SANDIA REPORT

SAND2010-8709
Unlimited Release
Printed December 2010

Redundant Computing for Exascale Systems

Rolf Riesen, Kurt Ferreira, Jon Stearley, Ron Oldfield, James H. Laros III, Kevin Pedretti, Ron Brightwell, Sandia National Laboratories

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-8709
Unlimited Release
Printed December 2010

Redundant Computing for Exascale Systems

Rolf Riesen
Kurt Ferreira
Jon Stearley
Ron Oldfield
James H. Laros III
Kevin Pedretti
Ron Brightwell
Scalable Computing Systems Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319

Abstract

Exascale systems will have hundred thousands of compute nodes and millions of components which increases the likelihood of faults. Today, applications use checkpoint/restart to recover from these faults. Even under ideal conditions, applications running on more than 50,000 nodes will spend more than half of their total running time saving checkpoints, restarting, and redoing work that was lost.

Redundant computing is a method that allows an application to continue working even when failures occur. Instead of each failure causing an application interrupt, multiple failures can be absorbed by the application until redundancy is exhausted. In this paper we present a method to analyze the benefits of redundant computing, present simulation results of the cost, and compare it to other proposed methods for fault resilience.

Contents

Figures

Tables

1 Introduction

Today's large-scale parallel machines experience outages from failed components, software bugs, human errors, and power disruptions. A common method to allow scientific applications to compute longer than the interval between interrupts is to checkpoint the application state at regular intervals and restart the application from the most recent successful checkpoint after a fault occurs. Checkpoint/restart works but is predicted to be inefficient in future machines because of the large number of expected faults [18, 37, 38].

Planing for exascale systems is under way and it is expected that the first general scientific computers operating at speeds faster than 10^{18} floating-point operations per second (flops) will appear by the end of this decade. In order to reach this performance, technology has to advance. For example, silicon feature size and power consumption has to decrease, while transistor count has to increase. Even with these predicted technological improvements, an exascale system will consist of a huge number of individual components, none of them more reliable than today's components, but potentially worse [9].

We wrote a library which allows MPI ranks to be replicated. Both ranks perform the same computation and if one of them fails, the other continues to function without forcing the application to restart [3]. This approach effectively increases the time between interrupts which results in fewer restarts, and less rework. This leads to better system throughput. Since the application can continue to work in the presence of some faults, it is now possible to increase the checkpoint interval and allow the application to make uninterrupted progress for a longer slice of time. The cost is a small performance degradation due to the replication protocol and, of course, the overhead of using more nodes than the application and problem would need otherwise.

In this paper we look at redundant computing as one of several approaches to isolate extreme-scale applications from failures in the underlying system. Redundant computing has been employed in mission-critical systems for several decades; e.g, [7, 36] and many others. However, until now it has not been considered a necessity for high performance computing (HPC). We believe redundant computing has its place and can improve the efficiency of large-scale future machines. In order to quantify this belief, we wrote an analysis tool which allows us to conduct various studies. This paper describes this tool and reports the results we have obtained using it.

1.1 Faults at exascale

Million-core machines for exascale computing will have so many parts that faults will be frequent. Studies have shown that the failure rate of a system is proportional to the number of processor chips, and that systems and their hardware do not grow more reliable as technology advances [44].

Running interfaces at higher clock rates for improved bandwidth, "wear-out" mechanisms of new devices; e.g., buildup of stray charges on a gate, smaller feature sizes, and using lower voltages with decreased margins will further exaggerate the problem [9]. That study looks at various approaches of how an exascale system could be built. Specifically, extrapolations of a "heavy-node" system, such as the current Cray XT series, a "light-node" system, such as the IBM Blue Gene system, and an "aggressive silicon" system, which assumes that a new system can be designed from scratch combining the best new technologies currently projected. Given these different scenarios, predicting

what will be contained in a node or socket, and how many of each will be present in an exascale system is somewhat difficult. In this paper we use the term node as the unit of replication and failure. It should be understood that, depending on the architecture of these future systems, this could be a socket with multiple cores and integrated memory, or a node in the more traditional sense of one or more CPUs coupled with memory chips and one or more NICs connected to the network.

The exascale study [9] also looks at resilience. Taking the number of components in an exascale system into account and using the appropriate Failure In Time (FIT) number (Errors per 10^9 hours of use) for the different types of components, the study computes that an exascale system will fail once every 35 to 39 minutes. These times include the assumption that about half of the failure will come from software. In this paper we do not consider the source of a failure. The impact on an application is the same whether the failure came from hardware, software, human intervention, or the environment. Any failure that interrupts an application causes lost work and a setback in completion time.

Note that the estimated Mean Time Between Failures (MTBF) of 35 to 39 minutes in [9] may be optimistic. It assumes 5 FITs per 1 GB DRAM chip, but [45] found that the observed uncorrectable error rate is much, much higher than that: 0.22% uncorrectable errors per DIMM, 25,000 – 75,000 FIT per Mbit = 55 - 165 FIT per Mbit. Other sources also indicate a higher FIT for DRAM chips [46] and an MTBF of about 20 minutes for an exascale system [19].

The system-wide MTBF will become so small that much more than 50% of an application's total execution time will be spent writing checkpoints and recovering from failures [3, 37].

In addition, applications are also susceptible to software errors which may also increase in these complex systems due to the increase in parallelism, the possibility of deadlock, and new race conditions.

The more failures occur during the execution time of an application, the longer it will take to finish its work. This decreases the throughput of the machine: fewer applications finish in a unit of time.

1.2 Other approaches

In the previous subsection we made the case that extreme-scale applications will have to contend with a higher failure rate. We now briefly look at two other approaches that are being considered. The first approach is to improve file systems and the storage system to enable faster checkpoints and recovery. In [44] the authors explain that for storage bandwidth to grow at the same rate as the number of processors, more storage devices will be needed since the bandwidth per disk drive grows much more slowly. That in turn increases the number of components in a system and, therefore, the number of failures. It also means that a higher portion of the system cost will need to be devoted to storage than has been traditionally the case.

In this paper we evaluate redundant computing in the context of coordinated checkpoint/restart because this method is widely deployed, easy to implement, and a natural fit for many scientific applications that have periodic synchronization points in their algorithms. In addition to specialized file systems [8] and improved non-volatile storage, more sophisticated checkpoint/restart methods

are often suggested to solve the problem of higher failure rates in exascale systems. Some of them may not help with self-synchronizing applications that are often run on high-end systems. Successful approaches will lower the checkpoint and restart time. A similar effect can be obtained by using Non-Volatile Read-Only Memory (NVRAM) in the form of Solid State Disks (SSD) [39]. In Section 6.3 we evaluate lowered checkpoint and restart times and compare it with redundant computing.

