summarized in Table I.

B. Application Performance Model

Previously [3], we created a performance model based on Daly's higher order model [5] to project the time to solution for an application using checkpoint/restart and compression. Daly's model assumes node failures are independent and exponentially distributed and takes as input: the mean time between failures (MTBF) for

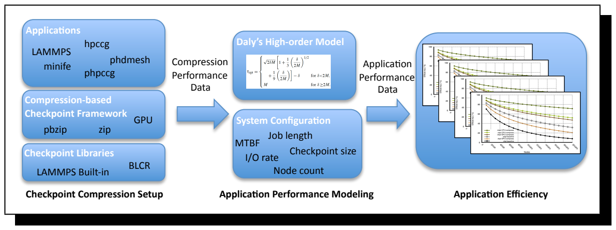

Fig. 1. Our Methodology: Empirically collected checkpoint compression data is input to an extension of Daly's Model. The results are used to compute application efficiency.

the system, the checkpoint commit time, the checkpoint restart time, the number of nodes used in the application and the time application would take to complete in a failure-free environment with no checkpointing . For the checkpoint commit time, this model includes the time to compress the checkpoint image as well as the time to write this compressed image to stable storage. Similarly, on restart we included the time to read the compressed checkpoint image and perform the decompression step.

We extended this model to integrate incremental checkpointing as well. As such, the model takes two additional parameters. The first new parameter specifies the size ratio of an incremental checkpoint to a full checkpoint. We assume that approximately the same fraction of the address space changes between each checkpoint. This assumption is based on the results of a previous incremental checkpointing study [21].

The second new parameter, the number of incremental checkpoints taken before taking the next full checkpoint, reflects the periodic desire to take full checkpoints. Increased recovery latencies and increased storage costs are two factors that motivate the desire for periodic full checkpoints. If an application fails and is recovered from the i^{th} incremental checkpoint after a full checkpoint, additional overhead is required to either coalesce the full checkpoint and the i increments or to recover the full checkpoint and iteratively recover the state in each increment. Incremental checkpointing necessarily increases storage costs since it requires maintaining a full checkpoint as well as subsequent increments. If each increment is on average $1/s$ the size of the full checkpoint, after s increments, storage costs would have doubled. We use Naksinehaboon et al's derivation of the optimal number of increments n between two full checkpoints as: $n = \left\lceil 4c/5r_{commit} - 1 \right\rceil$, where c is the size of a full checkpoint and r_{commit} is the rate a file can be committed to stable storage [35].

For simplicity we assume that taking incremental checkpoints and reconstructing a checkpoint from the increments will be free and not incur additional costs. There are a number of techniques, such as concurrent coalescing, that make this assumption possible.

C. Application Efficiency

Application efficiency is the ratio of an application's time to solution when the application is using some fault-tolerance mechanism to recover from failures as they occur to the application's time to solution assuming perfect conditions: no failures and, therefore, no need for any fault tolerance mechanisms. We use this simple calculation to compare the effectiveness of

Tool	App.	Factor	Comp./Decomp. Speed (MB/s)	Ckpt Bandwidth (MB/s)
pbzip2	HPCCG	91.55	23.74/396.06	41.01
pbzip2	PHPCCG	93.86	22.75/432.85	40.57
pbzip2	MINIFE	80.91	57.90/228.28	74.74
pbzip2	PhdMesh	86.16	84.56 307.00	114.25
pbzip2	LAMMPS	43.29	44.15/129.90	28.52
zip	HPCCG	86.21	61.77/151.58	75.67
zip	PHPCCG	88.49	67.98/156.63	83.90
zip	MINIFE	68.56	38.11/94.71	37.27
zip	PhdMesh	81.25	59.39/121.38	64.80
zip	LAMMPS	41.48	27.60/105.25	18.14

TABLE I
A SUMMARY OF OUR PREVIOUS CHECKPOINT COMPRESSION PERFORMANCE DATA.

different fault tolerance configurations: in the context of checkpoint/restart, the higher an application's efficiency, the greater the time spent executing the application's intended computation and the less the time spent taking checkpoints, recovering from failures or re-doing any lost computation due to failures.

