SAND2011-2634C

Evaluating the Viability of Process Replication Reliability for Exascale Systems

Kurt Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, and Ron Brightwell
Scalable System Software Department
Sandia National Laboratories*
{kbferre | jrstear | jhlaros | raoldfi | ktpedre | rbbrigh}@sandia.gov

Rolf Riesen
IBM Research, Ireland
rolf.riesen@ie.ibm.com

Patrick G. Bridges and Dorian Arnold
Department of Computer Science
University of New Mexico
{bridges | darnold}@cs.unm.edu

ABSTRACT

As high-end computing machines continue to grow in size, issues such as fault tolerance and reliability limit application scalability. Current techniques to ensure progress across faults, like checkpoint-restart, are increasingly problematic at these scales due to excessive overheads predicted to more than double an application's time to solution. Replicated computing techniques, particularly state machine replication, long used in distributed and mission critical systems, have been suggested as an alternative to checkpoint-restart. In this paper, we evaluate the viability of using state machine replication as the primary fault tolerance mechanism for upcoming exascale systems. We use a combination of modeling, empirical analysis, and simulation to study the costs and benefits of this approach in comparison to checkpoint/restart on a wide range of system parameters. These results, which cover different failure distributions, hardware mean time to failures, and I/O bandwidths, show that state machine replication is a potentially useful technique for meeting the fault tolerance demands of HPC applications on future exascale platforms.

1. INTRODUCTION

Process replication, generally referred to as *state machine replication* [37], is a well-known technique for tolerating faults in systems that target high-availability. In this approach, a process's state is replicated such that if the process fails, its replica is available or can be generated to assume the original process's role without disturbing the other application processes. Process replication can be costly in terms of space if replicas have dedicated resources or time if replicas are co-located with other primary processes. However, process replication can dramatically increase an application's mean time to interrupt (MTTI). Additionally, variants of this technique can detect or correct faults that do not crash a process but instead cause it to yield incorrect results [7].

Primarily due to its high costs, process replication has been examined only limitedly for high performance computing (HPC) systems [41]. Instead, HPC applications have relied primarily on rollback recovery techniques [13], particularly coordinated checkpoint/restart, where application state is periodically written to stable storage (checkpointed), and when failures occur, this state is used to recover the application to a previously known-good state. However, future exascale systems are expected to present a much more challenging fault tolerance environment to applications than current systems [4]. Additionally, recent studies conclude that for these systems, high failure rates coupled with high checkpoint/restart overheads will render current rollback-recovery approaches infeasible. For example, several independent studies have concluded that potential exascale systems could spend more than 50% of their time reading and writing checkpoints [12, 29, 39].

Such concerns have motivated the exploration of new techniques to enhance the scalability of checkpoint/restart [8, 21, 28]. Most of these approaches require additional system resources such local storage devices, inter-node communication, or memory capacity compared to the traditional approach. Researchers also have been reexamining the applicability of existing fault-tolerance mechanisms such as process replication in light of the fault tolerance challenges in upcoming exascale systems [39, 42, 15].

In this paper, we examine the viability of the process replication paradigm as a *primary* exascale fault tolerance mechanism, with checkpoint/restart providing *secondary* fault tolerance when necessary. Our goal is to understand the advantages and limitations of this approach for extreme scale computing systems. We focus on exascale MPI applications: redundant copies of MPI processes provide failover capabilities, which allow these applications to *transparently* run through most errors without the need for rollback. Checkpoint/restart augments our process replication scheme in

*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 EnergyâĂŹs National Nuclear Security Administration under contract DE-AC04-94AL85000.

cases where process replication is insufficient, for example, if all replicas of a process crash simultaneously or become inconsistent due to faults corrupting the machine state.

To summarize, we present a study of redundant computing for exascale applications to address the scalability concerns of disk-oriented coordinated checkpoint/restart techniques (Section 2.1) and the inability of checkpoint/restart methods to tolerate undetected hardware errors and non-crash failures (Section 2.2). Our results show that redundant computing should be considered as a viable approach for exascale systems because:

- Even at system scales less than those projected for exascale systems, our *model-based analysis* shows that process replication's hardware overheads are less than those of traditional checkpoint/restart (Section 5);
- Based on a full implementation MPI process replication that has been evaluated on more than 4000 nodes of a Cray XT-3/4 system, our *empirical analysis* shows that process replication overheads are minimal for real HPC applications (Section 6);
- Additional *simulation-based analysis* that includes both software and hardware overheads shows that process replication is a viable alternative to traditional checkpoint/restart on systems with more than 20,000 sockets (Section 7), depending on system checkpoint I/O bandwidth and per-socket mean time between failures (MTBF); and,
- Alternative approaches to scaling checkpoint/restart for exascale systems have hardware requirements comparable to replication-based approaches (Section 8).

2. BACKGROUND

2.1 Disk-based Coordinated Checkpoint/Restart

2.1.1 Current State of Practice

Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing systems for at least the last 30 years. In current large distributed memory HPC systems, this approach generally works as follows:

1. Applications periodically quiesce all activity at a global synchronization point, for example a barrier;
2. After synchronization, all nodes send some fraction of application and system state, generally comprising most of system memory, over the network to dedicated I/O nodes;
3. These I/O nodes store received checkpoint information data to stable storage, currently hard disk-based storage;
4. In the event of application crash, the stored checkpoint can be used to restart the application at a prior known-good state.

The dominance of coordinated checkpoint-restart as the *primary* HPC fault tolerance technique rests on a number of key assumptions that have thus far remained true:

1. Application state can be saved and restored much more quickly than a system's mean time to interrupt (MTTI);
2. The hardware and upkeep (e.g. power) costs of supporting frequent checkpointing are a modest portion (currently perhaps 10-20%) of the system's overall cost; and
3. System faults that do not crash (fail-stop) the system are very rare.

Heroic work in both I/O systems and hardware error detection has largely kept these assumptions valid through the present day, allowing large parallel applications to scale to over a petaflop of sustained performance in the face of occasional fault-induced system crashes.

2.1.2 Scaling of Coordinated Disk-based Checkpoint/-Restart

There are a number of excellent studies investigating the efficiency of disk-based checkpoint/restart for large scale systems, including past petascale systems [12, 30, 14] and upcoming exascale systems [39, 2, 4]. These projections invalidate essentially *all* of the assumptions on which traditional checkpoint/restart depend. For example, these studies suggest exascale MTTIs ranging from 3-37 minutes, checkpoint times for *low memory* systems taking up to five hours, and significant non-crash system failures, for example undetected DRAM errors.

