BAD-Check[§]: Bulk Asynchronous Distributed Checkpointing

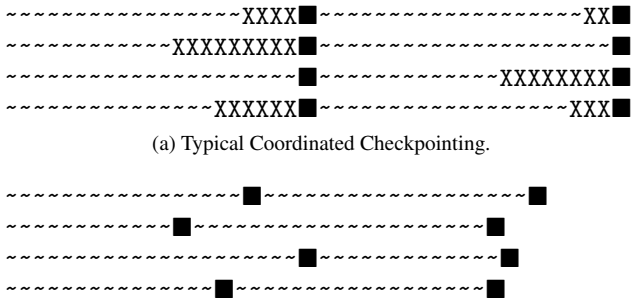
John Bent[*‡] Brad Settlemyer[†‡] Haiyun Bao[*]
Sorin Faibish[*] Jeremy Sauer[†] Jingwang Zhang[*]

Abstract

Leadership-scale scientific simulations running as tens of thousands of tightly-coupled MPI processes are vulnerable to interruption due to a single process or node failure. Due to the dependence of each state calculation on the successful completion of each of the prior state calculations, checkpoint-restart is the most widely-used technique to achieve fault tolerance. To write a consistent view of distributed state as a checkpoint, applications typically synchronize and pause while writing data to persistent media. In this paper we present a transactional protocol that enables asynchronous distributed creation of checkpoint data sets, and describe the conditions under which it is beneficial. With simulations, we demonstrate that scientific applications exhibiting computational variance without frequent synchronization can use our protocol to either reduce run time by up to 27% or reduce required storage system capability by up to 40%.

1 Introduction

Diverse computational science fields, such as climatology, fluid dynamics, and astrophysics, use large-scale, tightly-coupled simulations to advance scientific inquiry. Due to the frequent MPI-based communication between processes and the derivation of the current distributed simulation state from all prior simulation states, *checkpoint-restart* has become the dominant method for providing parallel application fault tolerance. In particular, scientific simulations have traditionally relied on coordinated checkpoint construction and bulk synchronous processing (BSP) in which all of the application processes barrier, or fence, and synchronously dump their state into one or more files. The large memory footprint required by these simulations makes it unrealistic for the computation to continue while the checkpoint data is being written. There typically is not enough spare memory space to copy the data locally to immutable buffers that can be written to the storage system asynchronously.



(a) Typical Coordinated Checkpointing.

(b) Checkpointing with BAD-Check.

Figure 1: **Parallel Computation and Checkpointing.** *The top figure shows a typical parallel application in which all processes are limited by the speed of their slowest member. Each process is represented with a single row, computation with '~', idleness with 'X', and IO with ■. The bottom shows the same application using BAD-Check, in which its fast processes are freed from this dependency.*

A BSP application, as shown in Figure 1a, alternates between computational phases where simulation progress is achieved, and checkpoint phases which serve to protect against hardware and software failures. Between compute and checkpoint phases occasional periods of idleness are necessary as the BSP model forces fast processes to wait for slow ones. Although some scientific applications are able to occasionally re-purpose the checkpoint data for visualization and analysis, since ideal checkpoint intervals are governed by the leadership system's mean time to interrupt, checkpoint frequencies may be too infrequent for useful analysis, and, further, are dependent upon the underlying machine architecture rather than the parameters of the scientific simulation.

Beyond the lack of simulation progress during checkpoint phases, bulk synchronous checkpoint techniques also affect how leadership-class storage systems are designed and built. In order to minimize the time spent writing checkpoint data, leadership-scale system architects must design storage systems that can satisfy extremely bursty I/O bandwidth requirements. That is, the storage system must be designed to provide extremely high ingest bandwidth that is always available for use, even though the available peak bandwidth will be used infrequently. Thus, even if the storage system is accessed by the entire data center, checkpoints must be serviced

[*]EMC OCTO: first.last@emc.com
[†]LANL HPC: {bws,jsauer}@lanl.gov
[‡]A portion of this work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-15-27175
[§]During the lead authors' formative years, the word *bad* was ironically, and hysterically, misappropriated by American popular culture to mean *good*.
Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
PDSW2015, November 15-20 2015, Austin, TX, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-4008-3/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2834976.2834981

