2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Shiraz: Exploiting System Reliability and Application Resilience Characteristics to Improve Large Scale System Throughput

Rohan Garg, Tirthak Patel, Gene Cooperman, and Devesh Tiwari
Northeastern University

Abstract

Large-scale applications rely on resilience mechanisms such as checkpoint-restart to make forward progress in the presence of failures. Unfortunately, this incurs huge I/O overhead and impedes productivity. To mitigate this challenge, this paper introduces a new technique, Shiraz, which demonstrates how to exploit differences in the checkpointing overhead among applications and knowledge of temporal characteristics of failures to improve both the overall system throughput and performance of individual applications.

1. Introduction

The Problem: Continued increase in computing power has enabled computational scientists to expedite the scientific research and discovery process in the past. Unfortunately, significant rise in the failure rates and a widening gap between compute and I/O system will significantly limit the usability of parallel computing systems in the future [13, 18, 25].

Computational science applications rely on resilience mechanisms such as checkpoint-restart to make forward progress in the presence of failures. Although checkpoint-restart mechanisms can keep scientific simulations moving forward, writing and reading application state incurs large I/O overhead, which impedes scientific productivity. Current large-scale scientific applications spend more than 15% of the total execution time on resilience mechanisms (e.g., checkpoint-restart) [13, 18]. At exascale, computational science applications will need to spend more than 40% of execution time on resilience mechanisms, due to orders of magnitude higher failure rate at exascale [18, 19, 40].

There have been numerous efforts to derive the optimal checkpointing interval (OCI) for an application, given the mean time between failures and the application's checkpointing overhead [14, 42]. Essentially, OCI attempts to maximize the amount of useful work done per failure for a given application. There have been several other studies that propose further refinements to OCI estimations. However, previous work has not explored how to maximize the useful work done per failure from the system's point of view, where multiple applications with different checkpointing overheads are available. To the best of our knowledge, no previous study has investigated maximizing the system throughput of a large-scale HPC system by leveraging variations in OCI's of scientific applications and knowledge of temporal characteristics of system failures.

Goal and Key Idea: The goal of this paper is to **demonstrate that variations in checkpointing overhead among scientific applications and knowledge of temporal characteristics of failures can be exploited to improve the overall system throughput**. *The key idea is to schedule applications with higher checkpointing overhead during periods of relatively high reliability (with a lower failure rate), while applications with lower checkpointing overhead are scheduled during periods with relatively low reliability (with a higher failure rate)*. The intuition behind this idea comes from the following insight. Applications with higher checkpointing overhead have a relatively large optimal checkpointing interval and hence, the amount of average lost work per failure is also higher. Therefore, scheduling an application with higher checkpointing overhead during periods of relatively higher reliability is likely to result in lower overall lost work. By scheduling those applications having lower checkpointing overhead during periods of lower reliability (higher system failure rate), the amount of lost work per failure can be decreased. Therefore, these schemes combined together can increase the useful work done per failure occurrence. But, it is challenging to effectively design a scheme based on this idea for several reasons.

Challenges: First, the scheme relies on timely and accurate identification of time periods with varying failure rates. Second, while the scheme improves the system throughput, it also needs to ensure that the performance of individual applications is not degraded. Third, the system failure rate continually changes over time. Therefore, it is critical to adapt to the changing failure rate by switching between applications with different checkpointing overheads.

To this end, this study answers the following questions: *(1) How to accurately identify and quantify changing reliability characteristics of a system? (2) How to leverage the above information to schedule applications with different checkpointing overheads, such that the overall system throughput is improved without hurting individual applications?* Our study is based on real system experiments, analytical models, and statistical techniques. This work is grounded by theoretical foundations, driven by extensive evaluation through real-world experiments and through simulation, and guided by real-world large-scale HPC system parameters.

2158-3927/18/$31.00 ©2018 IEEE
DOI 10.1109/DSN.2018.00021

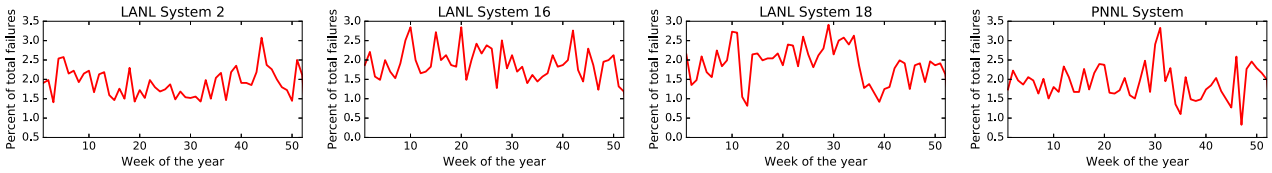

Figure 1: Temporal failure distribution on weekly basis for multiple HPC systems.

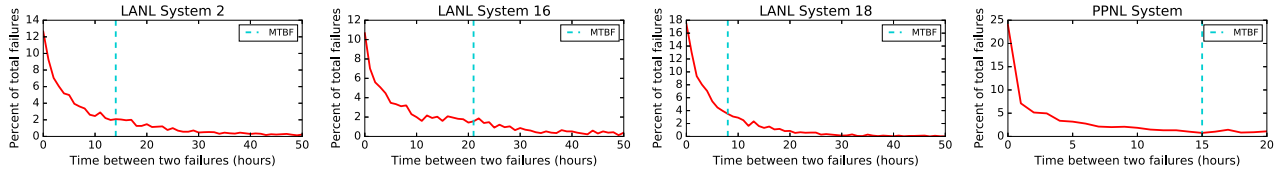

Figure 2: Inter-arrival failure distribution for multiple HPC systems (time between two failures).

Contributions: We leverage information about temporal characteristics of failures and variations in checkpointing overhead among applications to improve system throughput. This work introduces, *Shiraz*[1], a novel scheme, to improve the overall system throughput (defined as total useful work done per unit time) by intelligently scheduling applications with different checkpointing overheads under varying temporal characteristics of system failures. This paper also proposes a novel variant of Shiraz, called *Shiraz+*, which specifically reduces the overall checkpointing overhead of the system while improving the system throughput and maintaining individual application performance levels. Shiraz+ reduces the I/O pressure on the back-end and mitigates storage contention. Therefore, it can also potentially improve the effective I/O performance for other applications running on a large-scale HPC system.

Evaluation: Our evaluation results show that Shiraz improves system throughput under a wide variety of circumstances: on peta- and exa-scale platforms; on a range of checkpointing overheads; and with multiple real-world HPC applications. Our evaluation is based on extensive experimental, modeling, and simulation results, which are guided by real-world large-scale HPC system parameters. For a representative set of real-world large-scale HPC applications, Shiraz is shown to save up to $285,000 on a petascale system and $890,000 on a projected exascale system (with 5 years of anticipated system lifetime) and hence, can effectively pay towards future faster storage subsystem. Shiraz+ reduces the data movement by up to 52% for a variety of applications and system characteristics, without degrading the overall system throughput or individual application performance (Section 5).

[1] Shiraz is a conveniently chosen acronym of **SH**aring **I**ntelligently **R**eli**A**bility **Z**ones. Shiraz is also a type of red wine whose origin is a curious case.

2. Motivation

A naïve strategy for improving system throughput would be to identify periods when the system is distinctly more stable (or less stable) compared to the average period and schedule applications with higher checkpointing overhead (or lower checkpointing overhead). Figure 1 shows that for large-scale HPC systems such distinct periods of stability may not exist and brief stable periods are followed by long periods of fluctuation [1]. We also note that waiting for a period when the system is more reliable can lead to starvation for applications with large checkpointing overheads.

