[Technical Report CSRG-622 (University of Toronto)]

Checkpoint/Restart in Practice: When 'Simple is Better'

Nosayba El-Sayed Bianca Schroeder
Department of Computer Science, University of Toronto
{nosayba, bianca}@cs.toronto.edu

Abstract

Efficient use of high-performance computing (HPC) installations critically relies on effective methods for fault tolerance. The most commonly used method is checkpoint/restart, where an application writes periodic checkpoints of its state to stable storage that it can restart from in the case of a failure. Despite the prevalence of checkpoint/restart, it is still not very well understood in practice how to set its key parameter, the checkpoint interval. Despite a large body of theoretical work, practitioners still rely on crude rules-of-thumb such as "checkpoint once every hour".

Our goal in this work is to identify methods for optimizing the checkpointing process that are easy to use in practice and at the same time achieve high quality solutions. In particular, this work makes the following contributions: We evaluate an array of methods for optimizing the checkpoint interval, some previously known as well as new ones that we propose, using real-world failure logs. We show that a very simple closed-form solution can easily be adapted for use in practice and achieves near-optimal performance. We also find that more complex solutions only negligibly improve performance based on real world traces. We show that simple back-of-the envelope formulas can be used to accurately estimate the wasted work in HPC systems, and make projections of future HPC systems and requirements for their efficient use.

Keywords

High-performance computing; Fault tolerance; Checkpoint-restart; Performance;

I. Introduction

The most widely used method for fault tolerance in high-performance computing (HPC) applications is coordinated checkpointing, where a parallel application periodically stops execution to checkpoint its current state. In the case of a failure, the application recovers by restarting from the most recent checkpoint. It is important to note that under coordinated checkpointing all compute nodes involved in a parallel application stop simultaneously to write their individual checkpoints, and that a failure of any one node involved in a parallel application requires all nodes to restart from their most recent checkpoints.

Overhead due to faults and fault tolerance in systems using checkpointing comes from two different sources: the time that is spent writing periodic checkpoints and the time that is spent to recover in the case of a failure (i.e. the time to revert back to the state of the most recent checkpoint and the time to redo all the lost work that has been done since the most recent checkpoint). Hence the amount of time that is wasted, i.e. any time that is not spent on doing actual computation, depends on the system's failure rate, the amount of time it takes to write a checkpoint, and the frequency of checkpoints.

Given the importance of the problem it is not surprising that much research has been dedicated to optimizing the checkpointing process. Besides approaches to reduce the cost of a checkpoint, for example through data compression [10] or filesystem optimizations [4], a large body of work focuses on optimizing the choice of the checkpoint interval, i.e. the time between two consecutive checkpoints [5]–[7], [11], [14], [18]. While frequent checkpointing reduces the amount of lost computation in the case of a failure, it leads to a large amount of time spent checkpointing rather than performing computation. Conversely, the fewer checkpoints a system schedules, the higher the recovery overhead when failures happen.

Our motivation for this work comes from discussions with practitioners at a number of large HPC installations, who lament that in practice crude and ad-hoc rules of thumb are used to decide on the frequency of checkpoints, such as "checkpoint once every hour". Given the large body of literature on the topic and the immediate impact the choice of the checkpointing interval has on system efficiency this situation is unsatisfactory, to say the least. Further discussions identified as a reason the high complexity of existing solutions. They often assume detailed knowledge about the underlying failure process (such as the statistical distribution function of the time between

failures and its parameters), which is not readily available and which can furthermore change over the lifetime of a system. Moreover, existing solutions are perceived as too complex to implement, as they often do not provide straightforward closed-form solutions.

The goal of our work is to revisit the problem of optimizing coordinated checkpointing in tightly-coupled, HPC applications with a *practitioner's view* in mind. We look at solutions of varying degrees of complexity (including previously proposed solutions and our own) and provide a thorough evaluation based on real world failure traces (rather than synthetically generated data as most previous work) to answer the question of how much complexity is really needed. We explore the issue of how sensitive methods are to errors in their parameter estimates, and show that the key parameter can be estimated sufficiently accurately online using simple methods. We provide an easy to use back-of-the envelope formula to accurately estimate wasted work, which can be used by practitioners to configure their applications or in the planning of future systems. We also make projections of system efficiency as systems scale out and derive requirements for their efficient use.