This paper makes the following contributions: A). We created an open-source tool that mimics an application's work, checkpoint, restart, and rework cycles and allows the modeling of various combinations of node count, MTBF, and work to be performed (Section 3). We describe and validate this tool in Section 4. B.) The results of these simulations let us determine when it might be beneficial to use redundant computing (Section 5). The paper closes with a related work section (Section 6), and a summary (Section 7).

2 Motivation

In today's large parallel systems, whenever a component fails that an application is currently using, the application is interrupted and aborts. At a later time, after the system has been repaired or enough resources are available again, the application is restarted. During the restart an earlier checkpoint is read in from stable storage that lets the application continue from the point in time when the last successful checkpoint was written. In addition to the time the application needs to solve a particular problem, T_s, there is the overhead of occasionally writing checkpoints to stable storage, restarting when necessary, and redoing work that was lost since the last successful checkpoint.

In this paper we ignore the time an application spends in the batch queue after it has been interrupted and before it is restarted. Although this time increases the time to solution, it is very unpredictable and depends very much on the queuing policies in place and the load of the system. Furthermore, we only consider coordinated checkpoint/restart. Other methods exist, but are rarely used in practice, and coordinated checkpointing is easy to implement and natural for self-synchronizing applications. Assuming that other methods might be able to write checkpoints more quickly and restart faster, we will look at these benefits in Section 6.3. However, evaluating the full impact of different checkpoint methods, such us uncoordinated with message logging or RAID-style writes to other node's memories, is beyond the scope of this paper. Improving checkpoint and restart times is clearly beneficial, but the exact method and time requirements do not alter the fundamental results presented in this paper.

Let Figure 1 motivate our discussion. It has been generated from numbers computed by the analysis tool described in Section 3 and shows the overhead an application experiences. The application needs to finish $T_s = 168$ hours of work per node (weak scaling). For several node sizes we compute the overhead which is broken down into restart time, lost work, and time to write checkpoints. The graph in this example clearly shows that beyond 50,000 nodes the application spends only a fraction of the elapsed time doing the actual computation it was designed for. The MTBF of an individual node in this example is 43,800 hours (5 years).

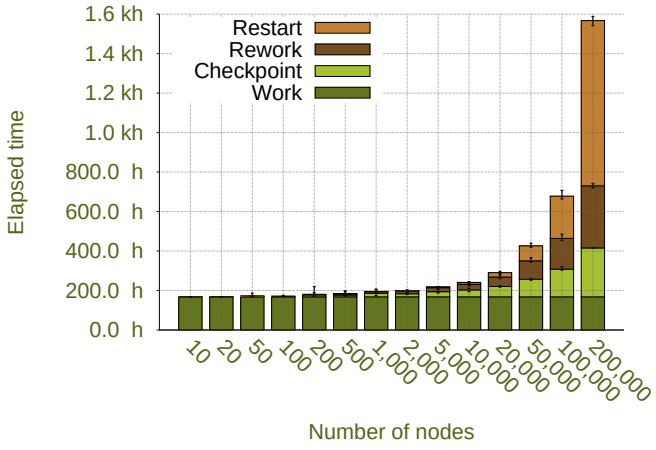

Figure 1. Example of overhead for a 168-hour application run.

The reason for the rapidly increasing overhead in Figure 1 is the lost work and number of restarts the application experiences as the number of faults increases. The right y-axis of Figure 2 shows the number of interrupts the application in this example experiences. While manageable to about 10,000 nodes, the number of faults increases exponentially for larger systems. Unless otherwise noted, all experiments in this paper were conducted seven times. The error bars show minimum, average, and maximum values. The error bars in these first two examples are barely visible due to their small spread.

Given a node MTBF Θ_{node} and the assumption that all nodes have the same MTBF and independent failure behavior, it is easy to compute the MTBF Θ_{sys} for an entire system consisting of n nodes [30]:

$$\Theta_{\text{sys}} = \frac{1}{\frac{1}{\Theta_1} + \frac{1}{\Theta_2} + \ldots + \frac{1}{\Theta_n}} = \frac{1}{n\frac{1}{\Theta}} = \frac{\Theta_{\text{node}}}{n} \tag{1}$$

The result of this calculation for a 5-year node MTBF and the number of nodes varying from 10 to 200,000 is shown using the left side y-axis in Figure 2. The two figures in this section make clear that large-scale applications running on future systems will have to contend with a lot of overhead, a large number of faults, and a low system MTBF.

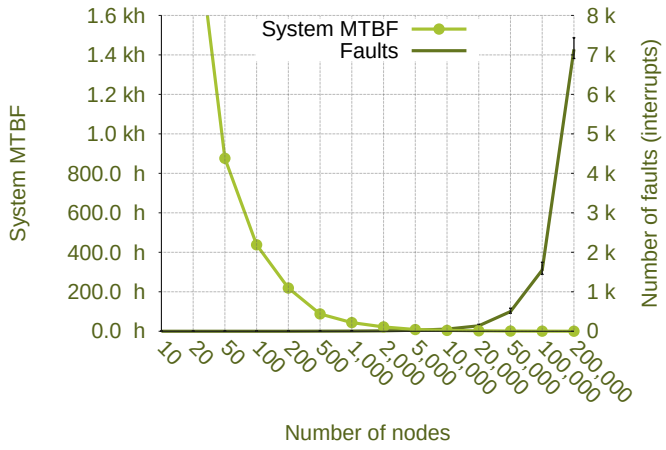

Figure 2. Number of faults (interrupts) for the example in Figure 1. The left y axis shows the calculated system MTBF.

2.1 Redundant computing

When two nodes are used to represent the same MPI rank, as our replication library does, then the failure of one or the other node 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 birthday problem.

One version of the birthday problem asks how many people need to be brought together until there are enough to have a 50% or better 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 in a parallel system, 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.

The equation to calculate the birthday problem is shown in Equation 2 [25, 35]. This equation is the so called Q-function described in [20] and examined by Knuth in [31] in the context of hashing. The answer for a n = 365-day year, and all days equally likely, is 24.6 people.

$$F(n) = 1 + \sum_{k=1}^{n} \frac{n!}{(n-k)! \cdot n^k} \tag{2}$$

The above equation is time consuming to compute since it requires an arbitrary precision calculator. Equation 3 is an approximation that can be calculated much more quickly and provides good results [25].

$$\tilde{F}(n) = \sqrt{\frac{\pi n}{2}} + \frac{2}{3} \tag{3}$$

In Figure 3 we calculate, using Equation 2, the expected number of faults that can occur before an application is interrupted. The total number of faults occurring in a system is still the same; actually it doubles, since we are using twice as many nodes. But redundant computing acts as a filter, absorbing many of the faults and lets the application progress uninterrupted for a longer period of time.