Fortunately, we can find changing failure rate characteristics when we analyze failure characteristics at finer granularity (i.e., inter-arrival times between two failures). Note that failures considered in this study are ones that cause an application to crash and recover from last checkpoint. Figure 2 shows that a large fraction of failures are likely to occur much before the MTBF. We refer to this as the temporal recurrence behavior of failures. This has been shown and modeled extensively for many other current and past HPC systems [9, 17, 36, 38, 40]. This property is captured by the hazard rate of the Weibull distribution which changes between consecutive failures (instead of being constant in case of the exponential distribution). The shape of the hazard rate is primarily characterized by the shape parameter (β). For $\beta < 1$, the hazard rate is high right after a failure, but it decreases over time until the next failure [32].

Multiple prior studies have shown that β varies from 0.4 to 0.7 for HPC systems [9, 36, 38, 40]. We find similar results, but since determining shape parameter is not a main contribution of this work, we omit those results. In summary, one can schedule applications within two failures to exploit changing reliability characteristics.

Next, we show evidence that large-scale scientific applications have significant variations in their checkpointing overhead. Table 1 shows the checkpointing cost of applications from different scientific domains running at different

Table 1: Differences in checkpointing cost among large-scale HPC applications.

Machine	Application Domain	Checkpointing Duration (sec.)
Titan (OLCF)	Climate Change Simulation with the Community Earth System Model	1.5
Hopper (NERSC) Franklin (NERSC)	20th Century Reanalysis	2
Jaguar (ORNL) Hopper (NERSC)	Molecular Simulation in Energy Biosciences	6
Carver and Euclid (NERSC)	Computational Predictions of Trans. Factor Binding Sites	50
Cori (NERSC)	Chombo-crunch	70
Hopper (NERSC)	Climate Science for a Sustainable Energy Future	150
Hopper (NERSC)	Laser Plasma Interactions	1800
Hopper (NERSC)	Plasma Based Accelerators	2000
Hopper (NERSC)	Plasma Science Studies	2700

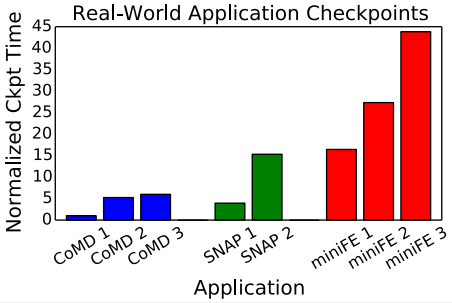

Figure 3: Normalized cost of checkpointing for CoMD, SNAP and miniFE applications for different configurations (experimentally measured using system-level checkpointing [7], normalized to CoMD config-1).

large-scale HPC centers [5, 6]. The checkpoint durations of the applications in the table range from a few seconds to more than half an hour. Other researches have also noted a difference of orders of magnitude in the checkpointing traffic among large-scale HPC applications [23].

To further confirm the existence of this trend, we conducted a real-system experiment where we experimentally measured the cost of checkpointing for three representative applications: CoMD, SNAP, and miniFE [22, 26] using DMTCP system-level checkpointing [7], under three different configurations (Figure 3).

We observed that (1) different applications have widely varying checkpointing overheads (up to a difference of more than 40x), and (2) even the same application can exhibit different checkpointing overheads, depending on the input parameters. These variations in checkpointing overheads open up opportunities for new optimizations in the presence of multiple applications performing checkpointing on large-scale systems that experience system failures.

Next, we will show how Shiraz exploits observations about temporal recurrence of failures and checkpointing overhead to improve overall system throughput.

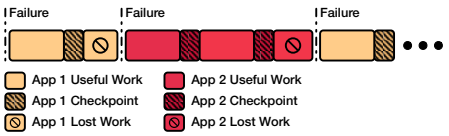

Figure 4: Conventional scheduling (Baseline): Switch between applications after every failure.

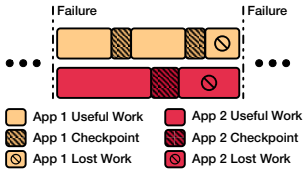

Figure 5: Heavy-weight application is likely to have higher average lost work per failure.

3. Shiraz: Design and Model

In a multi-application environment, a fair scheduler switches the applications at every failure, as shown in Fig. 4. By switching at every failure, the scheduler provides each application an equal chance to do useful work. This traditional approach does not exploit the two key factors discussed in Section 2: temporal recurrence characteristics of failures, and variation in checkpointing cost among applications.

First, we point out that the average lost work due to a failure is different for different types of applications. Fig. 5 shows that an application with higher checkpointing overhead (referred as heavy-weight application) is likely to have higher average lost work compared to an application with relatively lower checkpointing overhead (referred as light-weight application). This is because the optimal checkpointing interval (OCI) for the heavy-weight application is larger than the OCI of light-weight application, according to Daly's formula: $\sqrt{2M\delta} - \delta$, where M is the system MTBF and δ is the checkpoint overhead of the application. Thus, larger OCI leads to higher average lost work due to a failure (Fig. 5).

Implication: It is beneficial to schedule the heavy-weight application when the system MTBF is higher. Unfortunately, it is hard to find consistent higher MTBF periods during the operational time of a system and a suboptimal choice may result in performance degradation for the heavy-weight application (as discussed in Section 2). To address this challenge, we leverage the non-constant failure rate between two failures. The hazard rate decreases between two failures and hence, statistically, the probability of a failure is higher right after a failure and it decreases over time. This observation can be exploited by scheduling the light-weight application before scheduling the heavy-weight application.

Shiraz Key Idea: The key idea is to intelligently schedule applications with different checkpointing overheads between two failures. Shiraz schedules a heavy-weight application during periods with relatively lower system failure rate, while a light-weight application is scheduled during

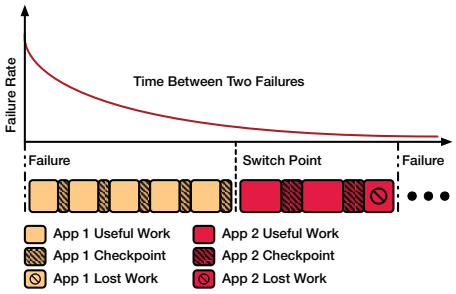

Figure 6: Shiraz switches two applications in between two failures to reduce the overall lost work per failure by scheduling the heavy-weight application during periods with relatively lower system failure rate.

periods with relatively higher system failure rate (as demonstrated in Fig. 6). Scheduling an application with high checkpointing overhead (i.e., larger OCI) during the later part of the failure rate curve is likely to result in lower overall lost work. Similarly, scheduling a light-weight application (i.e., smaller OCI) during the earlier part of the failure rate curve decreases the amount of lost work per failure. Therefore, it increases the useful work done per failure occurrence. However, this creates new challenges.

As Fig. 7 shows, while switching late, in order to avoid failures, may potentially save large amount of average lost work per failure for the heavy-weight application, it can also degrade the performance for the heavy-weight application. This is because the application cannot produce the same amount of useful work as in the baseline, where each application gets a fair share of the runtime. On the other hand, switching too soon (1) exposes the heavy-weight application to a higher failure rate, and (2) degrades the performance of the light-weight application. Therefore, Shiraz encapsulates an analytical model that determines the optimal switching point to dynamically adapt to the failure rate.

The formulation and details of this model are presented below. We refer to the light-weight application as LW and the heavy-weight application as HW. Using Daly's formula, the OCI's for the two applications can be expressed as:

$$\text{OCI}_{\text{LW}} = \sqrt{2M\delta_{\text{LW}}} - \delta \quad \text{and} \quad \text{OCI}_{\text{HW}} = \sqrt{2M\delta_{\text{HW}}} - \delta \tag{1}$$

Where system MTBF, checkpoint overhead for light weight application and heavy weight application are denoted by M, δ_{LW}, and δ_{HW}, respectively.

First, we need to estimate the baseline performance for the two given applications. Recall, that in the conventional scheme, the applications are switched at every failure. Let us suppose that both the applications are executed for a total of T_{total} time. We note that switching at a failure boundary is equivalent to switching after an infinite amount of time since the last failure. This helps in developing a unified framework for modeling both baseline and Shiraz.

