SAND2015-4456C

Canaries in a Coal Mine: Using Application-level Checkpoints to Detect Memory Failures

Patrick M. Widener[1], Kurt B. Ferreira[1], Scott Levy[2], and Nathan Fabian[1]

[1] Center for Computing Research, Sandia National Laboratories*, Albuquerque, NM
[patrick.widener|kbferre|ndfabia]@sandia.gov
[2] University of New Mexico, Albuquerque, NM, USA,
slevy@cs.unm.edu

Abstract. Memory failures in future extreme scale applications is a significant concern in the high-performance computing community and has attracted much research attention. We contend in this paper that using application checkpoint data to detect memory failures has potential benefits and is preferable to examining application memory. To support this approach, we describe the application of machine learning techniques to evaluate the veracity of checkpoint data. Our preliminary results indicate that supervised decision tree machine learning approaches can effectively detect corruption in restart files, suggesting that future extreme-scale applications and systems may benefit from incorporating such approaches in order to cope with memory failures.

1 Introduction

Fault-tolerance has been identified as a major challenge for exascale-class systems. As systems grow in scale and complexity, failures become increasingly likely. Due to the plateauing of CPU clock rates, a system $1,000$x more powerful than today's petascale systems will likely need $1,000$x more components to deliver this increased performance [1]. This increase in component count will likely lead to a commensurate increase in the system's failure rate. This is compounded by the fact that shrinking transistor feature sizes and near-threshold voltage logic needed to address energy concerns may further increase the hardware failure rates. Given these dire predictions and the dynamics of fault-tolerance techniques, significant effort has been and is being devoted to improving system resilience.

The current de facto standard for fault-tolerance on high-performance computing (HPC) systems is coordinated checkpoint/restart. The success of checkpoint/restart on current systems depends on two assumptions: 1) failures are not commonplace; and 2) systems receive notification of failures, i.e., silent data corruption is rare. While these assumptions hold on today's systems, whether they will continue to hold on next-generation extreme-scale systems is unclear.

* 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.

Silent data corruption (i.e., undetected bit flips) are of particular concern for future systems, and are not mitigated using checkpoint/restart. These undetected bit flip are such a concern that *application-* and *algorithmic-based* fault tolerance methods has become an active research area [2–5]. These methods, which are tailored to the computational characteristics of specific applications, are guaranteed to deliver a correct answer even in the presence of failures in application memory. These methods typically work by 1.) encoding redundant data into the problem such that data from failed nodes can be recomputed [2–4], or 2.) exploiting an algorithm-specific relationship between the parallel application and its individual data chunks. Data lost due to a failure impacts the result by possibly increasing the error of the solution or by running the surviving nodes for longer until the problem has converged. Therefore, the number of nodes lost determines the application time-to-solution or the error associated with the solution [5]. A key feature of these methods is that they typically must protect the entire memory footprint for the application, which can be substantial.

In contrast to current algorithmic methods, we propose using *application checkpoints*, rather than the application's entire memory footprint, to detect errors. In particular, we propose using them to detect silent data corruption in HPC applications. This position has a number of advantages over more traditional application-based methods. For example:

- Current extreme-scale algorithms already use checkpointing for fault-tolerance, and will therefore require little modification to take advantage of this method.
- By definition checkpoints represent the critical state of the application. The entire memory footprint can be regenerated from this critical state.
- Checkpoints are typically much smaller than the application's memory footprint and therefore will likely require lower overheads to protect.
- Errors that occur but do not eventually impact checkpoint data are ignored as they do not impact the applications critical state.
- Error detection methods may have lower overheads than the detection and correction mechanisms found in algorithmic approaches.
- Checkpoint verification can run in parallel with application computation, not interfering with application progress.
- Checkpoint data is widely used as input for downstream tasks such as analytics and visualization. Making checkpoints a basis of fault-detection efforts also protects the operation of such downstream computations.

