Lazy Checkpointing : Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems

Devesh Tiwari, Saurabh Gupta, Sudharshan S Vazhkudai
Oak Ridge National Laboratory
tiwari@ornl.gov, guptas1@ornl.gov, vazhkudaiss@ornl.gov

Abstract

Continuing increase in the computational power of supercomputers has enabled large-scale scientific applications in the areas of astrophysics, fusion, climate and combustion to run larger and longer-running simulations, facilitating deeper scientific insights. However, these long-running simulations are often interrupted by multiple system failures. Therefore, these applications rely on "checkpointing" as a resilience mechanism to store application state to permanent storage and recover from failures.

Unfortunately, checkpointing incurs excessive I/O overhead on supercomputers due to large size of checkpoints, resulting in a suboptimal performance and resource utilization. In this paper, we devise novel mechanisms to show how checkpointing overhead can be mitigated significantly by exploiting the temporal characteristics of system failures. We provide new insights and detailed quantitative understanding of the checkpointing overheads and trade-offs on large-scale machines. Our prototype implementation shows the viability of our approach on extreme-scale machines.

1. Introduction

Increase in the computational capability of supercomputers has enabled scientists to run larger simulations both in time and size, facilitating deeper scientific insights [1, 29]. Unfortunately, these long-running simulations are often interrupted by multiple system failures. Therefore, applications have traditionally relied on "checkpointing" as a resilience mechanism against failures. Checkpointing is a process by which applications periodically save their state to permanent storage so they can restart from a previously known stable state, in the event of a failure [16, 27].

Checkpointing and restoring the application state after a failure exerts severe pressure on the I/O subsystem, as it involves writing and reading a large amount of data from permanent storage [3, 6]. For example, GTC, a fusion application writes 20 TB of checkpoint data per hour at-scale and has to read back at every failure.

To illustrate this, Fig. 1 (top) shows the time spent on I/O, useful computation and wasted work[1] for different system sizes. As the system size increases, the time spent on I/O increases significantly to perform a fixed amount of computation because of the increased failure rate. The I/O overhead and wasted work are also dependent on the frequency of checkpoints. For example, comparatively less frequent checkpointing may decrease the I/O overhead (Fig. 1 (bottom) vs (top)), but will increase the wasted work, possibly increasing the application's total execution time. Therefore, checkpointing has implications to both storage and compute systems.

Unfortunately, both system administrators and the scientific application programmers have a limited understanding of the interplay between checkpointing, the I/O overhead and the compute resource wastage, due to the non-trivial trade-offs involved and the lack of a large-scale quantitative study. Therefore, the goal of this paper is to understand, quantify and mitigate the impact of checkpointing on the storage system on extreme-scale machines. The study is driven using analytical models, statistical techniques, and real large-scale computing facility parameters, logs and traces.

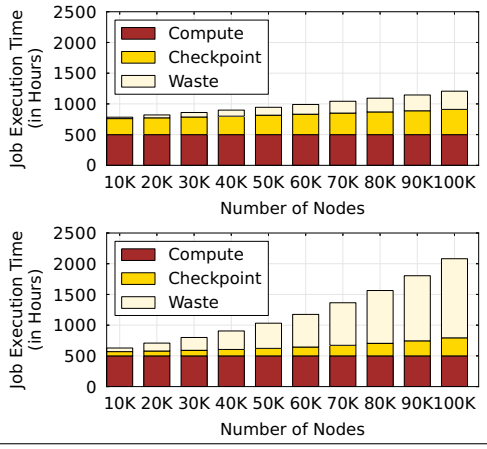

Figure 1. Impact of checkpoint/restart mechanism on a large-scale application checkpoint taken every hour (top), and every four hours (bottom). Checkpoint and restart time 30 mins and 15 mins, respectively. MTBF of each node is taken as 25 years and scaled according to the system size. Large-scale scientific applications are weak-scaling, i.e., the compute time per node remains a constant.

Contributions: First, we study the effect of the traditional periodic checkpointing technique for a variety of leadership computing applications, using an analytical model and simulation based validation. Our study reveals several interesting, previously unknown insights. We show that the analytically derived optimal checkpoint interval, though difficult to determine in a dynamic environment, can be approximated and works well in most situations.

Second, we investigate failures on multiple leadership computing facilities to understand their impact on I/O overhead and compute resource wastage. One of our interesting findings, from analyzing more than 9 years worth of failure data from supercomputing facilities, is that failures have a strong temporal locality. The probability of a failure is high soon after a failure has occurred (i.e., more failures occur on the heels of a failure).

Based on our observation of the temporal locality in failures, we propose two novel techniques, *Lazy Checkpointing and Skip Checkpointing*, that place checkpoints by taking advantage of the temporal locality in failures, instead of naively taking periodic checkpoints.

The temporal locality in failures indicates that a significant fraction of failures is likely to occur within a relatively shorter time-period (compared to the MTBF of the system) after a failure strikes.

[1] Wasted or lost work is the amount of work between the failure and the last checkpoint that can not be recovered.

Domain	Application	Checkpoint data size	Job run-time
Astrophysics	CHIMERA	160 TB	360 Hours
Astrophysics	VULCUN/2D	0.83 GB	720 Hours
Climate	POP	26 GB	480 Hours
Combustion	S3D	5 TB	240 Hours
Fusion	GTC	20 TB	120 Hours
Fusion	GYRO	50 GB	120 Hours

Table 1. Checkpoint data size and job run-time of characteristics of leadership applications.

Lazy checkpointing takes advantage of this observation by intelligently increasing the checkpointing interval between two failures with minimal or no performance degradation. Our proposed scheme dynamically increases the checkpointing interval using the statistical properties of failure inter-arrival times. To avoid potential performance degradation, we provide an upper bound to these increasing checkpointing intervals and reset the checkpointing interval to default optimal checkpointing interval at every failure.

Our evaluation demonstrates that Lazy checkpointing can significantly reduce the I/O overhead[2]. We present, *Skip checkpointing*, an alternative, simpler and static checkpointing technique that takes advantage of temporal locality in failures as well. Skip checkpointing scheme skips a "later" checkpoint after a failure instead of the ones that follow immediately. We present a thorough evaluation of the benefits and limitations of both proposed techniques.

Third, we have built a prototype implementation that integrates different checkpointing strategies. We evaluate the techniques using this prototype and actual supercomputing I/O traces and failure logs, and show that our technique can significantly mitigate the checkpointing overhead even in a dynamic environment.

2. Background and Methodology

2.1 Leadership Computing Facility and Scientific Applications' Requirements

Our work is primarily modeled and evaluated based on Titan, No. 2 on the Top 500 supercomputer list. Titan consists of 18,688 compute nodes (CPU and GPU) and more than 700 TB memory capacity. At the time of this study, it had a peak I/O bandwidth of 240 GB/s. Titan's peak performance is approx. 27 Petaflops. We have also included various system design points to show the relevance of our insights and the impact of our techniques for future exascale systems.

Table 1 shows the checkpoint size and run time for different leadership applications[3], based on traditional hourly checkpoints on Titan [1, 29]. Table 1 depicts application-level checkpointing, wherein an application only saves the data it deems necessary for a restart, as opposed to system-level checkpointing that saves the entire system state [16]. Therefore, depending on the nature of the application, the checkpoint size can vary significantly. Although with the variation in the checkpoint interval, the size of checkpoint data may vary, but for simplicity we assume it to be a constant.

2.2 Data Collection

I/O Data: I/O data, presented in this study, has been collected on the DDN RAID controllers on the Titan's Spider storage system (Lustre parallel file system). I/O statistics such as bandwidth and IOPS are collected at a granularity of 2 second intervals over these DDN controllers (negligible overhead is observed on Spider due to this monitoring). A corresponding database is populated with this data using a custom utility [24].

Failure Data: Failure related data, presented in this study, has been collected from multiple supercomputing facilities such as the Oak Ridge Leadership Computing Facility (OLCF) and the Los Alamos National Lab (LANL), over a total of 1000 failures. The OLCF data represents Titan's failure log data for approx. sixe months since it went into production (Mar'13). The failure logging is mostly automated at OLCF. Limited amount of console logs from the compute nodes are pushed to a Cray admin server where a daemon utility runs continuously to parse them and detect failures. The parser uses a simple event correlator to parse important events, based on a set of rules. The simple event correlator identifies all the events that may require some action, and not necessarily failures. The rules are periodically updated to reflect the addition of new components and events (e.g., new custom interconnect, GPU addition).

We have identified the events that cause application failure to avoid any under/overestimation of failures. We collect all of the system-level failures, including some software failures. We do not account for all storage-subsystem failure as not all of them will cause an application to fail-stop. However, some Lustre parallel file system related failures do fail-stop an application and they have been accounted for appropriately. Since the log provides exact time stamps, it can be used in a dynamic environment to reduce checkpointing overhead as shown later.

LANL failure logs are collected manually, but for a relatively longer duration (9 years) [33]. It has been used to support our findings and the applicability of our proposed techniques.