Estimating different components of the execution (useful work, checkpoint overhead, and lost work) requires knowing

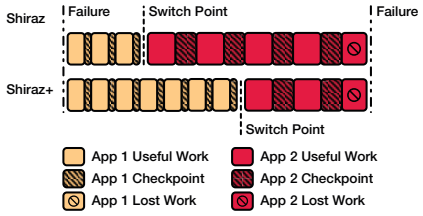

Figure 7: Effect of different switch points between failures.

the number of failures. The number of failures between two time instances (t_{start} and t_{end}) can be estimated as follows:

$$\text{Fail}^{\text{num}}_{(t_{\text{start}},t_{\text{end}})} = \frac{T_{\text{total}}}{M} \times (e^{-(\frac{t_{\text{start}}}{\lambda})^{\beta}} - e^{-(\frac{t_{\text{end}}}{\lambda})^{\beta}}) \tag{2}$$

Where λ and β are the scale and shape parameter for Weibull distribution, respectively (Section 2). We note that the scale parameter can be derived from the MTBF: $\lambda = \frac{M}{\Gamma(1+\frac{1}{\beta})}$. Eq. 2 can be used to derive the total number of failures in time T_{total} as follows.

$$\text{Fail}^{\text{num}}_{\text{total}} = \frac{T_{\text{total}}}{M} \times (1 - e^{-(\frac{T_{\text{total}}}{\lambda})^{\beta}}) \tag{3}$$

In the baseline case, where the application gets switched at every failure, each of the two applications essentially gets to run for $\frac{T_{\text{total}}}{2}$ time (in the baseline case $T_{\text{total}} = \frac{T_{\text{total}}}{2}$). Thus, the total lost work in the baseline case for both applications can be estimated as:

$$T^{\text{LW}}_{\text{lost-base}} = \epsilon \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \times \text{Fail}^{\text{num}}_{\text{total}} \tag{4}$$

$$T^{\text{HW}}_{\text{lost-base}} = \epsilon \times (\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \times \text{Fail}^{\text{num}}_{\text{total}} \tag{5}$$

Where ϵ is the average fraction of lost work per failure. For estimating useful work and checkpointing overhead, we can divide the time segment between two failures in chunks of optimal checkpointing interval plus checkpointing overhead (OCI$+\delta$). For probabilistic modeling, one can imagine that there are infinite such segments and calculate the probability of failure after each segment. Note that the average number of such segments is $\frac{M}{(\text{OCI}+\delta)}$. As discussed previously, the number of failures between time segments i and $i+1$ is given by $\text{Fail}^{\text{num}}_{(i\times(\text{OCI}_{\text{LW}}+\delta_{\text{LW}}),(i+1)\times(\text{OCI}_{\text{LW}}+\delta_{\text{LW}}))}$. As a short hand notation, we denote this as $\text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$. Successful completion of a segment results in useful work equivalent to the optimal checkpointing interval. Therefore, the useful work for the two applications in the baseline case can be mathematically expressed as:

$$T^{\text{LW}}_{\text{useful-base}} = \sum_{i=1}^{\infty} i \times \text{OCI}_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \tag{6}$$

$$T^{\text{HW}}_{\text{useful-base}} = \sum_{i=1}^{\infty} i \times \text{OCI}_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \tag{7}$$

Similarly, the checkpointing overhead per successful segment of $(OCI + \delta)$ is equal to the cost of one checkpoint. Therefore, the I/O overhead in the baseline case is:

$$T^{\text{LW}}_{\text{io-base}} = \sum_{i=1}^{\infty} i \times \delta_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \quad (8)$$

$$T^{\text{HW}}_{\text{io-base}} = \sum_{i=1}^{\infty} i \times \delta_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \quad (9)$$

This approach of modeling leads to an elegant formulation for the Shiraz case as well. The index for the summation terms does not range from 1 to ∞ now. Instead, for the light-weight application, the index will range from 1 to the switching point (k). We refer to the switching point as the number of checkpoints (say, k) the light-weight application takes before yielding to the heavy-weight application. Note that the total time period the light-weight application gets to run is $k \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$. For the heavy-weight application, the index will range from k to ∞. Note that for the heavy-weight application, each of the segments $(i, i+1, \dots)$ are still $(\text{OCI}_{\text{HW}} + \delta_{\text{HW}})$ long, but the first such segment starts after $k \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}})$ time since the last failure. Now, we can write the expressions for useful work, checkpointing overhead, and lost work for the Shiraz case as follows:

$$T^{\text{LW}}_{\text{useful-shiraz}} = \sum_{i=1}^{k} i \times \text{OCI}_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \quad (10)$$

$$T^{\text{HW}}_{\text{useful-shiraz}} = \sum_{i=k}^{\infty} i \times \text{OCI}_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \quad (11)$$

$$T^{\text{LW}}_{\text{io-shiraz}} = \sum_{i=1}^{k} i \times \delta_{\text{LW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \quad (12)$$

$$T^{\text{HW}}_{\text{io-shiraz}} = \sum_{i=k}^{\infty} i \times \delta_{\text{HW}} \times \text{Fail}^{\text{num}}_{i,i+1}(\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \quad (13)$$

$$T^{\text{LW}}_{\text{lost-shiraz}} = \epsilon \times (\text{OCI}_{\text{LW}} + \delta_{\text{LW}}) \times \text{Fail}^{\text{num}}_{\text{LW-fraction}} \quad (14)$$

$$T^{\text{HW}}_{\text{lost-shiraz}} = \epsilon \times (\text{OCI}_{\text{HW}} + \delta_{\text{HW}}) \times \text{Fail}^{\text{num}}_{\text{HW-fraction}} \quad (15)$$

We note that the failure can still occur before k checkpoints of the light-weight application. Our model is probabilistic and hence, sums up the probabilities over all the segments. $\text{Fail}^{\text{num}}_{\text{LW-fraction}}$ refers to the number of failures observed during the time light-weight application runs (i.e., after a failure until k checkpoints, summed over all such periods). Similarly, $\text{Fail}^{\text{num}}_{\text{HW-fraction}}$ refers to the number of failures observed during the time heavy-weight application gets to runs (i.e., after k checkpoints of the light-weight application until the next failure, summed over all such periods).

Where is optimal point (optimal value of k)?: If the goal is to simply maximize the system throughput (useful work done per unit time), one can simply set k to ∞ . However, this results in starvation of the heavy-weight application. In this approach, the system throughput improvement comes from favoring the light-weight application over the heavy-weight application at all times. The key constraint is that both applications should not see any performance degradation compared to the baseline. That is,

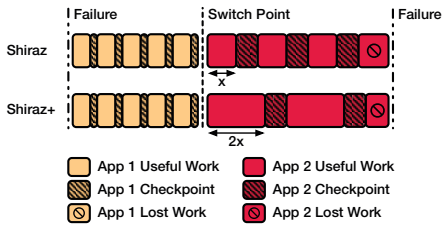

Figure 8: Shiraz+: Reducing the checkpointing overhead.

$$T^{\text{LW}}_{\text{useful-shiraz}} \geq T^{\text{LW}}_{\text{useful-base}} \quad \text{and} \quad T^{\text{HW}}_{\text{useful-shiraz}} \geq T^{\text{HW}}_{\text{useful-base}} \quad (16)$$

Note that a range of values for k will satisfy Eq. 16. The highest of the values of k in this range will be the theoretical optimal switching point. It will result in the maximum useful work done per unit time for the whole system. However, it will not necessarily be fair to both the applications. Recall that increasing k improves the light-weight application's performance (useful work done per unit time), however, it also decreases the heavy-weight application's performance. Therefore, choosing the highest such value of k that satisfies Eq. 16 will result in zero improvement for heavy-weight application. To address this issue, Shiraz choose a suboptimal value of k that provides fairness, i.e., equal benefits to both the applications. Therefore, Shiraz uses the following constraints to derive optimal value of k:

$$T^{\text{LW}}_{\text{useful-shiraz}} - T^{\text{LW}}_{\text{useful-base}} = T^{\text{HW}}_{\text{useful-shiraz}} - T^{\text{HW}}_{\text{useful-base}}$$
$$\text{s.t.} \quad (T^{\text{LW}}_{\text{useful-shiraz}} - T^{\text{LW}}_{\text{useful-base}}) \geq 0$$
$$\text{and} \quad (T^{\text{HW}}_{\text{useful-shiraz}} - T^{\text{HW}}_{\text{useful-base}}) \geq 0$$

Shiraz solves this optimization problem numerically to determine the optimal switching point (k) such that the system throughput is maximized but both applications are treated fairly. Shiraz will return $k = \infty$ if no system throughput improvement can be achieved in the above equation.