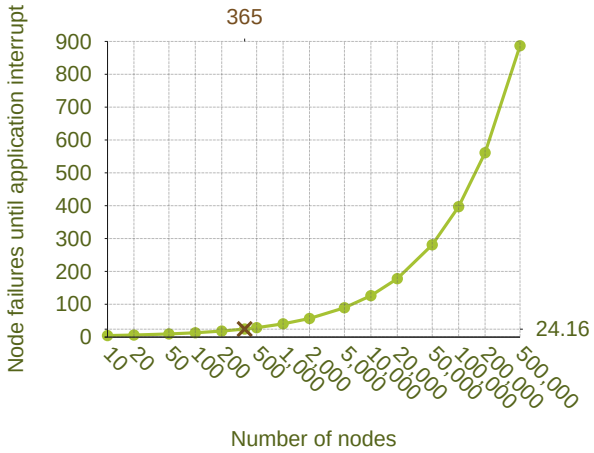

Figure 3. Number of faults before a redundant application gets interrupted.

Due to the absorption of faults, the Mean Time Between Interrupts (MTBI) Θ_{app} which the application experiences is larger than the MTBF of the system it is running on: Equation 4. $F()$ is the expected number of faults the application can absorb: Equation 2 or Equation 3 depending on

your patience. Note that we now need $2n$ nodes.

$$\Theta_{\text{app}} = \frac{\Theta_{\text{node}}}{2n} F(2n) \tag{4}$$

Note that we are not interested in the probability of a node failing. We want to know the time between interruptions of an application in case of redundant and non-redundant computing.

In Figure 4 we compare system MTBF Θ_{sys} with the application MTBI Θ_{app} when individual nodes have an MTBF Θ_{node} of 5 years. Even though both curves tend downward, the application MTBI at 200,000 nodes is still about 100 hours, while the system MTBF has dropped to 13 minutes.

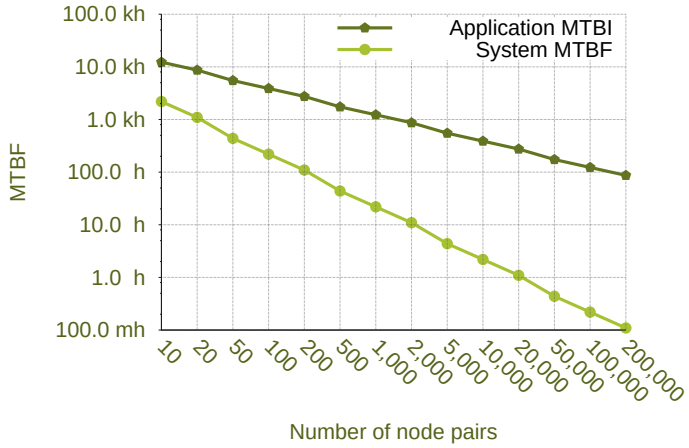

Figure 4. Comparing system MTBF and application MTBI when using redundant computing.

Since redundant computing increases the MTBI, an application running on such a system can increase its checkpoint interval; making it more efficient.

3 Checkpoint restart

In the previous section we have seen that redundant computing reduces the number of interrupts an application experiences. In this section we describe a tool that mimics an application cycling through its work, checkpoint, restart, and rework phases. The tool randomly generates node faults and determines whether the application has been interrupted. There are a slew of configuration parameters that determine how long the application needs to do its work, how many nodes have redundancy, the MTBF of a node, and so on.

Our simulation tool mimics an application by assuming it is always is in one of four states: work (making progress towards a solution), ckpt (writing state information to stable storage), restart (recover from an interrupt, and rework (recompute lost work). The state diagram in Figure 5 illustrates these phases. Application interrupts can occur during any of the phases, and the simulation continues until a fixed amount of work has been completed.

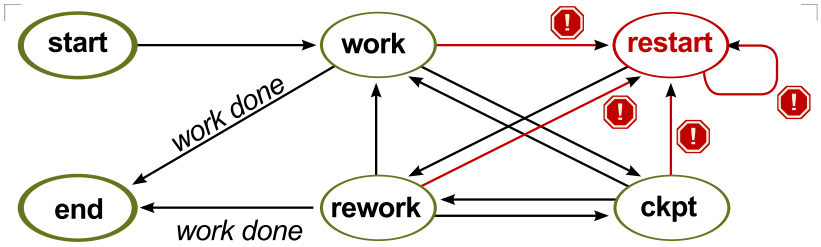

Figure 5. State diagram of the application simulator.

The simulator currently assumes coordinated check-pointing and a fixed amount of time to write checkpoints and restart. While more sophisticated checkpoint methods exist, they are seldom used in practice and would not change the basic results presented here. At best, they reduce the time needed to create checkpoints and would allow for a longer checkpoint interval. We explore the impact of checkpoint time in Section 6.3.

Our simulations assume a perfectly, weak-scaling application; i.e., all nodes perform the same amount of work, independent of the number of nodes used. The amount of work to be done is an input parameter to our tool. Our decision to simulate a weak-scaling application simplifies our experiments and makes the answers easier to understand.

When an interrupt occurs during any of the phases, a restart from the last successful checkpoint is initiated. The work that was lost since the last checkpoint has to be redone in the rework phase. After that, the regular cycle of work and check-pointing continues.

The transitions to the checkpoint state occur whenever the checkpoint interval timer expires. That timer is reset in the checkpoint state. Our simulator uses Equation 5 from [16] to calculate the optimal checkpoint interval. In this equation δ is the (fixed) amount of time it takes to write a

checkpoint, R is the time required to perform a restart, and Θ is the MTBI seen by the application.

$$\tilde{\tau}_{\text{opt}} = \begin{cases} \sqrt{2\delta\Theta}\left[1 + \frac{1}{3}\left(\frac{\delta}{2\Theta}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{\delta}{2\Theta}\right)\right] - \delta & \text{for } \delta < 2\Theta \\ \Theta & \text{for } \delta \geq 2\Theta \end{cases} \tag{5}$$

For all of our work, we assume that the checkpoint interval τ_{opt} is calculated using Equation 5. The checkpoint interval applies to work and rework phases. If the rework phase does not consume the entire interval, then the remaining time until the next checkpoint is used to continue regular work. When all the successfully completed work phases add up to the total work time an application needs to perform, then the application will end.

One of the parameters to control the simulator is the node MTBF of the simulated system. The simulator generates random events that are exponentially distributed around the node MTBF. Each event is a fault that we feed into a model of our replication library. The model determines which node has failed and whether the application receives an interrupt or can continue doing its work. If the model determines that an application interrupt should occur, it forces a transition to the restart state in the state diagram in Figure 5. These transitions are indicated by the exclamation point signs in the diagram. We are assuming that for a restart, the same number of nodes will be available again. This is the same assumption that is made for applications using checkpoint/restart running without redundant nodes.