We have conservatively assumed the per-node MTBF to be 25 years, which is higher than what is observed in our system. System MTBF is obtained by dividing the per-node MTBF by the system size.

3. Understanding I/O Overheads on Leadership Scale Computing Systems

Large-scale scientific applications traditionally checkpoint hourly depending on the estimated I/O overhead. This stems from the fact that they are not aware of the system failure rate or the MTBF. Unfortunately, this approach does not lead to an optimal execution time because it fails to account for the lost work due to failures and the nature of the failure distribution. Therefore, first we build an analytical model to derive a checkpoint interval, one that minimizes the overall runtime. Second, we verify this model with event-driven simulation and derive insights. Third, we present the results for our portfolio of leadership applications and the implications of these results on current and future systems.

3.1 Analytical Modeling of I/O Overhead and Checkpoint Interval

We model a large-scale scientific application's execution as a sequence of computation and checkpointing activity chunks. We denote α as the computation period after which a checkpoint is taken periodically; every α is followed by the checkpointing activity for a duration, β (as shown in Fig. 2). Put another way, α indicates how long an application can compute before it needs to checkpoint. Therefore, we also refer to α as the *checkpoint interval*, and β as the *time-to-checkpoint*.

These activities, when interrupted by failures, incur restart overheads (e.g., reading the last saved checkpoint) for a period of time, say, γ. In a failure-free environment, the application completes its execution after S such activity chunks, each chunk being $\alpha + \beta$ long. However, due to failures we incur additional overhead (T_{waste}), the work that is "lost" due to failures. We can express the total running time of the application as follows:

[2] We use the terms I/O overhead and checkpointing cost interchangeably.

[3] Leadership-scale computing refers to supercomputers facilitated by the Department of Energy, and we refer to the large-scale application run on these supercomputers as leadership applications.

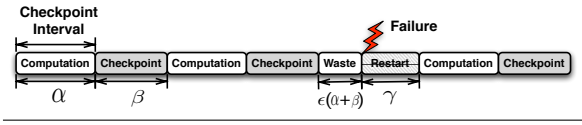

Figure 2. Periodic computation and checkpoint phases of scientific application.

$$T_{total} = T_{compute} + T_{checkpoint} + T_{waste} \quad (1)$$

where the total compute time, $T_{compute}$, is equal to the checkpoint interval times the total number of steps in a failure-free environment ($T_{compute} = S\alpha$). Similarly, the time spent towards checkpointing can be expressed as follows:

$$T_{checkpoint} = (S-1)\beta \quad (2)$$

$$= (\frac{T_{compute}}{\alpha} - 1)\beta \quad (3)$$

Total overhead due to failures can be broken down into two components. First, each failure will cause a certain fraction, say ϵ, of the computation and checkpointing duration, $\alpha + \beta$ to go waste. Second, each failure will have an associated recovery overhead, γ. Thus, the total overhead due to failures can be expressed as:

$$T_{waste} = N_f(\epsilon(\alpha+\beta) + \gamma) \quad (4)$$

where N_f is the total number of failures. Both N_f and ϵ are dependent on the nature of the failure distribution. Next, we derive an expression for N_f assuming that failures follow an exponential distribution, as assumed in previous studies [37, 36, 30, 7, 28]. We will revisit the validity of this assumption using the failure logs collected from supercomputer facilities (Section 4). We also quantitatively estimate the value of ϵ under these assumptions (Section 4.2).

The number of failures can be expressed as the difference between the total number of trials needed to complete S chunks without encountering a failure and the number of times the chunks complete successfully (S). Recall that each chunk is a pair of compute and checkpointing activity, ($\alpha+\beta$). The number of trials can be further estimated as S divided by the probability of not failing before the period $\alpha + \beta$ (i.e., $1 - Pr(t < (\alpha+\beta))$). Therefore,

$$N_f = \frac{S}{1 - Pr(t < (\alpha+\beta))} - S \quad (5)$$

For an exponential distribution, the probability of failure before time t is given by $Pr(X \leq t) = 1 - \mathrm{e}^{-\frac{t}{M}}$, where M is the MTBF. Using this, the above expression can be simplified as:

$$N_f = S(\mathrm{e}^{\frac{\alpha+\beta}{M}} - 1) \quad (6)$$

Putting it all together, the total job execution time (Eq. 1) can be obtained as a complete function of the checkpoint interval, α, by substituting S with $\frac{T_{compute}}{\alpha}$ as follows:

$$T_{total} = T_{compute} + (\frac{T_{compute}}{\alpha} - 1)\beta + \frac{T_{compute}}{\alpha}(\mathrm{e}^{\frac{\alpha+\beta}{M}} - 1)(\epsilon(\alpha+\beta)+\gamma) \quad (7)$$

For the range, where $\alpha + \beta \ll M$, we can simplify the above expression:

$$T_{total} = T_{compute} + (\frac{T_{compute}}{\alpha} - 1)\beta + \frac{T_{compute}}{\alpha}(\frac{\alpha+\beta}{M})(\epsilon(\alpha+\beta)+\gamma) \quad (8)$$

Optimal checkpoint interval, α_{oci}, that will minimize the total execution time can be obtained by solving $\frac{d}{d\alpha}(T_{total}) = 0$. The

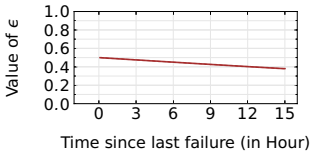

Figure 3. Value of ϵ, i.e., lost work fraction for exponential distribution.

above formula can be differentiated to get the following:

$$\frac{1}{M}(\epsilon - \frac{\epsilon\beta^2}{\alpha_{oci}^2} - \frac{\epsilon\gamma}{\alpha_{oci}^2}) - \frac{\beta}{\alpha_{oci}^2} = 0 \quad (9)$$

Solving this we get the expression for **optimal checkpoint interval (OCI)**:

$$\alpha_{oci} = \sqrt{\beta^2 + \frac{\beta\gamma}{\epsilon} + \frac{M\beta}{\epsilon}} \quad (10)$$

Previous studies have done similar theoretical exercise and derived different variants [37, 36, 7, 30, 22, 28]. However, our exercise is slightly different as it retains the average fraction of lost work, ϵ, in the equation, which leads to a better understanding when we compare this model with real world supercomputer logs (Section 4). The average fraction of lost work, ϵ, becomes the key to understanding the difference between the model and the real-world and its impact on the total execution time.

3.2 Model Validation and Model Driven Study

In this section, we compare our model based results against the results from an event-driven simulator that we have developed. We study the optimal checkpointing interval (OCI) estimation from these two approaches for current and future large-scale systems.

Recall that our analytical model can predict both the total runtime (Eq. 8) and OCI (Eq. 10). To drive this model, we use the parameters obtained from supercomputing facilities (Section 2). The compute-time of a job is assumed to be 500 hours, though individual leadership applications may have varied compute-time requirements (Table 1). The checkpoint time is taken as 0.5 hours, typical of multiple leadership computing facilities [6]. MTBF of one node is taken to be 25 years (Section 2) and adjusted according to the system size.

We empirically obtain the fraction of lost work, ϵ (Fig. 3), by generating one million samples from an exponential distribution (MTBF 10 hours) and estimating the lost work for a given time interval. Note that it is not the same as the probability of a failure in that interval. Fig. 3 shows the value of ϵ beyond the MTBF interval. A value of 0.50 for ϵ reduces the OCI estimation as approximated by Daly's formula as well [7]. We revisit the significance and implications of the "fraction of lost work", ϵ, again in Section 4.2, when analyzing supercomputer failure logs.

To validate our model results, we built an event-driven simulator that simulates the execution of an application given certain parameters, e.g. type of failure distribution (exponential distribution), checkpoint time, restart time, MTBF, and compute time. It does not rely on any mathematical equation, instead it mimics an application execution on a leadership machine. For example, the application experiences probabilistically generated failures and recovers from it. Ideally, modeling results should match the simulation-based results.

Fig. 4 shows the total runtime of a scientific application obtained from both our analytical model and the event-driven simulation. The figure depicts a "hero" run that uses all the nodes in a system (e.g., 20K and 100K node runs). The OCI in the figure is the point where the total execution is at a minimum. First, we observe that the OCI decreases as the system size grows (left and right charts). Second, the modeling and simulation results closely track each other. For a petascale system (Fig. 4 (left)), the model-

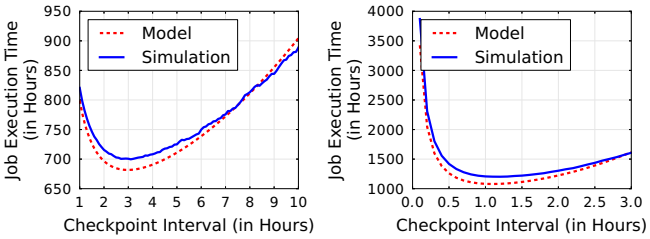

Figure 4. Comparing model and simulation-based results for (left) current petascale systems consisting 20K nodes, and (right) future exascale systems consisting of 100K nodes.