Shiraz+ for reducing I/O overhead: Shiraz model demonstrates that choosing optimal switching point can lead to an improvement in system throughput without performance degradation for individual applications, but it does not specifically address the problem of high data movement caused by checkpointing. Checkpointing causes excessive pressure and contention on the I/O subsystem. Therefore, reducing checkpointing overhead leads to alleviating the I/O pressure, reduction in data movement (i.e., higher energy efficiency), and potentially better performance for other applications too. Shiraz+ works on top of Shiraz and trades the additional performance gain obtained by Shiraz to reduce the checkpointing overhead.

The key idea is to increase the checkpointing interval of heavy-weight application (Fig 8). The intuition behind this

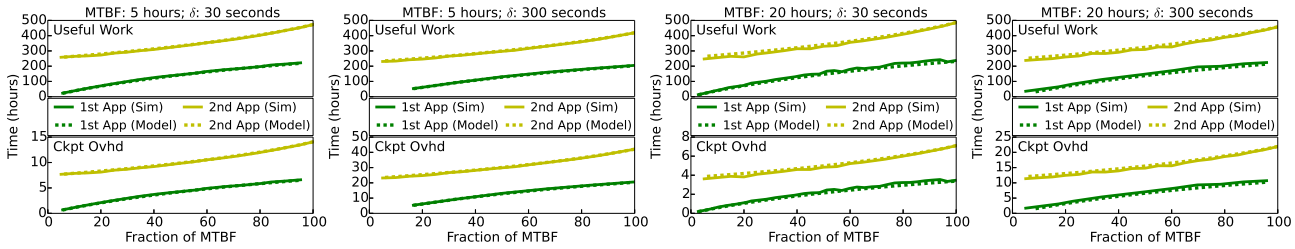

Figure 9: Shiraz model matches with the discrete-event based simulator for a wide range of parameters and scenarios.

idea is simple: heavy-weight application observes effectively higher MTBF and hence, can afford to run at a checkpointing interval that is larger than its OCI (and thus, reduce the I/O cost), though at the risk of losing performance.

Determining the new checkpointing interval for heavy-weight application is a new optimization problem that Shiraz and Shiraz+ open up. But, for this work, Shiraz+ takes a relatively simpler approach and explores increasing the OCI_{HW} by an integer factor $(2\times, 3\times, \dots)$ and evaluating its impact on performance and checkpointing overhead (Section 5). We also note that this is a more practical approach since it does not require the application programmers to change the checkpointing interval to some new value (e.g., $2\times$ stretch in OCI can be emulated at the system level).

Interestingly, Shiraz's choice of a suboptimal value of k helps Shiraz+. The suboptimal value of k ensures that the heavy-weight application's performance also improves. This allows Shiraz+ to trade this performance improvement for a lower checkpointing overhead. If theoretically optimal value of k was chosen, no such opportunity would exist, and increasing OCI for the heavy-weight application will lead to performance degradation.

Shiraz+ does not alter the checkpointing interval of light-weight application for two reasons: (1) it has a lower impact on the overall checkpointing overhead (due to the lower cost of taking one checkpoint); and (2) it requires a re-adjustment to the the optimal switch point, which would further complicate the determination of optimal switching point. Note that Shiraz+ has no impact on the performance or checkpointing overhead of light-weight application. We evaluate Shiraz+ thoroughly under different scenarios, and analyze its impact on I/O overhead and performance in Section 5.

4. Shiraz Model Validation

In this section, we validate the Shiraz model with a discrete-event simulator that simulates multiple applications with different characteristics running on an HPC system.

The goal of the validation is demonstrate that our probability theory based model has accurate estimations when compared to the discrete event simulation. This validation exercise will also form the basis for demonstrating that the optimal switch point predicted by Shiraz model matches with the optimal switch point obtained via extensive simulation (Section 5).

Our discrete-event simulator executes an application with a given checkpointing overhead on a system with a given MTBF. The application takes periodic checkpoints at optimal checkpointing interval. The application restarts from the latest checkpoint when a failure strikes. The failures are generated from a Weibull distribution. Since we are interested in analyzing the switching points between two different types of applications, we simulate two scenarios: (1) *first application*: an application is executed first and after some specified time, it is switched out. From the validation perspective, it is irrelevant what happens after the application of interest is switched out. We need to validate the application characteristics for the time frame for which the application of interest was run; and (2) *second application*: in this case, after some specified time, the application of interest is scheduled and run. From the validation perspective, it is irrelevant what happens before the application of interest is scheduled. Essentially, the goal is to not make any assumptions about relationship between the two applications being run (for example, one light-weight and other heavy-weight).

We simulated a wide range of scenarios and validated our model against the simulation. For brevity, we show validation results only for representative parameters (Fig. 9). We ran both the cases described earlier. An application is switched out or started at different times (expressed as fraction of MTBF for easy interpretation; results were similar for longer time periods.). We simulated an application for a total of 1000 hours under different scenarios: MTBF of 20 hours and 5 hours (representative of peta- and exa-scale systems, respectively), and 30 seconds and 300 seconds checkpointing overhead. Failure events were generated from the Weibull distribution using the shape parameter β of 0.6. The scale parameter is determined by the MTBF and the shape parameter. We only show the useful work and the checkpointing overhead here, since the lost work, by definition, will be validated, if these other two components match. The average fraction of lost work ϵ parameter was estimated to be 0.45. These parameter values match with other studies and are true for many systems [9, 13, 35, 38, 40]. We obtained similar results with other values in the range.

From Fig. 9, we make several observations. First, Shiraz model matches closely with the simulation across a number of parameters and scenarios. We observed similar results for the time longer than the MTBF and other parameters. For example, in the case of the second application, the average

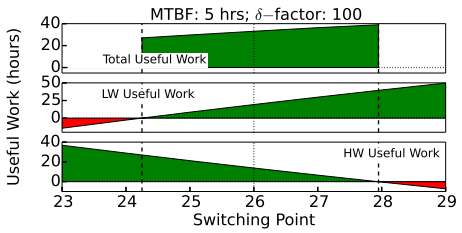

Figure 10: Shiraz identifies optimal switching point and region of interest. Switching point k varies from 24 to 28 – region of interest (no performance degradation). Shiraz's optimal k = 26. The total runtime is 1000 hours; the δ-factor is $100\times$; the MTBF is 5 hours.

difference in the checkpointing overhead between the model and the simulation for petascale and exascale systems is 0.06 hours and 0.14 hours, respectively.

Second, Shiraz model matches the simulation for both the application execution cases. For example, the average difference in the useful work between model and simulation for first and second applications is 2.1 and 2.2 hours, respectively. One can note the lack of data points in the case of the first application in Fig. 9. This is because the data points are limited to integer multiples of application's OCI. Recall that when an application is executed first, it can be switched out only after a checkpoint, and it can invoke a maximum of $\frac{\text{MTBF}}{\text{OCI}}$ checkpoints before getting switched out. However, for second application case, we can assume any arbitrarily small OCI for the application that ran first and was switched out. This implies that we can switch in the second application at any point resulting in a smoother validation curve.

Finally, the model matches well for both applications for both useful work and checkpoint overhead. For example, application with 300 sec checkpointing overhead for petascale system observes less than 3 hours and 0.5 hours of average difference between model and simulation. This is critical as the choice of an optimal switching point relies on accurate estimation of both components.

