Exploring Reliability of Exascale Systems through Simulations

Dongfang Zhao*, Da Zhang*, Ke Wang*, Ioan Raicu*†
***Department of Computer Science, Illinois Institute of Technology, Chicago, IL**
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
{dzhao8, dzhang32, kwang22}@hawk.iit.edu, iraicu@cs.iit.edu

Keywords: Exascale Computing, Checkpointing, Fault Tolerance, Parallel Filesystems, Distributed Filesystems

Abstract

Exascale computers are predicted to emerge by the end of this decade with millions of nodes and billions of concurrent cores/threads. One of the most critical challenges for exascale computing is how to effectively and efficiently maintain the system reliability. Checkpointing is the state-of-the-art technique for high-end computing system reliability that has proved to work well for current petascale scales. This paper investigates the suitability of checkpointing mechanism for exascale computers, across both parallel filesystems and distributed filesystems. We built a model to emulate exascale systems, and developed a simulator, RXSim, to study its reliability and efficiency. Experiments show that the overall system efficiency and availability would go towards zero as system scales approach exascale with checkpointing mechanism on parallel filesystems. However, the simulations suggest that a distributed filesystem with local persistent storage would offer excellent scalability and aggregate bandwidth, enabling efficient checkpointing at exascale.

1. INTRODUCTION

Exascale computing [1, 2], i.e. 10^{18} FLOPS, is predicted to emerge by the end of 2018 with current trend. Millions of nodes and billions of concurrent data access are expected with the exascale. This degree of computing capability is similar to that of a human brain and will enable the unraveling of significant scientific mysteries and present new challenges and opportunities. The US President made the building of exascale systems a top national priority, stating that it will "dramatically increasing our ability to understand the world around us through simulation and slashing the time needed to design complex products such as therapeutics, advanced materials, and highly-efficient autos and aircraft" [3].

One of most critical challenges for exascale computing is how to maintain the exascale computer reliable. Failures are unavoidable in high end computing (HEC) systems, making the partially-done work useless if no recovery mechanism exists. The reliability of a system is how strong the system is to prevent failures and/or recover after a failure. With millions of nodes and billions of cores and concurrent requests, keeping the entire exascale system reliable is extremely hard.

In order to bring the system back into the last correct state, checkpointing records system's (correct) states periodically. These states need to be stored on the persistent storage, because otherwise they are gone permanently if the system encounters a failure. Checkpointing is a general mechanism to maintain system's reliability, which is independent of any particular system. The fact of dealing with persistent storage implies potentially huge overhead. Therefore, besides other factors like how frequently to save the states, the question of improving the checkpointing degenerates to the question of how to elevate the storage I/O bandwidth.

In general, there are two major approaches to checkpointing in distributed systems. The first one is called coordinated checkpointing, where all nodes work together to establish a coherent checkpoint. The second one, called Communication Induced Checkpointing (CIC), allows nodes to make independent checkpoints on their local storage. Current HEC systems, e.g. IBM BlueGene/P supercomputer, adopt the first approach to write states to the parallel filesystem on the network attached storage (NAS). Applying CIC is not a viable option in this case, since no local storage is available on the work node of BlueGene/P.

To answer the question how the current checkpointing mechanism would work for exascale systems among other HEC systems, we built a model to emulate exascale systems, designed and implemented a simulator RXSim (Reliability of eXascale computers by Simulation) to study its reliability and efficiency. Results show that, unfortunately, current checkpointing mechanism on parallel filesystems is incapable to effectively recover the system from failures. However, it suggests that a distributed filesystem with local persistent storage, e.g. [4], would offer an excellent scalability and aggregate bandwidth, which in turn enables efficient checkpointing at exascale.

The contributions of this paper are:

1. *Proposed a simple and effective model to simulate large-scale high end computing system, particularly at exascale levels.*
2. *Designed and implemented a simulator to simulate checkpointing performance with different filesystem architectures (parallel and distributed).*
3. *Extensive evaluation of both synthetic workload and real IBM BlueGene/P logs demonstrates that state-of-the-art*

parallel filesystems would not be able to support exascale checkpointing, and distributed filesystems would scale well and delivers high application efficiency and system reliability.

The remainder of this paper is structured as follows. Section 2 describes the assumptions and empirical specifications we use to build the model of a million-node exascale system. We present the design and implementation of RXSim in Section 3. Section 4 presents the evaluation results. We review some related work in Section 5. Section 6 concludes the paper and discusses future work.

2. MODELING THE HEC SYSTEMS

Application Efficiency is defined as the ratio of application up time over the total running time:

$$E = \frac{up\_time}{running\_time} \times 100\%,$$

where *running_time* is the summation of up time, checkpointing time, lost time and rebooting time. **Up time** is when the job is correctly running on the computer. **Checkpointing time** is when the system stores the correct states on persistent storage periodically. **Lost time** measures the time when a failure occurred, the work since the last checkpointing would be lost and needs to be recalculated. **Rebooting time** is simply the time for the system to reboot the node.