Optimal Checkpoint Interval (OCI) (in hours)					
CHIMERA	VULCAN	POP	S3D	GTC	GYRO
12.4	**0.02**	**0.13**	1.87	3.81	**0.18**

Table 2. Estimated OCI for leadership applications.

based OCI (approx. 3 hours) is only 3% off from the simulation-based OCI. The corresponding execution time difference is only 2.8%. For an exascale system (Fig. 4 (right)), the model and the simulation both estimate the same OCI. The small difference in the OCI estimation is also due to a relatively flat area near the minima of the execution time curve. In fact, for a petascale system, even if the estimated OCI were half an hour more/less, the resulting change in the model-estimated execution time would have been only less than 0.5%. In summary, while the total execution time may not match the simulation exactly, the model-based OCI leads to the same minima as the simulation study. Therefore, in practice the model-based OCI can be used to guide the checkpointing interval as a simulation-based parameter exploration is often more time consuming.

Observation 1. *Optimal checkpoint interval decreases as the system size increases, and the model-estimated OCI is fairly accurate to be used to guide the checkpoint interval of applications.*

Next, we evaluate the benefit of OCI for applications on current Oak Ridge Leadership Computing Facility (OLCF) like systems.

3.3 Evaluation of OCI benefits

Table 2 shows the calculated OCI for different applications on the Titan supercomputer. It is based on the applications' checkpoint size and an observed I/O bandwidth of 10 GB/s on the Spider storage system (Section 2). While the peak bandwidth of such large-scale parallel storage systems may be much higher, users may observe a lower bandwidth due to file-alignment, stripping, and contention issues [23]. We note that our insights are not specific to a particular I/O bandwidth, instead we highlight the key trends that matter under relatively low I/O bandwidth periods. Simulation studies highlight the trends under high I/O bandwidth scenario.

Table 2 suggests that different applications favor different optimal checkpoint intervals; the current practice of one-size-fits-all (hourly checkpoint) is not optimal. In fact, applications with less data to checkpoint, and consequently less I/O overhead, (e.g. VULCAN, POP, GYRO, highlighted in a grey box) should checkpoint more often. This also implies that we will need to checkpoint more often on faster, SSD-based storage systems to achieve the minimum execution time. This may seem counter-intuitive at first, but this is because of the tension between the lost work and the I/O overhead (as indicated by Eq. 10) .

Next, we show the benefit of OCI compared to the traditional hourly checkpointing (Fig. 5). We note that OCI is calculated assuming exponential distribution fits the failure inter-arrival times. It can be seen that for all applications OCI provides a significant reduction in execution time. However, for some applications (e.g,

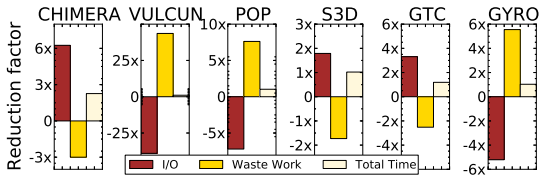

Figure 5. Benefit of OCI to leadership application compared to traditional hourly checkpointing. Positive value shows decrease by that factor compared to the hourly checkpointing case.

VULCUN, POP), it seems to increase the I/O overhead instead of decreasing it. This is because hourly checkpointing is higher than their OCI, which increases their I/O overhead, but reduces the amount of wasted work and results in a net gain overall.

Observation 2. *As emerging storage technologies (e.g, SSD) provide faster I/O bandwidth, the total time spent in checkpointing may increase due to shorter OCI. However, this results in less wasted work at the cost of higher I/O overhead. Overall, this results in a net performance improvement.*

We have shown that the OCI, obtained from the analytical model, can work well in practice and provide significant performance and I/O improvements to leadership applications, when compared to the naive, hourly checkpointing approach.

The model for OCI estimation is based on the assumption that failures follow an exponential distribution. While this assumption simplifies the math and can provide us with tangible benefits, it remains unclear if the OCI is indeed the "true" optimum on current leadership computing facilities, where failures may or may not follow the same characteristics assumed in this model. We need to understand the failure characteristics if we are to achieve the true optimum.

For this reason, we conduct an analysis (Section 4) of machine failure characteristics and its implications to the OCI, the I/O overhead, and the total application runtime on current and future extreme-scale systems.

4. Analyzing and Understanding Characteristics of Failures on Large-scale Computing Systems

4.1 Temporal Locality in Failures

In this section, we present the temporal characteristics of failures on large-scale systems. Our study is based on several years worth of logs from multiple HPC sites. Fig. 6 shows a histogram of failures over time from multiple HPC centers (Section 2.2). Interestingly, a significant fraction of the failures occur much before the observed MTBF. For example, on the OLCF system approximately 45% of the failures occur within 3 hours of the last failure, despite an MTBF of 7.5 hours.

These results indicate that there exists a strong temporal locality between failures. In other words, the occurrence of certain failures may be correlated or caused by previous failures. This is likely to happen in a large-scale computing facility where components are tightly packed and interconnected. For example, a sudden temperature rise may cause a node outage; the resulting increase in fan activity may cause near-by components or links to experience a failure. This finding also implies that the average work lost due to a failure would be less, because a significant fraction of failures occur soon after a previous failure. We will exploit this observation to design a new checkpointing technique (Section 5).

Observation 3. *Failure characteristics of large-scale systems suggest that there is a strong temporal locality between them. The likelihood of a failure occurring on the heels of past failure is high, though the observed MTBF of the system may be much higher.*

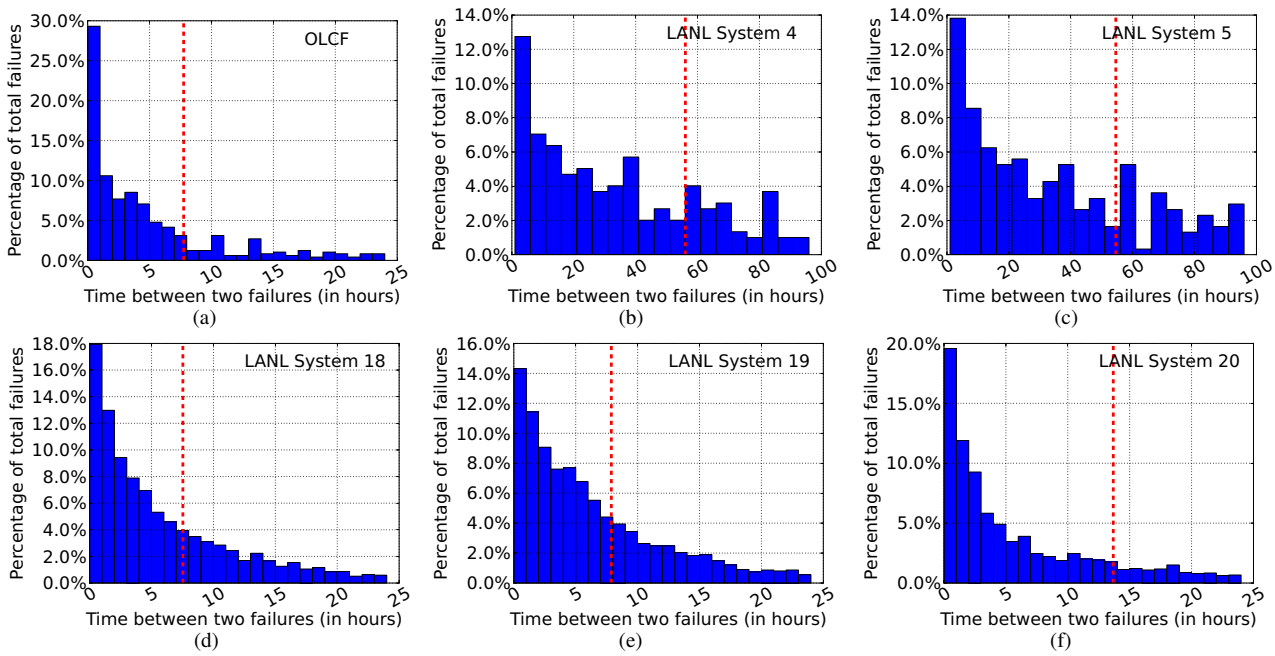

Figure 6. Temporal characteristics of failures from multiple HPC systems. The dashed vertical line indicates the "observed" mean time between failures (MTBF). Multiple failures that occur beyond the x-axis limits are not shown here for clarity, but they contribute toward MTBF calculation.