IV. The Impact of Compression Factor

As we described in Section II, checkpoint data volume reduction is arguably the most significant user-controllable factor that impacts checkpoint-restart performance. Therefore, an important question is what are the limits of checkpoint data volume reduction via compression. A secondary related question is whether it is worth considering compression algorithms that specifically target checkpoint data. In this section, we provide novel insights into these questions by using information theory to theorize about the compression performance of off-the-shelf utilities and evaluate the additional impact of a custom algorithm that achieves optimal compression. In this discussion we will use the metric *compression factor* which is the inverse of the compression ratio, therefore higher compression factors are higher performing.

A. An Application-specific Case Study

Based on the data in Table I and our previous work [3], we focus on checkpoint/restart for the LAMMPS application. LAMMPS exhibits the poorest checkpoint compressibility and, hypothetically, the greatest opportunity for improvement for all the applications tested. We use knowledge of the LAMMPS on-disk checkpoint format to translate application-specific checkpoint data into its composite data elements. Using this, we compute the entropy of LAMMPS checkpoints using Shannon's information theory [36].

Shannon's theorem tells us the minimal number of bits needed to represent a certain amount of information. Using our understanding of the LAMMPS checkpoint format, we calculated a frequency distribution for the values in the checkpoint file. We calculated this distribution in a representation independent way; for example, the double 0.0 is interpreted to be the same value as the integer, 0, as they contain the same information. Using this frequency distribution, we then calculated the entropy of this newly created ”checkpoint language” for LAMMPS checkpoints. This entropy calculation gives us a minimal encoding.

Table II shows the results of this minimal checkpoint encoding. This checkpoint contained about 3.5 million total symbols of which about 1 million were unique, resulting in an entropy of 10.59 or a theoretically maximal compression factor of 79.5%. Comparatively, our bzip2-encoded strings for the same checkpoint (excluding the bzip2 dictionary and headers, as we do not include this information in the entropy calculation above) had a compression factor of 67.6%, a significant difference in compression performance . Therefore, a hypothetical optimal checkpoint compression algorithm tailored specifically for the information contained within it will compress the checkpoint to 20% of its original size, in comparison to bzip2, which compressed the checkpoint to 32%.

Next, we use this LAMMPS checkpoint compression comparison data to model how LAMMPS performance would improve with this optimal algorithm that could better compress its checkpoints. Optimistically, we keep compression speed constant, assuming that the optimal algorithm would take no longer to run than bzip2. We look at three different scenarios, systems with 10K, 50K and 100K total sockets. Figure 2 shows the impact on application efficiency as compression factor varies,

Total Symbols	Unique Symbols	Entropy	Optimal Compression Factor	Bzip Compression Factor
3,584,043	1,023,367	10.59	79.5%	67.6%

TABLE II
COMPARING A THEORETICAL MINIMAL ENCODING WITH BZIP2.