II. Starting simple: Young's formula

The first and simplest approach for computing the checkpoint interval is the closed-form solution proposed by Young [18] in 1974. Young's formula determines the checkpoint interval Δ_{Young} based on only two quantities, the system's mean time to failure (MTTF) and the checkpoint cost C:

$$\Delta_{Young} = \sqrt{2 \cdot C \cdot MTTF} \tag{1}$$

Young's formula relies on the following set of unrealistic assumptions, which make its value questionable and has spurred a sizeable body of follow-up work that provides more complex, but presumably more accurate results:

1) Young assumed that failures follow a Poisson process. Prior work [8], [9] reports dependencies between failures and non-exponential inter-arrival time distributions. Work in [6], [13] extends the analysis to Weibull distributions, known to be a better model of empirical distributions.
2) Failures do not happen during checkpoints. In practice, there are applications with a checkpoint cost that is high enough that the probability of experiencing a failure during a checkpoint is significant. Several recent papers [6], [7], [11], [13] take into account the probability of failure during checkpoints.
3) Failures happen on average half-way between two checkpoints. Work by Liu [12] removes this assumption by approximating the excess lifetime distribution of the time between failures.
4) The system's MTTF and checkpoint cost C are known accurately and in advance, and do not change over time. In reality, failure rates of a system change over time (e.g. with age) and are not known a-priori by administrators. We are not aware of any work that addresses this problem.

In the remainder of this section our goal is to evaluate the impact of the unrealistic assumptions 1.-3. in practice using trace-driven simulations based on a large array of real-world failure logs, to evaluate Young's accuracy for realistic scenarios. The next section will deal with practical implementation issues brought up in 4. and whether Young's theoretical performance can be achieved in practice.

A. How accurate is Young's formula for real traces

In this subsection we use trace-driven simulations to evaluate the quality of the solution provided by Young. The traces we use are available online [2] and cover nearly a decade worth of failure logs from 20 different HPC systems at Los Alamos National Lab (LANL). The data contains records of all node outages that occurred during the measurement period (a total of 23,600 node failures).

Our simulations assume a "hero run" of an application that uses all available nodes. For each system, we use the entire log and and simulate periodic checkpoints at fixed intervals that we compute using Young's formula (recall Equation (1)) and record the fraction of time that is lost due to writing checkpoints or redoing lost computation. (Note that we do not include the time needed to restart the application to the state of the most recent checkpoint, as this time does not depend on the checkpointing interval and hence will be the same for any checkpointing policy). We run simulations varying the checkpointing cost C from as low as 20 seconds to as high as 60 minutes, and we obtain the MTTF for a system from the corresponding trace. (Note that this is not feasible in practice as one can not know a system's MTTF beforehand. Section III will deal with how one

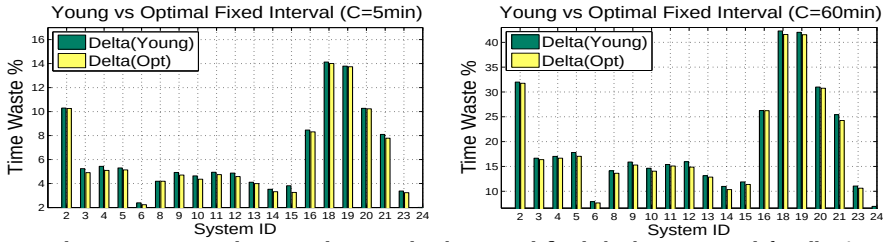

Fig. 1. Wasted time under Young compared to wasted time under the optimal fixed checkpoint interval for all LANL systems.

would obtain MTTF in practice). To measure how good the provided solution is we compare it to the wasted work the system would have experienced under the *optimal* fixed checkpointing interval, i.e. Δ_{Opt} that leads for a given trace to the smallest fraction of wasted work. We obtain Δ_{Opt} by searching through the entire range of Δ values and using our simulator to identify the one that performs best.

Figure 1 compares the fraction of time that is wasted under Δ_{Young} to that under Δ_{Opt}, for all LANL systems, and under different C costs. Results shown assume that C is constant, but we also experimented with C following an exponential or uniform distribution and reached the same results.

We observe that the fraction of wasted time under Young's formula is very close to the optimal, in most cases within 2%. Interestingly, even for input scenarios that deviate significantly from Young's assumptions checkpointing using Δ_{Young} is near optimal. For example, Table II in the appendix provides for each system the shape parameter of the best Weibull fit to the data and shows that systems that deviate the most from an exponential distribution (recall that an exponential would have a shape parameter of 1) do not exhibit worse performance than others. Also, a higher checkpointing cost (which will increase the chance of failures during checkpoints) does not reduce performance of Δ_{Young} compared to Δ_{Opt}.