System	K-S test D-Statistics			Critical D-value	Weibull Shape Parameter
	Log Normal	Exp.	Weibull		
OLCF	0.090	0.184	0.038	0.062	k = 0.64
LANL System 4	0.114	0.087	0.048	0.078	k = 0.82
LANL System 5	0.095	0.075	0.036	0.078	k = 0.86
LANL System 18	0.059	0.082	0.019	0.022	k = 0.82
LANL System 19	0.073	0.039	0.016	0.024	k = 0.90
LANL System 20	0.038	0.210	0.041	0.028	k = 0.65

Figure 7. Result of Kolmogorov-Smirnov test (K-S test) for failure logs of multiple systems. Null hypothesis that the samples for a given system comes from a given probability distribution function is rejected at level 0.05 if k-s test's D-statistics is higher than the critical D-value. Comparison between D-statistics and critical D-value shows that Weibull distribution is a better fit in all cases except the last one.

We take advantage of this observation to reduce the checkpointing overhead on large-scale HPC systems by changing the checkpointing intervals such that it captures the temporal locality in failures. Towards that, we use two statistical techniques to fit our failure inter-arrival times data against four distributions, normal, Weibull, log normal, and the exponential distribution. First, Fig. 7 shows the results from the Kolmogorov-Smirnov test for different distributions [15]. We notice that Weibull distribution fits our sample data better than the exponential distribution. We also present the QQ-plot for visualizing the fitness of these distributions (Fig. 8), which reaffirms the K-S test results.

We note that a Weibull distribution is specified using both a scale parameter (λ) and shape parameter (k). If the value of shape parameter is less than one, it indicates a high infant mortality rate (i.e., the failure rate decreases over time). We point out that shape parameter (k) is less than one for the Weibull distributions that fit our failure sample data. This has also been observed by other researchers for various other systems [21, 31, 35, 17], indicating a larger applicability of the new techniques presented in this work (Section 5), which are based on this observation. Next, we show how does a better fitting Weibull distribution affect the OCI and the total execution time (as opposed to the previously discussed analytical model and simulation-based results that assumed exponential distribution of failure inter-arrival times).

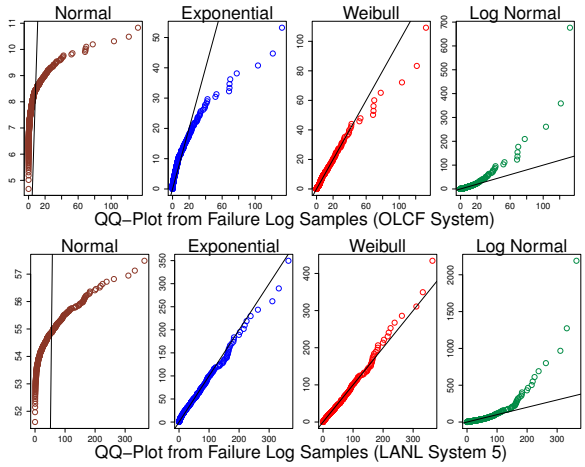

Figure 8. QQ-Plot for graphical representation of fitting different probability distribution functions (PDF). The quantiles drawn from the sample (failure log) are on the x-axis and y-axis shows theoretical quantiles. If the samples statistically come from a particular distribution function then points of QQ-plot fall on or near the straight line with slope=1. Only three representative failure logs are plotted due to space constraints, the rest show similar behavior.

4.2 Effect of Temporal Locality in Failures on OCI

We found that the failure inter-arrival times are better fitted by a Weibull distribution than an exponential distribution. Therefore, in this section we present the results from our event-driven simulator to study how the OCI and total execution time are affected if failure events are drawn from a Weibull distribution instead of an exponential distribution (as assumed in previously discussed analytical model). Fig. 9 shows the total execution time of a "hero" run on three different systems (10K, 20K and 100K nodes). We notice that the Weibull distribution curve is always below the exponential distribution curve. This result suggests that if failure events are drawn from a Weibull distribution, it will result in an overall execution

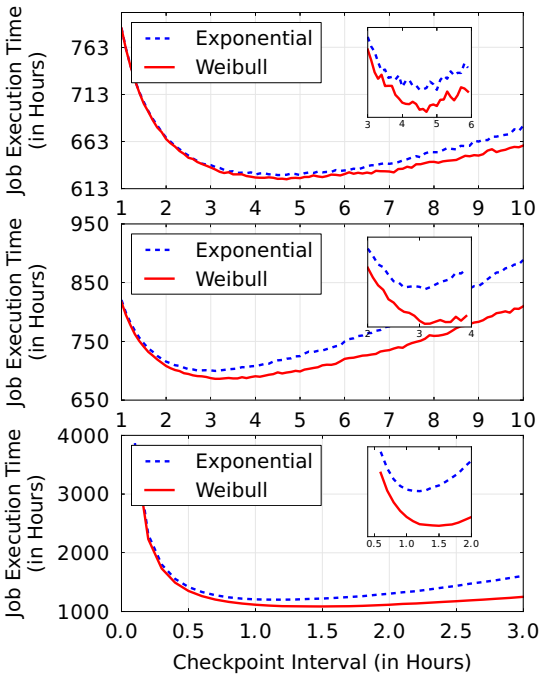

Figure 9. Effect of distribution function on the total execution time and OCI : 10K node system (top), 20K node system (middle) and 100K node system (bottom). The zoomed in section shows that the OCI estimation, which assumes an exponential distribution is not affected even though the actual run time may differ slightly.

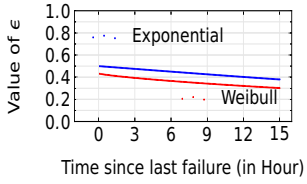

Figure 10. Difference in average lost work fraction between Weibull and exponential distributions.

time that is lower than the exponential case. The underlying reason can be explained using our previous result (Fig. 6), which shows that a large fraction of failures occur soon after the last failure, resulting in less wasted work per failure on an average. We further support this result by showing (Fig. 10) that the lost work fraction, ϵ is lower for a Weibull distribution than for an exponential distribution.

What is of significant interest is that, while the execution time differs, the OCI for these two distributions are quite similar (as shown by the zoomed in section of Fig. 9). Both curves achieve the minima for nearly the same OCI.

Observation 4. *The OCI is not affected significantly by the underlying distribution of failure inter-arrival being Weibull vs exponential. However, the Weibull distribution does result in a lower overall execution time compared to the exponential counterpart because the average lost work per failure is lesser compared to the exponential case.*

While our findings about the temporal locality in failures do not affect the OCI estimation, it does provide an opportunity to improve current checkpointing strategies by exploiting this observation.

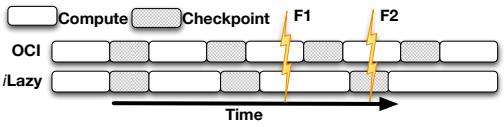

Figure 11. iLazy Checkpointing: increasing checkpointing interval does not always lead to more waste work.

5. Exploiting Temporal Locality in Failures for Reducing Checkpointing Overhead

Lazy Checkpointing Overview: We have shown that OCI based checkpointing is quite effective (Section 3.3), however it inherently fails to capture the temporal locality in failures. Towards this end, we propose to make OCI based checkpointing temporal locality aware.

We showed that failures have high temporal locality. That is, the failure rate decreases over time since the last failure (and until the next failure).

To support this, we plot the failure rates of both distributions for a fixed MTBF of 10 hours for illustration (Fig. 12). The figure shows that while the failure rate for the exponential distribution remains a constant, it decreases for the Weibull distribution.

Note that the failure rate of an exponential distribution is given by $1/M$, where M is the MTBF. The failure rate for a given Weibull distribution ($1 - \mathrm{e}^{-(\frac{t}{\lambda})^k}$) is given as $\frac{k}{\lambda}(\frac{t}{\lambda})^{k-1}$, where λ is the scale parameter, k is the shape parameter, and t is the time since the last failure. In Fig. 12, we determine λ using a Γ (Gamma) function for $k = 0.6$ (representative of an OLCF-like system, Fig. 7), such that the MTBF of this Weibull distribution remains the same as the exponential distribution (M).

We observe that since the failure rate decreases over time, one may accordingly get "lazy" in taking checkpoints as more time passes by since the last failure. Essentially, we should increase the checkpointing interval over time such that it has the same slope as the corresponding Weibull distribution's failure rate curve. Therefore, a simple formula to achieve this incrementally increasing checkpoint interval, α_{lazy}, is as follows:

$$\alpha_{lazy} = \alpha_{oci} \left(\frac{t}{\alpha_{oci}} \right)^{(1-k)} \tag{11}$$

where α_{oci} is the same OCI as previously determined and t is the time since the last failure. Note that the checkpoint interval increases inversely to the slope of failure rate curve $(k-1)$.

We call this technique *iLazy* checkpointing (increasingly lazy, or simply *Lazy checkpointing*) as the new checkpointing interval (α_{lazy}) keeps increasing over time until the next failure; at that point the checkpointing interval is reset to α_{oci}. When failures are exponentially distributed, the iLazy technique automatically reduces to the OCI case, guaranteeing no harm or benefit.

iLazy reduces the checkpointing overhead for failures that occur late, while potentially increasing the wasted work. Since iLazy increases its checkpointing interval over time, it may seem that the waste work penalty is always higher in the iLazy case when compared to the OCI case. However, we illustrate that this is not the case necessarily. As shown in Fig. 11, failure F1 will result in more lost work for OCI than iLazy; however, the reverse is true for failure F2. The cumulative lost work may be higher than the OCI depending on all the failures and their arrival times. This requires an understanding of application execution over its full run.