We therefore make the following contributions in this paper:

- We introduce an application-independent strategy of verifying application checkpoints as proxies for application memory corruption.
- We describe the application of particular unsupervised and supervised machine learning methods to the problem of verifying checkpoint data.
- Using checkpoints from an execution of a well-known simulation, we examine the clustering and prediction accuracy of our chosen methods. While unsupervised clustering does not appear to distinguish corrupted data well, we found that a supervised learning method trained on bit-level errors can classify corrupted checkpoints with reasonable accuracy.

2 Checkpointing on Current Systems

Checkpoint/restart is the most widely studied and deployed set of techniques for handling failure in large-scale high-performance computing (HPC) systems. The basic idea is that applications can mitigate the cost of recovering from failure by periodically saving their critical state, a *checkpoint*, to some form of storage that is likely to survive the failure. Upon failure, the last known good checkpoint is retrieved from stable storage, loaded into memory, and computation resumes.

Checkpoints contain the critical data of an HPC application–the data needed to fully recreate the state of the application after a failure. They are typically highly optimized and considerably smaller than the runtime memory footprint of the application. For example, Table 1 shows the per-process memory footprint for two key production applications, CTH and LAMMPS. LAMMPS is a molecular dynamics code [6] from Sandia National Labs and CTH is a shock physics code [7] also from Sandia. Each of these applications are key US Department of Energy DOE applications which run for long periods of time in production modes and exhibit a range of different communication structures. From this table, we see that these application-based checkpoints are significantly smaller than the entire memory footprint. The CTH checkpoints are roughly 5% of the application's memory footprint. The LAMMPS checkpoints are approximately 19% of the memory footprint for this LAMMPS problem, EAM. [3]

Application	Memory Footprint (MB)	APP Checkpoint Size (MB)	CKPT % of Footprint (%)
CTH	583	26.1	4.5%
LAMMPS (EAM)	3,256	608.0	18.7%

Table 1. Per-process memory footprint and application-based checkpoint sizes for two key production workloads, CTH and LAMMPS. The memory footprint represents the average across the lifetime of the application. The application checkpoint (APP Checkpoint) is the average across all checkpoints and nodes. The final column is the percent the memory footprint the checkpoint occupies.

Because application checkpoints capture the critical state of an application but are a small fraction of the size of the application's entire memory footprint, it is our position that checkpoints can be effectively and efficiently exploited to protect against silent data corruption in the memory of HPC applications. The fact that they contain the critical state of the application means that we can identify any errors that would corrupt the solution produced by the application. The fact that checkpoints are so much smaller than the application's memory footprint means that the overhead of examining them to identify the effects of data corruption will likely be much lower than if we were to consider all of application memory directly.

[3] This is the largest checkpoint over tested LAMMPS inputs (EAM, LJ, SNAP, CHAIN, RHODO). The average checkpoint size is considerably smaller–7%.

3 Approach: Using Checkpoints as Failure Detectors

We identify two types of checkpoint corruption: (i) indirect corruption; and (ii) direct corruption. Indirect corruption occurs when a silent error in application memory is captured and preserved in a checkpoint. In this case, if we detect that a checkpoint has been corrupted it indicates that the application has been corrupted and may produce an untrustworthy result. As a result, recovery requires restarting the application either from a known-good checkpoint or from the beginning. Direct corruption occurs when the checkpoint itself is corrupted without affecting the state of the application. Although the application is unaffected, if a failure occurs before the next checkpoint is taken, restoring the state of the application from the checkpoint would allow the corruption to propagate into the application's memory. Recovery in this case could be accomplished by either re-checkpointing the application or rolling back to an earlier known-good checkpoint.

Automatic classification using selected machine learning techniques can help us determine whether a checkpoint contains one or more errors. By training a supervised learning classifier using known-good checkpoints along with ones that include known errors, newly-generated checkpoints can be identified as valid or not. This identification carries a degree of certainty which will vary according to the learning method chosen, the semantics of the checkpoints themselves (necessarily an application-dependent factor), and the amount and variety of checkpoints used for training data.