We begin by assuming a 15-minute checkpoint time, a modest improvement over the approximate 20 minute checkpoint time that supercomputers both a decade ago (ASCI Red [26]) and today (BlueGene and Jaguar [6]) achieve. We also assume a one hour system MTTI, again a generous assumption given recent studies. Daly's model [10] estimates that such a system should checkpoint once every 27 minutes and would achieve only 44% system utilization. Scaling the I/O system to achieve a utilization greater than 80% would require checkpoint times of approximately one minute. Assuming that the I/O system supporting that 15 minute checkpoint took only 10% of the system's original budget and I/O throughput scaled up perfectly, a simple Amdahl's law calculation shows that an I/O supporting such checkpoint speeds would comprise 63% of the total cost of the system!

Checkpoint-restart is also problematic when dealing with non-crash failures such as so-called "soft errors". In particular, checkpoint-restart *preserves* the impact of failures that corrupt application state. Addressing this would require application developers to either restart an application from scratch or analyze the contents of their checkpoints looking for one prior to when the fault that corrupted application state occurred.

2.2 State Machine Replication

Redundant computation, process replication, and state machine replication have long histories and have been used extensively in both distributed [17, 9] and mission critical systems [27, 3, 31, 37] as a technique to improve fault tolerance. State machine replication, the focus of this paper, maintains one or more replicas of each node and assumes every node computes deterministically in response to a given external input, for example a message being received. It then uses an ordering protocol to guarantee all replicas see

the same inputs in the same order, and additional communication to detect and recover from failures.

State machine replication offers a different set of tradeoffs compared to rollback recovery techniques such as checkpoint/restart. In particular, it completely masks a large percentage of system faults, preventing them from causing application failures *without the need for rollback*. Some forms of state machine replication can also be used to detect and recover from a wider range of failures than checkpoint/restart, potentially including Byzantine failures [7]. Unlike checkpoint/restart, however, state machine replication is not sufficient by itself to recover from all node crash failures; faults that crash all of a node's replicas will cause a computation to fail.

This approach has previously been dismissed in HPC as being too expensive for the meager benefits that are seen at present machine scale. For the reasons described above in Section 2.1, however, several authors have recently suggested using this technique in HPC systems [39, 42, 15]. In the remainder of this paper, we examine the suitability of a specific type of state machine replication in HPC systems.

3. REPLICATION FOR MESSAGE PASSING HPC APPLICATIONS

3.1 Overview

State machine replication is conceptually straightforward in message passing-oriented HPC applications. In this approach, each replica is created on independent hardware for every processor rank in the original application of which failure cannot easily be tolerated. Note that we do not require *all* ranks to be replicated—in master/slave-style computations where the master can recover from the loss of slaves, only the master might be replicated.

The replication system then guarantees that every replica receives the same messages in the same order and that a copy of each message from one rank is sent to each replica in the destination rank. In addition, the replication system must detect replica failures, repair failed nodes when possible, and restart failed nodes from active replicas. The replication system may also periodically check that replicated ranks have the same state.

Checkpoint/restart recovery is still required in this approach for recovery from faults that fail *all* replicas of a particular process rank. It is also used to recover from situations where replica state becomes inconsistent, for example due to silent (undetected) failures.

3.2 Costs and Benefits

This approach requires significantly increased computational resources—at least double the hardware for replicated ranks. In cases where only portions of an application must be replicated, these requirements are potentially modest. For many HPC applications (e.g. traditional stencil calculations), however, this approach *doubles* the required hardware—$2N$ nodes are required to fully replicate a job that would otherwise run (perhaps much more slowly) on N nodes. In addition, there are runtime overheads for maintaining replica consistency.

With these costs, however, come significant advantages:

- **Dramatically increased system MTTI.** This approach dramatically reduces the number of faults visible to applications. Specifically, the application only sees faults that crash (or otherwise fail) *all* replicas of a particular rank.

- **Significantly reduced I/O requirements.** Increased system MTTI reduces the speed at which checkpoints must be written to storage to allow applications to effectively utilize the system. A smaller fraction of the system cost and power budget must as a result be spent on the I/O system.

- **Detection of "soft errors".** By comparing the state of multiple replicas (e.g. using memory checksums) prior to writing a checkpoint, replication can detect if application state has been corrupted and trigger restart from a previous checkpoint.

- **Increased system flexibility.** The extra nodes used for redundant computation when running the largest jobs can be used for providing extra system *capacity* when running multiple smaller jobs for which fault tolerance is less of a concern. A system that uses N nodes and an expensive I/O system to reach exascale can only run 100 10PF jobs at a time, for example. A system that uses $2N$ nodes and a less expensive I/O system to reach exascale, however, can potentially run 200 10PF jobs at a time.

4. EVALUATING REPLICATION IN EXASCALE SYSTEMS

The advantages described in Section 3 provide a compelling reason to examine the viability of state machine replication for extreme-scale HPC systems. Without quantifiable performance benefits compared to other approaches, however, state machine replication will not be viable for use in exascale systems. The remainder of this paper therefore examines the performance costs and benefits of state machine replication.

4.1 Comparison Approach

Our primary performance evaluation criteria is; at what node counts, if any, state machine replication provides quantitative performance *advantages* over past approaches particularly in terms of system utilization, *after* accounting for the overheads of state machine replication. If, for example, state machine replication achieves 46% utilization at a given system socket count and another technique only achieves 40% system utilization, we regard state machine replication as superior at that point.

We use traditional checkpoint/restart fault tolerance as the baseline technique against which to compare because its performance characteristics are well-understood. We hope that comparing against a well-understood baseline will facilitate future comparisons against other proposed exascale fault tolerance techniques as their costs and benefits at scale are more fully quantified. A brief qualitative comparison with several such techniques, however, is provided in Section 8.

4.2 Assumptions

Because we are comparing a new technique on projected hardware systems, our comparisons make a number of assumptions that are important to make explicit:

1. Full dual hardware redundancy for all applications, resulting in a maximum possible efficiency for state machine replication of 50%.

2. The MPI library is the only potential source of nondeterminism in the application.

3. Machines suffer only crash failures, not more general failures from which checkpoint/restart may not be able to recover.

4. System MTTI decreases linearly with increased system socket count which, based on past study results [38].

5. MODEL-BASED ANALYSIS

We first examine the performance benefits of state machine replication compared to its fundamental redundant hardware costs. For this initial comparison, we assume every MPI process is replicated, and make very simple assumptions about system characteristics, particularly that there is no software overhead for maintaining replica consistency, that the system can checkpoint in a fixed amount of time regardless of scale, and that all failures follow a simple exponential distribution. These assumptions will be successively relaxed in the following sections.

When two nodes are used to represent the same MPI rank, the failure of one node in a pair does not interrupt the application. Only when both nodes fail does the application need to restart. The frequency of that occurring is much lower than the occurrence of a single node fault and can be characterized using the well-known *birthday problem*.