Therefore, to gain a better understanding of how the proposed iLazy strategy works for an application's execution, we compare it with the OCI in terms of checkpoint overhead, wasted work and computation (Fig. 13). This example illustrates how the

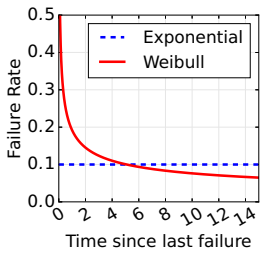

Figure 12. Failure rate (MTBF 10 hrs).

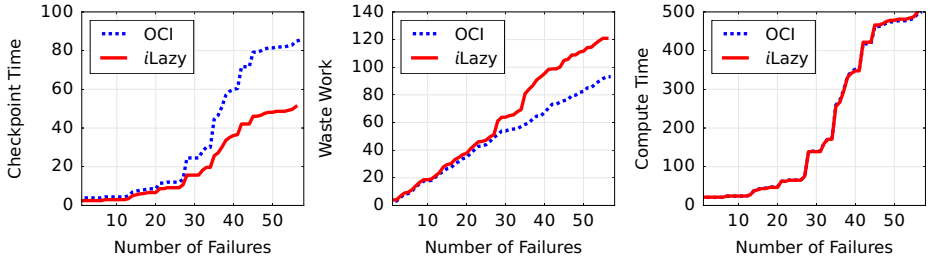

Figure 13. Comparing execution progress of iLazy and OCI techniques.

	Petascale (20K nodes)		Exascale (100K nodes)	
	Checkpoint Time	Total Run Time	Checkpoint Time	Total Run Time
iLazy	34%	-0.45%	24%	1.76%
Increased OCI	25%	0.15%	19%	1.75%
iLazy on top of Increased OCI	51%	-3.45%	38%	0.93%

Figure 14. Effect of applying iLazy on top of increased OCI on different scale of systems. Note that in this case, increased OCI achieves a slight performance improvement over OCI, since the OCI was determined assuming exponential distribution instead of Weibull distribution, and hence, may not be the true optimum.

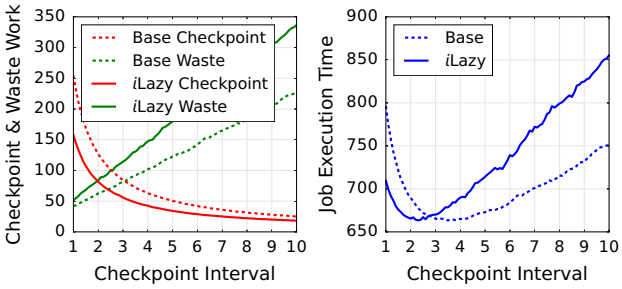

Figure 15. Effect of applying iLazy for different checkpointing intervals. The base case refers to the simulation of an application run on 20K nodes, using different checkpoint intervals without iLazy.

iLazy checkpointing strategy significantly reduces the checkpointing overhead, albeit with increase in the amount of lost work.

Fig. 13 shows results from our event-driven simulator for a run across 20K nodes, with a computational time of 500 hours per node, a time-to-checkpoint of 30 minutes, a Weibull failure distribution with $k = 0.6$, and model-estimated OCI of 2.98 hours. For a fair comparison, both the iLazy and OCI schemes use the same failure arrival times. We notice that the cumulative checkpointing overhead reduces significantly (iLazy is better than OCI by 34% in the checkpoint overheads) with increase in cumulative lost work, resulting in only 0.45% performance hit. *By reducing the checkpoint overhead, Lazy checkpointing is able to reduce the load and contention on the storage subsystem, and amount of data moved.*

Is iLazy more beneficial than simply increasing the OCI?: iLazy reduces the checkpointing overhead significantly with minimal performance degradation. However, one may argue that this reduction in checkpointing overhead can also be possibly obtained with a larger checkpointing interval compared to the OCI since the execution time curve is relatively flat near the OCI region (Fig. 9).

To test this, we increased the OCI by the same percentage gain achieved by iLazy for the checkpoint overhead (Fig. 14). For example, for a petascale system since iLazy provides 34% checkpoint time reduction, we increased the OCI by 34% (referred as Increased OCI). Increasing the OCI results in a 25% checkpoint time reduction. Next, we apply our iLazy technique assuming increased OCI as our base OCI (i.e., increased OCI becomes the α_{oci} in Eq. 11) to assess if iLazy can still reduce the checkpointing overhead. We note that applying iLazy on top of that *further* reduces the overhead significantly compared to the original OCI (by 51% and 38%, third row in Fig. 14), albeit with a small performance degradation.

Observation 5. *The iLazy checkpointing technique can provide more reduction in checkpointing overhead than what is possible by simply increasing the OCI by the same proportion.*

While iLazy does mitigate the checkpointing overhead, it affects the performance, especially when applied with increased OCI (Fig. 14). Therefore, we dig deeper to understand the strengths and limitations of iLazy.

Understanding the strengths and limitations of iLazy:

Fig. 15 plots (using the event-driven simulator) the checkpoint overhead, the wasted work and the total execution time for different checkpoint intervals, for both iLazy and the base case. The base case is simply a plot of the above three aspects of the application at various checkpoint intervals (including OCI).

First, we observe (Fig. 15 (left)) that iLazy consistently provides checkpoint savings for all intervals. This is consistent with our previous result that iLazy reduces the checkpointing overhead even when OCI is increased. Also, the checkpointing overhead curve for the base case explains the decrease in checkpoint overhead when checkpoint interval is increased beyond OCI.

Interestingly, we observe from Fig. 15 (right) that if the "operating checkpointing interval" is smaller than the OCI or near the OCI, then iLazy provides both a significant reduction in checkpointing and runtime as the I/O savings offset the lost work in this region. However, iLazy's total runtime may still not be lower than the OCI's runtime.

However, if the "operating checkpointing interval" is much larger than the OCI, the checkpoint savings decrease significantly and the performance degradation is noticeable. The reason is that as the checkpointing interval grows, the wasted work relative to the base case increases; at the same time, checkpointing savings decrease due to longer checkpoint intervals.

Observation 6. *The iLazy checkpointing technique can significantly mitigate the checkpoint overhead if the checkpoint interval being used is smaller or nearby the OCI. The iLazy checkpointing can be viewed as a technique to reap the same benefits as OCI, even when OCI may not have been very accurately estimated.*

iLazy vs incrementally increasing checkpointing interval:

Next, we investigate the effects of shaping the checkpointing intervals with an alternative function to the one used in iLazy (Eq. 11). Essentially, we wish to understand the additional benefits of checkpoint placement, guided by a Weibull distribution. We compare against a simple linearly increasing function, i.e., α_{oci},

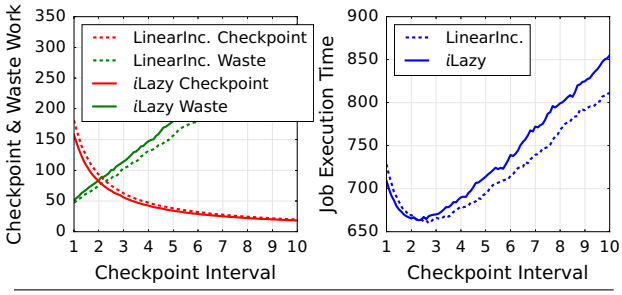

Figure 16. Comparing iLazy with an alternative linearly increasing checkpointing strategy.

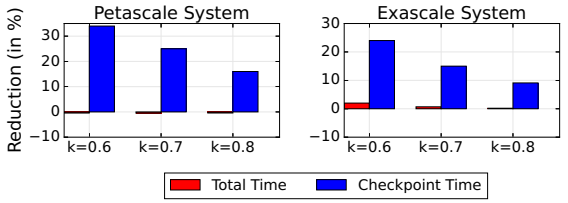

Figure 17. Impact of system size and shape parameter (k) on iLazy benefits.

$\alpha_{oci} + x$, $\alpha_{oci} + 2x$, and so on. Unfortunately, a linear function does not capture the slope of the failure rate (shape parameter k), therefore it requires tuning. Fig. 16 compares iLazy with this approach, with tuned x value of 0.10 for $k = 0.6$. Due to its linear nature, the lost work is less compared to iLazy, and as expected the checkpoint savings are comparatively less as well. Overall, this may work well in practice as an approximation to iLazy and can be useful when the operational OCI is much larger than the true OCI (providing less performance degradation compared to iLazy).