Choice of features for training data is an interesting issue in this case. Checkpoint metadata or provenance information can contribute meaningfully. For example, a checkpoint size which differs from expectations might be a sign of a truncation. Also, a particular node with known memory issues or other significant maintenance history might be represented as additional features in a potential model. Determining features based on the checkpoint data itself will likely prove more complicated. Assigning features based on checkpoint data semantics is most straightforward but is necessarily application-specific; however, this would allow for considerable reduction of feature dimensionality by eliminating highly-correlated data. More generic approaches such as considering raw bit patterns in the output are possible, but present undifferentiated feature ranges and will likely pose scalability issues; consider that checkpoint files with sizes measured in hundreds of megabytes are not uncommon. Also, the choice of learning method is dependent on how checkpoint data and metadata are mapped into a feature space: decision trees or Bayesian methods may work better with heterogeneous data while support vector machines are more appropriate for data which can be scaled into a common numeric range.

Ideally, such a classification step would be performed immediately upon checkpoint generation. However, it may not be feasible for an application to perform classification itself at each checkpoint without seriously degrading solve time. If *in situ* classification is not possible, it may be possible to maintain in-band detection by providing a communication channel whereby an application can be notified when a previous checkpoint has been determined invalid. It may be possible to accomplish classification within an acceptable checkpoint interval through the use of co-processors such as GPGPUs or reserved processor cores or machine nodes. This is similar to how analytics are performed in so-called "in-transit" solutions, where classification tasks would be treated as another

type of data analysis. This would provide feasibility advantages in that perturbation effects are typically already addressed to varying degrees in these environments, and also in that such approaches are already using checkpoint data as input to visualization and analysis tasks. If classification cannot be performed in a checkpoint interval, an out-of-band method, potentially requiring the retention of checkpoint data for lengths of time greater than a checkpoint interval, will be required.

4 Using learning approaches to classify checkpoints

To explore the possible application of ML approaches to checkpoint error detection, we conducted a series of experiments which we detail in this section.

4.1 Application checkpoint description

For this work, we consider errors in the aforementioned LAMMPS production molecular dynamics application. From LAMMPS, we consider the SNAP potential. SNAP is a computationally intensive, quantum accurate potential that uses the same kernel as GAP [8]. This potential was chosen for a number of reasons: 1) this represents an important scalable workload used at extreme-scale, and 2) due to the small number of atoms, checkpoints are exceedingly small–92 bytes per MPI process. These 92 byte checkpoints allow us to investigate the efficacy of our position while not having to worry about performance overheads.

4.2 Modeling checkpoint data

We chose for our investigation to treat the 128,000 92-byte restart files from LAMMPS as samples of 92 one-byte features. This choice disregards potentially useful information that might be derived from the files. For example, an examination of the logical structure of the restart files would result in fewer features per sample, as each restart file is a serialization of a C++ application object using data types which are multiple bytes in length. Using the logical structure of the restart file could also give an indication of how features should be weighted when applying different ML methods.

Despite these considerations, we believe our approach still provides useful insight. We are most concerned with the detection of single-bit errors, which should be better isolated using a larger number of single-byte features. Also, as different applications write different information in their checkpoints, exploiting the structure of the restart files is necessarily an application-dependent modeling approach that would need to be repeated for other applications. This information can also be difficult to acquire if access to the source code for the application in question is unavailable. Our approach provides a way of looking at checkpoints with arbitrary structure.

We synthesized corrupted restart files with different scales of errors. To represent silent data corruption or "bit-flips", we introduced errors by flipping one, two, or three bits in a single restart file. This was done by choosing a byte at random in the restart file, and then choosing a random bit within that byte to invert; multiple bit flips were done by repeating this process. We synthesized larger-scale errors by choosing either

one byte or three bytes at random and substituting a random bit pattern. This patterns are consistent with the errors patterns found on current systems [9].

We use the `scikit-learn` [10] Python toolkit for our experiments. `scikit-learn` contains implementations of many different ML algorithms, allowing implementation changes with relatively little overhead. Several other toolkits of this kind exist, most notably the Weka class library [11]. While performance of the classification algorithms was not a primary concern in this work, attempting online or near-line classification might require parallelized or more performant approaches for applications with larger restart files.