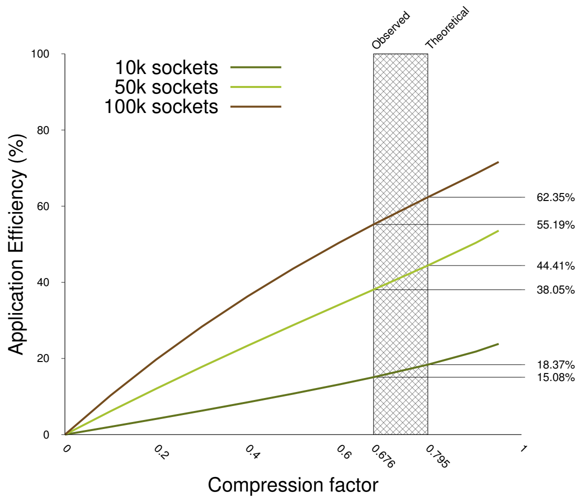

Fig. 2. Varying compression factor

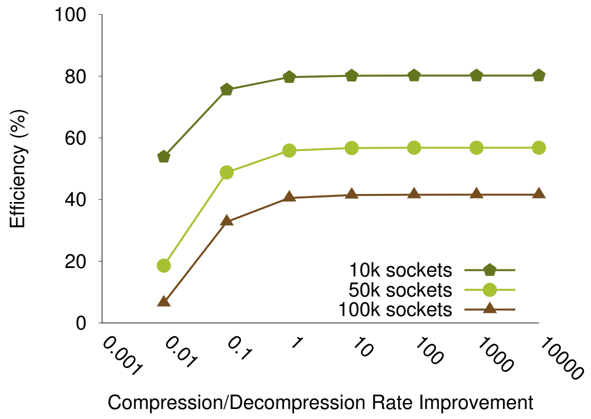

Fig. 3. Varying compression/decompression speed

highlighting our observed compression factor and our theoretic maximum compression factor. For each of the three scenarios, we observe that optimal compression would yield a relatively small increases in application efficiency – the largest being an additional 7.2% of efficiency in the 100K socket scenario. Therefore, we conclude that exploring checkpoint-specific compression algorithms is unlikely to yield significant improvement over standard text-based compression algorithms. In fact, with the expected growth of I/O on future systems, these differences in efficiencies will further decrease, supporting our position that current compression algorithms are sufficient for future systems as well.

V. Impact of Compression Speed

While compression factor likely is the biggest determinant of the performance impact of checkpoint compression, it is still prudent to understand the importance of compression speed. In this section, we evaluate the benefits of accelerating our top performing (in terms of compression factor) algorithm, either using algorithmic enhancements or hardware technologies, for example, GPUs.

To explore this question, we used our compression-enhanced application performance model described previously. Using the compression performance exhibited by pbzip2 on phpccg checkpoints (our top performer for compression factor) as a baseline, we varied compression and decompression rates in a range from a slow-down of 100X to a speed-up of 10,000X.

The results, shown in Figure 3, show that a four orders of magnitude improvement in speed would yield an insignificant improvement in application efficiency on current systems. While this is an important result, it is not so surprising: given current checkpoint commit rates (based on available per process I/O bandwidth to checkpoint storage), the time spent compressing a checkpoint is insignificant to the time spent committing the checkpoint to stable storage. What is unclear is the impact of compression speed increases with the expected I/O bandwidth increases expected in future systems.

Figure 4 shows the increase in application efficiency as a function of the per-node checkpoint commit bandwidth. Similar to previous work in this paper, we assume a 5 year socket MTBF and use optimal compression factors. The Y-axis in this is the difference in application efficiency in the accelerated and non-accelerated case. For the accelerated case, we assume a hypothetical compression of 100 times the CPU compression speeds. These optimal speedups have been observed with carefully crafted codes and workloads with GPUs [37]. We model these overheads for a number of node counts between 10k and 200k. From this figure, we see that a two

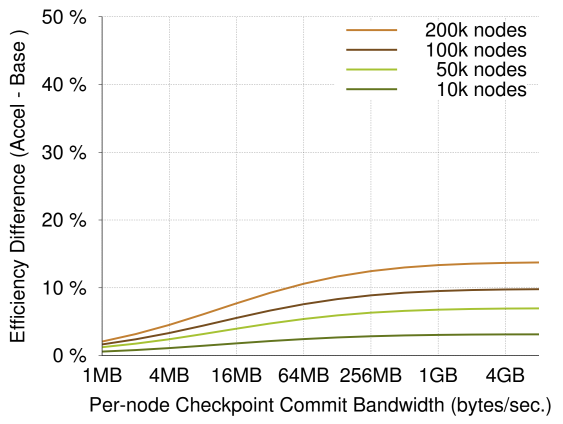

Fig. 4. Efficiency increase for a number of node counts as a function per-node checkpoint commit speeds assuming a compression/decompression speed a factor of 100 greater than what we see on current systems. The efficiency difference is defined as the accelerated efficiency minus the efficiency using current speeds

order magnitude increase in compression/decompression speeds lead to only marginal increases in application efficiency. This result suggests that the effort involved in accelerating compression.decompression speeds may not be worth the performance return.