5. Evaluation

In this section, we answer the following questions:
Q1. *Does an optimal switching point between two applications with different checkpointing overheads exist?*
Q2. *Can Shiraz determine optimal switching point accurately and improve the overall system throughput?*
Q3. *Is Shiraz effective with real-world applications and produce significant energy savings?*
Q4. *Can Shiraz+ reduce the data movement caused by checkpointing under different scenarios? If so, what is the impact on system throughput and application performance?*
Q5. *Are Shiraz and Shiraz+ effective in improving throughput and reducing I/O overhead for representative applications on a real-system?*

Optimal Switching Point: First, we show that an optimal switching point exists, given two applications with different checkpointing overhead, such that the overall useful work is

Table 2: Shiraz model predicts the optimal switching point correctly across scenarios.

System Type	δ-factor	Model Optimal Switch Point	Sim Optimal Switch Point
Exascale	5×	6	6
Exascale	25×	13	13
Exascale	100×	26	26
Exascale	1000×	81	79
Petascale	5×	12	11
Petascale	25×	26	24
Petascale	100×	51	51
Petascale	1000×	161	161

increased without degrading the performance of individual applications. To demonstrate this, we use Fig. 10 as an example. Fig. 10 illustrates that Shiraz finds an optimal point (i.e., k = 26) and Shiraz improves the overall useful work by 33 hours at this optimal point for the two given applications with a checkpointing overhead ratio (δ-factor) of $100\times$. The total runtime is 1000 hours and the MTBF is 5 hours. We use 20 hours and 5 hours MTBF to represent the failure rate of a petascale and exascale systems, respectively [13, 25]. Note that these failure rates are conservative estimates.

For deeper analysis, Fig. 10 also shows region of interest where none of the two applications is being hurt and there is an opportunity for improving the overall system throughput. Simulation results (which can take more than a few hours in some cases) confirm the same optimal point as the model predicts (which takes a few seconds). In fact, Table 2 shows that Shiraz model estimates the same optimal switching point as the simulation across different scenarios — the maximum difference in the estimations is 2, which results in a difference of less than 0.5% in the throughput improvement. The δ-factor is the ratio of checkpointing overheads of the heavy-weight and the light-weight applications (the heavy-weight application's checkpoint takes 30 mins).

In summary, Shiraz model can successfully identify regions of benefit and determine the optimal switching point much more quickly than extensive simulation based method. The next question to investigate is: how does the improvement vary across scenario and the reasons behind that?

Impact of Shiraz on system throughput and individual application performance: Next, we demonstrate that Shiraz improves overall system throughput without hurting individual applications' performance for different situations. Fig. 11 shows that Shiraz's optimal switching point improves system throughput (overall useful work done per unit time) (a) as the scale of the system changes (MTBF changes), and (b) as the checkpointing overhead ratio between the heavy-weight and light-weight application changes (δ-factor changes). From Fig. 11, we make following observations:

(1) Shiraz improves the system throughput in all cases and does not penalize individual applications. In fact, Shiraz improves the performance of individual applications in all cases. In the exascale case, both light-weight and heavy-weight applications on an average observe approximately

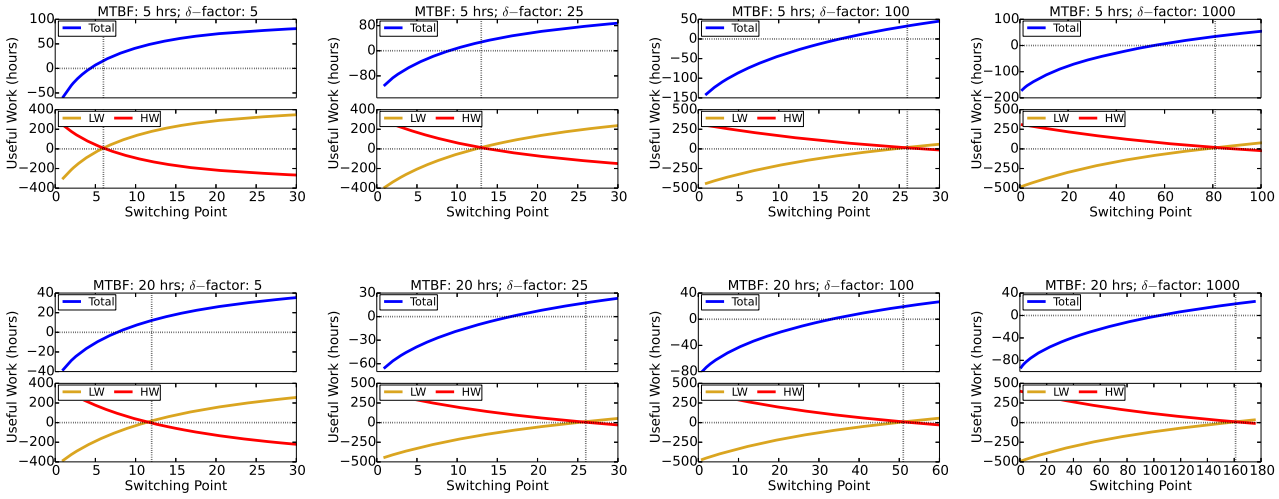

Figure 11: Shiraz provides improvements across different scenarios. For all the cases, the total runtime is 1000 hours, and the checkpoint duration (δ) of the heavyweight application is 0.5 hours.

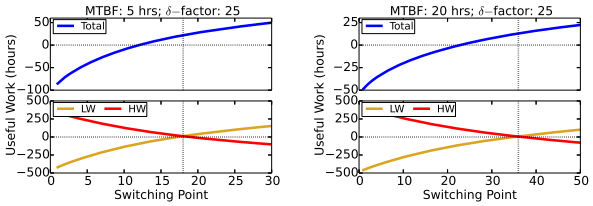

Figure 12: Shiraz improves throughput across system scale with heavyweight application checkpoint duration (δ) of 0.25 hours.

14 hours of individual performance improvement on average, leading to an overall average improvement of 28 hours. Therefore, Shiraz improves both system throughput and individual performance (latency).

(2) Shiraz's overall improvement in useful work increases as the δ-factor increases. This is expected since a high δ-factor provides more potential for Shiraz to eliminate lost work. Interestingly, the overall improvement in useful work increases as the MTBF decreases. For example, the overall improvement in useful work increases from 19 hours to 33 hours as the system changes from petascale to exascale when the δ-factor is fixed at 100. Essentially, Shiraz minimizes the lost work due to failures and this opportunity is higher with a low MTBF. This demonstrates that Shiraz will continue to be effective on future systems.

(3) Shiraz's optimal switching point also increases as the checkpointing overhead ratio between the heavy-weight and light-weight application (δ-factor) increases. For example, Fig. 11 shows that the switching point increases from 6 to 83 when δ-factor increases from 5 to 1000. This is because the light-weight application is able to perform more checkpoints in the same time period.

Shiraz's optimal switching point also increases with MTBF for a fixed δ-factor factor. For example, Fig. 11 shows that that the switching point increases from 6 to 12 when system changes from exascale to petascale. This is because between two failure points, the hazard rate drops less quickly with a higher MTBF and hence, it is beneficial to run the light-weight application for a longer time.

Finally, we note that Shiraz delivers improvement in overall useful work as the checkpointing overhead of the heavy-weight application varies. We reduce the checkpointing overhead of the heavy-weight application from 0.5 hours to 0.25 hours. Fig. 12 shows that the total throughput improvement is 21.8 hours with 5 hours MTBF system and 12.9 hours with 20 hours MTBF system.