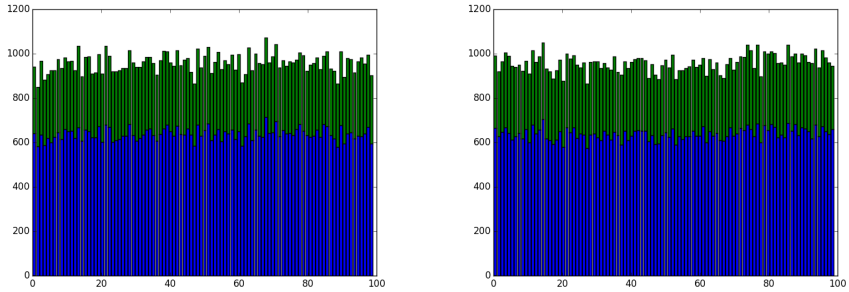
4.3 Choosing an ML technique

Although we expect there are correlations between files written out as part of a group checkpoint, they may not necessarily relate to whether errors occur within any single file. For simplicity we consider each checkpoint file as an i.i.d. sample of potential error. Because each byte is a feature, the space we are considering here is not contiguous, and in fact certain byte values for certain features will probably never arise for a checkpoint value, and should be construed as erroneous. This makes a high dimensional solution space of good regions marked by large voids, or bad regions, where there is no plausible observation. Our expectation here is very much a non-linear relationship among the features, observations into the solution space, and the likelihoods of a bit error occurring in any given file. Based on this, we consider both a k-means clustering [12] and CART a tree classifier, provided by scikit-learn, as appropriate first passes to capture those regions and the relationship between them and detected errors, the results of which are discussed further on.

Moving beyond these initial attempts to capture the space, several possibilities exist for exploration in future work. Unsupervised approaches, self-organizing maps or growing neural gas [13], may provide better information about the layout or structure of the file space, especially to understand whether and where these void regions lay between the good. In particular k-means clustering provides an opportunity to explore proximity measures beyond Euclidean distance and determine if there is a better comparison between points in the high dimensional space. With exploration of the features themselves, including imposing structure from the definition of the file, we may find additional correlations in the features allowing some reduction from raw byte counts, such as bag-of-words or Latent Dirichlet Allocation [14] used in text. Following Rennie et al. [15], we may find good results using Naive Bayes with more information rich features.

4.4 Unsupervised learning: clustering with k-means

We first investigated an unsupervised learning method, k-means clustering [16]. Our goal here was to explore whether explicit labeling is a necessity for this set of data. We used the `scikit-learn` *KMeans* module to perform clustering. The input samples were the set of uncorrupted restart files combined with a set of corrupted restart files with a specific error type as described above, and we used 100 as K. We conducted three separate clustering tests, with the results displayed in Fig. 1. These figures stack



(a) Clustering with 100 clusters, 1-bit errors (b) Clustering with 100 clusters, 3-byte errors

Fig. 1. K-means clustering results. The X-axis is cluster index, and the Y-axis is the number of samples in each cluster.

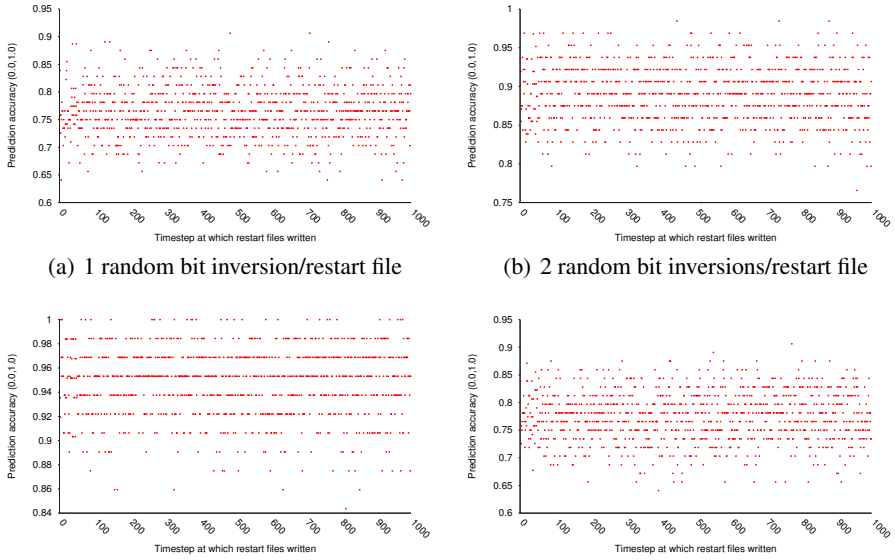
the histogram of the uncorrupted clusters on the bottom with the corrupted clusters histogram on top.