One version of the birthday problem asks how many people on average need to be brought together until there are enough to have a greater than 50% chance that two of them share the same birth month and day. If we equate days in a year with nodes and let the number of people represent the faults occurring, we can use the birthday problem to calculate how many faults can occur until both nodes in a pair are damaged and cause an application interrupt.

Equation 1, from [25, 20], shows how to calculate this version of the birthday problem.

$$F(n) = 1 + \sum_{k=1}^{n} \frac{n!}{(n-k)! \cdot n^k} \approx \sqrt{\frac{\pi n}{2}} + \frac{2}{3} \qquad (1)$$

Essentially, replicas act like a filter between the system and an application, and the birthday problem helps us estimate how many faults can be absorbed before the application is interrupted.

Figure 1 estimates the resulting application efficiency with optimal checkpoint intervals for both state machine replication and using only traditional checkpoint/restart. MTTI was computed directly from the birthday problem approximation in Equation 1, while the resulting efficiency is computing using Daly's higher-order checkpoint/restart model and optimal checkpoint interval [10]. These calculations assume a 43800 hour (5 year) per-socket MTBF based on past studies [38, 19], 15 minute checkpoint times as discussed in Section 2.1, and a 168 hour application solve time.

These results show the dramatic increase in system MTTI that state machine replication provides, allowing it to maintain efficiency close to 50% as system socket count increases dramatically towards the 200,000 heavyweight sockets suggested for exascale systems [4]. In contrast, the efficiency of a checkpointing-only approach drops precipitously as system scales approach those of upcoming exascale systems.

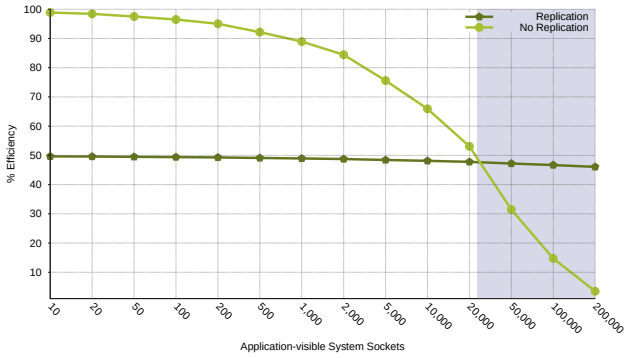


Figure 1: Modeled application efficiency with and without state machine replication for a 168-hour application, 5-year per-socket MTBF, and 15 minute checkpoint times. Shaded region corresponds to possible socket counts for an exascale class machine [4]

6. RUNTIME OVERHEAD OF REPLICATION

While the previous sections demonstrate that state machine replication is viable at exascale in terms of the basic hardware costs, they do not evaluate the runtime overhead of the necessary consistency management protocols. Transparently supporting state machine replication for MPI applications on HPC systems requires maintaining sequential consistency between replicas. It also requires protocols for detecting and repairing failures. As mentioned in Section 2.2, these are potentially expensive in communication-intensive HPC systems as every replica must see messages arrive in the same order.

To study this overhead, we designed and implemented *r*MPI, a portable user-level MPI library that provides redundant computation transparently to deterministic MPI applications. *r*MPI is implemented on top of an existing MPI implementation using MPI profiling hooks. In the remainder of this section, we outline the basic design and implementation of *r*MPI and measure the runtime overhead of this implementation for several real applications on a large scale Cray XT-3/4 system. A complete description of *r*MPI, including low-level protocol and implementation details is available elsewhere [16, 5].

6.1 *r*MPI Design

The basic idea for the *r*MPI library is simple: replicate each MPI rank in an application and let the replicas continue when an original rank fails. To ensure consistent replica state, *r*MPI implements protocols that assure identical message ordering between replicas. Unlike more general state machine replication protocols [37, 7], these protocols are specific to the needs of MPI in an attempt to reduce runtime overheads. In addition, *r*MPI uses the underlying Reliability, Availability and Servicablity (RAS) system to detect node failures, and implements simple recovery protocols based on the consistency protocol used.

6.1.1 *Basic Consistency Protocols*

rMPI implements two different consistency protocols, named mirror and parallel, to assure that every replica receives a copy of every message and to order message reception at replicas. Both protocols take special care when dealing with MPI operations that could potentially result in different message orders or MPI results being seen at different replicas. Note that collective operations in rMPI call the point-to-point operations internal to rMPI.

Figure 2(a) shows the basic organization of how the mirror protocol assures that all replicas see the same messages. In this figure, A and B represent distinct MPI ranks and A' and B' are A's and B's replicas respectively. In this protocol, each sender transmits duplicate messages to each of the destinations. Similarly, receivers must post multiple receives for the duplicate messages, but only require one of those messages to arrive in order to progress. This eases recovery after a failure, but doubles network bandwidth requirements.

The parallel protocol, in comparison, is shown in Figure 2(b). In this approach, each replica has a *single* corresponding replica for each other rank with which it communicates in non-failure scenarios. In the case of failure, one of the remaining replicas of a rank takes over sending and receiving for the failed node. This failure detection requires frequent message-based interaction with the reliability system on current systems. As a result, the parallel protocol will initiate approximately double the number of messages for each send operation. These extra messages contain MPI envelope information and are small. Therefore, the parallel protocol reduces network bandwidth requirements for an increased number of short messages.

6.1.2 MPI Consistency Requirements

rMPI assumes that only MPI operations can result in non-deterministic behavior, and there are a few specific MPI operations that can result in application-visible non-deterministic results. For example, rMPI must address non-blocking operations, wildcard (e.g. `MPI_ANY_SOURCE` and `MPI_ANY_TAG`) receives, and operations such as `MPI_Wtime`. As a first step, both rMPI protocols use the notion of a leader node for each replicated MPI rank, while non-leader nodes are referred to as replicas or redundant nodes. When a leader drops out of a computation, the protocol chooses a new replica from among those remaining for a rank to take over as leader. rMPI uses one high order bit in the tag to distinguish messages from leader and replica nodes.

For the remainder of consistency protocol discussions, we focus on the mirror protocol implementation; the parallel protocol implementation is generally similar and described in more depth elsewhere [16]. For blocking non-wildcard receives, one of the the most common forms of MPI communication, rMPI posts a receive for *both* senders A and A' into the buffer provided by the user. Since the data in the two arriving messages is identical, there is no danger of corrupting the user buffer. If multiple messages from the replica set A arrive with the same tag, rMPI must make sure that the first active and first redundant message arrive in the first buffer, and the second active and second redundant in the second buffer. rMPI achieves this by using one high-order tag bit, setting it on all outgoing redundant messages and setting the same bit for all receives of redundant messages.

Due to MPI message-passing semantics and the possibility of wildcard source receives, this basic approach is not completely sufficient. To handle `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, rMPI relies on explicit communication between the leader of each rank and other replicas. Essentially, rMPI allows only one actual wildcard receive to be posted at any time on a node, and then only on the leader. When a wildcard receive is matched, the leader then sends the MPI envelope information to replica nodes which then post for the actual message needed. The situation is more complicated for non-blocking wildcard receives, test, and wait operations, requiring a queue of outstanding wildcard receives, but the basic approach is similar.