Optimal Checkpointing Interval is the optimal checkpointing interval as modeled in [5]:

$$OPT = \sqrt{2\delta(M+R)} - \delta,$$

where δ is the checkpointing time, M is the system mean-time-to-failure (MTTF) and R is the rebooting time of a job.

Memory Per Node is modeled as the following based on the specifications of IBM BlueGene/P. When the system has fewer than 64K nodes, each node has 2GB memory. For larger systems, the per-node memory is calculated (in GB) as

$$2 \cdot \frac{\#nodes}{64K}.$$

We have two different models of **Storage Bandwidth** for parallel filesystems (PFS) and distributed filesystems (DFS), respectively, since they have completely different architectures. We assume PFS is the state-of-the-art parallel filesystem used in production today, e.g. GPFS [6], whose bandwidth (in GB/sec) is modeled as

$$BW_{PFS} = \frac{\#nodes}{1000}.$$

And for DFS, it is a hypothetical new storage architecture for exascale. There are no real implementations of a DFS that can scale to exascale, but this study should be a good motivator towards investing resources to the realization of DFS at exascale. The bandwidth of DFS in our simulation has the following bandwidth

$$BW_{DFS} = \#nodes \cdot (\log \#nodes)^2.$$

These equations are based on our empirical observations on the IBM BlueGene/P supercomputer.

For rebooting time, DFS has a constant time of 85 seconds because each node is independent to other nodes. For PFS, the rebooting time (in seconds) is calculated as the following:

$$\lceil 0.0254 \cdot \#nodes + 55.296 \rceil,$$

which is also based on the empirical data of the IBM Blue-Gene/P supercomputer. The above formulae indicate that DFS has a linear scalability of checkpointing bandwidth, whereas PFS only scales sub-linearly. The sub-linearity of PFS checkpoint bandwidth would prevent it from working effectively for exascale systems.

3. DESIGN AND IMPLEMENTATION

For any job running on an HEC system, RXSim has three states: running, repairing, restarting, as shown in Figure 1. This transmission works as follows: 1) when a job is running, repairing or restarting, if a failure occurs then the job will be hanged and enters repairing state; 2) after repaired, the job will restart, i.e. reboot nodes occupied by this job; 3) after the job completes, it restarts its nodes; 4) after restarting, if job is just repaired from a failure then the job continues its work; otherwise the job has completed its work.

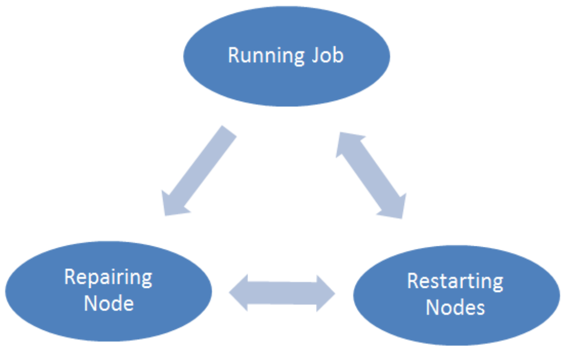


Figure 1. State transmission of RXSim.

RXSim is implemented in Java with only less than 2K lines of code, and will be released as an open source project. Some key modules include job management, node management, and time stamping. We will discuss each of them respectively in the following subsections.

3.1. Job Management

The job management module is used to keep tracks of any job-related information during the run time. Every job in the workload is an instance of the *Job* class. A job has common attributes like *jobID*, *walltime*, *size*, *endTime*, and some state variables, like *state_up*, *state_repair*, etc.

We do not keep the entire workload globally. Rather, each time the generator generates a new job, it is inserted into the running queue. Once a job is completed, the allocated nodes are restarted and released.

3.2. Node Management

Because the workload is randomly assigned work nodes, and there may be many jobs running on the HEC system at the same time, nodes need to have the information on which jobs are running on them. This is implemented by adding a *jobID* attribute to the *Node* class. Nodes management is analogous to traditional memory management.

An array is fulfilled with instances of *Node* class, to keep all information e.g. node ID, working state, etc. A free list is to keep and track all idle parts of the HEC system, so that each time a job requests some computing resources (nodes), RXSim will first check if there are enough idle nodes left. If so, RXSim retrieves the first idle part of the HEC system and keeps doing so till the job gets enough nodes. After a job is completed, the nodes occupied by this job will not be released immediately. These nodes would be occupied by the completed job until they are successfully rebooted.

3.3. Time Stamping

TimeStamp class is the event class where each *timeStamp* instance is an event with some information related to time stamping. For example, if simulator encounters a failure at some point, it creates a *timeStamp* instance including the incident's time, type, node ID, job ID.

There are four types of TimeStamp:

- Job ends successfully: with time and job ID.
- Job recovers: with time and job ID.
- Node reboots successfully: with time and job ID.
- Node failure: with time and node ID.

The TimeStamp queue is implemented as a TreeSet. The benefit of TreeSet is that it will automatically sort the data, so it is easy to retrieve the latest event from this queue. An obvious drawback of TreeSet is that its elements are hard to be modified. Unfortunately modification is a frequent operation since the simulator needs to update events in a regular basis. To fix the problem, we maintain another list called *uselessEventList*, which keeps tracks of all idle TimeStamp. The simulator would simply skip such an idle TimeStamp and try to retrieve the next available one.