VI. Compression and Other Optimizations

Lastly, we put the performance of checkpoint compression in context by comparing against a number of popular software, hardware, and mixed hardware/software solutions. Also, we investigate the performance of scenarios where checkpoint compression can be combined with these techniques. More specifically, we compare checkpoint compression performance against a software-only, incremental checkpointing solution, showing performance of the combination of both incremental checkpointing with compression. We then compare these software-only checkpointing solutions against state-of-the-art and considerably more costly hardware-based solutions: checkpointing to SSDs (solid-state device) and the multi-level checkpointing solution Scalable Checkpoint Restart (SCR) [11].

A. Increment and Compression-based Optimizations

First, we use our previously-collected compression performance data and application performance model from Section III-B to compare checkpoint methods and their impact on application efficiency. Based on observed workloads at Sandia National Laboratories, we again assume each process uses 2GB of memory and checkpoints $\frac{1}{3}$ of that memory [21], a five year node MTBF [38] and a per process I/O rate of 1 MB/s. This latter value was chosen optimistically from a performance study on Argonne National Laboratory's 557 TFlop Blue Gene/P system (Intrepid) [39].

Additionally, we use our best-performing compression scenario, pbzip on phdmesh, which showed an 86% compression factor, 84 MB/s compression rate and 307 MB/s decompression rate, and the optimal incremental checkpointing compression found in [21] (80% compression). An additional assumption in this work is that incremental checkpoints have similar compression ratios as the standard full checkpoints. This assumption has been validated using the incremental checkpointing library described in [21].

Figure 5 shows these scenarios compared with the baseline standard coordinated checkpointing and allows us to make several observations:

1) Perhaps unsurprisingly, all combinations of compression-based and increment-based optimizations outperform standard coordinated checkpointing (labeled "baseline" in the figure).
2) Compression yields greater application efficiency than pure, optimal incremental checkpoint (labeled "ickpt"). This result is more notable than it may first appear: our model does not include the potentially high-overhead of the mechanisms used in incremental checkpoints to detect updated memory regions or introspective application knowledge. So in environments where this overhead is prohibitively excessive or application characteristics unknown, checkpoint compression is a simple solution that can achieve better performance and with no programmer burden.
3) The combination of compression-based and increment-based optimizations yields the best performance of these software-only methods.

Therefore, checkpoint compression can dramatically improve application efficiency on large-scale machines. Most importantly, this method can be combined with other checkpoint optimizations to further improve application efficiency.

B. Hardware and Multi-level Checkpointing

Next, we compare our checkpoint compression technique against the performance of two hardware-based checkpoint optimizations. More specifically, we compare against a local SSD checkpointing solution [40] and a multi-level solution(SCR) that uses local and remote memory, SSDs, a parallel file system, and a software RAID to ensure reliability [11]. It is important to note that these hardware checkpointing solutions are considerably more expensive than a software only solution such

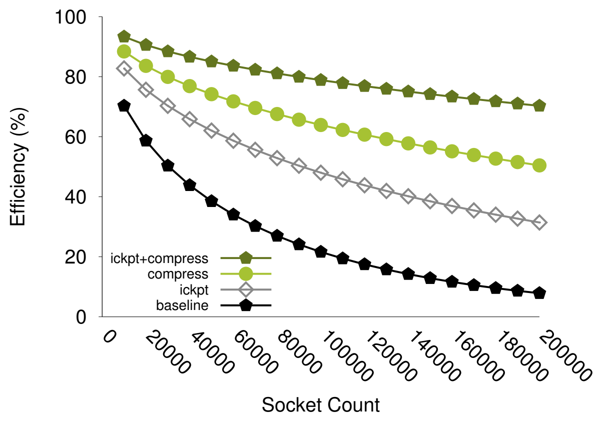

Fig. 5. Impact of the software-only optimizations checkpoint compression and incremental Checkpointing on application efficiency.