The MTBF parameter for the application simulator is the MTBF of a single node. The fault generator within the simulator uses that MTBF to generate exponentially distributed faults for the individual nodes. This is shown in Figure 6. The state machine requests the next time an application interrupt will occur from the replication model. The replication model then determines at what time to cause an application interrupt. If redundant computing is used, it will be the earliest time both nodes in a bundle have failed. When there are no redundant nodes, each node fault causes an application interrupt.

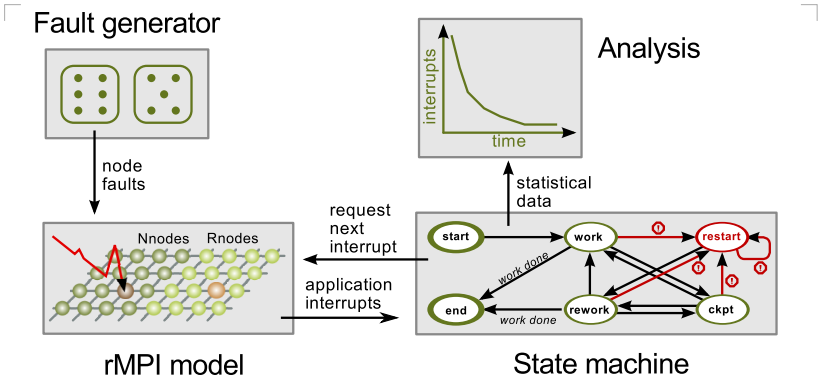

Figure 6. Block diagram of the simulator.

The application interrupt times are fed into a analysis module which computes various values and compares them to simulation results. That module also prints various statistics about the run.

Any attempt at redundant computing has some overhead in addition to using twice the resources. We have evaluated this overhead in [3] and found it to be minimal for actual applications. The overhead is application-specific and dwarfed by the potential savings offered by redundant computing. Therefore, our simulation tool does not consider that small overhead. Our tool has been released as open source and is available from the authors [4].

Using this tool we can now conduct experiments with redundant computing. Figure 7 shows the result for the same configuration we used for Figure 1 but adding an additional n nodes for backup. We keep the y-scale the same for both plots to illustrate the dramatic savings in elapsed time offered by redundant computing.

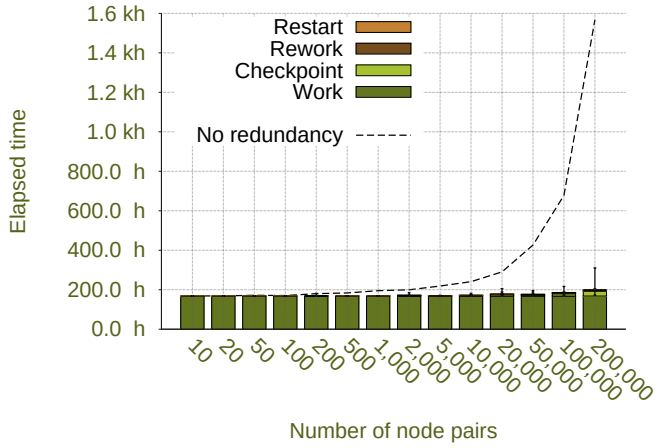

Figure 7. Example of overhead for a 168-hour application run with redundant nodes. The line labeled "No redundancy" corresponds to the total elapsed time of Figure 1.

4 Validation

4.1 Expected application MTBI

In Section 3 we explained that the application simulator takes the node MTBF as one of its input parameters. It simulates node failures over the time an application needs to complete its task. We can output the time of each application interruption and calculate the mean. When we do that for simulations without redundant nodes we get the application MTBF as calculated by Equation 1. This is a good indication that the replication model within the simulator is doing its job correctly. The simulation is within 1% of the calculated value, as long as the work time is long enough to allow for enough interrupts to occur to make the average calculation meaningful.

Similarly, we can use Equation 4, which uses the system MTBF and the birthday problem to predict the application MTBI, to evaluate our simulation for the case where each node is part of a redundant bundle.

4.2 Interrupt-to-fault ratio

Based on our discussion of the birthday problem in Section 2.1, the simulator must for a given number of nodes show the same ratio of application interrupts to node faults as is the expected value of the birthday problem. We ran 500,000 hour workload simulation to let enough interrupts occur to get meaningful results. We ran each experiment seven times and show the minimum, average, and maximum as error bars in Figure 8.

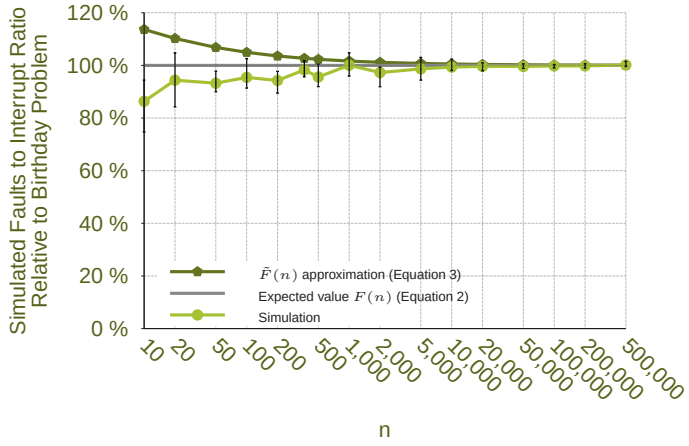

Figure 8. Simulated interrupt-to-fault ratio compared to expected value for the birthday problem.

We normalize against he expected value of the birthday problem from Equation 2. The plot shows that the simulator slightly underestimates the interrupt/fault ratio for smaller number of nodes, but matches the expected value very closely starting with a few thousand nodes. The reason for that is that with fewer nodes, even with a 500,000 hour workload, the number of interrupts is small. That also explains the larger error bars when simulating fewer nodes.

For reference, Figure 8 also shows the value of Equation 3 normalized against the expected value from Equation 2.

We ran an experiment for a 5,000-hour workload on 200,000 nodes (half of them redundant). The application experienced 43 interrupts during that run. At each interrupt we counted the number of faults that had occurred since the last interrupt. Figure 9 shows the result. While the number of faults leading up to an interrupt varies considerably, the figure clearly shows that they cluster around the expected value of 561.166 calculated using Equation 3 for 200,000 nodes. The average number of faults per interrupt for this experiment was 567.093, just 1% above the expected value.