Finally, rMPI must guarantee that operations such as `MPI_Wtime()` return the same value on active and redundant nodes, as some applications make decisions based on the amount of time elapsed. For these situations, the leader node sends its computed value to the redundant node. As an option, rMPI can synchronize the `MPI_Wtime()` clocks across the nodes [24].

6.1.3 Failure Detection

rMPI's failure detection requirements are relatively modest, and it uses the underlying supercomputer RAS system can provide much of this functionality. Both mirror and parallel protocols require that messages from failed nodes will be consumed and do not deadlock the network or cause other resources, such as status in the underlying MPI implementation, to be consumed. Furthermore, failing nodes must not corrupt state on other nodes. I.e., corrupted or truncated messages in flight must be discarded. Most systems already do this using CRC or other mechanisms to detect corrupt messages. The RAS system is responsible that the machine stops the retry of messages from and to failed nodes.

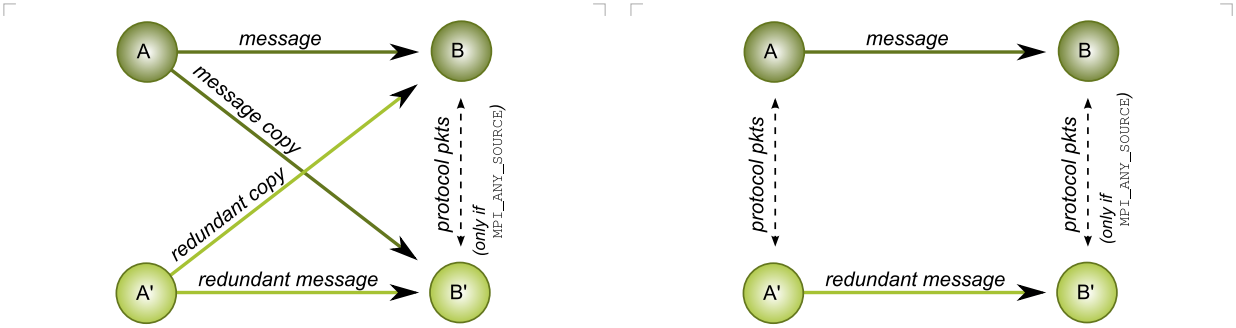
For the parallel protocol we expect that there is a method to learn whether a given node is available or has failed. On Red Storm we emulate a RAS system at the user-level. This is a table which rMPI consults, and the RAS system updates, when a node's status changes. It could also be an event mechanism that informs rMPI whenever the RAS system detects a failed node.

6.2 Evaluation

6.2.1 Methodology

From the discussion in the previous sections it should be clear that rMPI will add overhead and lengthen the execution time of an application. To measure this overhead we ran multiple tests with applications on the Cray Red Storm system at Sandia National Laboratories compiled with both rMPI and the original unmodified Cray MPI library. Red Storm is a XT-3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.2 GHz quad-core AMD Opteron processor and 8 GB of main memory. To ensure leader and replica are on separate physical nodes, and to avoid memory and bandwidth bottlenecks on the nodes themselves, we only used one CPU on each node.

We used four applications tested on up to 2,048 application-visible nodes (4096 total nodes in the case of replication): CTH [11], SAGE [23], LAMMPS [34, 35], and HPCCG [36]. These application represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads to both the US DOD and DOE. These four applications represent different communication characteristics and compute to communication ratios. Therefore, the overhead of rMPI affects them in different ways.



(a) Mirror Protocol **(b)** Parallel Protocol

Figure 2: Basic replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as `MPI_ANY_SOURCE`

Because a given node allocation may impact the performance of an application, we ran our tests in three different modes. The first mode, called *forward*, assigns rank $n/2$ as a redundant node to rank 0, rank $n/2+1$ to rank 1, and so on resulting in a mapping like this: ABCD|A'B'C'D'. *Reverse* mode is ABCD|D'C'B'A', and *shuffle* mode is a random shuffle (Fisher/Yates) such as ABCD|C'B'D'A'.

6.2.2 LAMMPS

Figure 3 shows the performance impact of *r*MPI with both the mirror and parallel protocol. The impact of each redundancy protocol is less than 5%, independent of the nodes used, while the baseline overhead for each is negligible.

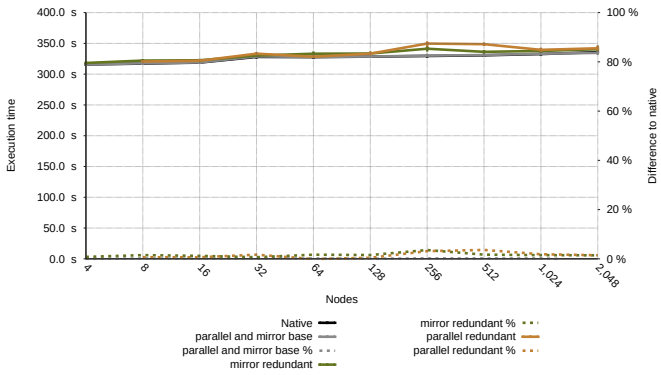


Figure 3: LAMMPS *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

6.2.3 SAGE

Figure 4 shows the *r*MPI performance for SAGE. Similar to LAMMPS, the baseline performance degradation is negligible. Also similar to LAMMPS, the parallel protocol performance remains nearly constant and performance decrease is negligible in the tested node range; with performance overhead generally less than 5%. In contrast, full redundancy for the mirror protocol loses about 10% performance over native, with performance increasing with scale. We attribute the performance degradation for SAGE to the factor of two increase of large network messages sent by SAGE and the limited available network bandwidth.

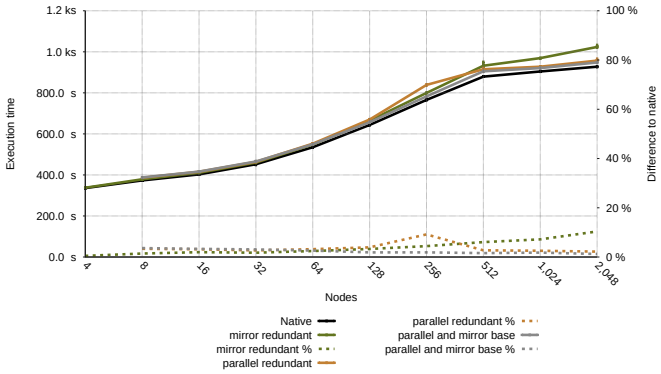


Figure 4: SAGE *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

6.2.4 CTH

In Figure 5 we see the impact of our consistency protocols for CTH at scale. Again, baseline for both mirror and parallel shows little performance difference. For CTH, mirror has the greatest impact on performance with full redundancy. This impact, which is nearly 20% at the largest scale, is due to CTH's known sensitivity to network bandwidth [32] (the greatest of each of the applications tested) and the increased bandwidth requirements of the mirror protocol. Interestingly, the parallel protocol version of CTH runs slightly *faster* then the native versions (around 5-8%) for forward, reverse, and shuffle replica node mappings. Though further testing is needed, current performance analysis results suggest this decrease in application runtime is due to parallel reducing the number of unexpected messages received.