4. EVALUATION

Experiments can be categorized into three major types. We first compare RXSim results to existing valid results with the same parameters and workload to verify RXSim. Then variant workloads are dispatched on RXSim to study the effectiveness and efficiency of checkpointing at different scales of HEC systems. Lastly, we apply RXSim on a 8-month log of an IBM BlueGene/P supercomputer, and emulate the checkpointing at exascale. Metrics *Uptime*, *Check*, *Boot* and *Lost* refer to the definitions of **up time**, **checkpointing time**, **rebooting time** and **lost time**, respectively, defined in Section 2.

4.1. Experiment Setup

The single-node MTTF is set to 1000 years, optimistically, as claimed by IBM. We assume it takes 0 second (again, very optimistically) to repair a single node. Simulation time is set to 5 years, where each time step is 1 second.

4.2. Validation

[7] shows how the applications look like for 3 cases: No-Checkpointing, PFS with checkpointing, and DFS with checkpointing, as shown in Figure 2.

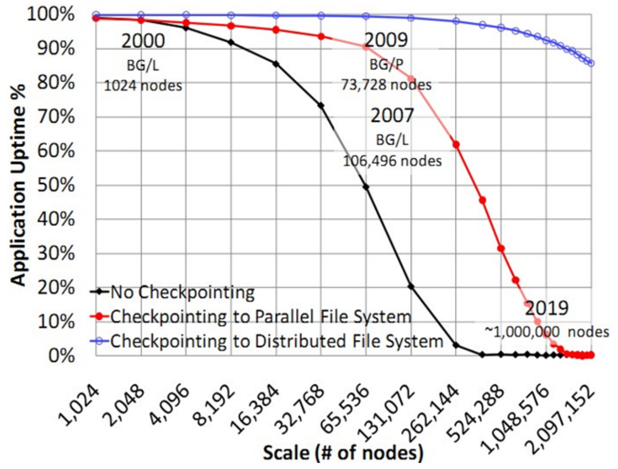


Figure 2. Comparison between checkpointings on different filesystems [7].

The result of RXSim with the same workload of Figure 2 is shown in Figure 3. RXSim result is quite close to the published results: two lines have negligible difference, which is only due to the random variables used in the simulator.

Figure 4 shows the system reliability with checkpointing disabled. As we can see, the system is basically not functioning beyond 400K nodes.

Figure 5 shows the system reliability when enabling checkpointing on a PFS. We observe that the system up time is much longer than Figure 4. This is expected, since checkpointing proves to be an effective mechanism to improve system reliability. However, the efficiency is quite low ($< 10\%$)

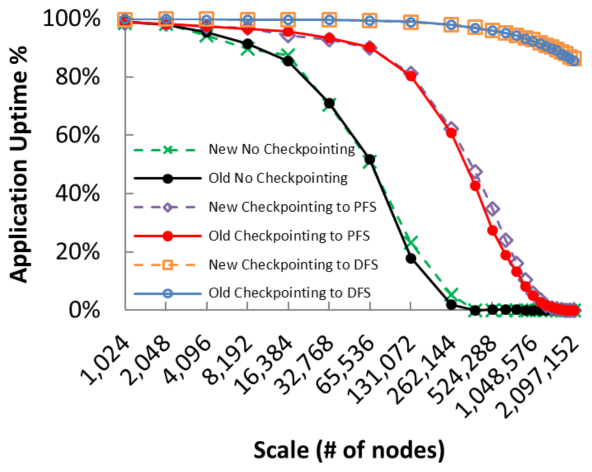


Figure 3. Comparison between RXSim and [7]

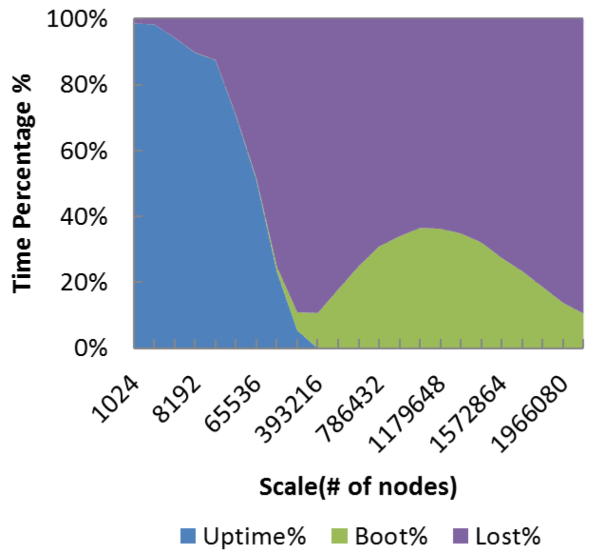


Figure 4. A no-checkpointing system stops functioning when having more than 400K nodes.

at exascale (1 million nodes), meaning that PFS is not a good choice for checkpointing.