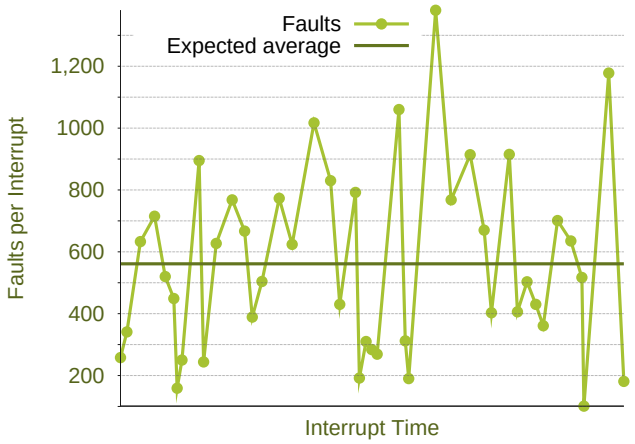

Figure 9. Observed number of faults leading to each interrupt. The expected average value for 200,000 nodes is 562.166.

4.3 Comparison to model

John Daly presents Equation 6 in [16], from which he derives an optimal checkpoint interval $\tilde{\tau}_{\text{opt}}$ (Equation 5). In Equation 6, $T_w(\tau)$ is the total wall clock time the application needs to complete its assigned task. The checkpoint interval τ is calculated using Equation 5. Θ is the MTBI for the application and T_s is the solve time, the amount of time the application needs to complete its work. R is the time needed to restart an application, and δ is the time it takes to write a checkpoint.

$$T_w(\tau) = \Theta e^{\frac{R}{\Theta}}(e^{\frac{\tau+\delta}{\Theta}} - 1)\frac{T_s}{\tau} \quad \text{for} \quad \delta << T_s \tag{6}$$

Using the model from Equation 6 we should be able to validate our simulator. In Figures 3, 4, and 5 of [16] Daly plots the total elapsed time of an application and varies the checkpoint interval. He does this for MTBF of 24, 6, and 0.25 hours. The work time is 500 hours, restart time is 10 minutes, and the checkpoint time δ is 5 minutes. Using these parameters we can recreate Daly's experiment.

Initial tests show that for an MTBF of 6 hours our simulation predicted an average elapsed time that was 11.65% higher, and 5.88% higher for the 24-hour MTBF. Unfortunately, for the 0.25 hour

MTBF it was 51.7% higher. When the MTBF is that small it is near the checkpoint time δ. At that point even very small changes in δ or the application MTBI cause very large changes in the total elapsed time $T_w(\tau)$.

When using redundant computing, but otherwise the same system configuration as above, and recalculating τ accordingly, simulation results agree with the model very nicely. We get -.28%, 1.27%, and 3.14%. Of course, the application MTBI is much higher now due to the redundancy and, since we kept δ at 5 minutes, much larger than δ.

For an additional test we compare the model and the simulator for a 700-hour application and a five-year node MTBF for increasing number of nodes. The two nearly overlapping lines are shown in [Figure omitted for space in draft]. We run our simulator 200 times for each simulation point.

For this paper we chose simulation over an analytical model because it is easier to adapt to new conditions, such as choosing a different random distribution, and varying the level of redundancy; e.g., triple redundancy for soft-error correction or only a portion of the nodes replicated. A limitation of our simulator is that we assume the application will checkpoint at the precise moment the optimal checkpoint interval ($\tilde{\tau}_{\text{opt}}$) indicates. For real application this is sometimes difficult to achieve because they have to be in a consistent state and network activity has to be quiescent. Self-synchronizing applications usually use the granularity of a time step computation to select the time when to checkpoint.

Although our paper focuses on scientific exascale computing systems, the results presented here may be applicable to large data centers as well. Non-scientific applications may not have the long execution times that some scientific applications have, but a lower number of system interruptions should result in a higher system throughput and increase productivity of such centers.

4.4 Performance

Simulation performance depends on the number of nodes we simulate and how many application interrupts we have to process. For most simulation results presented in this paper, simulator execution time was less than one second on a desktop PC. Only when the node count exceeds a million does the simulator need more than a few seconds; especially in non-redundant experiments when many more application interrupts need to be processed.

5 Results

In this section we conduct parameter studies using the simulator described in the first part of this paper.

For the experiments in this section, unless we state otherwise, we use the following parameters for our simulation runs: Checkpoint time $\delta = 5$ minutes, restart time $R = 10$ minutes, a work time T_w of 168 hours (one week), and a node MTBF $\Theta = 43{,}800$ (five years).

Manufacturers often claim a higher MTBF for their products. However, [43] found an MTBF of about four years more realistic for a large high-performance-computing site. The MTBF we are considering is not purely due to hardware faults. Any interruptions that causes an application restart adds to the overhead an application experiences. Even scheduled maintenance is not handled properly by many applications and causes work to be lost. For comparisons we also include some results assuming an MTBF of one year. While probably not common, it has been used in the literature [18], and provides a lower bound on what to expect.

Note that in the data we present in this section, we are not considering the slowdown our replication library introduces, since it is application specific and dwarfed by the time savings when running on large numbers of nodes.

5.1 Application efficiency

Comparing Figure 1 and Figure 7 shows that redundant computing can make a significant difference at large scale. Because of the time it takes to write checkpoints, restart, and rework, an application's efficiency $\frac{t_s}{t_w}$ suffers.

Figure 10 shows three workloads with and without redundant computing. The amount of work an application needs to complete: 168 hours (a week), 700 hours (a month), and 5,000 hours (seven months) in Figure 10 has no impact on application efficiency. Whether redundant computing is used or not, has a huge impact. Even at 500,000 nodes, efficiency only drops to 90% when redundant nodes are used (one million in this case), while applications running without redundancy drop below 2% (below 11% at 200,000 nodes).

Application efficiency impacts system throughput. If an application occupies n nodes ten to forty times longer in order to complete its job, that application is taking resources away from other applications. An inefficient application takes longer to complete and lowers system throughput. Using redundant computing we can bring efficiency up to 90% and even though it requires $2n$ nodes, can complete tens of jobs in the time it takes to finish a single non-redundant job.

5.2 Number of application interrupts

The reason redundant computing offers better efficiency is because the number of application interrupts is significantly reduced. Table 1 illustrates this for a 5,000-hour application.