Summary: Despite a number of unrealistic assumptions that the derivation of Young's formula relies on, it achieves performance nearly identical to that under the optimal checkpoint interval (identified offline through exhaustive search). This is the case even for input scenarios that significantly deviate from Young's unrealistic assumptions.

III. Making Young's formula work in practice

Employing Young's formula in practice requires two types of information: the cost C of a checkpoint and the MTTF. Our evaluation in Section II-A assumed perfect knowledge of these two quantities, which in practice need to be estimated before the run of an application. Below we first study how sensitive the performance of Young is to errors in the estimation of those parameters, and then show how sufficiently accurate estimates can be obtained in practice.

A. Sensitivity to accuracy in parameter estimation

To understand the sensitivity of Young's formula to the accuracy of the parameter estimation we apply the same trace-based simulation we relied on in Section II-A, but rather than determining the checkpoint interval based on the actual values for C and MTTF we assume that they were estimated with varying degrees of error. We range the degree of error between $(1/5)X$ (i.e. the C or MTTF was underestimated by a factor of 5) to a degree of error of $5X$, i.e. the C or MTTF was overestimated by a factor of 5.

Figure 2 shows the results when the MTTF is estimated with varying degrees of error while C is estimated accurately in two representative LANL systems (systems 2 and 20). (We also performed experiments with an error in C instead of MTTF, with similar results.) The X-axis shows the degree of error and the Y-axis shows the resulting wasted time normalized by the wasted time that would have resulted under error-free parameter estimation. We observe that there is a large range of MTTF values that achieve almost identical, near optimal performance. For example, to stay within a range of 5-10% of the wasted work achieved under accurate parameter estimates one can tolerate errors of a factor of 2 (either over- or underestimation) in estimating the MTTF.

Summary: We conclude that checkpointing based on Young's formula is quite robust against reasonable errors in the estimation of C or MTTF, which motivates us to explore ways to implement Young in practice.

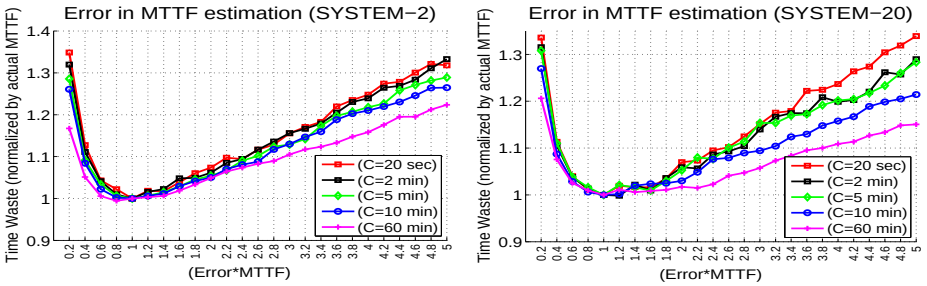

Fig. 2. Wasted time assuming an error in the MTTF estimation.

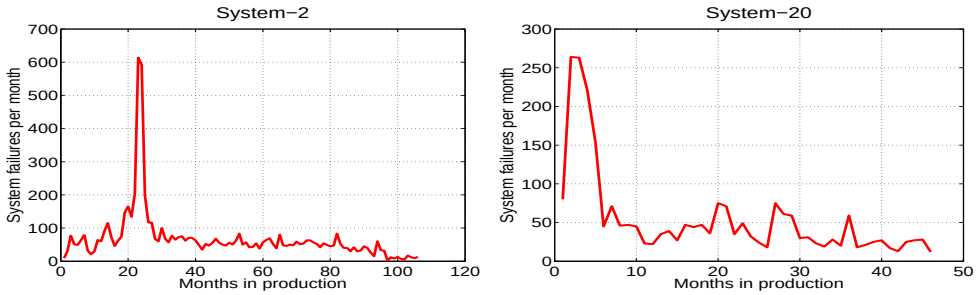

Fig. 3. Failure rates as a function of system age for two LANL systems.

B. Parameter estimation in practice

The good results for Young in Section III-A were obtained by "cheating": we assumed a-priori knowledge of the checkpoint cost and the MTTF (as we just used the entire log for a system to determine the MTTF before running our simulations) – an approach that is used by all previous work as well, but not realistic in practice. In this section we turn to methods for estimating the parameters in practice.