6.2.5 HPCCG

Figure 6 shows the performance impact of *r*MPI on the HPCCG mini-application. In contrast to the other results presented in this section we present the mirror and parallel results separately. Though the results presented in Fig-

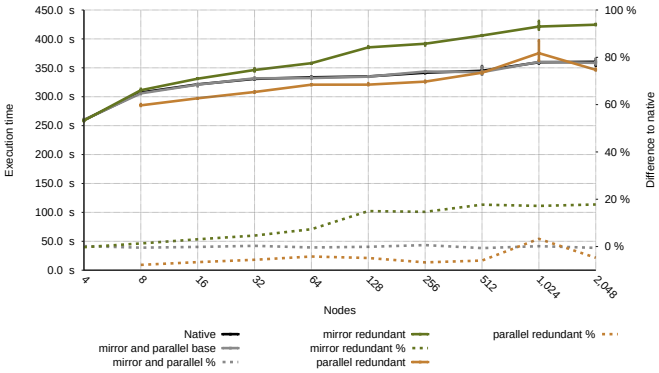


Figure 5: CTH *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

ure 6(a) and Figure 6(b) represent the same computational problem, the native results of each vary due to different node allocations between the two plots. Allocation issues aside, we see that mirror has very little impact. Parallel on the other hand shows a significant impact at higher node counts, with slowdowns of around 10% at 1,024 nodes. Also, in contrast to all the other applications tested, impact from the parallel protocol is greater than that of mirror. This is because unlike other applications, HPCCG stresses the system's message rate and parallel's synchronization messages are causing it to reach the maximum messaging rate of a node.

6.3 Analysis and Summary

Our results evaluating the runtime overhead of state machine replication show that the runtime costs of implementing state machine replication for a wide range of production HPC applications at significant scale is minimal. In particular, for each application either the parallel or mirror protocol provides almost negligible performance impact. Examining the best protocol for each application, SAGE has the highest net overhead, 2.2% at 2048 application-visible nodes. A logarithmic curve can be fit to the overhead for this worst-case, with the fit curve shown in Equation 2.

$$g(S) = \frac{1}{10} \log S + 3.67 \quad (2)$$

This curve would result in a 4.9% additional overhead on a projected exascale system with 200,000 sockets.

7. SIMULATION-BASED ANALYSIS

7.1 Overview

In this section, we use a simulation-based approach to verify, integrate, and expand the results from the previous sections into a more complete analysis of the costs and benefits of state machine replication for HPC systems. This approach allows us to examine real failure distributions derived from studies of failures of real HPC systems in addition to the exponential distributions assumed analytical models such as those of the Daly model or the birthday problem. We also use it to examine additional machine parameters and their impact on the viability of state machine replication, particular variations in available I/O system bandwidth.

In the remainder of this section, all results assume software runtime overheads as shown in Equation 2; efficiency results also include a factor of two reduction for replication because of the required redundant hardware. Unless otherwise stated, we also continue to assume checkpoint and restart times of 15 minutes as in previous sections.

7.2 Combined Hardware and Software Overheads

As a first study, we reexamine state machine replication under exponential failure distributions with a 5 year per-socket MTTI as in Sections 2 and 5, but this time including projected software runtime overheads from Section 6. As can be seen in Figure 7, these results are similar to those of Figure 1, with the break-even point for state machine replication shifted to a somewhat higher socket count due to the additional software runtime overheads. Despite this, state machine replication still outperforms traditional checkpoint/restart at socket counts currently projected for use in exascale systems.

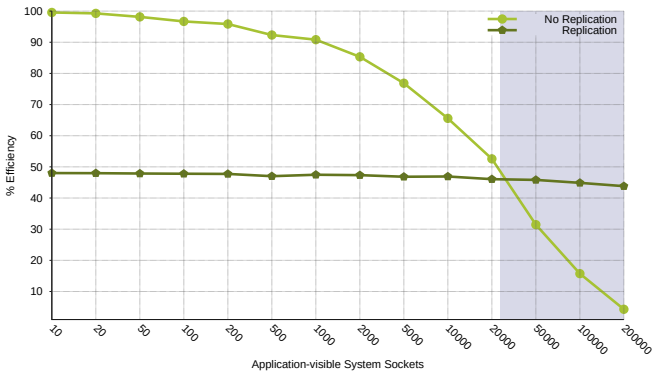
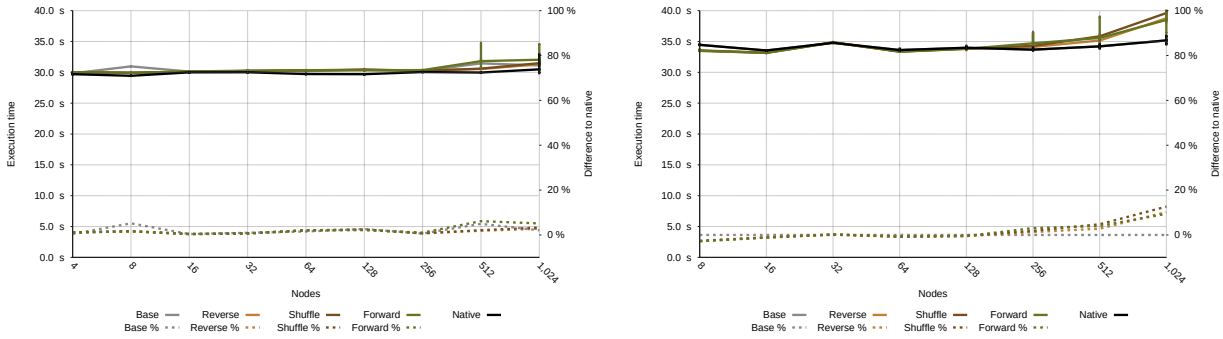


Figure 7: Simulated application efficiency with and without replication including *r*MPI run time overheads. Shaded region corresponds to possible socket counts for an exascale class machine [4]

7.3 Scaling at Different Failure Rates

While the 5 year per-socket MTBFs used above are based on well-known studies of large-scale systems, the challenges of exascale systems make changes to these reliability statistics likely. For example, more reliable nodes could be deployed to address fault tolerance concerns, or power conservation, miniaturization, or cost concerns could lead to a *reduced* per-socket MTBF. Because of this, we also examined the viability of state machine replication over a range of per-socket MTBFs.