In Figure 6, we show the trends of system MTTF, the overhead of doing a checkpoint, and the checkpointing circle (summation of checkpoint time and optimal checkpointing interval) in PFS. When system MTTF becomes less than the checkpointing circle time (which is the case for 1 million nodes), it basically means the system does not have enough time to complete one round of checkpointing. In other words, the system cannot recover from failures: checkpointing helps nothing but adding more overhead.

For DFS with checkpointing, we see an excellent application efficiency and scalability, as shown in Figure 7. The uptime portion is still as high as 90% for exascale.

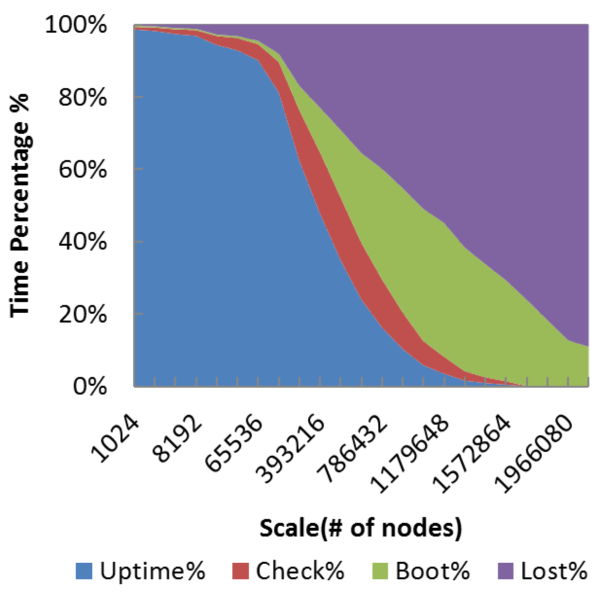


Figure 5. System reliability when enabling checkpointing on a PFS: efficiency is quite low (< 10%) at exascale (1 million nodes).

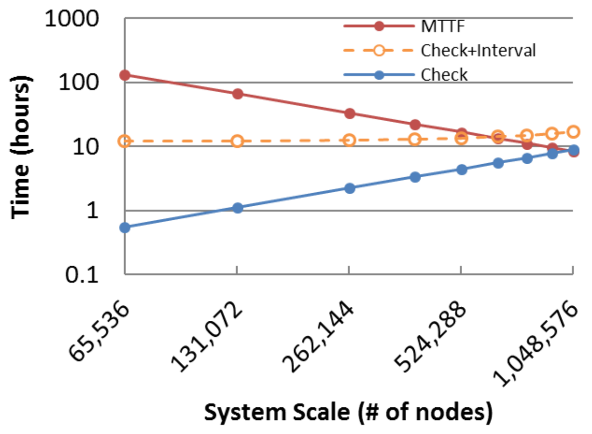


Figure 6. Trends of System MTTF, the overhead of doing a checkpoint, and the checkpointing circle: checkpointing fails to recover the system for 1M nodes.

As shown in Figure 8, DFS is perfectly fine to allocate enough time slice for checkpointing at exascale. This can be best explained by the fact that DFS has less checkpointing overhead when writing to the local storage as opposed to NAS (networked attached storage).

4.3. Synthetic Workloads

We carried out two more workloads with different job size and wall time on RXSim in this subsection.

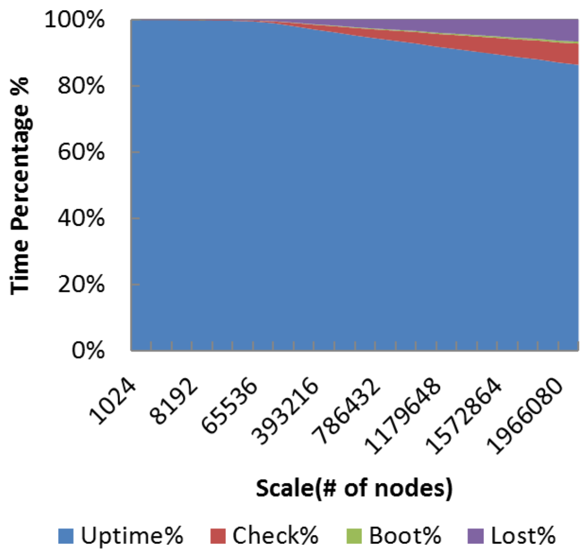


Figure 7. System reliability when enabling checkpointing on a DFS: excellent uptime and scalable to beyond exascale systems.

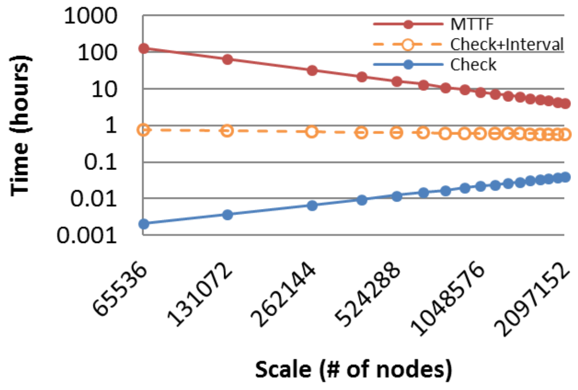


Figure 8. System reliability when enabling checkpointing on a DFS: excellent uptime and scalable to beyond exascale systems.