Evaluating iLazy for different shape parameter, system-scales, and I/O bandwidth:

Next, we quantify the benefits of iLazy under various scenarios. Figs. 17 and 18 show iLazy's benefit for different shape parameters (k), different system sizes, and I/O bandwidth (time-to-checkpoint).

First, as the shape parameter increases (Fig. 17 (left)), the benefits decrease relatively, because the temporal locality decreases with the increasing shape parameter. But, the checkpoint savings are significant with minimal performance degradation (more than 10% checkpointing savings with less than 0.5% performance degradation).

Second, iLazy's improvements are sustained across different system sizes. At exascale (Fig. 17 (right)), iLazy is expected to provide more benefits than petascale as the OCI decreases (hence, more checkpoints). However, due to the increased failure rate, iLazy checkpointing suffers from increased wasted work. Thus, while the benefits may not increase compared to a petascale system with the same I/O bandwidth, it remains significant (approx. 25%, 15% and 10% for different shape parameters in the presence of low I/O bandwidth with less than 1% performance hit).

Third, our results show that iLazy provides more improvement under high I/O bandwidth availability for both peta and exascale systems (Fig. 18). This is because under better I/O bandwidth (lower time-to-checkpoint), OCI decreases and hence, more checkpoints occur. Consequently, there is more opportunity for iLazy to improve upon. This is particularly important for future generations of supercomputers, where SSD-based storage systems are likely to provide much higher I/O bandwidth. Typically a checkpoint saving technique looses some of its shine when high-bandwidth storage

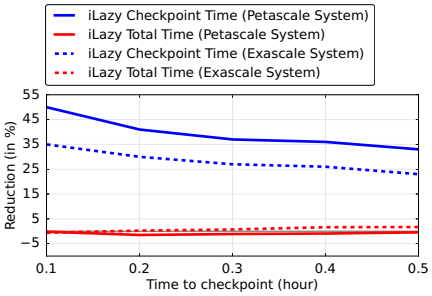

Figure 18. Impact of I/O bandwidth (time-to-checkpoint) on iLazy benefits.

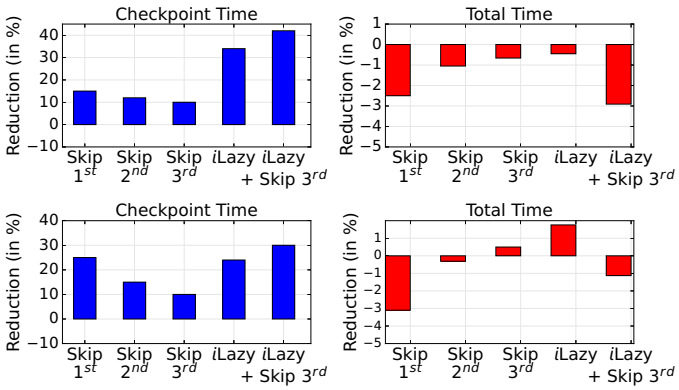

Figure 19. Comparing Skip and Lazy Checkpointing technique for petascale (20K) system (top), and exascale (100K) system (bottom).

systems are adopted, however our iLazy checkpointing becomes even more attractive.

Observation 7. *The iLazy checkpointing technique is likely to become even more attractive in the future as supercomputing facilities will adopt high-speed SSD-based storage systems.*

Skip checkpointing strategy:

We also propose an easy alternative method to reduce checkpointing overhead. We refer to it as "Skip" checkpointing as it skips certain checkpoints after each failure. The intuition behind this simple strategy is that a later checkpoint after a failure (say, third checkpoint) is relatively less costly to skip, in terms of performance degradation, compared to skipping checkpoints immediately after a failure (for example, first checkpoint). The underlying reason is that the temporal locality in failures suggests that failure is relatively less likely to occur much after a failure.

Fig. 19 shows the performance and checkpoint overhead of different variations of the Skip Checkpointing technique. As expected, skipping the first checkpoint after a failure results in more savings in checkpointing time than skipping second or third checkpoint. This is because the total number of second or third checkpoints are lesser than the number of first checkpoints as failures are likely to happen soon after a failure than much later. However, skipping the first checkpoint after a failure results in higher performance degradation. Skipping later checkpoints may still provide significant checkpointing savings without incurring much performance degradation. Therefore, the Skip checkpointing strategy can act as an useful, static checkpoint overhead reduction technique.

Observation 8. *Skipping later checkpoints after a failure, due to the temporal locality of failures, can reduce the checkpoint overhead as well. Coupled with iLazy, it mitigates the checkpointing overhead more than what iLazy technique alone can achieve.*

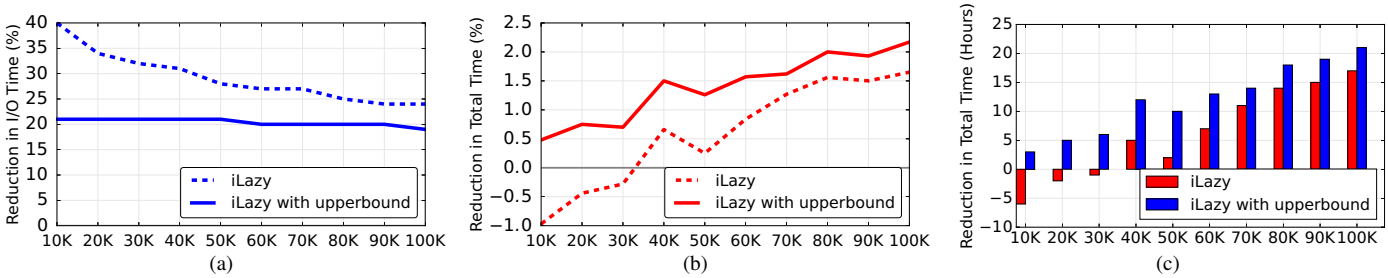

Figure 20. Benefits of iLazy with an upper bound checkpointing interval compared to the base OCI case across different scale of systems (number of nodes on the x-axis): (a) checkpoint time, (b) total time, and (c) absolute performance savings in hours. (checkpoint time = 30 minutes, k =0.6) note the difference in y-axis scales.

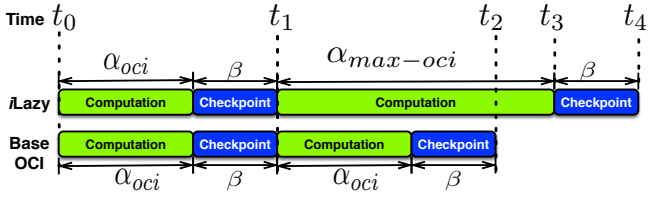

Figure 21. Modeling the upper bound on the checkpointing interval for no performance degradation.

Providing analytical performance bounds on the iLazy checkpointing strategy:

We note that iLazy checkpointing technique reduces I/O overhead significantly, however it may slightly increase the job runtime even when the OCI is correctly estimated (Fig. 19). In some situations, even such minimal performance loss may not be desirable. Therefore, we develop a mathematical model to provide no-performance loss guarantees, at the cost of potentially decreased reduction in I/O overhead.

The primary reason for performance loss lies in the inherent nature of the iLazy checkpointing scheme: the checkpoint interval becomes increasingly large. Consequently, in the cases where inter-arrival time between failures may be considerably high, the amount of lost work may negate the savings coming from infrequent checkpointing. Therefore, to avoid any performance degradation, at any given point we have to estimate if the checkpoint cost saving is not smaller than the potential lost work. If so, the checkpoint interval can not be larger than that. These trade-offs bound how large a checkpoint interval can be. Unfortunately, estimating this at the run time is difficult because it involves calculating the checkpointing cost saving, which in turn requires it to be compared with the traditional OCI based scheme.

We reduce this problem to a relatively simpler case. We propose a simpler and conservative estimation of the largest possible checkpointing interval such that no performance loss is incurred. We focus only on how large the second checkpointing interval can be, without degrading performance.

The proposed scenario is depicted in Fig. 21. If the second checkpoint interval ends at t_3, then we ask what is the maximum value of t_3 (resulting in the maximum allowed checkpointing interval, $\alpha_{max-oci}$) such that the potential benefit of reducing the checkpointing cost is more than the amount of lost work compared to the base OCI (α_{oci}).

The amount of "additional" lost work compared to the OCI case can be estimated in two steps: (1) calculating the probability of failure in the time period between the end of second checkpoint in the OCI case and the iLazy case (i.e., t_2 and t_4), and (2) multiplying this probability by the additional lost work ($(\alpha_{max-oci} - \alpha_{oci})$), if a failure did occur in this time window.

$$\text{performance loss} = (\alpha_{max-oci} - \alpha_{oci})(\mathrm{e}^{-(\frac{t_2}{\lambda})^k} - \mathrm{e}^{-(\frac{t_4}{\lambda})^k})$$
$$= (\alpha_{max-oci} - \alpha_{oci})(\mathrm{e}^{-(\frac{2(\alpha_{oci}+\beta)}{\lambda})^k} - \mathrm{e}^{-(\frac{\alpha_{max-oci}+\alpha_{oci}+2\beta}{\lambda})^k}) \quad (12)$$

Note that the probability that an event happens between time t_x and t_y is given by $Pr(t_x, t_y) = \mathrm{e}^{-(\frac{t_x}{\lambda})^k} - \mathrm{e}^{-(\frac{t_y}{\lambda})^k}$ (for Weibull distribution).