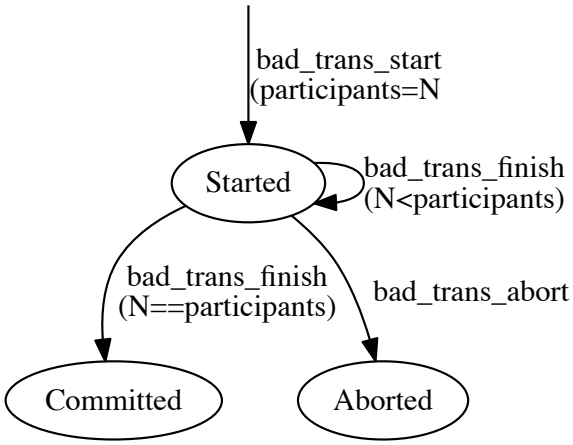


Figure 2: **BAD Transactional States.** *This state-transition diagram shows the valid states of BAD-Checktransactions. Notice that BAD-Check uses reference counting to provide data consistency via an atomic commit of a distributed set of asynchronous modifications.*

immediately; otherwise, the leadership-class computer, usually the most expensive resource within the data center, will idle which results in lost scientific productivity.

In contrast, asynchronous checkpointing, as shown in Figure 1b, has the potential to both reduce overall application runtime and lessen the peak bandwidth requirement of the storage system. For maximum benefit, as will be discussed further in Section 2, asynchronous checkpointing must support prior state dependencies and also cannot create additional copies of large data sets. In this paper we describe a new checkpointing protocol, Bulk Asynchronous Distributed Checkpointing (BAD-Check) that satisfies these requirements. Further, we identify the application requirements that enable it and quantify the degree to which these requirements must be present in order to benefit from it.

2 Related Work

Fundamentally, there are only a few general techniques to decrease the checkpointing overhead of large parallel applications. One technique is to increase the storage bandwidth using mechanisms like burst buffers [3, 14], staging areas [15, 16], and I/O reorganization [4, 25]. These techniques are generally interested in accelerating synchronous checkpoint performance rather than changing the checkpointing paradigm. Additionally, each of these checkpoint mechanisms require the addition of dedicated resources to the computation machine to improve checkpoint performance. In contrast, BAD-Check is focused on lowering the burst bandwidth requirement thereby reducing the overall required peak storage bandwidth.

A second technique is to reduce the quantity of checkpoint data using techniques such as compression [11], deduplication [18], and incremental checkpointing [19]. These techniques are largely orthogonal to our asynchronous checkpointing scheme; further, we believe that they can be applied to asynchronous checkpoint protocols.

```
bad_ret_t bad_trans_start(bad_container_t,
    bad_trans_id_t, uint32_t participants, bad_mode_t);

bad_ret_t bad_trans_finish(bad_container_t,
    bad_trans_id_t, int flag);

bad_ret_t bad_trans_query(bad_container_t,
    bad_trans_id_t, bad_trans_status_t *);

bad_ret_t bad_write(bad_obj_t,
    bad_trans_id_t, const char *buf, offset_t, ssize_t len);

bad_ret_t bad_read(bad_obj_t,
    bad_trans_id_t, char *buf, offset_t, ssize_t len);
```

Figure 3: **BAD API.** *The relevant details of the BAD API include a mode to indicate read or write and a flag to indicate whether to commit or abort an open transaction; other parameters are self-explanatory. Many details have been elided to allow focus on the relevant transactional features. The complete characteristics of this API, which grew from the 2014 DOE Storage and IO FastFoward project [1], include asynchrony, object containers, semantic objects, hints, checksums, versioning, garbage collection, and batch IO.*

A third checkpoint technique allows computation and checkpointing to proceed mostly in parallel by protecting the memory region and applying copy-on-write as necessary [13, 21]. Although these techniques reduce idleness as does BAD-Check, they differ by requiring available free memory to buffer the subsequently modified memory regions until they can be committed to persistent storage. Large scientific applications are computationally intensive and frequently under significant memory pressure. The combination of these requirements means that memory is modified frequently and few additional resources exist for buffering.