When redundant computing is not used, each fault causes an application interrupt. This can be seen in columns two and three of Table 1. Because redundant computing increases resilience, the

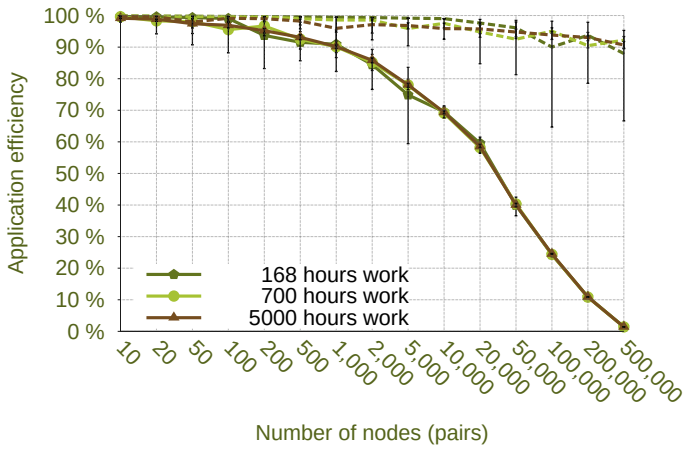

Figure 10. Application efficiency $\frac{t_s}{t_w}$ for three different work sizes. Solid lines are non-redundant, dashed lines use redundant nodes.

number of interrupts an application experiences is drastically reduced: column 5. Since redundant computing uses twice as many nodes, the number of faults also doubles: column 4. Note, however, that this is not true for larger node counts: A larger non-redundant application will take much longer to complete and experiences more faults and interrupts due to remaining in the system for a longer time period.

The last few rows of Table 1 exemplify why redundant computing is so efficient at high node counts. While a non-redundant application has to restart four million times, using twice as many nodes requires just over 100 restarts. A sensible checkpoint/restart strategy is still necessary to recover from these interrupts, but it does not dominate anymore the time the application uses the system.

5.3 Level of redundancy

Sometimes it may be desirable to use less than twice the number of nodes for redundant computing. The replication library supports this and we can evaluate whether it makes sense to use only a few, half, or almost twice the nodes required by the application.

Figure 11 shows the impact of partial redundancy on the number of interrupts an application sees. The longer an application runs, the more impact even a small increase in redundancy has. For example, going from using 50% more nodes for redundancy to 60% reduces the number of interrupts a 720-hour job sees from 6,709 to 4,427; a 34% improvement. Since we ran these experiments for 200,000 nodes, that 34% improvement comes at a cost of an additional 20,000 nodes.

The same 20,000-node investment for a 24-hour job is less beneficial in the number of interrupts saved. At a 50% redundancy level, a 24-hour job takes 231 interrupts and at the 60% redundancy level it takes 167 interrupts; a savings of 64 interruptions.

Table 1. Number of faults and interrupts for a 5,000-hour application.

nodes (pairs)	not redundant faults	not redundant interrupts	redundant faults	redundant interrupts
100	11	11	24	1
200	27	27	45	1
500	60	60	117	3
1,000	126	126	231	3
2,000	264	264	447	5
5,000	739	739	1,153	9
10,000	1,653	1,653	2,384	13
20,000	3,902	3,902	4,784	19
50,000	14,228	14,228	11,973	30
100,000	46,565	46,565	24,304	43
200,000	209,909	209,909	48,930	62
500,000	4,031,114	4,031,114	125,811	101

Figure 11. Level of redundancy versus number of interrupts.

In Figure 12 we show the same experiment but plot the elapsed time on the left y-axis. The larger savings in number of interrupts for a 10% increase in the level of redundancy translates directly into a larger savings of elapsed time. The right y-axis and the dashed lines show the impact on application efficiency, $\frac{t_s}{t_w}$, for the three jobs. When we increase the level of redundancy, we increase application efficiency. With full redundancy (100%) we achieve almost 100% efficiency because very little time is wasted on writing checkpoints and, more importantly, recovering from failures.

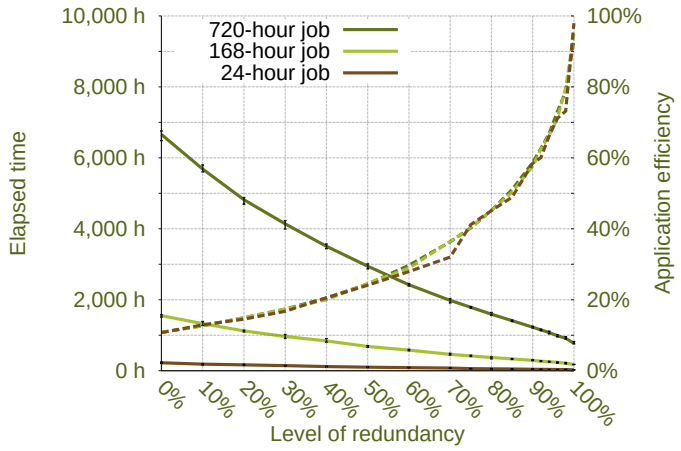

Figure 12. Level of redundancy versus elapsed time. The dashed lines show application efficiency $\frac{t_s}{t_w}$.

5.4 Rebooting nodes

Future systems will try to address fault resilience during the design stage. One approach that might help, is hot-swappable nodes. When a node goes down it can be replaced, rebooted, and reintegrated into the running application. Another approach that is available today, except for the re-integration part, is to simply reboot failed nodes. Very often the faults that brought them down are so called soft-faults, and a reboot brings the node back into operation.

This is especially useful for redundant computing. If one of the two node fails and can be rebooted and reintegrated into the application before the other node fails, that node bundle will be fully restored and can absorb another fault.

Anecdotal evidence suggests that rebooting a failed node in a production system succeeds about 50% of the time. We further assume that a node requires five minutes to reboot. The simulator will interrupt the application only, if the second node in the bundle fails during that five minute window. However, if the second node comes back online (with our chosen 50% probability), both nodes in that bundle will have to fail before it causes another application interrupt.

We assumed that this should further reduce the number of interrupts an application experiences. However, our simulations show no significant difference, and application execution time remains about the same, with or without node reboots.

6 Discussion and related work

6.1 Implementing software redundancy

Using our replication library, we evaluated empirically the cost of implementing redundant computing [3, 5], while in this paper we analyze the impact of redundant computing at exascale on the throughput of a system and the application efficiency.

The replication library implements redundant computing transparently to the application at the profiling layer of the MPI library. At start time the user chooses how many ranks of the application should be replicated. The replication library then uses these additional MPI ranks and assigns them as redundant partners to the initial set of ranks in a one-to-one fashion, or according to some permutation chosen by the user. The later feature is used to evaluate the impact of distributing ranks and their redundant partners across the nodes of a system. To increase fault independence, the two ranks should be physically as far apart from each other as possible; hopefully using different power supplies, colling systems, and network components. This increases communication latency between a rank and its redundant partner which could impact the efficiency of the replication protocol needed between the ranks to maintain replication state. It turns out that on a system with more than 10,000 nodes placing of replicas has very little impact on application performance.

When we first implemented the replication library we assumed that it would need to reside at a very low layer in the communication stack and be thoroughly integrated into the Reliability, Availability, and Security (RAS) system. In the end it was possible to implement the library in user space between an application and the MPI library. Its requirements of the RAS system are rather minimal. It needs to be informed in one fashion or another about ranks that have experienced faults and have become unresponsive. Furthermore, the system must be able to discard messages in flight to disabled nodes; i.e., even messages to dead nodes must be consumed and must not deadlock communication channels. Since the replication library relies on the underlying MPI implementation, that library must be able to survive the loss of individual ranks without stopping the application.