If clustering is to be effective at distinguishing restart file corruption, we would expect to see a majority of the samples corresponding to corrupted restart files collected in a subset of the clusters, and similarly for those samples corresponding to uncorrupted restart files. The figure shows that, instead, both sets of samples are relatively uniformly distributed across the set of clusters. Note also that the general uniform shape does not change as the scale of error changes across the subfigures. Similar results hold for $K = 10, 50, 200, 300$, not presented here due to space constraints. Our approach here is straightforward, without any significant tuning of the clustering algorithm. However, for the purposes of our exploration, an unsupervised learning approach does not seem to be effective in distinguishing restart files with errors. We therefore turn our attention to explicitly labeled samples and a supervised method.

4.5 Supervised learning with decision tree methods

As detailed above, we chose a decision tree-based method as a test case for supervised learning on our checkpoint data. As with the unsupervised K-means experiments, we treat each 92-byte restart file as a sample with 92 features, namely the constituent bytes of the file. We first trained a separate decision-tree classifier on each type of our synthetically corrupted data (1,2,3-bit and 1,3-byte). For this experiment, we trained each classifier on the entire set of uncorrupted restart files (labeled as good inputs) and on the type-specific set of corrupted files (labeled as bad inputs). We then evaluated the prediction accuracy on the entire set of bad files. Since the classifier had already built a model using the bad samples, we expected high accuracy as this experiment served as a best-case trial of the decision tree method on these inputs.

These results are presented in Fig. 2. Each point on each plot is the prediction accuracy for the restart files, across all nodes at each timestep for which corrupted data was synthesized (if a restart file was written by a node at a timestep in the uncorrupted data,



(a) 1 random bit inversion/restart file

(b) 2 random bit inversions/restart file

(c) 3 random bit inversions/restart file

(d) 1 random byte error/restart file

Fig. 2. Prediction accuracy of each classifier on the same type of error, across all ranks and timesteps of the set of restart files.

we synthesized a file with each type of data corruption). The data indicates that the bit-scale-trained classifiers have better accuracy detecting bit-scale errors than do the byte-scale classifiers on byte-scale errors. The 2-bit and 3-bit classifiers in particular are able to correctly classify the corrupted files with high accuracy.

In order to get a more general idea of the ability of the different classifiers to correctly identify multiple types of corrupted data, we then conducted an MxN comparison where we measured the prediction accuracy of each classifier on each type of our synthetic corrupted data. In this trial we also include classifiers trained using the AdaBoost ensemble method [17] for an example of an additional supervised learning method. Additionally, for this experiment we separated the corrupted data into training and testing sets, only running the prediction on the testing set of error data. Fig. 3 displays the results of this experiment. All save one of the classifiers performed best at identifying restart files with at most 3 inverted bits. The notable exception in this case was the classifier trained on 3 random bytes, which exhibited poor prediction performance on every type of corrupted data tested. In addition, the prediction accuracy of all the classifiers suffered when tested against the higher scale (1-byte and 3-byte) errors. While our experiments were not intended to conclusively explain all behaviors of the set of classifiers, we hypothesize that increasing the scale of errors essentially introduces noise into the feature set, complicating accurate prediction.