This evaluation focuses on determining the *break-even point* in number of system sockets for state machine replication compared to traditional checkpoint/restart. This is the number of sockets above which state machine replication is more efficient than traditional checkpoint/restart even accounting for replication's software and hardware overheads. At socket counts greater than or MTBFs less than this break-even point, replication is preferable; at socket counts less than this or MTBFs above it, traditional checkpoint/restart is preferable.



(a) Mirror protocol

(b) Parallel protocol

Figure 6: HPCCG *r*MPI performance comparison. Varying performance for native and baseline between mirror and parallel protocols is due to different node allocations.

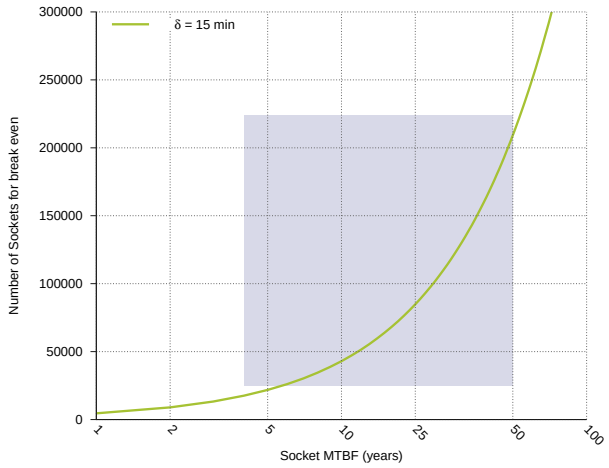


Figure 8: Simulated replication break even point assuming a constant checkpoint time (δ) of 15 minutes. Shaded region corresponds to possible socket counts and MTBFs for an exascale class machine [4]

Figure 8 shows these results for per-socket MTBFs up to 100 years; socket counts and per-socket MTBF commonly discussed for exascale systems (socket counts above 25,000 and MTBFs between 4 and 50 [4]) are shaded; the shaded area above and to the left of the break-even curve represents the portion of the exascale design space in which state machine replication is beneficial.

These results show that state machine replication is viable for a large range of socket MTBFs and node counts in the exascale design space, but not the entire space. In particular, state machine replication performs worse that traditional checkpoint/restart for low socket-count systems with MTBFs greater than about 10 years. For socket MTBF above 50 years, state machine replication is outperformed by traditional checkpoint/restart at all expected socket counts.

7.4 Scaling at Different Checkpoint I/O Rates

We also examined the viability of replication at a wide range of checkpoint I/O rates. Because checkpoint I/O is an area of active study, including work on a wide range of hardware and software techniques to improve its performance for exascale systems (as described later in Section 8), understanding the potential impact of this research on exascale fault tolerance approaches is critical.

For this analysis, we used recent modeling work which extends Daly's checkpoint modeling work to account for how variations in checkpoint system throughput impact checkpoint times and system utilization [29]. We assume each socket in the system has 16 GB of memory associated with it, and again examine the break-even point for replication over checkpoint/restart at a range of checkpoint I/O bandwidths and socket MTBFs. We choose an aggressive range of such bandwidths ranging from 500 GB/sec to 30 TB/sec to fully understand the impact of dramatic increases in I/O rates on the viability of replication.

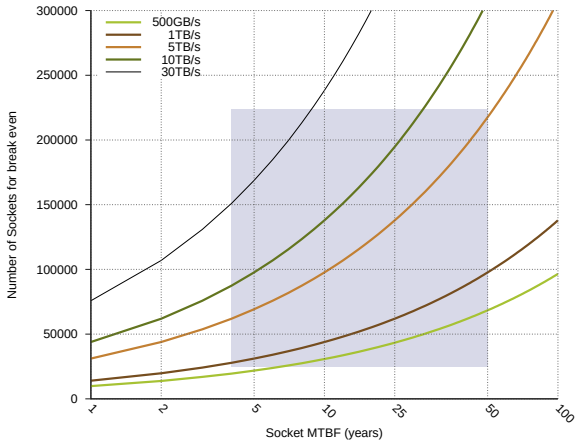


Figure 9: "Break even" points for replication for various checkpoint bandwidth rates. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [4]. State machine replication is a viable approach for most checkpoint bandwidths, but with a checkpoint bandwidth greater than 30TB/sec, replication is inappropriate for most of the exascale design space.

Figure 9 shows the results of this analysis. Replication outperforms check pointing for the vast majority of the exascale design space at checkpoint I/O bandwidths of 1 TB/sec or less. However, beginning at I/O bandwidths of approx-

imately 5 TB/sec, checkpoint/restart becomes competitive for a substantial fraction of the design space, particularly systems with high per-socket MTBFs and low numbers of sockets. At checkpoint bandwidths of 30 TB/sec or higher, several orders of magnitude faster than current I/O systems, checkpoint/restart is preferable across a large majority of the design space.

7.5 Non-Exponential Failure Distributions

Finally, we also examine the viability of replication with more realistic failure distributions. For failure information, we use numbers from a recent study of failures on two BlueGene supercomputer systems, a 16,384 node system at Rennesseler Polytechnic Institute (RPI) and a 4,096 node system at École Polytechnique Fédérale de Lausanne (EPFL) [19].

This study shows shows that failures in these systems are best described by a Weibull distribution with MTBFs of 6.6 hours (11.7 years/socket) and 8.4 hours (3.9 years/-socket), and shape (β) values of 0.156 and 0.469, respectively. These β values ($\beta < 1.0$) describe distributions that decrease in probability over time; in HPC systems, this indicates that failures are more likely to happen at the start of a system's lifetime or an application run and reduce in frequency as the system runs.

To examine the impact of these failure distributions, we build on the results of the previous subsection and examine how the efficiency of replication and checkpoint/restart change under Weibull failures assuming a fixed 1 TB/sec checkpoint bandwidth and 16 GB of memory per socket. Note that the systems from which these distributions were measured experienced a significant number of I/O system failures, and it is unclear how these failures should be properly scaled up to larger systems. As a result, we use the failure data from the larger of the two systems (the RPI system), and focus on how Weibull distributions change the efficiency of replication and checkpoint/restart approach as opposed to the specific efficiency crossover point.

Figure 10 presents impact of these failure distributions on both a replication-based approach and a purely checkpoint-based approach. These results show that Weibull failures experienced by real-world systems result in a much more challenging fault tolerance environment, reducing the effectiveness of both replication and traditional checkpointing approaches. However, replication is less severely impacted than traditional checkpointing, again pointing to the potential more viability of a replication-based fault tolerance approach for exascale systems.