The replication library supports several replication protocols to maintain state of the computation and the availability of individual ranks. The most efficient of these protocols has a very low impact on application performance. The overhead of replicating messages and synchronizing a rank and its redundant partner is easily measurable with micro-benchmarks. However, the impact of that overhead is greatly diminished for real applications which perform enough compute work between communications to mask most of this overhead. We measured an overhead of well below 10% for several scientific applications and application benchmarks, and saw a worst case of 20% for one application in one specific case. These overheads are easily dwarfed by the performance gains of redundant computing.

A major contributor to overhead is the handling of any-source MPI receives. The library must ensure that these messages arrive at a redundant rank in the same order as they did on the original rank. This requires tight synchronization between these two ranks and increases latency greatly. The library detects any-source receives and uses the more expensive synchronization protocol only in those cases where application behavior mandates it.

The current implementation of the replication library is a prototype and has several limitations, the main one being that it does not handle I/O. It also replicates a large portion of the MPI library

itself in order to stay informed about the state of communication. Integrating replication into the MPI library would be beneficial.

6.2 Lowering the cost of checkpoint/restart

Redundant computing is a costly approach to application resilience at the exascale. We have seen in Section 5 that it can pay off at large number of nodes and that it is not too difficult to implement for MPI in [3]. However, it is somewhat controversial in HPC and researchers are looking at other methods to make applications more resilient to faults.

In Section 1.2 we discussed other checkpoint/restart schemes, and improvements to file and storage systems that are alternatives to redundant computing. [17] is a good survey of checkpoint restart methods and [34] looks at seven specific implementations for clusters. Although some of these methods promise better performance than coordinated checkpointing, very few of them have been implemented and used by scientific applications at large scale. Many scientific applications are self-synchronizing and coordinated checkpointing is a natural fit for these applications. For other types of applications, uncoordinated (asynchronous) checkpointing has been proposed; e.g., [47]. The issue for large-scale systems is that a consistent-state recovery line has to be established after a fault which may be prone to rollback propagation. Rollback may be worse at larger scales, although we have not investigated that yet. Communication-induced checkpointing [1] allows for some checkpoints to be local, but does not scale well.

Message logging schemes in conjunction with uncoordinated checkpointing may be good candidates for larger systems [17, 26–28, 32, 33]. Most of them are difficult to implement and have not been evaluated using real applications at large scale.

Several hybrid approaches, improvements, and alternate methods have also been proposed. Using incremental checkpoints to reduce the checkpoint overhead is described in [40, 41]. Our experiments show that at large scale in systems with many faults the main contributor to overhead is restart time. It is therefore very important to improve restart times along with checkpointing performance.

New technologies expected to be available in future exascale systems include node-local NVRAM and SSD which promise faster access times and higher bandwidth. One way of using these devices would be to checkpoint locally to NVRAM and then trickle the data in the background to stable storage off-node (either SSD or spinning media). Until data reaches off-node storage, it may not be available to a restarting process. One interesting idea is to use intermediate nodes in a system to coordinate checkpointing to external storage and thereby better utilize a system's resources and improve checkpoint and restore times [37].

6.3 The need for a new solution

Applications will need to adapt to exascale systems [2]. Concerns include billion-fold parallelism, locality, and a simple, understandable execution model. While dealing with these difficult challenges, the idea of adapting applications to an environment with a higher fault rate is rarely mentioned. Therefore, systems researchers need to find ways to isolate applications from the faults that occur in the underlying system.

There is a cost associated with that. Hardware manufacturers could make their components more reliable, perhaps using redundancy at the hardware level; machine owners could buy additional resource to allow for software redundancy, as proposed in this paper; or, advances in new technologies, such as NVRAM, advanced checkpointing methods, or faster file systems might offer a solution. Transaction-based systems require a lower error rate than the underlying hardware provides. For these systems, users are paying a performance penalty that would not be acceptable in a HPC system [10].

Which method, or combination of methods, will prevail remains to be seen. We can use the simulator described in this paper to set limits on when a method becomes more cost-effective than redundant computing. Until now we have used a fixed checkpoint time of $\delta = 5$ minutes and a fixed restart time of $R = 10$ minutes for all of our experiments. While these values may be unrealistically low for today's technology extrapolated to the exascale, newer technologies and approaches to fault resilience aim to lower the checkpoint and restart times.

In Figure 13 we look at checkpoint times δ of five minutes or less. We assume a restart time of $R = 2\delta$. We simulate four large node counts, with and without redundant computing. For each run we calculate and use the optimal checkpoint interval τ. To obtain a y-axis value, we compute the ratio of the average of seven redundant runs and the average of seven non-redundant runs. As long as this ratio remains below 0.5, redundant computing is more cost effective.

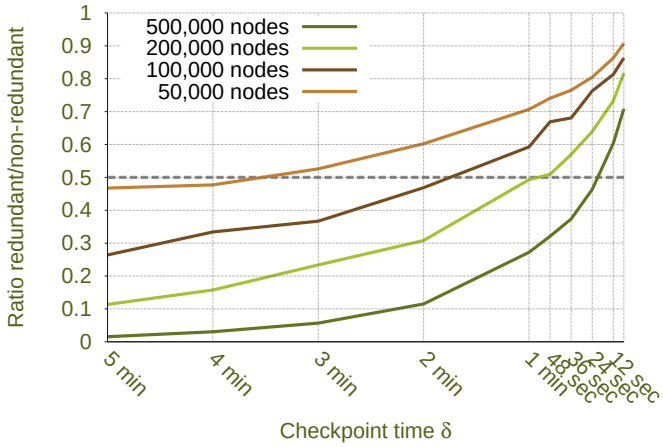

Figure 13. Redundant/none-redundant elapsed time ratio for various checkpoint times δ.

Figure 13 lets us determine how low δ needs to be for a given number of nodes, to rule out redundant computing as a feasible solution. For example, for 500,000 nodes, δ needs to be less than 24 seconds in order to beat redundant computing. Specifically, a typical 500,000-node job that does 168 hours of work (T_s) runs for about 370 hours without redundant computing and a checkpoint time of 24 seconds. Using the same parameters, but using twice as many nodes, the job completes in 170 hours. If we lower the checkpoint time some more to 12 seconds, the non-redundant job finishes in 290 hours, while the redundant job still takes about 170 hours. At that point it is no longer cost-effective to use twice the number of nodes for redundant computing.