Finally, although our experiments were not designed to comprehensively measure performance, we measured relative execution time of the different classifiers for the MxN comparison. Presented in Fig. 4, our results show that the AdaBoost ensemble

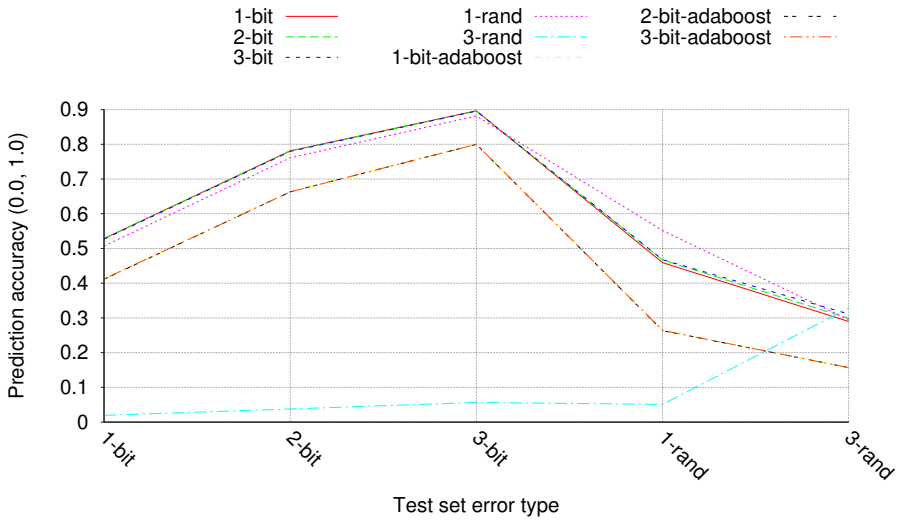


Fig. 3. Prediction accuracy of each classifier against the test set of each type of random errors (1-3 bit, 1 and 3 byte)

classifiers took considerably longer than the single decision tree classifiers for each type of data corruption we examined. When considered against the results of Fig. 3, it seems clear that a large cost in prediction execution time for the AdaBoost classifiers produces no advantage in predictive accuracy. It is dangerous to generalize from this result, as it reflects dependencies on our test data as well as a lack of tuning of the training methods for the classifiers. However, for future applications of these types of learning methods for automatic classification of restart files, these types of investigations may prove useful.

5 Related work

To the best of our knowledge, we are unaware of any existing work which addresses silent data corruption in HPC application by ensuring the veracity of checkpoints. That being said, resilience methods which ensure progress in failures, both detected and undetected is diverse and popular. In this section, we briefly outline some of the more closely related studies and contrast them with our work.

Checkpointing: The most prevalent method of defensive fault-tolerance mitigation in modern applications, coordinated checkpointing periodically writes global application or system state to stable storage [18]. Consistent application state snapshots are enforced through global barrier synchronization. When a process fails, all application processes can then be restarted from a known-good, globally consistent state. Algorithmic approaches borrowed from the distributed computing domain [19] allow applications to generate consistent checkpoints without using barriers, avoiding increasingly expensive global synchronization.

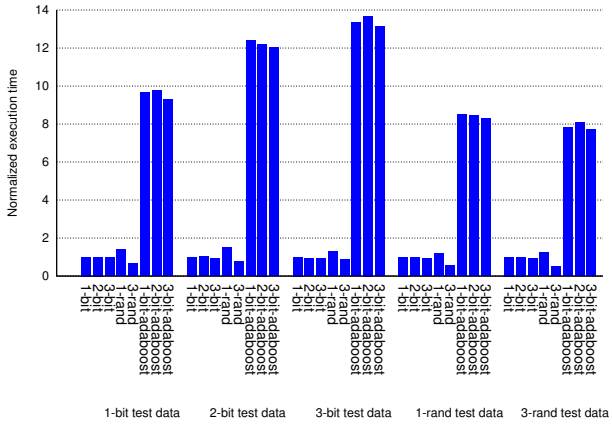


Fig. 4. Classifier execution time normalized to the performance of the 1-bit classifier.

Coordination and limited bandwidth to the stable storage typically used to store checkpoints have given rise to uncoordinated or asynchronous checkpointing [18, 20–22]. In these systems, nodes do not synchronize or coordinated in any way when they checkpoint, but they also keep a log of their sent messages on stable storage. Nodes restoring from local asynchronous checkpoints can then reconstruct a local state consistent with the application global state by replaying inbound messages from other nodes (using their logs).