8. ALTERNATIVE APPROACHES

In addition to state machine replication, a number of alternatives have also been suggested for scaling fault tolerance methods to HPC systems. Essentially all of these approaches attempt to improve the performance of checkpoint/restart. In the remainder of this section, we describe these approaches and briefly discuss their potential benefits and costs as an exascale fault tolerance methods.

8.1 High-speed Storage for Checkpoint/Restart

High speed local storage, for example local disk and flash memory systems has periodically been proposed to speed up checkpoint/restart systems by placing large amounts of high-speed storage near the data that must be checkpointed. The Exascale planning report cited earlier notes that placing spinning storage and a flash RAM in each system node would allow nodes to checkpoint in between 4 minutes and 1 second. This would in turn increase system utilization to from 59% to 97% ([4], Table 7.12, revised using Daly's second order model.)

Actually deploying large amounts of local non-volatile storage in an exascale system is potentially very challenging, however. Local disk-based storage has traditionally been avoided because of the increased failures it causes, for example. Upcoming non-volatile phase change PCRAM and resistive RRAM devices provide high bandwidth and reliability, but are potentially very expensive. Unless their cost per bit rivals that of DRAM, using such technologies for checkpoint/restart purposes would result in checkpointing hardware that makes up a much larger portion of the system cost.

Modern NAND and NOR flash technologies are potentially the most promising for buffering and storing local checkpoints because of their comparatively low cost, high density, and high reliability. NAND flash write bandwidths are currently in the low GB/s range, allowing them to checkpoint a node in a few minutes. Assuming that exascale MTTIs can be kept at or above one hour, this would result in system utilizations of 80% or higher. However, their write durability would require periodically replacing all flash memory in the system.

8.2 Asynchronous Checkpointing and Message Logging

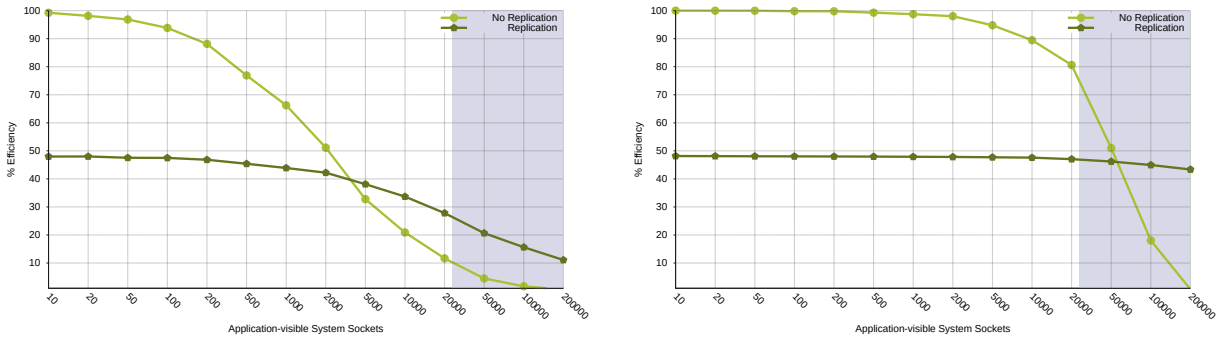
Another approach that has been suggested to improve the performance of checkpointing systems is uncoordinated or asynchronous checkpointing [1, 21, 22]. In these systems, nodes generally checkpoint and restore from local storage without the synchronization used by coordinated checkpointing. To support a node restoring from a local asynchronous checkpoint, nodes in this approach keep a log of recent messages that they have sent. When a node restores from a previous checkpoint, it can then replay reception of messages using remote nodes' logs.

While this approach can increase checkpointing performance, it also generally assumes the availability of local storage. In addition, logging increases the latency of messaging operations and potentially takes significant amounts of space. Finally, asynchronous checkpointing approaches can result in cascading rollbacks; recent work attempts to bound the amount of rollback that may be necessary [18], but also places non-trivial limits on application behavior. We are unaware of any studies examining performance of more general message logging approaches at large scales (e.g. thousands of nodes or larger).

8.3 Other Checkpointing Systems

Memory-based checkpointing [33, 40] uses the the memory of a remote machine to checkpoint node state. Unless node memory is primarily read-only (in which case RAID 5-like techniques can be used), this approach doubles the memory demands of an application. Since memory is regarded as a key budget and power constraint in exascale systems, it is unclear if the benefits of replicating only memory are superior to the qualitative advantages of state machine replication described in Section 3.

Multi-level checkpointing [28] is a library-based approach for controlling checkpointing to multiple storage targets, in-



(a) Weibull $\beta = 0.156$, socket MTBF = 12 years

(b) Exponential, socket MTBF = 12 years

Figure 10: Comparison of simulated application efficiency with and without replication, including *r*MPI run time overheads, using a Weibull and Exponential fault distribution. In both these figures we assume a checkpoint bandwidth of 1TB/sec. The shape parameter (β) 0.156 and MTBF corresponds to the RPI BlueGene system [19]. Shaded region corresponds to possible socket counts for an exascale class machine [4]

cluding memory-based checkpoints, local checkpoint storage, and remote checkpoints, into a single system. Because of this, it shares some of the advantages and disadvantages of memory-based checkpointing and local storage techniques. Unlike these techniques, however, multi-level checkpointing has the flexibility to choose between multiple levels of storage based on system design parameters, making it a promising technique for exascale systems.

9. SUMMARY AND FUTURE WORK

In this paper, we evaluated the suitability of replication, an approach well-studied in other fields, as the primary fault tolerance methods for upcoming exascale high performance computing systems. A combination of modeling, empirical evaluation, and simulation were used to study the various costs and benefits of state machine replication over a wide range of potential system parameters. This included both the hardware and software costs of state machine replication for MPI applications, and covered different failure distributions, system mean time to interrupt ranges, and I/O speeds.

Our results show that a state machine replication approach to exascale resilience outperforms traditional checkpoint/restart approaches over a wide range of the exascale system design space, though not the entire design space. In particular, state machine replication is a particularly viable technique for the large socket counts and limited I/O bandwidths frequently anticipated at exascale. However, replication-based approaches are less relevant for designs that have per-socket MTBFs of 50 years or more, less than 50,000 sockets, and checkpoint bandwidths of 30 terabytes per second.

Outside of its performance benefits, using replication as the primary exascale fault tolerance methods provides a number of other advantages. First among these is that it can be used to detect and aid in the recovery from faults that corrupt system state instead of crashing the system, sometimes referred to under the banner of silent errors. Checkpoint-based approaches, on the other hand, potentially preserve such errors. In addition, while the extra hardware nodes needed to support replication-based approaches can also be used to increase the capacity of exascale systems when it runs more but smaller (e.g. 1-10 petaflop-scale) jobs.