Interestingly, our analysis reveals that the optimal switching point is not necessarily half of the MTBF value. As an example, the optimal switch point is 6 when the δ-factor is $5\times$ and the MTBF is 5 hours (Fig. 11). This implies that the switch happens at 6.6 hours, which is higher than the MTBF. Similarly, for the 20 hours MTBF case, the switching happens after 25.2 hours. As discussed in Section 3, since the light-weight application observes effectively higher MTBF, it needs to run for a longer duration in the beginning to gain improvement in the overall useful work. A naïve strategy to switch applications at half of the MTBF or slightly a higher value will lead to a significant decrease in the overall useful work. This demonstrates the need and efficacy of Shiraz.

Analysis of impact of Shiraz+ on checkpointing overhead: Next, we evaluate and analyze the effect of Shiraz+ on the overall checkpointing overhead and throughput. Recall that Shiraz+ increases the checkpointing interval of the heavy-weight application (Section 3). Thus, it is intuitive that it will reduce the checkpointing overhead. However, this may also result in a loss of throughput, since the heavy-weight application is no longer operating at its OCI.

Fig. 13 shows that when Shiraz+ is applied on top of Shiraz, it significantly reduces the overall checkpointing overhead across different scenarios. Note that Shiraz+ op-

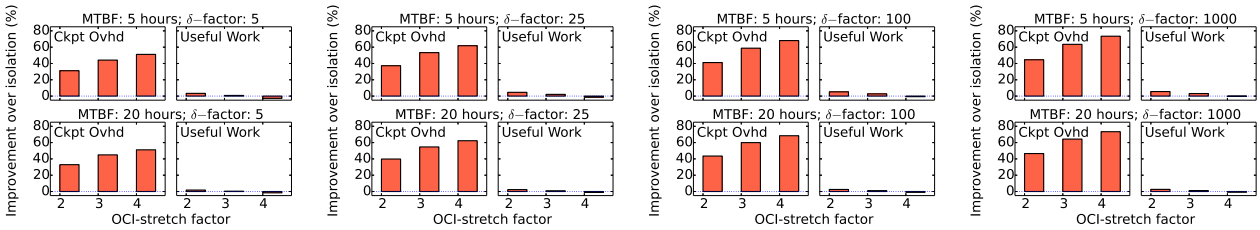

Figure 13: Impact of Shiraz+ on checkpointing overhead and useful work: checkpointing interval is increased by different factors ($2\times$ - $4\times$) under varying system scale and checkpoint overhead ratios. The checkpoint duration of the heavy weight application is set to be 30 minutes. The baseline refers to switching between applications at every failure.

erates at the optimal switching point determined by Shiraz. From Fig. 13, we make several observations. First, as the checkpointing interval is stretched from $2\times$ to $4\times$ for the heavy-weight application, the checkpointing overhead reduces drastically. This observation is true across changes in different parameters: system MTBF, application checkpointing overhead, and δ-factor. The average reduction in checkpointing overhead is approximately 40%. When the OCI-stretch factor is $4\times$, the checkpointing overhead reduces by more than 60% in many cases.

Second, interestingly, while the checkpointing overhead drops significantly, the corresponding performance degradation is minimal. In fact, using a $2\times$ OCI-stretch always keeps a part of the performance improvement obtained by Shiraz; in some cases, the throughput improvement remains up to 5.6% (with no performance degradation for any application). Even with $3\times$ and $4\times$ OCI-stretch factors, the maximum performance degradation across petascale and exascale systems is less than 1.4% and 4.8%, respectively. The underlying insight is that Shiraz schedules the heavy-weight application in a lower failure rate region (i.e., effective higher MTBF) and hence, the effective OCI also increases. We note that Shiraz+ also has the opportunity to eat off the performance improvement provided by Shiraz and hence, sees no performance degradation in the $2\times$ OCI-stretch case.

In this work, we do not explicitly determine the optimal OCI-stretch factor for different situations, since application programmers and system resource managers are likely to increase the checkpointing interval by an integer factor. Due to practical constraints, many applications do not adopt techniques that alter the checkpointing interval dynamically. In other words, Shiraz+ has chosen to value practical feasibility over theoretical optimum point — which will be an interesting avenue for future work.

Shiraz in multi-application environment and energy savings: Shiraz can determine the optimal switching point between two given applications and improve the overall system throughput. The next question is: can Shiraz scale and be effective in the presence of multiple applications? Fortunately, it turns out that Shiraz can be naturally scaled to the multiple applications scenario. It can be achieved in multiple possible ways. One easy way to achieve this is to make

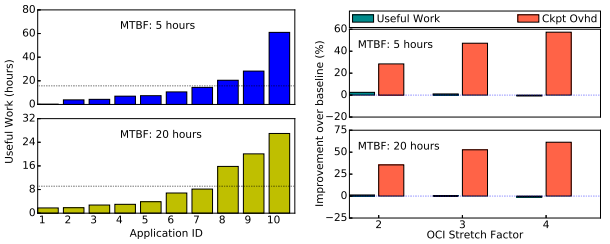

Figure 14: Shiraz provides improvement in real-world multi-application mix selected from Table 1 and simulated for year-long time period (left). The horizontal lines denotes the average improvement in useful work per application. Shiraz+ decreases checkpointing overhead significantly for the same mix of applications (right).

pairs of applications with different checkpointing overheads and run one such pair between two failures using Shiraz, and switching to a different pair after every failure. Optimal strategy to make such pairs is to combine the application with the highest checkpointing overhead with the application with the lowest checkpointing overhead, until we exhaust the available applications. The theoretical proof is not provided for brevity; the intuition behind such a strategy is simple: it maximizes the average of the ratios of checkpointing overheads. We also experimented with another strategy: making random pairs. We found that while it may not deliver the maximum possible improvement, it is relatively easier to implement.

To evaluate Shiraz in a multi-application environment, we experimented with the latter strategy using 10 applications and noted the corresponding throughput gains. The application list is composed from the real-world application characteristics from Table 1. We used the Shiraz model to obtain the optimal switch point for the different application pairs, and simulated the scenario where these applications ran for one calendar year (8,700 hours). To ensure that our results are statistically stable, we repeated the simulation over 15,000 times and report average of all runs.

Figure 14 (left) shows the overall system throughput improvement and impact on individual job performance for all the 10 applications. We make a few interesting observations. First, no application suffers a performance degradation, and the average throughput improvement is 15 hours. Second, Shiraz improves the total useful work by approx. 91 hours

and 157 hours for the petascale and the exascale systems, respectively.

Our results also demonstrate that Shiraz+ is also effective in the multi-application scenario. Figure 14 (right) shows that Shiraz+ (with $3\times$ OCI stretch factor) decreases the checkpointing overhead by up to 52%, without incurring any loss in the overall system throughput for both exascale and petascale systems. When the OCI stretch factor is increased to $4\times$, only then the system incurs degradation (less than 1%) the total useful work, while the checkpointing overhead decreases by up to 60%.

To show the results in a conservative scenario, we conduct an experiment with 40 jobs, with 5 heavy-weight applications, and the rest 35 light-weight applications. The 35 light-weight applications are selected at random from the three least heavy applications from Table 1. Shiraz improves the total useful work done 57 hours and 89 hours for the petascale and the exascale systems, respectively.

Finally, we evaluate the potential energy savings enabled by Shiraz for the exascale (5 hours MTBF) and the petascale systems (20 hours MTBF). Since Shiraz increases the useful work done per unit time at the whole system level, it effectively saves energy that would have been spent on lost work (due to failures). In order to simplify the evaluation and interpretation, we estimate the yearly energy savings. Taking a conservative electricity rate of $0.1 per kW-Hour [2], the energy and monetary savings on the exascale (5 hours MTBF and 20MW power consumption) system would translate to 1.78 MW-Hour and $178,000 per year, respectively. For the petascale (20 hours MTBF and 10MW power consumption) system, the energy and monetary savings would translate to 0.57 MW-Hour and $57,000 per year.