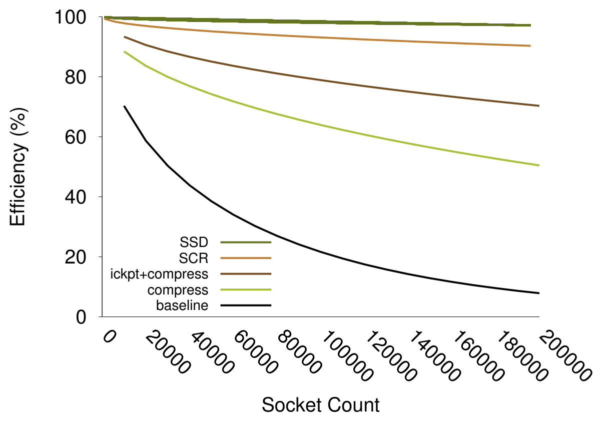

Fig. 6. Comparison of hardware/multi-level checkpointing techniques with pure software techniques like compression and incremental checkpointing

as incremental and compression-based checkpointing. In fact, the device reliability required for the SSD only solution maybe prohibitively expensive even at smaller scale as recent studies have shown that in 15% of failures, the checkpoint cannot be recovered from current SSD technology [11] and may require a highly reliable backing store like a parallel file system. Also, the SCR approach, in addition to using additional hardware, uses a portion of on-node memory to store checkpoints. This point is especially important for future extreme-scale systems; with the dramatic core count increases, we are moving from a compute-scare environment to one where we have an abundance of compute cycles but a scarcity of memory.

Again, we assume each process uses 2GB of memory and checkpoints $\frac{1}{3}$ of that memory. We also assume a 5 year MTBF and a per-process I/O rate of 1MB/s for the compression and incremental checkpointing case. For the SSD only case, we assume a 2GB/s checkpoint commit rate and a 8GB/sec checkpoint read rate. Lastly, for SCR, we assume a per-process mean checkpoint commit rate of 211MB/s for both read and write. This mean commit rate is calculated from [41], where the authors presented a user-space file system, CRUISE, which dramatically improve the performance of SCR. The take-away here is that the per-process checkpoint commit rates of these hardware based solutions are several orders of magnitude larger than the software solutions.

Figure 6 shows a comparison of compression with the hardware-based techniques outlined in this section. For comparison we also include the efficiency of standard rollback/recovery to the parallel filesystem shown previously. From this figure we make the following observations:

1) As expected, the hardware-based solutions perform significantly better than the software solutions
2) The SSD only solution has nearly 100% efficiency through the socket count tested, though as pointed out previously recent work suggests this solution may not be achievable.
3) The multi-level checkpointing approach which uses multiple levels of the system storage and can recover from all observed failures, performs similarly to an SSD only approach.
4) The optimal software-only approach (ickpt+compress), though two orders magnitude slower commit speeds, only performs 20% worse than the other approaches.

Not shown in this work is the combination of the SSD and SCR approach with compression. At compression rates observed on current architectures, these hardware approaches perform slightly worse with compression. This is due to the fact that the compression bandwidths are considerably slower than the commit bandwidths and the compression step becomes the performance bottleneck. At the compression speeds that might be expected using an accelerator (e.g. a GPU), the combination of compression with these hardware approaches shows only marginal (less than 2%) increases in application efficiency.