The value of the checkpoint cost C depends on the particular application and the amount of data it needs to checkpoint to be able to restore a previous state of execution. For the user of an HPC system, estimating this quantity is easier than estimating the MTTF, as it can be determined based on measurements taken during some test runs. We therefore focus our attention on estimating the MTTF which is more challenging as perfect estimation would require knowledge about the system's future failure behavior.

Estimating the MTTF is further complicated by the fact that in reality a system's MTTF is not stable over time. We illustrate this by plotting the failure rates over a system's lifetime for two representative LANL systems in Figure 3. Failure rates are often higher early in a system's life (see Figure 3 (right)), as different hardware and software issues get exposed during the execution of real world workloads. Random spikes in failure rates can also happen later in a system's lifetime (see Figure 3 (left)), for example due to the upgrade or installation of new software. Based on these observations, we experiment with three implementations of Young that dynamically maintain an estimate of the MTTF, based on the system's recent failure history:

Young(SMA): This approach uses a Simple Moving Average, i.e. it simply calculates th MTTF as the average value of the failure inter-arrival times within an observation window consisting of the last w days and uses that average as an estimate of the expected time to the next failure. w is a parameter that needs to be determined.

Young(WMA): This method works like Young(SMA), but uses a Weighted Moving Average, i.e. it assigns a weight for each value in the observation window, with more recent observations having greater weights. WMA, therefore, considers recent failure inter-arrival times more predictive of the time to next failure, than older ones.

Young(EMA): Young(EMA) uses an Exponential Moving Average to estimate the MTTF, i.e. the weights of older observations decrease exponentially, giving past values a diminishing contribution to the calculated average. Unlike SMA and WMA that only consider values within the observation window, EMA is a cumulative calculation that includes all the historical observations, taking into account the entire history of failure inter-arrivals.

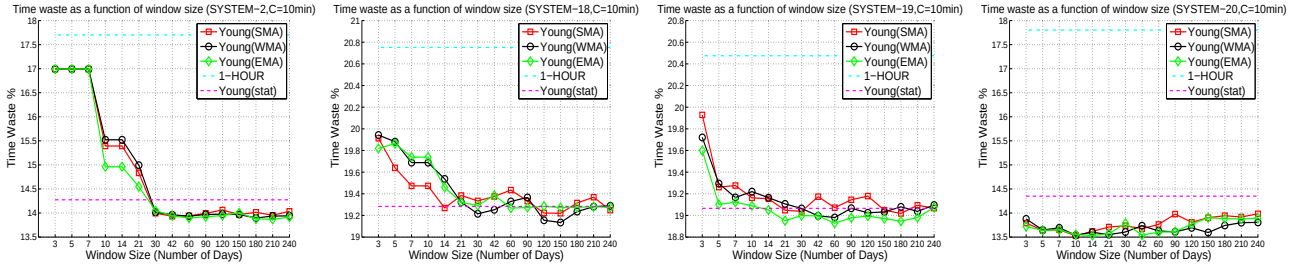

Fig. 4. Checkpointing using Young's formula and different techniques to estimate the MTTF.

The only requirement of the above algorithms is that the system maintains a log of the the most recent times of failures. When an application first starts running, the initial checkpoint interval is computed based on Young's formula applied to the MTTF estimate obtained using one of the three moving averages. Every time a failure occurs and the applications needs to roll-back and restart, the MTTF estimate is updated (taking the most recent failure into account), and the checkpoint interval is recomputed based on the new MTTF estimate.

C. Performance in practice

We use trace driven simulations based on the LANL data to evaluate the performance of Young(SMA), Young(WMA), and Young(EMA). Figure 4 shows the results for four LANL systems, systems 2, 18, 19, 20 (the four systems that the largest amount of data is available). Table II in the appendix shows the complete results for all LANL systems.

Each graph plots for one of the systems the wasted time for each of the three algorithms as a function of the window size w that was used. In the case of the EMA method, which does not rely on a specific time window, the parameter w on the X-axis is used to vary the smoothing coefficient α. In particular, α is computed as $\alpha = 2/(w+1)$.

To evaluate the quality of the produced solutions each graph shows for comparison the wasted time that would have resulted under Young's static checkpoint interval, Young(stat); i.e. the theoretical algorithm that knows the system's MTTF a-priori and uses the checkpoint interval over the entire trace. We also compare our results to the case of checkpointing every one hour, a scenario commonly applied in practice. We observe the following:

- The performance of all moving averages methods and their ability to estimate the MTTF is nearly the same.
- For all systems one can achieve performance comparable to that of Young(stat), for a large range of w values.
- For some systems (system 2 and system 20) running Young on the MTTF estimates actually performs slightly better than Young(stat). The reason is that for these two systems failure rates are more variable over the course of their life (recall the graphs in Figure 3). Using a dynamic estimate of the MTTF, rather than the average across the entire trace, can actually slightly improve performance.
- The performance is not overly sensitive to the choice of w making tuning easy. Values for w of 30 days or more performed well for all systems. Except for system 2, only a few days worth of data provide nearly optimal performance, which is good news when introducing any of the proposed methods on a system with no prior recording of failures.