These savings could be invested towards faster storage systems and more computing power in the future — which would further increase the profits due to faster completion times. For the petascale system, the cost savings due to energy expenditure cuts enabled by Shiraz translate to $285,000 over 5 years (anticipated lifetime of a system). At 0.2 GB/USD for SSD-based burst buffers [3, 4] (the total cost of infrastructure pessimistically assumed to be $3\times$ of the hardware cost due to packaging, assembly, firmware and integration cost), the monetary savings could pay for 5.7% of the cost of the burst buffers (0.285M USD out of 5M USD) for the petascale system, with 1 PB of storage. For the exascale system, the cost savings enabled by Shiraz would amount to $890,000 over 5 years. We note that this analysis is on the conservative side, as it does not include the energy cost reduction due to the reduction in data movement enabled by Shiraz+.

We note that in a multi-application environment, Shiraz can produce different individual performance improvements for the same application depending upon on the pairing and application-mix since the runtime improvement provided by Shiraz depends on the δ-factor. This can possibly lead to small amount of unpredictability in the runtime, although Shiraz will improve the individual runtime in all such cases. Improving predictability in a dynamic application-mix will be a worthy goal for future works.

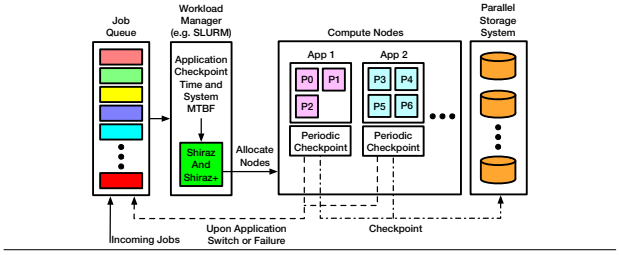

Figure 15: Prototype of Shiraz and Shiraz+.

In summary, our results show that Shiraz leads to significant energy and monetary saving for real-world applications that can act as positive feedback loop and result in compounded returns over years.

Prototype implementation and evaluation of Shiraz and Shiraz+ using system-level checkpointing: We developed a prototype of Shiraz and Shiraz+ to evaluate its effectiveness on real-world applications. We developed a scheduler plug-in that implements the core scheduling algorithm of Shiraz and Shiraz+. It maintains records of the checkpointing overhead for different applications, temporal characteristics of system failures, and takes checkpoints using a system-level checkpointing package, and schedules applications based on the Shiraz model. To demonstrate the effectiveness, we evaluated the prototype using two real-world HPC applications: Co-Design Molecular Dynamics Proxy (CoMD) [26] and Finite Element Solver (miniFE) [22]. CoMD represents a variety of scientific applications including SPaSM, and miniFE is an approximation of unstructured finite element and finite volume codes including HPCCG and pHPCCG. We used DMTCP [7], a system-level checkpointing library, to perform checkpoints, and the optimal switch point was decided based on the checkpointing overhead obtained experimentally. We note that our plugin is not tied to a particular implementation of checkpointing library and can be ported across systems and resource managers (e.g., SLURM) (schematic shown in Figure 15). The ratio of the checkpointing overhead of miniFE (heavyweight application) to that of CoMD (lightweight application) is 30x, as experimentally measured using DMTCP.

Statistically sound evaluation of such a prototype implementation is challenging since it requires dedicated time (in order of months) on a large-scale supercomputer. To address this challenge, we emulated the setting by feeding a failure trace with the same characteristics as large-scale supercomputers (discussed in Section 2) but at a higher frequency. We also scaled down the program input size to ensure that the runs completed on a local cluster within a month. We performed an effectively 200-hour long run by scaling the failure-frequency and program size, and did this run 30 times for each point, to obtaining stable results. We injected er-

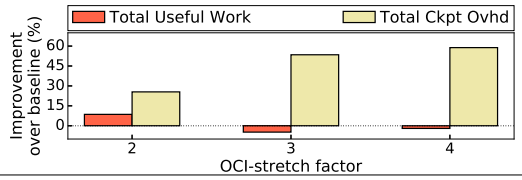

Figure 16: Impact of Shiraz+ on CoMD and miniFE application performance and checkpointing overhead.

rors in the local cluster that crash the application and used checkpoints to recover from errors without any human intervention during the experiments. At the end of run, we collected runtime statistics (useful work, checkpoint overhead, and lost work) to compare Shiraz with the baseline.

We found that Shiraz results in 10.2% more useful work system-wide using CoMD and miniFE application, compared to the baseline case, where applications are switched at every failure. Since these experiments take prohibitively long, we did not explore the optimal switching point using experiments. Instead, we used the Shiraz model to obtain optimal point offline and results show improvements.

We also evaluated Shiraz+ using this prototype. Fig. 16 shows that Shiraz+ reduces the checkpointing overhead significantly with minimal or no performance degradation. For example, the overall checkpointing overhead is reduced by approximately 35.8% when using a $2\times$ OCI-stretch factor, while still maintaining the overall improvement in useful work at approximately 7%. When Shiraz+ applies $3\times$ and $4\times$ OCI-stretch, the overall checkpointing overhead is reduced by 69.6% and 77.6%, respectively, while the performance degradation is under 3%. Overall, the evaluation shows that when operating at the optimal switching point obtained by Shiraz model, Shiraz+ is effective in reducing the data movement caused by checkpointing and still retains some of the performance benefits provided by Shiraz.

6. Related Work

A large number of previous HPC works have focused on performing failure analysis and developing checkpointing methods for fault tolerance. HPC system and application logs are extensively studied to extract information about the characteristics of failures [11, 17, 21, 31, 35, 36]. More recent studies have used neural networks, statistical learning and big-data analytics to model failure characteristics and provide potential root causes [33, 34].

In order to improve the checkpointing overhead, past studies have provided different derivations for application-specific OCI [14, 24, 30, 38, 40, 42]. Our work relies on and is complementary to these studies as it schedules jobs using the proposed OCI values to improve system throughput. Some recent studies have also proposed to use multi-level checkpointing: a strategy that checkpoints at different levels (memory, SSD, PFS) to tolerate different types of failures, based on the temporal and spacial distribution of the failures [10, 15, 16, 27]. Another method called incremental checkpointing proposes to only store the state of the data which has been modified since the last checkpoint, thus potentially reducing I/O overhead [20, 29]. On the other hand, several studies have seeked to use faster storage options such as SSD-based burst buffers to reduce the overhead of writing the checkpoint files [8, 23, 37]. All of the above optimizations, which target different methods of reducing checkpointing overhead, can be used in conjunction with Shiraz, which targets efficient scheduling as a way to improve system throughput and decrease the checkpointing overhead.

Bouguera et al. [12] propose an application-oriented resilience scheme that combines predictive, proactive and preventive checkpointing by tracking and drawing correlation graphs between faults and failures. Several other works have developed reliability-aware task scheduling strategies that optimize the degree of job replication to reduce communication interference and/or energy consumption [28, 39, 41, 43]. However, replication for increased reliability also increases consumption of valuable compute and energy resources. Tiwari et al. [40] introduced Lazy checkpointing that uses temporal locality of failures to dynamically adjust the checkpointing frequency of an application to reduce I/O overhead.

Lazy checkpointing results in non-equidistant checkpoints because the rate of increase of interval depends on the hazard rate. Unfortunately, non-equidistant checkpoints are unattractive for many applications because some domain scientists may use checkpoints to monitor the progress of the simulation; non-equidistant checkpoints make it difficult to monitor such progress. On the contrary, both Shiraz and Shiraz+ provide equidistant checkpoints. Shiraz+ shows that it is possible to increase the OCI by a factor, reduce I/O overhead and still achieve significant performance improvement unlike Lazy checkpointing. Therefore, techniques proposed in this work are more practical strategies, which improve performance, I/O overhead and work even when scheduling multiple applications unlike Lazy checkpointing [40].

In the above implementations, applications tune their internal parameters to improve resilience, reduce I/O overhead, and increase their useful work. However, no work explores how these individual optimizations can be combined together, without disrupting an application's local optimization methods such as OCI tuning, to improve the system's throughput. This work proposes a scheduling technique that exploits failure characteristics and works with multiple applications, while ensuring individual performance fairness.