Overall, this shows the benefit of this compression approach. With no application specific knowledge, no additional hardware, minimal memory overhead, using standard and freely available compression algorithms, and using checkpoint commit bandwidths observed on today's systems, we can get within 20% of a costly hardware solution. Compression based approach can be made to be readily available to existing systems while for the hardware based solutions we need to make changes to existing systems and install hardware to support it. In the next subsection we are going to present the cost performance analysis for both the systems.

C. Evaluating the cost of an SSD-based system

In this section, we examine the cost efficiency of these hardware, software, and mixed methods. As hardware based approaches require additional storage, we factor monetary cost and performance for several NAND-based SSD storage devices and compare this to the cost of a software only approach. Although these devices have higher throughput compared to disks drives, they also have limited write-erase cycles and must be replaced after a certain number of write cycles. There are three types of NAND flash technology used in this comparison: Single layer cell (SLC), Multi-level cell (MLC), and and Three-level cell (TLC). These technologies vary in both their storage density and write endurance, two properties with an inverse relationship. Therefore, while TLC provides cheap storage capacity, it does so at the expense of durability and may fail after limited numbers of writes.

To evaluate the costs for the hardware based solutions in comparison to the top performing software-based approach (i.e. incremental checkpointing with compression), we will compare the cost and amount of work done for two equivalent systems. One of these systems will be equipped with on-node SSD devices. For both the systems we will calculate a total of five years of operational life and assume that the system is completely utilized (i.e. 100% utilization rate). As the efficiency for the software and hardware based checkpoint-restart optimizations are different, the amount of *application* work done for a fixed five years of operation will be different for both systems.

Each node of our hypothetical cluster contains 2 sockets per node, 8 cores per socket, 2GB/core memory and to support on node storage for checkpoints a 256GB SSD drive per socket. As the price per node of future systems is expected to vary, we use a per node price range from \$500 to \$3000. For both software and hardware-based approach we will use the same node configuration with the only difference in performance due to the SSD device. The type of flash, name, write-erase cycle number(f_{rating}) of the SSD drives used for this study are given in Table III. In addition, the table contains the number of weeks(t) till the device will be saturated and the current maximum write-erase cycle number (f_{max}) [42]. Based on this configuration data and the range of per-node prices, we calculate the total ownership cost per node for five years. We assume that the the additional energy costs for the SSD devices are negligible and both type of systems have equal operational costs. The multilevel checkpointing solution (SCR) also suffers an additional cost due to on-node memory used for staging. We will not include this cost in our calculations.

To quantify the SSD costs, we construct a simple model to calculate the number of weeks till saturation of the SSD device.

$$s_{cap} \times f = n_{ckpt} \times n_{proc} \times s_{ckpt} \times t$$

where s_{cap} is the initial capacity of the SSD drive, f is the write- erase cycle number, n_{ckpt} is the number of checkpoints being written for the job per week, n_{proc} is the number of processes writing the SSD, s_{ckpt} is the size of the checkpoint and t is the total number of weeks we can reliably use the SSD. Since we need to replace SSD after every t weeks, and we are looking for a five years operational life for our systems, we present the total cost per node for a SSD-based system as follows

$$c_i = c_{node} + 2 \times c_{ssd} \times \left\lceil \tfrac{5\times 52}{t} \right\rceil$$

where c_i is the total cost for node i, c_{node} is the price of a node without the SSD device and c_{ssd} is the price of the SSD device. Using the above two equations, we calculate the total work done w_i per dollar for each node, i, using the following equation

$$w_i = \tfrac{w \times e}{c_i}$$

where w is the total work, in our case 5 years of work hours and e the efficiency of our system.