The final checkpoint acceleration technique are IO mechanisms that enable uncoordinated checkpointing [6, 24]. These systems enable asynchronous checkpointing for individual processes. However, these techniques also require message logging in order to fully capture the distributed state [12]. An additional benefit provided by these systems is that not all processes must necessarily roll-back upon restart*. A disadvantage of these systems is that complex inter-dependencies sometimes force the roll-back to the very beginning of the computation (*i.e.* without benefiting from any of the previously saved checkpoints) [23]. Our BAD-Check protocol is similar to uncoordinated checkpointing. However, rather than leveraging message logging, we rely on coordination within the distributed storage system to ensure a consistent checkpoint data set. We note that adding message logging to BAD-Check may add further improvement by enabling global restarts without restarting every process.

3 Design

Although most scientific simulations currently use BSP, asynchronous programming models that scale in the presence of computational jitter [22] are increasing in importance [2, 9, 10]. To support them, and to address the limited scalability of the POSIX storage interface [7], the DOE commissioned the Fast Forward Storage and IO project [1] to

*Recent work [5] suggests optimizing restart can be counterproductive

build a new storage interface for exascale; BAD-Check grew from this project.

Figures 2 and 3 show the protocol states and API for constructing a BAD-Check distributed transaction. Each participant in a transaction must establish with its peers a version identifier for each transaction, as well as agree on the number of peers participating in the transaction. With this knowledge, instead of simply writing to a shared file or directory, each participant writes to a shared data set within a shared transaction. The storage system, BAD-Check, does reference counting on the participants and is responsible for the atomic commit of the asynchronous distributed modifications upon transaction completion. Comparable pseudo-code for BSP and BAD-Check,

```
for ckpt in range(0,max_ckpts):
  for ts in range(0,max_timesteps):
    compute();
    exchange(exchange_group);
  if mode==BSP:       # BSP mode
    barrier();        # barrier
    open("ckpt.%d" % ckpt);
    write(data);
    close();
  else:               # BAD-Check mode
    # no barrier
    bad_trans_start(tid=ckpt,
              participants=sizeof(exchange_group));
    bad_trans_write(data);
    bad_trans_finish();
```

shows the key absence of the barrier in the BAD-Check branch. For clarity's sake, we elide the opening and closing of BAD-Check objects which could either be done within, or outside of, the main *for* loop.

This new interface does require application modifications. We considered an interposition middleware layer that would convert POSIX IO from unmodified applications into BAD-Check. However, as knowledge in the middleware layer about the number of writers to a shared file is not available, we did not implement this.

Another challenge is that applications must agree upon which consistent version of the data (i.e. which simulation time step) to store as a checkpoint. Although this decision process exists in current BSP codes, the decision will need to be mutually agreed upon beforehand, or determined asynchronously (using non-blocking collectives, a dedicated thread, or both) to be consistent with asynchronous programming models.

4 BAD Experimental Methods

To evaluate our design, we characterized the run time behavior of HIGRAD/FIRETEC, a computational fluid-dynamics model used at Los Alamos National Laboratory to study the multi-dimensional interactions between fire and its environment [8] using a Python-based simulation package. The simulator, modeling HIGRAD/FIRETEC, models a square number of processes, `Job Size`. Each process computes on a region of a two-dimensional virtual grid (e.g. representing a wildfire burning through a forest). Initially, each process's time is spent in a calculation phase, which we refer to as a timestep. Each processes simulated time in each timestep value is calculate by randomly adjusting `Compute Time` with a variance between +/-`Compute Variance` (*i.e.* between 5 to 7 seconds). Following the calculation, the processes immediately enter a message passing phase with a configurable number of neighbors, `Comm Size`. This messaging synchronizes all processes within a communication group

The simulated processes repeat this cycle `Timesteps Per Ckpt` times and then simulate the time required to create a checkpoint. In the BSP simulation, the processes barrier before each checkpoint. The checkpoint time is `Checkpoint Time` without any variance (we explain the reasoning behind this decision later). The complete workload finishes after `Runtime` checkpoints. The final important detail of the simulator is that the initially randomly assigned `Compute Time` values are periodically shifted by some number of cells every some number of timesteps; this simulates the movement of *hot spots* throughout the computational grid. Table 1 summarizes each simulation input parameter, describes the application characteristic the parameter is designed to control, and defines the default value used as input to the simulator. Default values are based on observations of HIGRAD/FIRETEC.

Figure 1 shows this basic workflow for two checkpoint phases comparing them for BSP and BAD-Check. In this contrived example, the reason that BAD-Check finishes earlier is that there is large compute variance and that the fastest process in the first checkpoint phase becomes the slowest process in the second. As described earlier, BAD-Check cannot significantly reduce the runtime for workloads with frequent global communication. Our simulation experiments are designed to identify and quantify the degree to which various asynchronous program characteristics enable the BAD-Check protocol to improve overall performance by allowing checkpoint phases to proceed without synchronization.