7. Conclusion

This paper introduced, Shiraz, a novel scheme, to improve the overall system throughput by intelligently scheduling applications with different checkpointing overheads. This paper also introduced, Shiraz+, a novel scheme to reduce checkpointing overhead. Evaluation results show that Shiraz improves system throughput under a wide variety of circumstances. Shiraz can save up to $285,000 over the lifetime of a petascale supercomputer and Shiraz+ reduces the data movement by up to 52% for a variety of applications.

Acknowledgment We thank anonymous reviewers for their constructive feedback. This work was partially supported by Northeastern University, NSF Grant ACI-1440788, Grant 2014-345 from "Chaire d'attractivité" de lIDEX Université Fédérale Toulouse Midi-Pyrénées and resources from the Mass Open Cloud (MOC).

References

[1] CFDR Data. https://tinyurl.com/yd6ornwa. [Online; accessed 28-Nov-2017].

[2] EIA - Electricity Data. https://tinyurl.com/ya6o3eas. [Online; accessed 04-Dec-2017].

[3] Intel DC P3608 SSDPECME040T401. https://tinyurl.com/ybng8ll3. [Online; accessed 04-Dec-2017].

[4] Samsung PM1725a Series 1.6TB TLC. https://tinyurl.com/yd8rcy55. [Online; accessed 04-Dec-2017].

[5] Large Scale Computing and Storage Requirements for Biological and Environmental Science: Target 2017. Technical Report LBNL-6256E, LBNL, 2012.

[6] Large Scale Production Computing and Storage Requirements for High Energy Physics: Target 2017. Technical report, LBNL, 2012.

[7] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *IPDPS 2009*, pages 1–12. IEEE, 2009.

[8] L. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large-scale Systems. In *CCGrid 2010*, pages 63–72. IEEE Computer Society, 2010.

[9] Leonardo Bautista-Gomez et al. Reducing Waste in Extreme Scale Systems Through Introspective Analysis. In *IPDPS 2016*, pages 212–221. IEEE, 2016.

[10] Anne Benoit, Aurélien Cavelan, Valentin Le Fèvre, Yves Robert, and Hongyang Sun. Towards Optimal Multi-level Checkpointing. *IEEE Trans. Comput*, 66(7):1212–1226, 2017.

[11] R. Birke, I. Giurgiu, L. Y Chen, D. Wiesmann, and T. Engbersen. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *DSN 2014*, pages 1–12. IEEE, 2014.

[12] Mohamed Slim Bouguerra et al. Improving the Computing Efficiency of HPC Systems Using a Combination of Proactive and Preventive Checkpointing. In *IPDPS 2013*, pages 501–512. IEEE, 2013.

[13] Franck Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *IJHPCA*, 23(3):212–226, 2009.

[14] John T Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[15] S. Di, M. S Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. In *IPDPS 2014*, pages 1181–1190. IEEE, 2014.

[16] Sheng Di, Yves Robert, Frédéric Vivien, and Franck Cappello. Towards an Optimal Online Checkpoint Solution Under a Two-level HPC Checkpoint Model. *TPDS 2017*, 28(1):244–259, 2017.

[17] N. El-Sayed and B. Schroeder. Reading Between the Lines of Failure Logs: Understanding How HPC Systems Fail. In *DSN 2013*, pages 1–12. IEEE, 2013.

[18] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for Peta-scale Systems: A Look into the Future of Practical Rollback-Recovery. *TDSC 2004*, 1(2):97–108, 2004.

[19] Kurt Ferreira et al. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *SC 2011*, page 44. ACM, 2011.

[20] Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, and Ron Brightwell. Accelerating incremental checkpointing for extreme-scale computing. *Future Generation Computer Systems*, 30:66–77, 2014.

[21] Ana Gainaru, Franck Cappello, and William Kramer. Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems. In *IPDPS 2012*, pages 1168–1179. IEEE, 2012.

[22] M Heroux. MiniFE: Finite Element Solver. https://tinyurl.com/y7hslf65. [Online; accessed 15-Apr-2018].

[23] Ning Liu et al. On the Role of Burst Buffers in Leadership-class Storage Systems. In *MSST 2012*, pages 1–11. IEEE, 2012.

[24] Yudan Liu et al. An Optimal Checkpoint/Restart Model for a Large-scale High Performance Computing System. In *IPDPS 2008*, pages 1–9. IEEE, 2008.

[25] Robert Lucas. Top Ten Exascale Research Challenges. In *DOE ASCAC Subcommittee Report*, 2014.

[26] Jamaludin Mohd-Yusof, S Swaminarayan, and TC Germann. Co-Design for Molecular Dynamics: An Exascale Proxy Application, 2013.

[27] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC 2010*, pages 1–11. IEEE, 2010.

[28] A. Namazi, M. Abdollahi, S. Safari, S. Mohammadi, and M. Daneshtalab. Reliability-Aware Task Scheduling using Clustered Replication for Multi-core Real-Time Systems. In *NoCArc 2016*, pages 45–50. ACM, 2016.

[29] Bogdan Nicolae and Franck Cappello. AI-Ckpt: Leveraging Memory-access Patterns for Adaptive Asynchronous Incremental Checkpointing. In *HPDC 2013*, pages 155–166. ACM, 2013.

[30] Ron A Oldfield et al. Modeling the Impact of Checkpoints on Next-generation Systems. In *MSST 2007*, pages 30–46. IEEE, 2007.

[31] Narasimha Raju, Y Liu Gottumukkala, Chokchai B Leangsuksun, Raja Nassar, and Stephen Scott. Reliability Analysis in HPC Clusters. In *HAPCW*, pages 673–684, 2006.

[32] Marvin Rausand and Arnljot Hoyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-IEEE, 3 edition, November 2003.

[33] Andrea Rosà, Lydia Y Chen, and Walter Binder. Predicting and Mitigating Jobs Failures in Big Data Clusters. In *CCGrid 2015*, pages 221–230. IEEE, 2015.

[34] Andrea Rosà, Lydia Y Chen, and Walter Binder. Failure Analysis and Prediction For Big-data Systems. *TSC 2016*, 2016.

[35] Ramendra K Sahoo, Mark S Squillante, A Sivasubramaniam, and Yanyong Zhang. Failure Data Analysis of a Large-scale Heterogeneous Server Environment. In *DSN 2004*, pages 772–781. IEEE, 2004.

[36] B Schroeder and Garth Gibson. A Large-scale Study of Failures in High-performance Computing Systems. *TDSC 2010*, 7(4):337–350, 2010.

[37] Jim Stevens, Paul Tschirhart, and Bruce Jacob. Fast Full System Memory Checkpointing with SSD-aware Memory Controller. In *MemSys 2016*, pages 96–98. ACM, 2016.

[38] Omer Subasi, Gokcen Kestor, and Sriram Krishnamoorthy. Toward a General Theory of Optimal Checkpoint Placement. In *CLUSTER 2017*, pages 464–474. IEEE, 2017.

[39] Xiaoyong Tang, Kenli Li, Renfa Li, and Bharadwaj Veeravalli. Reliability-aware Scheduling Strategy for Heterogeneous Distributed Computing Systems. *JPDC*, 70(9):941–952, 2010.

[40] D. Tiwari, S. Gupta, and S S Vazhkudai. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-scale Systems. In *DSN 2014*, pages 25–36. IEEE, 2014.

[41] S. Wang, K. Li, J. Mei, G. Xiao, and K. Li. A Reliability-aware Task Scheduling Algorithm Based on Replication on Heterogeneous Computing Systems. *JGC*, 15(1):23–39, 2017.

[42] John W Young. A First-order Approximation to the Optimum Checkpoint Interval. *CACM*, 17(9):530–531, 1974.

[43] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, and K. Li. Maximizing Reliability with Energy Conservation for Parallel Task Scheduling in a Heterogeneous Cluster. *Information Sciences*, 319:113–131, 2015.