For a workload of 196000 processes, checkpointing one third of $2GB$ of memory per process, we can use Daly's model to calculate that the total number of checkpoints taken during a week long job is 760. Similar to our previous discussion, we assume 8 process per socket and one 256GB SSD per socket and use the efficiency number presented in the previous subsection- 90.94% for hardware-based system(SCR) and 71.025% for software-based approach(compressed incremental checkpointing). Using this model, we present the comparative costs of these methods in Figure 7

Figure 7 shows that for lower per-node costs a software based approach produces more work per dollar per node and is more efficient. As the node prices increases, however, the overhead due to the added SSD hardware

Type	Name	f_{rating}	t(weeks)	Price(USDA)	f_{max}
TLC	Sansung 840Pro	750	47.4	200	2500
MLC	OCZ Revo drive 3	3000	189.5	460	10,000
SLC	OCZ Z drive R2	100000	6315	4800	100,000

TABLE III
PRICES AND READ-ERASE CYCLE NUMBERS FOR SSD USED IN OUR MODEL

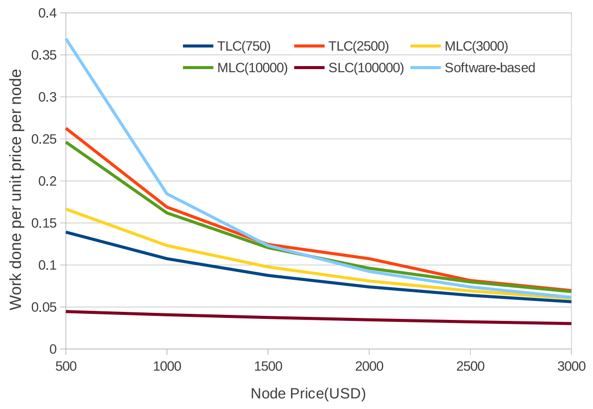

Fig. 7. Comparison of work done per unit price per node for a system with different types of SSD device compared against software-based solution. (higher is better).

is amortized and the hardware-based approaches perform similar to the software based approach. Note, our model assumes only checkpoint data is written to the SSD devices and they wear uniformly and perform according to their specifications. Several studies have shown that depending on the write patterns, these devices can wear out significantly faster; in some cases 10-30 times faster than the device specified rating [43]. This makes our model optimistic as SSD devices may need to replaced more often. As a result, the hardware-based approaches will have additional overheads.

VII. CONCLUSIONS

In this paper, we studied the performance limits of checkpoint compression and put the results of this technique in the context of the current state-of-the-art in checkpointing. Specifically, we used information theory to show that current compression techniques are close enough to a theoretical optimal that no difference in application efficiency will be seen. In addition, we showed that a dramatic increases in compression speeds (factor of 100 or more) results in a less than 15% increase in application efficiency. We showed that checkpoint compression outperforms another popular software-based checkpoint optimization, incremental checkpointing, and a combination of both leads to further performance increases. Also, we showed that a compression+incremental checkpointing technique can get within 20% of the performance of the current state-of-the-art in checkpointing, while not requiring additional costly non-volatile storage. Finally, we studied the additional cost of such non-volatile storage devices and showed that our software-based checkpoint/restart optimization produces more work per unit cost than the hardware-based approaches as long as per-node procurement costs are kept low. A remaining open question we are currently investigating is the power/energy considerations of checkpoint compression and their impact on large-scale systems.

REFERENCES

[1] B. Schroeder and G. A. Gibson, "A Large-scale Study of Failures in High-performance Computing Systems," in *Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006.

[2] K. Ferreira, R. Riesen, P. Bridges, D. Arnold, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, and R. Brightwell, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *SC*. ACM, Nov 2011.

[3] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," *2012 41st International Conference on Parallel Processing*, vol. 0, pp. 148–157, 2012.

[4] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. De Supinski, and R. Eigenmann, "MCRENGINE: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012.

[5] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[6] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-recovery Protocols in Message-passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[7] D. Z. Pan and M. A. Linton, "Supporting Reverse Execution for Parallel Programs," in *1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*. Madison, WI: ACM Press, 1988, pp. 124–129.