4.3.1. 1/10 Job Size

In this workload, each job size is 1/10 of the full system scale and wall time is set to 7 days. The efficiency is generally better than full system scale (see Figure 3). In particular, PFS with checkpointing on 1 million nodes is improved from 5% to 70%. This fact demonstrates that the major bottleneck of PFS is the shared storage between jobs. The more concurrent jobs trying to access the shared storage, the less efficient PFS becomes.

We will show more details of time breakdown for each case. For no-checkpointing, not surprisingly, the major two portion of costs are up time and lost time, as shown in Figure 10.

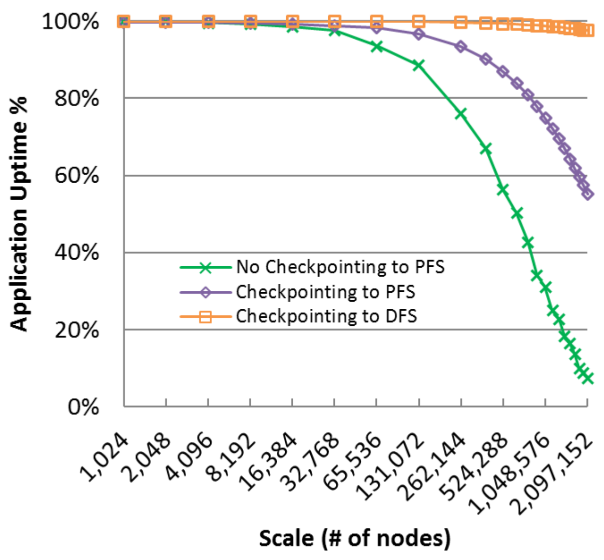


Figure 9. Comparison of different checkpointings with 1/10 job size.

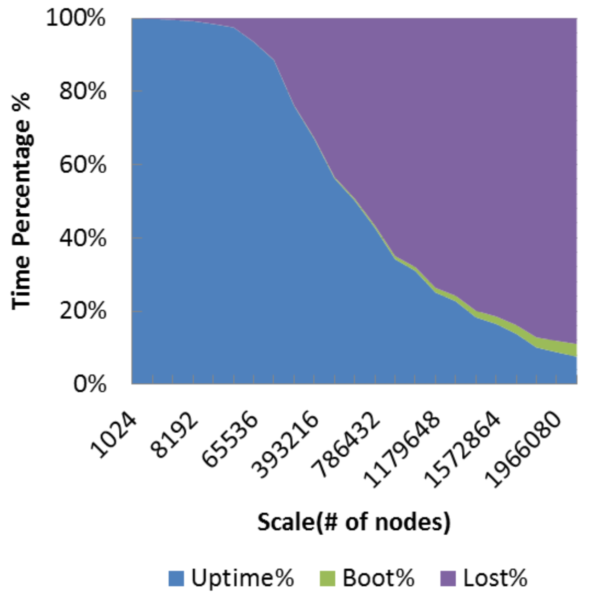


Figure 10. Cost breakdown of a no-checkpointing filesystem with 1/10 job size.

The cost breakdown for PFS with checkpointing is shown in Figure 11. Now the up time and checkpointing overhead are the top two portions, as expected.

We further examine how PFS with checkpointing would behave for system recovery. As shown in Figure 12, there is still a significant gap between MTTF and checkpointing time, which suggests that PFS with checkpointing might work well for 1/10 job size. In particular on 1 millions nodes, the MTTF is about 10 hours and the checkpointing takes only 1 hour.

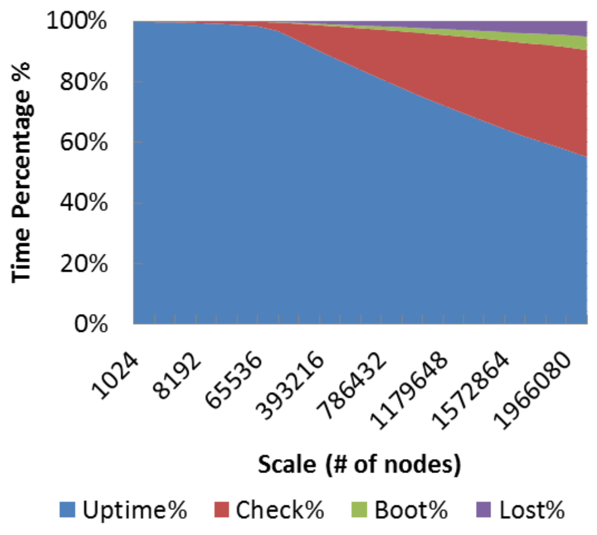


Figure 11. Cost breakdown of a PFS with checkpointing for 1/10 job size.