5 BAD Results

Using parameter sweeps of each of these application characteristics, we have quantified the types of workloads which can benefit from BAD-Check as well as quantifying the degree to which they can do so. Visualizing how BAD-Check improves total throughput, Figure 4 shows a small sample of snapshots of application progress through thirty checkpoint phases using BAD-Check on the left and BSP on the right. Each successive downward frame shows elapsed progress from the one above. Within each frame each process is shown as a colored cell with its position within the 2D computational grid corresponding to its position in the x and y axes. The height of each cell shows that process' progress towards workload completion; the numbers on the z-axes represent the checkpoint phase. The color of each cell represents each process's progress relative to each other; red for fast and blue for slow. When all cells are red, they are converged; when all are blue, they have not yet begun.

The first two rows show that BAD-Check and BSP perform identically until they reach their first checkpoint at which

PARAMETER	DESCRIPTION	DEFAULT
`Compute Time`	The simulated amount of compute time within each timestep.	6 seconds
`Compute Variance`	The maximum randomized difference between compute timesteps.	16.7%
`Timesteps Per Ckpt`	The simulated number of timesteps between checkpoints.	45
`Checkpoint Time`	The simulated checkpoint latency.	30 seconds
`Checkpoint Variance`	The maximum randomized difference between checkpoints.	0%
`Runtime`	The total simulated runtime.	300 Checkpoints
`Job Size`	The number of processes in the simulated 2D compute grid.	1024^2
`Hotspot Movement`	The speed of hotspot movement within the 2D compute grid.	1:1
`Comm Size`	The size of peer groups exchanging data between timesteps.	Neighbors

Table 1: **Simulation Parameters.** *This table lists characteristics which affect a workload's ability to benefit from BAD-Check, as well as the default values used in the simulation study. Defaults were derived from observations of the HIGRAD/FIRETEC application [8]. Notice that the defaults result in the application checkpointing for approximately 10% of its runtime as is typical of large parallel applications [17, 20].*

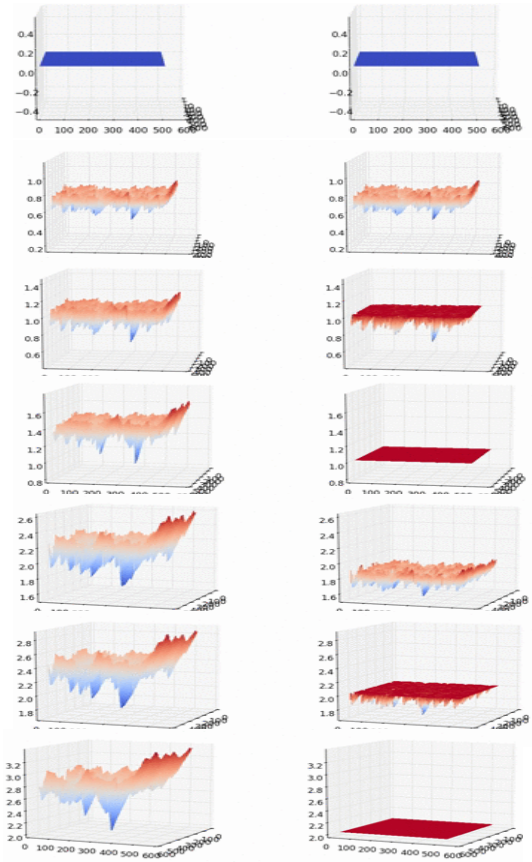


Figure 4: **Time Elapsed Progress.** *These pictures show application progress (from top to bottom) through thirty checkpoints using BAD-Check on the left and BSP on the right. The y-axis shows the current checkpoint; notice how the frequent BSP synchronizations (i.e. the flat planes) slow its progress. These pictures are extracted frames from a full video available here: http://johnbent.com/badcheck.gif*

point each process in BAD-Check continues at its current rate whereas the fast processes in BSP must wait in order to converge with their slower siblings as is shown in the fourth row. Following the initial checkpoint, it is obvious how the BAD-Check processes continue to benefit from asynchrony whereas the BSP processes are more tightly coupled.