[8] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, Concurrent Checkpoint for Parallel Programs," in *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*. Seattle, Washington: ACM, 1990, pp. 79–88.

[9] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '95/PERFORMANCE '95. New York, NY,

USA: ACM, 1995, pp. 64–73. [Online]. Available: http://doi.acm.org/10.1145/223587.223596

[10] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 10, pp. 972–986, oct 1998.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[12] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *International Parallel Processing Symposium*. Honolulu, HI: IEEE Computer Society, April 1996, pp. 526–531.

[13] V. C. Zandy, B. P. Miller, and M. Livny, "Process Hijacking," in *8th International Symposium on High Performance Distributed Computing (HPDC '99)*, Redondo Beach, CA, August 1999, pp. 177–184.

[14] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[15] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software – Practice & Experience*, vol. 29, no. 2, pp. 125–142, 1999.

[16] Y. Chen, K. Li, and J. S. Plank, "CLIP: A Checkpointing Tool for Message-passing Parallel Programs," in *SuperComputing '97*, San Jose, CA, 1997. [Online]. Available: http://citeseer.ist.psu.edu/chen97clip.html

[17] E. N. Elnozahy, D. B. Johnson, and W. Zwaenpoel, "The Performance of Consistent Checkpointing," in *11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, 1992. [Online]. Available: http://citeseer.ist.psu.edu/elnozahy92performance.html

[18] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *IEEE International Symposium on Parallel&Distributed Processing*, 2009, pp. 1–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1586640.1587642

[19] K. Li, J. F. Naughton, and J. S. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, August 1994.

[20] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995, pp. 213–224.

[21] K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold, "Libhashckpt: Hash-based Incremental Checkpointing Using GPUs," in *Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[22] C.-C. Li and W. Fuchs, "CATCH-compiler-assisted techniques for checkpointing," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, jun 1990, pp. 74–81.

[23] A. Moshovos and A. Kostopoulos, "Cost-Effective, High-Performance Giga-Scale Checkpoint/Restore," University of Toronto, Tech. Rep., November 2004.

[24] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 2, no. 2, pp. 62–67, 1994.

[25] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," University of Tennessee, Tech. Rep. CS-95-302, August 1995. [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-95-302.html

[26] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974. [Online]. Available: http://doi.acm.org/10.1145/361147.361115

[27] A. Geist *et al.*, "Fault management workshop final report," U.S. Department of Energy, Tech. Rep., August 2012.

[28] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratory, Tech. Rep. SAND2009-5574, 2009.

[29] S. J. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal Computation Physics*, vol. 117, pp. 1–19, 1995.

[30] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006.

[31] J. G. Elytra, "Parallel Data Compression With Bzip2."

[32] P. Deutsch, "Deflate Compressed Data Format Specification." [Online]. Available: ftp://ftp.uu.net/pub/archiving/zip/doc

[33] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," Ph.D. dissertation, Australian National University, February 1999.

[34] "7Zip Project Official Home Page." [Online]. Available: http://www.7-zip.org

[35] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 783–788. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2008.109

[36] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 623–, july, october 1948.

[37] A. Colic, H. Kalva, and B. Furht, "Exploring nvidia-cuda for video coding," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, ser. MMSys '10. New York, NY, USA: ACM, 2010, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/1730836.1730839

[38] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *Journal of Physics Conference Series*, vol. 78, no. 1, 2007.

[39] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654100

[40] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Proceedings of the nternational Parallel and Distributed Processing Symposium*, ser. IPDPS '13. New York, NY, USA: ACM, 2013.

[41] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 pb/s file system to checkpoint three million mpi tasks," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 143–154.

[42] AnandTech, "Understanding TLC NAND," 2012, http://www.anandtech.com/show/5067/understanding-tlc-nand/2.

[43] R. Templeman and A. Kapadia, "Gangrene: Exploring the mortality of flash memory," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, ser. HotSec'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372387.2372388