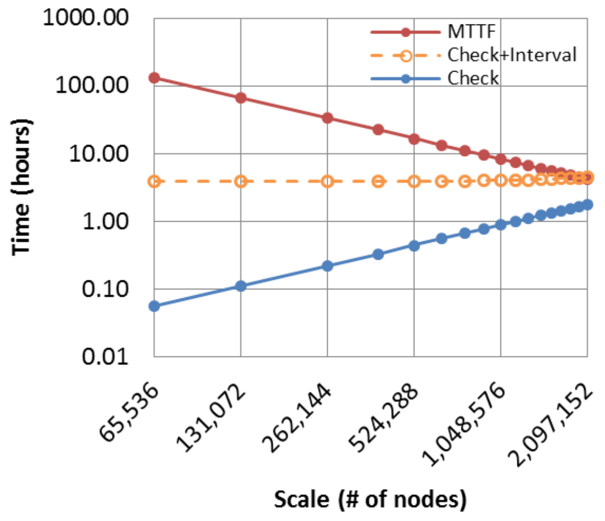


Figure 12. Trends of MTTF and checkpointing time for PFS: PFS with checkpointing might work well for 1/10 job size.

At last we show how DFS deals with 1/10 job size by plotting the cost breakdown. The result is shown in Figure 13. The up time is dominant, and keeps taking over 95% percentage even for 2 million nodes.

4.3.2. One-Day Wall Time

This workload keeps the job size as the full system scale, shortening the wall time from 7 days to 1 day. We compare these relatively short jobs in 3 different filesystems as shown in Figure 14. Again, DFS outperforms other two and keeps high efficiency of 90% on 1 million nodes. However, PFS is

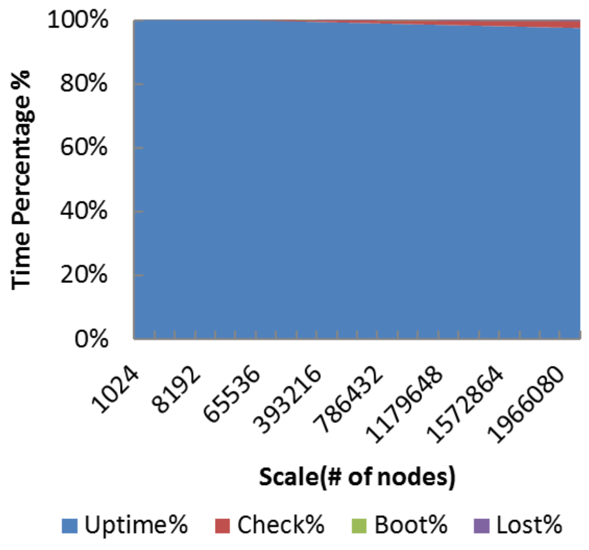


Figure 13. Cost breakdown of the DFS with checkpointing for 1/10 job size: up time is dominant.

degenerated to be worse than no-checkpointing.

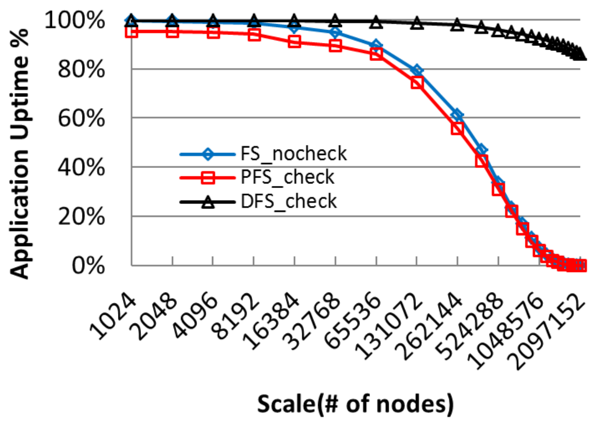


Figure 14. Efficiency of different checkpointing scenarios for short jobs: DFS works fine, but PFS with checkpointing does not help improve up time.

We show the cost breakdown of PFS and no-checkpointing in Figure 15 and Figure 16 respectively, in order to investigate why PFS has such poor performance. The cost distributions of both cases are about the same, except for PFS has some additional time spent on checkpointing which only takes a small portion ($< 10\%$). The reason is most likely that the wall time of each job was much shorter, which implies less lost-time during a failure. The checkpointing interval and checkpointing overhead are quite sensitive to wall time, therefore shortening jobs dramatically hurts the application efficiency of PFS with checkpointing. The implication, in order, is that, for short jobs no-checkpointing might do as equally well as

PFS with checkpointing enabled.

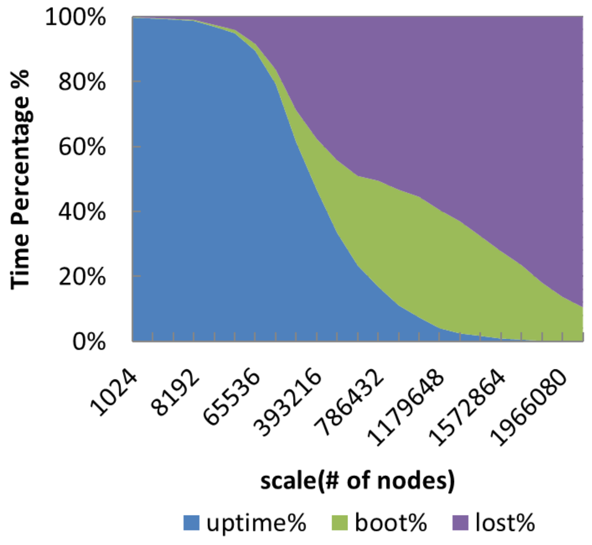


Figure 15. Cost breakdown of no-checkpointing filesystem for 1-day jobs.