The benefit can be estimated as one checkpointing cost saving (β) multiplied by the probability that the failure happens beyond time t_3.

$$\text{performance gain} = \beta \mathrm{e}^{-(\frac{t_3}{\lambda})^k} \quad (13)$$

Therefore, by solving the following inequality, we can obtain the maximum value of $t3$ that guarantees no performance degradation:

$$\beta \mathrm{e}^{-(\frac{\alpha_{max-oci}+\alpha_{oci}+\beta}{\lambda})^k} = (\alpha_{max-oci} - \alpha_{oci})\mathrm{e}^{-(\frac{2(\alpha_{oci}+\beta)}{\lambda})^k}$$
$$-(\alpha_{max-oci} - \alpha_{oci})\mathrm{e}^{-(\frac{\alpha_{max-oci}+\alpha_{oci}+2\beta}{\lambda})^k} \quad (14)$$

If the α_{lazy} (Eq. 11) is larger than the "upper bound" of the checkpoint interval as determined by the above equation (Eq. 14), then the checkpoint interval is capped at $\alpha_{max-oci}$.

Note that our estimation of the maximum value of checkpoint interval is conservative as doing the same cost-benefit analysis at a later point (after actually saving multiple checkpoints) is likely to result in higher benefits. In our evaluation (Fig. 20), we found that even this conservative estimation obtained by a first-order approximate cost-benefit model can work fairly well. In our experience, we found the trends to remain the same across different settings. However, this analysis depends on multiple factors, such as the base OCI, time-to checkpoint and shape parameter, therefore, a more detailed model, analysis and tuning may be required in certain cases. Our results show that the the iLazy strategy with an upper bound retains a significant amount of the I/O reduction provided by the base iLazy scheme, without degrading performance. We note that it provides approx. 20% reduction in checkpoint time. In fact, it may improve performance by a few hours as opposed to the naive iLazy checkpointing scheme that results in slight performance degradation at low node counts (Fig. 20(c)).

Observation 9. *Using probabilistic estimation, an upper bound on the increasing checkpoint interval can be achieved to avoid performance degradation for the iLazy checkpointing strategy. This capping retains a significant amount of original checkpointing cost savings in the presence of other varying factors, with no performance degradation guarantee.*

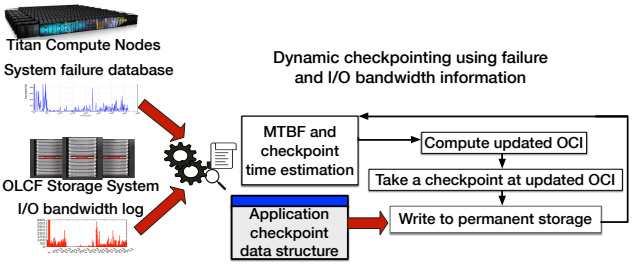

Figure 22. Schematic of our prototyped dynamic checkpointing tool.

6. Prototype Implementation

Previous sections presented model and simulation-based results, using statistically generated events to mimic real-world scenarios. In this section, we discuss the prototype implementation of the proposed checkpointing schemes, its integration with a practical checkpoint restart library, and the evaluation based on real logs from the Titan supercomputer. Our trace-driven study uses approx. six months of real failure and I/O traces from Titan and Spider to evaluate the prototype system. We perform trace-driven evaluation since it is not possible to do such long-duration experiments on a supercomputer due to allocation restrictions. Traces encapsulate the dynamic I/O and failure behavior of these systems – the aspect that we want to evaluate.

6.1 Checkpoint Schemes with Checkpoint/Restart Library

We implemented support for several checkpointing strategies (i.e. static OCI, dynamic OCI, iLazy, and Skip strategies) in an application-level checkpoint/restart (C/R) library [18] from the Indiana University. Static OCI uses historical machine MTBF and historically observed average I/O bandwidth. However, dynamic OCI scheme uses a moving average of failure inter-arrival times and estimates average I/O bandwidth (when the application writes its first checkpoint) to calculate the OCI. Hence, under the dynamic OCI scheme the OCI may change over time (and across different runs) reflecting the variations in observed MTBF (and I/O bandwidth). The C/R library allows users to provide a pointer to a data structure that needs to be saved. Library function calls are provided to easily backup this checkpoint to a persistent store, and restart from a previously saved checkpoint. Our implementation adds adaptive control of checkpointing intervals in a separate thread. Fig. 22 shows a block diagram of the components added to the C/R library.

We have implemented a failure log agent and an I/O log agent within the C/R library to query the failure and I/O log databases that Titan and Spider make available. Spider updates the I/O throughput data from the controllers periodically to the I/O database; Titan updates the system/console logs in the failure database. The failure log agent queries the database to obtain inter-arrival times of any new failure events; the I/O log agent queries Spider data for current and historical I/O throughput. We have implemented the I/O and failure log agents within the C/R library to show that it can be integrated within a checkpoint library. However, these entities can also stand alone as system-wide services that can be queried upon by applications or checkpoint libraries.

We need to determine an appropriate time to start the next checkpoint in accordance with the checkpointing strategies (i.e., static OCI, dynamic OCI, Skip or iLazy). A checkpoint timer attribute in the checkpointing thread is set to expire at the start time of the next checkpoint. Recall that the Skip and iLazy strategies are temporal locality-aware and therefore the checkpoint timer is dependent upon time elapsed since the last failure. Therefore, we

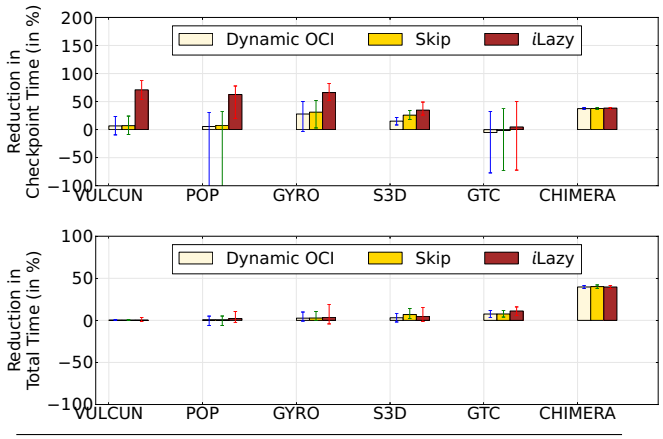

Figure 23. Log driven evaluation results showing the impact of dynamic checkpointing on checkpoint I/O time and total execution time, with respect to the static OCI scheme (with min and max bars).

retain the timestamp of the most recently observed failure in the failure agent. Due to possible lag in updating the failure database, this timestamp is maintained in the C/R library by adding additional attributes to specify a restart timestamp when the application resumes from a previously stored checkpoint. A lag in updating I/O log does not affect our approach because we use an average observed statistics. Note that we have used the base iLazy scheme (with no upperbound) in our prototype for simplicity and show that even the base iLazy is effective in improving performance and reducing checkpointing cost in a dynamic environment.

6.2 Results of Log-Driven Evaluation

We evaluate the C/R library with our strategies using six months worth of failure logs and I/O logs from Titan and Spider. Fig. 23 shows the savings in execution time and checkpoint I/O time observed for different scientific applications (with min and max bars). The applications are run multiple times over the failure and I/O log (without any look-ahead or prediction).

For applications with a relatively small checkpoint size (VULCUN, POP and GYRO) the checkpointing interval of static OCI is relatively small. While the dynamic OCI and Skip strategies can adapt the checkpoint interval on-the-fly and save I/O by skipping some checkpoints, iLazy achieves the most I/O savings. Although the relative savings in I/O for S3D and GTC is small compared to VULCUN, POP and GYRO, we see more impact on their total execution time. For CHIMERA, with a 160TB checkpoint, the average I/O bandwidth used by the static OCI causes it to spend significant amount of time in checkpoint I/O. On the other hand, the dynamic OCI reduces the checkpoint frequency and shows significant savings in I/O time as well as total execution time. iLazy may not be able to provide I/O savings in some pathological cases (e.g. GTC) due to I/O vagaries, but it still provides performance gains compared to static OCI. In summary, iLazy is effective in saving I/O time by up to 70% with respect to static OCI.