Therefore, technologies and algorithms that lower the checkpoint time are worthwhile to inves-

tigate. However, the cost of these methods need to be taken into consideration and compared to redundant computing. For example, on a 200,000 node system with 16 or 64 GB of memory per node, a 1 GB checkpoint size per node does not seem unreasonable. In order to write 200 TB of data in less than one minute (the break-even point for 200,000 nodes in Figure 13), the system aggregate bandwidth to stable storage needs to be 3,333 GB/s. That is unrealistic, when current projections are closer to 50 GB/s [37].

Other ideas, such as checkpointing to the NVRAM of neighboring nodes, are more promising because of the higher bandwidth that can be achieved. Note that this data still needs to be trickled off to stable storage, or some other method, such as using far-away neighbors, needs to be employed to keep the checkpoint data available in case of a cabinet-wide failure. In the 500,000-node non-redundant scenario above, a checkpoint is written every 1.32 minutes, which is not much time to trickle data to stable storage. Saving only a fraction of the checkpoints from NVRAM to stable storage helps, but increases the risk of more lost work and makes this less efficient. Further studies are needed to determine the break-even points for the various approaches at the exascale.

6.4 Other work

Extreme-scale resiliency is an active research topic. We discussed in detail why a higher failure rate is likely in exascale systems in Section 1.1, and several papers show that future systems will have a low system MTBF [18, 29, 37, 38], which requires solutions that go beyond traditional checkpoint/restart.

The requirements for RAS systems have been studied before [6] and newer work more directly aimed at large-scale systems is under way [24]. Integrating the RAS system with the replication library, or an MPI implementation that includes redundant computing will be essential. Systems that are specifically built to reduce the number of faults, and use redundancy to do so, have been available [7, 36] and designing hardware with fault resiliency in mind has a long history [42]. Software solutions for redundancy have also been proposed [11, 15]. The issue with proposed large-scale systems is that many of the components are off-the-shelf and not specifically designed to operate at that scale, and that these systems are specifically purchased for their performance and capacity.

Several groups are investigating application resilience in the context of extreme-scale systems: e.g., [12, 18, 34]. Looking toward extreme scale in particular are: [13, 14, 21, 38]. Several of these papers point at [43] which shows that the observed system error rate is often much higher than the theoretical prediction for the hardware alone.

Large-scale and frequent faults are typical in distributed systems. Solutions in that domain are interesting for exascale systems. For example, [22, 23]. However, performance and throughput are paramount in exascale systems, while they are less critical in distributed systems.

7 Summary

Redundant computing has been used in mission-critical systems but has been viewed as too expensive for large-scale HPC systems. This paper discusses a simulator for redundant computing in large-scale systems. We show that redundant computing can be more cost effective at large scale. Alternative methods that improve checkpoint and restart time need to improve by two orders of magnitude before they approach the throughput improvements possible with redundant computing.

References

[1] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *FTCS*, pages 242–249, 1999.

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

[3] Anonymous. Hidden for double blind review. Technical report ????, Some organization, October 2009.

[4] Anonymous. Hidden for double blind review. `http://www...`, March 2010.

[5] Anonymous. Hidden for double blind review. In *Hidden*, Lecture Notes in Computer Science. Springer Verlag, 2010.

[6] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993.

[7] Joel F. Bartlett. A nonstop kernel. In *Symposium on Operating Systems Principles (SOSP)*, pages 22–29, 1981.

[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *ACM/IEEE Supercomputing Conference (SC)*, 2009.

[9] Keren Bergman and et al. Exascale computing study: Technology challenges in achieving exascale systems. `http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, 2008.

[10] Ricardo Bianchini and et al. System resiliency at extreme scale. `http://institute.lanl.gov/resilience/docs/IBM%20Mootaz%20White%20Paper%20System%20Resilience.pdf`, 2009.

[11] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[12] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[13] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, 2009.

[14] Franck Cappello, Al Geist, Bill Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. Technical report TR-JLPC-09-01, Illinois-INRIA Joint Laboratory on PetaScale Computing, 2009.

[15] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *Symposium on Operating Systems Principles (SOSP)*, pages 12–25, 1995.

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

[17] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

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

[19] Christian Engelmann and Frank Lauer. Facilitating codesign for extreme-scale systems through lightweight simulation. In *Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC)*, 2010.

[20] Philippe Flajolet, Peter J. Grabner, Peter Kirschenhofer, and Helmut Prodinger. On Ramanujan's Q-function. *J. Comput. Appl. Math.*, 58(1):103–116, 1995.

[21] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *International Conference on Supercomputing (ICS)*, pages 269–277, 2006.

[22] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault management in p2p-mpi. *International Journal of Parallel Programming*, 37(5):433–461, 2009.

[23] Stéphane Genaud and Choopan Rattanapoka. P2p-mpi: A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5:27–42, 2007.

[24] Rinku Gupta, Pete Beckman, Byung-Hoon Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar Panda, Andrew Lumsdaine, and Jack Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *International Conference on Parallel Processing (ICPP)*, pages 237–245, 2009.

[25] Lars Holst. The general birthday problem. In *International seminar on Random graphs and probabilistic methods in combinatorics and computer science*, pages 201–208, 1995.

[26] Q. Jiang and D. Manivannan. An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[27] Qiangfeng Jiang, Yi Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *J. Parallel Distrib. Comput.*, 68(12):1575–1589, 2008.

[28] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *ACM symposium on principles of distributed computing*, pages 171–181, 1988.

[29] William M. Jones, John T. Daly, and Nathan A. DeBardeleben. Application resilience: Making progress in spite of failure. In *International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 789–794, 2008.

[30] Dimitri B. Kececioglu. *Reliability Engineering Handbook*, volume 2. DEStech Publications, Inc, 2002.

[31] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley, 1998.

[32] Pierre Lemarinier, Aurelien Bouteiller, Geraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *Int. J. High Perform. Comput. Netw.*, 2 (2-4):146–155, 2004.

[33] Kai Li, Jeffrey F. Naughton, and James S. Planck. Checkpointing multicomputer applications. *Reliable Distributed Systems*, pages 2–11, 1991.

[34] Andrew Maloney and Andrzej Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*, 2009.

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

[36] Dennis McEvoy. The architecture of tandem's nonstop system. In *Proceedings of the ACM conference*, 1981.

[37] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, 2007.

[38] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *International Parallel and Distributed Processing Symposium (IPDPS) - Workshop 18*, 2005.

[39] Xiangyong Ouyang, Sonya Marcarelli, and Dhabaleswar K. Panda. Enhancing checkpoint performance with staging IO and SSD. In *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 13–20, 2010.

[40] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 45–53, 2009.

[41] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[42] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[43] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[44] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[45] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *International joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 193–204, 2009.

[46] Charlie Slayman. Impact and mitigation of DRAM and SRAM soft errors. `http://www.ewh.ieee.org/r6/scv/rl/articles/Soft%20Error%20mitigation.pdf`, 2010.

[47] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):546–554, 1995.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911


Sandia National Laboratories