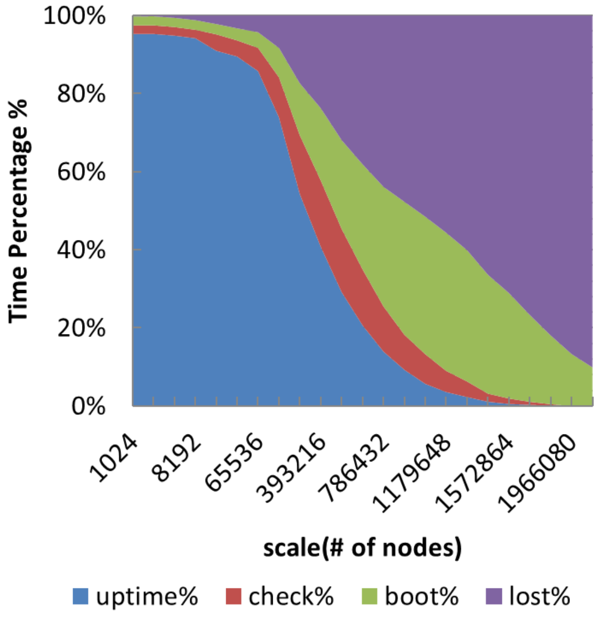


Figure 16. Cost breakdown of PFS with checkpointing for 1-day jobs.

4.4. Real Logs of IBM BlueGene/P

We carried out experiments on real workloads (8-month log) from IBM BlueGene/P supercomputer (a.k.a. Intrepid) at Argonne National Laboratory (ANL). Intrepid has a peak of 557TFlops, has 40 racks, and comprises 40960 quad-core nodes (163840 cores in total), associated I/O nodes, storage servers (NAS), and high bandwidth torus network interconnecting compute nodes. It debuted as No.3 in the top 500 supercomputer list released in June 2008.

The log in the experiment contains 8 months of accounting records of Intrepid. The log data is in swf (standard workload format). We scaled the job size on the log in proportion to scale RXSim from 1024 to 2 million nodes. Note that the data beyond 160K nodes are predicted by RXSim, since the BlueGene/P only has 160K cores.

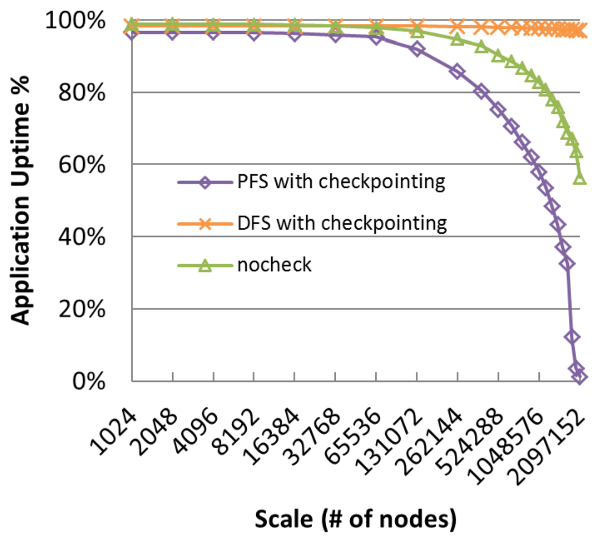


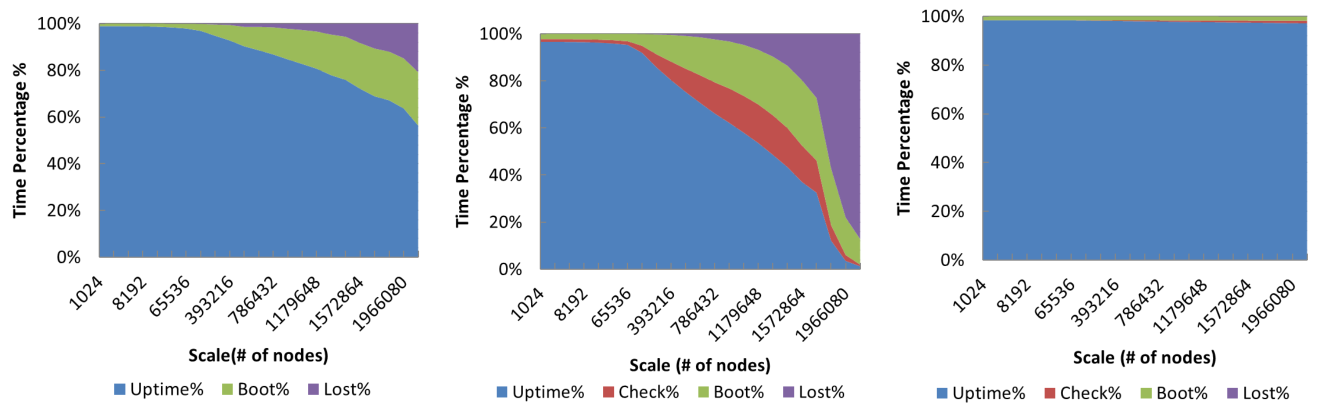
Figure 17. Application efficiency for BlueGene/P jobs.