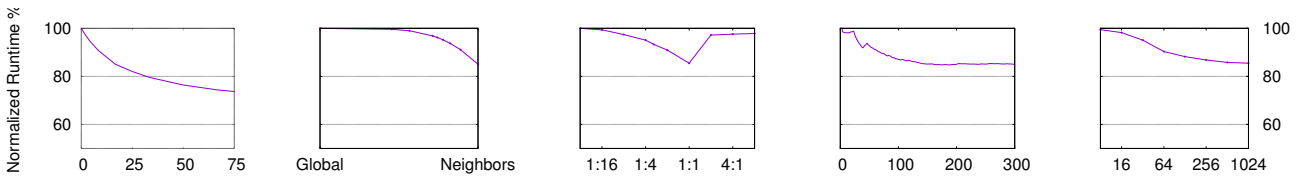
5.1 BAD Application Characteristics

The results of our simulated measurements are shown in Figure 5. For all figures, the y-axis shows the normalized total runtime of BAD-Check relative to BSP (lower is better). For all experiments, the values for all variables with the exclusion of the particular independent variable were set to their default values as shown in Table 1.

Figure 5a studies the effect of computational variance between the processes and shows that BAD-Check is only beneficial when there such variance exists. This is intuitive because processes that proceed at the same rate will arrive at the checkpoint phase simultaneously thereby preventing any asynchrony from entering the system. Other experiments, not shown, included checkpoint variance, which furthers improve the performance gains possible with BAD-Check. However, to focus our study on computational variance we eliminated checkpoint variance from these experiments.

Figure 5b similarly shows that BAD-Check relies on the cooperative processes becoming skewed. In this figure, the x-axis is the number of siblings with whom messages are exchanged during the compute phase. Since message passing is synchronous and blocking for most message passing interfaces, when the application does global message passing, all processes effectively proceed through the computation at the same rate and arrive simultaneously at each checkpoint. Conversely, local message passing in which data is only passed to immediate neighbors, as is done in many applications including HIGRAD/FIRETEC, allows the skew integral to BAD-Check to build within the system.

Figure 5c graphs the third of our experiments measuring the importance of skew. In this graph, the x-axis is the speed with which hotspots move throughout the computation. Maximum benefit from BAD-Check is possible when hotspots move neither too slowly nor too quickly relative to the checkpoint frequency. When hotspots move too slowly (as on the left-side of the graph), then total runtime cannot be improved since the processes which start slow never become fast and will never converge with their faster siblings. Conversely, hotspots which move too quickly also reduce skew: when every process is both slow and fast within the same compute phase, they effectively arrive simultaneously at the check-



(a) Compute Variance (%) (b) Comm Size (c) Hotspot Movement (cells:timestep) (d) Runtime (# checkpoints) (e) Job Size ($sqrt(np)$)

Figure 5: **BAD Performance.** *The ability of BAD-Check to reduce workload runtimes depending on different workload characteristics.*

point. In our experiments, hotspots that moved at the rate of the checkpoints maximized runtime improvements.

Figures 5d and 5e show the scalability of BAD-Check as a function of both runtime and job size respectively. Larger jobs benefit more as there will be more skew and longer running jobs benefit more than shorter running jobs until they flatten at the maximum benefit which is related to the average skew across processes.

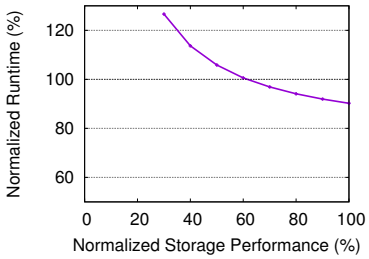


Figure 6: **BAD Efficacy.** *This graph shows how BAD-Check can be used either to improve performance or to reduce the capabilities of the storage system without reducing performance.*

The previously described experiments examined the potential for performance improvements. We now consider another possible benefit of BAD-Check which is to reduce the capability of the storage system (thereby reducing the total cost of ownership of the storage) without sacrificing performance. These results are shown in Figure 6. As before the y-axis is the normalized runtime; the x-axis however is the normalized capability of the storage system. Using our default parameters, we find that the capability of the storage system can be reduced by 40% without sacrificing performance.