These checkpointing methods provide no facilities for dealing with SDC in the application. Therefore, protection requires algorithmic- or application-based mechanisms to ensure these failures are detected and/or properly dealt with.

Algorithmic-based fault tolerance: Algorithmic-based Fault Tolerance Mechanisms are based on notion of designing algorithms that are capable of ignoring errors while delivering a correct answer, or are capable correcting errors using techniques such as redundant data or computation. Algorithmic-specific data redundancy methods work by encoding additional data into the problem such that data from failed nodes can be recomputed. In addition, the algorithm is modified to update the encoding as computation progresses [2–4]. In contrast, computation redundancy relies on algorithm-specific relationship between the parallel application and its individual data chunks. If data is lost due to a failure, this impacts the result by possibly increasing the margin of error or by running the surviving nodes for longer until the problem has converged [5, 23].

Machine learning for anomaly detection: A great deal of research has used machine learning techniques for different kinds of anomaly detection and classification. Li et al. used 1-gram and naive Bayes methods to identify different file types [24]. Others have considered anomaly detection in the context of: malware behavior analysis [25], network intrusion detection [26], and internet traffic classification [27]. To our knowledge, these techniques have not yet been applied to checkpoint/restart artifacts.

6 Conclusion

Preserving the reliability of future extreme scale applications will require new approaches for ensuring the veracity of the computation performed. We contend that validating checkpoint data is a more feasible approach than examining application memory footprints. In this initial study, we have described how checkpoint data can be leveraged to detect errors in application memory that are captured in a checkpoint. As a proof of concept, we investigated the usefulness of machine learning methods for automatically detecting checkpoint corruption.

Our initial results indicate that supervised learning approaches may be preferable to clustering for detection of checkpoint corruption. In particular, we determined that decision trees show promise for detecting small errors (as many as 3 inverted bits out of a 92-byte checkpoint) with reasonable accuracy. Based on these results, we are pursuing additional refinements to our machine learning techniques to improve its performance and to account for a broader range of failure modalities. More generally, we hope that these results prompt more investigation of machine learning approaches. Many machine learning alternatives exist, with different abilities to tune for accuracy and different performance characteristics, and much of the promise of this approach is yet unexplored.

References

1. Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.
2. Zizhong Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.
3. Zizhong Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In *IPDPS*, pages 1–8. IEEE, 2008.
4. Huang Kuang-Hua and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33:518–528, June 1984.
5. Christian Engelmann and George A. (Al) Geist. Super-scalable algorithms for computing on 100,000 processors. In *Lecture Notes in Computer Science: Proceedings of the* 5^{th} *International Conference on Computational Science (ICCS) 2005, Part I*, volume 3514, pages 313–320, Atlanta, GA, USA, May 22-25, 2005. Springer Verlag, Berlin, Germany.
6. Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.
7. Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382, July 1993.
8. A.P. Bartók, M.C. Payne, R. Kondor, and G. Csányi. Gaussian approximation potentials: the accuracy of quantum mechanics, without the electrons. *Physical review letters*, 104(13):136403, 2010.
9. V. Sridharan et al. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

10. F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
11. Mark Hall et al. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1), 2009.
12. Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.
13. Bernd Fritzke et al. A growing neural gas network learns topologies. *Advances in neural information processing systems*, 7:625–632, 1995.
14. David Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.
15. Jason D Rennie, Lawrence Shih, Jaime Teevan, David R Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, volume 3, pages 616–623. Washington DC), 2003.
16. Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.
17. Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.
18. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
19. K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
20. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant mpi for volatile nodes. In *Conference on High Performance Networking and Computing (SC2002)*, pages 1–18, Baltimore, MD, november 2002.
21. P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *IEEE International Conference on Cluster Computing*, pages 115–124, 2004.
22. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.
23. Patrick Bridges, Mark Hoemmen, Kurt B Ferreira, Michael Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/os DRAM fault recovery. *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the Euro-Par Conference, Lecture Notes in Computer Science*, 2011.
24. Wei-Jen Li, Ke Wang, Salvatore J Stolfo, and Benjamin Herzog. Fileprints: Identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 64–71. IEEE, 2005.
25. Konrad Rieck et al. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.
26. Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.
27. Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.