Figure 17 shows that a no-checkpointing filesystem outperforms PFS with checkpointing, which is counter intuitive at the first glance. The reason is that the BlueGene/P jobs have an average wall time of 5k seconds, which is less than 2 hours and far shorter than 1 day in Figure 14. So this result in fact justifies our previous conclusion that short jobs would badly hurt the application efficiency by enabling checkpointing.

We show the cost breakdown of different filesystems in Figure 18. PFS has a significant portion of checkpointing overhead, booting time and lost-time in exascale ($\geq$ 1 million nodes), as shown in Figure 18(b). DFS, on the other hand, introduces negligible overhead ($<$ 5%) in Figure 18(c).

5. RELATED WORK

Tikotekar et al. [8] developed a simulation framework to evaluate different fault tolerance mechanisms (checkpoint/restart for reactive fault tolerance, and migration for pro-active fault tolerance). The framework uses system failure logs for the simulation with a default behavior based on logs taken from the ASC White at LLNL. A non-blocking checkpointing mode is proposed in [9] to support optimal parallel discrete event simulation. This model allows real con-



(a) No checkpointing (b) PFS with checkpointing (c) DFS with checkpointing

Figure 18. Cost breakdown of BlueGene/P: DFS delivers an excellent efficiency from 1024 to 2 million nodes; PFS performs poorly, even worse than disabling checkpointing.

currency in the execution of state saving and other simulation specific operations (e.g. event list update, event execution), with the aim at removing the cost of recording state information from the parallel application. An incremental checkpointing/restart model is built in [10], which is applied to the HPC environment. The model aims at reducing full checkpointing overhead by performing a set of incremental updates between two consecutive full checkpoints. Some recent research was focused on XOR-based methods, for example, [11] proposed reliable and fast in-memory checkpointing for MPI programs and [12] presented a distributed checkpointing manner using XOR operations. None of these related works explored exascale systems, and none addressed the checkpointing challenges through a different storage architecture (e.g. distributed file systems).

6. CONCLUSION AND FUTURE WORK

In this paper we simulated exascale systems with different filesystem architectures to study the reliability. We developed RXSim to simulate checkpointing performance for exascale computing. RXSim suggests distributed filesystems are more optimistic than state-of-the-art parallel filesystems for reliable exascale computers. In particular, we found that local persistent storage would be dramatically helpful to leverage data locality in the context of traditional distributed filesystems. Our study shows that local storage would be one of the key points to succeed in maintaining the reliability for exascale computers. The results are coincident with the findings in [13], where a hybrid of local/global checkpointing mechanism was proposed for the projected exascale system.

The next step of this work is to apply the RXSim-suggested architecture to the checkpointing module of a scalable distributed filesystem called FusionFS [4]. FusionFS is a FUSE-based distributed filesystem, deployed on the local non-volatile memory (NVM) of each work node in a supercomputer. FusionFS+RXSim will bring a new understanding on reliable exascale computers.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation (NSF) under grant OCI-1054974. This research used resources of the Argonne Leadership Computing Facility (ALCF), which is supported by the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] Michael A. Heroux. Software challenges for extreme scale computing: Going from petascale to exascale systems. *Int. J. High Perform. Comput. Appl.*, 23(4):437–439, November 2009.

[2] Josep Torrellas. Architectures for extreme-scale computing. *Computer*, 42(11):28–35, November 2009.

[3] Barack Obama. A strategy for american innovation: Driving towards sustainable growth and quality jobs. National Economic Council, 2009.

[4] Dongfang Zhao and Ioan Raicu. Distributed File Systems for Exascale Computing. In *Doctoral Research, Supercomputing '12*, Salt Lake City, UT, 2012.

[5] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *ICCS*, pages 3–12, Berlin, Heidelberg, 2003.

[6] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[7] Ioan Raicu, Ian T. Foster, and Pete Beckman. Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance*, LSAP '11, pages 11–18, New York, NY, USA, 2011. ACM.

[8] Anand Tikotekar, Geoffroy Vallee, Thomas Naughton, Stephen L. Scott, and Chokchai Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 303–311, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Francesco Quaglia and Andrea Santoro. Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):593–610, June 2003.

[10] Nichamon Naksinehaboon, Yudan Liu, Chokchai (Box) Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen 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*, CCGRID '08, pages 783–788, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Gang Wang, Xiaoguang Liu, Ang Li, and Fan Zhang. In-memory checkpointing for mpi programs by xor-based double-erasure codes. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 84–93, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Ge-Ming Chiu and Jane-Ferng Chiu. A new diskless checkpointing approach for multiple processor failures. *IEEE Trans. Dependable Secur. Comput.*, 8(4):481–493, July 2011.

[13] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2):6:1–6:29, June 2011.