Although many may assume that applications with these characteristics are rare, our simulations, based on observations of the HIGRAD/FIRETEC application, achieved up to 18% improvements in total runtime. With more extreme computational variance, we measured runtime improvements up to 27%. To understand the significance of this performance improvement we presented our findings to members of the HIGRAD/FIRETEC team; one of whom responded,

> " We have taken serious measures (under synchronous io) to ensure great load-balancing. However in multiphysics scenarios (e.g. Monte Carlo radiation calculation in fire scenarios, or what is essentially lagrangian particle tracking for wind turbine blade elements) we can turn the difficult algorithm-disparity, load balancing nightmare into a benefit, by dynamically allowing those types of tasks to be performed by the fastest arriving cores and providing some overlap time for slow arriving cores to catch up. Essentially we could create the same types of imbalance we struggle so hard to circumvent now. " *LANL scientist.*

This note confirms our suspicion that reducing computational variance is difficult. Fortunately, by using uncoordinated checkpointing, reducing computational is not required in order to construct consistent checkpoints. Further, we believe that efforts to reduce this variance are increasingly futile as the scale, processing heterogeneity, and jitter of systems increases in the pursuit of exascale.

However, modifying applications to use uncoordinated checkpoint remains significantly challenging. A better method for adopting efficient, uncoordinated checkpoint techniques, of which BAD-Check is one, is to integrate them into emerging task-based parallel programming models such as Legion [2] that embrace asynchrony.

6 Conclusions

> "He's a big bad wolf in your neighborhood;
> not bad meaning bad but bad meaning good."
> *Run DMC, from 'Peter Piper'*

We have presented a design of an alternative storage interface which uses distributed asynchronous transactions to allow uncoordinated checkpointing. With simulated experiments, we determined the characteristics which allow workloads to benefit from uncoordinated checkpointing:

1. Computational variation exists between checkpoints.
2. This variation moves from one process to another.
3. Message exchange is not globally synchronous.

References

[1] E. Barton, J. Bent, and Q. Koziol, "Fast forward storage and io program documents," in *LLNS subcontract no. B599860 For Extreme-Scale Computing Research and Development (Fast Forward) Storage and I/O*, 2014. [Online]. Available: https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[3] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, April 2012, pp. 1–5.

[4] 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 *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[5] J. Bent, B. Settlemyer, N. DeBardeleben, S. Faibish, D. Ting, U. Gupta, and P. Tzelnic, "On the non-suitability of non-volatility," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. Santa Clara, CA: USENIX Association, Jul. 2015. [Online]. Available: https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/bent

[6] B. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach," in *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*, Oct 1988, pp. 3–12.

[7] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 1–17. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522712

[8] J. J. Colman and R. R. Linn, "Separating combustion from pyrolysis in higrad/firetec," *International Journal of Wildland Fire*, vol. 16, no. 4, pp. 493–502, 2007. [Online]. Available: http://dx.doi.org/10.1071/WF06074

[9] R. Cook, E. Dube, I. Lee, C. Shereda, F. Wang, and L. Nau, *Survey of Novel Programming Models for Parallelizing Applications at Exascale*, Nov 2011. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1107306

[10] A. Hammouda, A. Siegel, and S. Siegel, "Overcoming asynchrony: An analysis of the effects of asynchronous noise on nearest neighbor synchronizations," in *Solving Software Challenges for Exascale*, ser. Lecture Notes in Computer Science, S. Markidis and E. Laure, Eds. Springer International Publishing, 2015, vol. 8759, pp. 100–109. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-15976-8_7

[11] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 302–311. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29740-3_34

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[13] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpoint for parallel programs," in *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/99163.99173

[14] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[15] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. New York, NY, USA: ACM, 2008, pp. 15–24.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[17] NERSC and the Alliance for Computing at Extreme Scale, *Trinity / NERSC-8 Request for Proposal*, 2013. [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/

[18] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, United States, May 2013, pp. 19–28. [Online]. Available: https://hal.inria.fr/hal-00781532

[19] B. Nicolae and F. Cappello, "Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462918

[20] Oak Ridge, Argonne, and Livermore National Labs, *CORAL Request for Proposal B604142*, 2014. [Online]. Available: https://asc.llnl.gov/CORAL/

[21] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844162

[22] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 55–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

[23] B. Randell, "System structure for software fault tolerance," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 437–449. [Online]. Available: http://doi.acm.org/10.1145/800027.808467

[24] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges, "Alleviating scalability issues of checkpointing protocols," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 18:1–18:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389021

[25] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733