While the research described outlines most of the potential costs and benefits of HPC-oriented state machine replication, there is a wide range of additional work that remains. In terms of state machine replication, more work is needed quantifying the software costs of using replication to detect silent errors. While such techniques are well known in other communities, it is unclear what their cost would be for HPC applications; the quantitative results in this paper do not attempt to measure these costs and focus only on using replication to mask the pressing issue of frequent crash failures on exascale systems.

In addition, more detailed studies of the scaling, benefits, and hardware costs of the various alternative methods to scaling exascale fault tolerance described in Section 8 are needed. While state machine replication appears viable at exascale, other approaches may still be superior; careful investigation of such approaches is needed to understand their comparative costs and benefits.

Finally, we are investigating an alternative method to enable redundant computing that has a lower resource overhead then what is presented here. Rather than having a replica specific to a particular rank, we are looking into methods that would aggregate a number of replicated ranks onto one node and spread state throughout all of these aggregate replicas in a job. This would allow an aggregate replica, on demand, to replace a certain rank and could possibly allow the application to avoid checkpointing altogether.

10. REFERENCES

[1] Ahn, J. 2-step algorithm for enhancing effectiveness of sender-based message logging. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference* (2007), pp. 429–434.

[2] Amarasinghe, S., and et al. Exascale software study: Software challenges in extreme scale systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`, Sept. 2009.

[3] Bartlett, J. F. A nonstop kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles* (1981), pp. 22–29.

[4] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Kogge, P., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., and Yelick, K. Exascale computing study: Technology challenges in achieving exascale systems. `http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, Sept. 2008.

[5] Brightwell, R., Ferreira, K. B., and Riesen, R. Transparent redundant computing with mpi. In *EuroMPI* (2010), R. Keller, E. Gabriel, M. M. Resch, and J. Dongarra, Eds., vol. 6305 of *Lecture Notes in Computer Science*, Springer, pp. 208–218.

[6] Cappello, F. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA 23*, 3 (2009), 212–226.

[7] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS) 20*, 4 (Nov. 2002), 398–461.

[8] Chakravorty, S., and Kalé, L. V. A fault tolerant protocol for massively parallel systems. In *Proceedings of the International Parallel and Distributed Processing Symposium* (Santa Fe, NM USA, April 2004), IEEE Computer Society Press.

[9] Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. Hive: fault containment for shared-memory multiprocessors. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM, pp. 12–25.

[10] Daly, J. T. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst. 22*, 3 (2006), 303–312.

[11] E. S. Hertel, J., Bell, R. L., Elrick, M. G., Farnsworth, A. V., Kerley, G. I., McGlaun, J. M., PetneY, S. V., Silling, S. A., Taylor, P. A., and Yarrington, L. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves* (July 1993), pp. 377–382.

[12] Elnozahy, E., and Plank, J. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on 1*, 2 (Apr. 2004), 97–108.

[13] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (2002), 375–408.

[14] Elnozahy, E. N., Alvisi, L., Wang, Y. M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (SEP 2002), 375 – 408.

[15] Engelmann, C., Ong, H. H., and Scott, S. L. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the* 8^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009* (Innsbruck, Austria, Feb. 16-18, 2009), ACTA Press, Calgary, AB, Canada, pp. 189–194.

[16] Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., III, J. H. L., Pedretti, K., and Brightwell, R. rMPI: Increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, 2011.

[17] Gärtner, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys 31*, 1 (March 1999), 1–26.

[18] Guermouche, A., Ropars, T., Brunet, E., Snir, M., and Cappello, F. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium* (May 2011).

[19] Hacker, T. J., Romero, F., and Carothers, C. D. An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput. 69* (July 2009), 652–665.

[20] Holst, L. The general birthday problem. In *Random Graphs 93: Proceedings of the sixth international seminar on Random graphs and probabilistic methods in combinatorics and computer science* (New York, NY, USA, 1995), John Wiley & Sons, Inc., pp. 201–208.

[21] Jiang, Q., and Manivannan, D. An optimistic checkpointing and selective approach for consistent global checkpoint collection in distributed systems. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium* (Mar. 2007).

[22] Johnson, D. B., and Zwaenepoel, W. Recovery in distributed systems using asynchronous and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), pp. 171–181.

[23] Kerbyson, D. J., Alme, H. J., Hoisie, A., Petrini, F., Wasserman, H. J., and Gittings, M. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the ACM/IEEE conference on Supercomputing* (2001), pp. 37–48.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[25] Mathis, F. H. A generalized birthday problem. *SIAM Review 33*, 2 (1991), 265–270.

[26] Mattson, T. G., and Henry, G. An overview of the Intel TFLOPS supercomputer. *Intel Technology Journal*, Q1 (1998), 12.

[27] McEvoy, D. The architecture of tandem's nonstop system. In *ACM '81: Proceedings of the ACM '81 conference* (New York, NY, USA, 1981), ACM, p. 245.

[28] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[29] Oldfield, R. A., Arunagiri, S., Teller, P. J., Seelam, S., Varela, M. R., Riesen, R., and Roth, P. C. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on*

Mass Storage Systems and Technologies (Sept. 2007), pp. 30–46.

[30] Oliner, A. J., Sahoo, R. K., Moreira, J. E., and Gupta, M. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18* (2005), p. 299.2.

[31] Packard, H. HP NonStop computing. `http://h20338.www2.hp.com/NonStopComputing/cache/76385-0-0-0-121.html`.

[32] Pedretti, K. T., Vaughan, C., Hemmert, K. S., and Barrett, B. Application sensitivity to link and injection bandwidth on a Cray XT4 system. In *Proceedings of the 2005 Cray User Group Annual Technical Conference* (Helsinki, Finland, May 2008).

[33] Plank, J. S., Kim, Y. B., and Dongarra, J. J. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers* (Pasadena, CA, USA, June 1995), Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1995, pp. 351–360.

[34] Plimpton, S. J. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys 117*, 1 (1995), 1–19.

[35] Sandia National Laboratory. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Apr. 10 2010.

[36] Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo`, Apr. 10 2010.

[37] Schneider, F. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[38] Schroeder, B., and Gibson, G. A. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)* (June 2006).

[39] Schroeder, B., and Gibson, G. A. Understanding failures in petascale computers. *Journal of Physics: Conference Series 78*, 1 (2007), 012022.

[40] Silva, L. M., and Silva, J. G. An experimental study about diskless checkpointing. In *24th EUROMICRO Conference* (Vasteras, Sweden, August 1998), IEEE Computer Society Press, pp. 395 – 402.

[41] Uhlemann, K., Engelmann, C., and Scott, S. Joshua: Symmetric active/active replication for highly available hpc job and resource management. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing* (Los Alamitos, CA, USA, 2006), IEEE Computer Society.

[42] Zheng, Z., and Lan, Z. Reliability-aware scalability models for high performance computing,. In *Cluster'09: Proceedings of the IEEE conference on Cluster Computing* (2009).