In Table 3, the average volume of checkpoint data written to a persistent storage system is shown for our scientific applications. This further elaborates the impact of the strategies on the I/O subsystem. The dynamic OCI, Skip, and iLazy schemes show a significant reduction in the write volume (4.02 PB, 4.48PB, and 5.18PB respectively). The relative saving in the data volume is consistent with the observed reduction in I/O time. This shows that the savings in I/O time is not a result of fortunate placement of checkpoints when higher I/O bandwidth is available. Also, while

Average volume of checkpoint written (in TB)				
Application	Static OCI	Dynamic OCI	Skip	iLazy
VULCAN	84	77	77	11
POP	193	146	142	47
GYRO	103	73	70	26
S3D	1,070	829	720	563
GTC	1,177	1,467	1,412	1,283
CHIMERA	21,752	17,668	17,367	17,143

Table 3. Total volume of checkpoint data written for leadership applications with different checkpointing strategies.

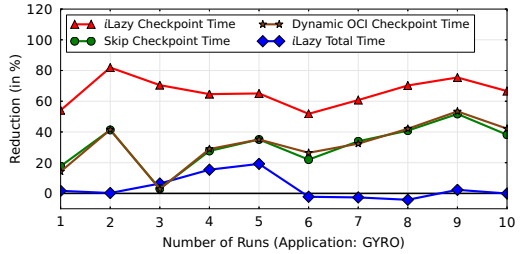

Figure 24. Case study of application GYRO showing the I/O and total execution time for 10 different runs.

the Skip strategy may not always provide a reduction in run-time, it is effective in reducing the data movement.

Fig. 24 shows results for application GYRO for its ten different runs, based on our log-driven evaluation. In general, iLazy achieves significantly better I/O time reduction for all phases and only 4 out of 10 runs see an increase in the total runtime. This increase is small and overall GYRO enjoys a reduction in runtime by up to 19.2% and 3.6% on average, while saving 66.1% in I/O time (74.8% in write volume) on average.

Observation 10. *Our prototype, based on failure locality-aware checkpointing schemes, provides significant I/O improvements in most cases under dynamic environment (unpredictable I/O bandwidth and failure strikes), as shown by our log-based evaluation.*

7. Related Work

Both checkpointing and failure analysis have been long-studied topics in HPC community. Several studies have explored the theoretical underpinnings of the optimal checkpoint interval [37, 36, 7]. Other works have built a Markovian queuing model for combined multiple checkpointing strategies and modeling their effect on the overall performance [26]. They have modeled the effects of mode changes on computer systems performance [20] and derived the optimal checkpointing frequency for minimizing the job response time in a queue [19].

There has also been extensive research on analyzing failure logs of HPC systems [9, 21, 31, 14, 35, 34, 12]. Our study reaffirms some of the previous findings and shows that they hold true in other HPC facilities as well. We also show that optimal checkpointing interval can still work well in practice as an approximation even if failure inter-arrival times are better fitted by Weibull distribution (instead of exponential distribution as assumed by prior works). However, none of the previous works have exploited the temporal locality in failures to mitigate the checkpointing overhead on HPC systems. Using the statistical properties of failure inter-arrival times, the Lazy checkpointing technique shows how to intelligently increase the checkpointing interval such that the I/O overhead is reduced significantly without degrading the overall performance.

Prior works on checkpointing have primarily focused on optimizing checkpointing process and avoiding checkpointing (due to well-known high I/O overhead of checkpointing). Techniques that seek to optimize checkpointing process either reduce the checkpoint data to be written [2] or provide mechanisms for writing checkpoints more quickly [32, 25, 27]. These approaches are complementary to our Lazy and Skip checkpointing strategies. We have also shown that our techniques remain effective even when the time-to-checkpoint is reduced. Therefore, such techniques can be combined to further reduce the I/O overheads.

Checkpoint avoidance, although orthogonal to our proposed schemes, is also a promising way to mitigate the I/O overhead on HPC systems. Checkpoint avoidance is typically achieved either via redundant execution [10] or developing algorithmically fault tolerant codes [4, 8]. Unfortunately, redundant execution wastes excessive compute resources and may require three replicas for correct execution. Algorithm-based fault tolerance techniques are not generic and require significant algorithmic rethinking and implementation efforts. Many applications, including legacy codes, may not benefit from this approach.

Recent studies [13, 5] have tried to reduce the checkpointing overhead by predicting failures and speculatively placing checkpoints accordingly. These schemes rely on machine-learning techniques to analyze large training data and are susceptible to environment changes. Failure prediction, the basis for these activities, is fundamentally harder [11, 5] and requires detailed logging information, which may not be turned on in production systems for performance reasons. In contrast, our scheme is relatively simple and only requires minimal, high-level information, making it relatively more practical.

8. Conclusion

Using both analytical modeling and simulation-based verification, we studied the interplay between checkpointing, the I/O overhead and the compute resource wastage due to system failures. We discovered that system failures on leadership computing facilities have temporal locality. We proposed two techniques, Lazy and Skip checkpointing, to take advantage of temporal locality in failures. We throughly evaluated our techniques using both simulations and a prototype based on large-scale system I/O traces and failure logs.

We believe that our findings will be useful for end-users and system designers in understanding the trade-offs of checkpointing and resource wastage at different system scales. Also, our techniques can possibly be applied in other checkpointing domains where fault inter-arrival times may be fitted by a Weibull distribution. For example, these proposed techniques may also be extended to hardware checkpointing strategies to recover from soft errors, dynamic DVFS scaling after failures.

9. Acknowledgement

We thank the reviewers for constructive feedback that has significantly improved the paper. This work was supported by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725). Authors thank OLCF members for help, especially Don Maxwell and Raghul Gunasekaran.

References

[1] Computational science requirements for leadership computing, 2007, `http://www.olcf.ornl.gov/wp-content/uploads/2010/03/ORNL_TM-2007_44.pdf`.

[2] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286. ACM, 2004.

[3] John Bent, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. Storage challenges at los alamos national lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5. IEEE, 2012.

[4] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[5] Mohamed Slim Bouguerra, Ana Gainaru, Leonardo Bautista Gomez, Franck Cappello, Satoshi Matsuoka, and Naoya Maruyam. Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 501–512. IEEE, 2013.

[6] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[7] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[8] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[9] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail, DSN. 2013.

[10] Kurt Ferreira, Jon Stearley, James H Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2011.

[11] Ana Gainaru, Franck Cappello, Joshi Fullop, Stefan Trausan-Matu, and William Kramer. Adaptive event prediction strategy with dynamic time window for large-scale hpc systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, page 4. ACM, 2011.

[12] Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale hpc systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1168–1179. IEEE, 2012.

[13] Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale hpc systems. In *Euro-Par 2011 Parallel Processing*, pages 52–64. Springer, 2011.

[14] Bianca S Garth. A large-scale study of failures in high-performance computing systems. 2006.

[15] DeGroot Morris H. and Mark J. Schervish. Probability and statistics. 3rd ed. boston, ma: Addison-wesley, 2002.

[16] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[17] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[18] Joshua Hursey, Scott S. Hampton, Pratul Agarwal, and Andrew Lumsdaine. An adaptive checkpoint/restart library for large scale hpc applications. *14th SIAM Conference on Parallel Processing and Scientific Computing,*, 2010.

[19] VG Kulkarni, VF Nicola, and KS Trivedi. Effects of checkpointing and queuebsfg on program performance. *Stochastic models*, 6(4):615–648, 1990.

[20] Vidyadhar G Kulkarni, Victor F Nicola, and Kishor S Trivedi. On modelling the performance and reliability of multimode computer systems. *Journal of Systems and Software*, 6(1):175–182, 1986.

[21] T-TY Lin and Daniel P Siewiorek. Error log analysis: Statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.

[22] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.

[23] Adam Manzanares, John Bent, Meghan Wingate, and Garth Gibson. The power and challenges of transformative i/o. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 144–154. IEEE, 2012.

[24] Ross Miller, Jason Hill, G Raghul DDA, GM Shipman, and D Maxwell. Monitoring tools for large scale systems. *CUG10*, 2010.

[25] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[26] Victor F Nicola and Johannes M Van Spanje. Comparative analysis of different models of checkpointing and recovery. *Software Engineering, IEEE Transactions on*, 16(8):807–821, 1990.

[27] Bogdan Nicolae and Franck Cappello. Blobcr: efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 34. ACM, 2011.

[28] 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 *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 30–46. IEEE, 2007.

[29] Preparing for Exascale: ORNL Leadership Computing Facility Application Requirements and Strategy, 2009, `http://www.olcf.ornl.gov/wp-content/uploads/2010/03/olcf-requirements.pdf`.

[30] James S Plank and Wael R Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 48–57. IEEE, 1998.

[31] Ramendra K Sahoo, Mark S Squillante, A Sivasubramaniam, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Dependable Systems and Networks, 2004 International Conference on*, pages 772–781. IEEE, 2004.

[32] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 19. IEEE Computer Society Press, 2012.

[33] Bianca Schroeder and Garth A Gibson. The computer failure data repository (cfdr). In *Workshop on Reliability Analysis of System Failure Data (RAF'07), MSR Cambridge, UK*, 2007.

[34] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[35] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, 2010.

[36] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *Computers, IEEE Transactions on*, 46(8):942–947, 1997.

[37] John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.