Summary: Using a combination of Young's formula and simple moving averages of past failure one can achieve performance comparable to the (hypothetical) case where the optimum checkpoint interval is known a-priori. This method, which is easily implementable in practice, obviates the need for complex (and theoretically more accurate) methods.

IV. More advanced techniques

The previous section showed that our adaptation of Young, which estimates the MTTF online based on a sliding window history (rather than assuming precise a-priori MTTF information), can not only match, but sometimes even slightly exceed the performance compared to using the optimum (static) checkpoint interval for a given trace. This observation motivates us to investigate more advanced methods that adapt the checkpoint interval on the fly based on a system's failure characteristics. We investigate methods that exploit three different characteristics of real world failures: decreasing hazard rates, autocorrelations and dependencies between different types of failures.

System	Failure Type	MTTF (minutes)	Frequency
2	Environment	1231.992	2.32%
	Hardware	878.7485	54.15%
	Network	453.4721	4.04%
	Software	724.9778	26.28%
	Memory	884.1197	15.20%
	CPU	749.408	0.50%
18	Environment	1392.401	0.50%
	Hardware	462.2575	74.48%
	Network	516.8922	0.43%
	Software	380.6104	17.19%
	Memory	403.1434	14.18%
	CPU	477.6332	50.90%
19	Environment	1410.08	0.27%
	Hardware	495.0095	77.68%
	Network	791.712	0.40%
	Software	352.6268	18.15%
	Memory	428.2856	6.20%
	CPU	499.8085	60.35%
20	Environment	1373.748	0.48
	Hardware	797.7577	69.45%
	Network	911.7771	0.56%
	Software	801.5455	26.07%
	Memory	1013.431	15.57%
	CPU	730.4533	46.60%

TABLE I
MTTF FOR DIFFERENT FAILURE TYPES IN THE FOUR LARGEST LANL SYSTEMS.

A. Failure type specific MTTF estimation

We observe in our previous work [8] that the occurrence of certain types of failures greatly increases the probability of later failures. To provide one example, we find in [8] that for some systems the probability of a failure during the week following a network failure is 3.7X times larger than during an average week. This leads us to the idea of adapting the checkpoint frequency based on the type of the most recent failure.

More precisely, we propose to use the root cause information that is provided with each failure recorded in the LANL dataset, to more accurately estimate the time until the next failure. Each failure is attributed to one of five root cause categories, depending on whether it was due to problems with software, hardware, network, the environment of the system, or human error. Instead of just maintaining one moving average of the MTTF, we keep one estimate for each type of failure. For example, the moving average for software failures considers only the software failures in the observation window and averages the time between a software failure and the next follow-up failure (of any type). Any time a failure happens, the moving average that corresponds to that failure type is updated, the new MTTF estimate is computed, and the time of the next checkpoint is calculated using Young's formula and that new MTTF estimate. We refer to this approach as ETTF(Type).

The column labeled "%wasted ETTF(Type)" in Table II reports the results achieved under ETTF(Type). We find that failure-type specific information does only slightly improve the fraction of wasted time compared to Young(stat). To explain the results we take a closer look at the failure data. Table I shows the expected time until the next failure, depending on what type the most recent failure was, for the four largest LANL systems. We see that the ETTF does differ across the different categories of failures, so taking failure-type information into account provides additional information on the ETTF. However, we find that failures are dominated by two large categories, hardware and software failures, and those two types of failures tend to have a very similar effect on the ETTF. The failure types whose follow-up ETTF differs the most from the overall ETTF, such as environmental failures, are relatively rare, and hence the benefits reaped from taking them into account are limited.

We considered using more detailed information on the root cause of a failure than the five high-level categories described above, as the LANL data provides sub-categories for each high-level root-cause category. For example, hardware failures are grouped into failures due to CPU, memory, node board issues, fan problems, etc. Unfortunately, we find that also the ETTF for the sub-categories of failures within each of the high-level categories are quite similar, so little can be gained.

Summary: For the systems at LANL and the data we have, the benefits of taking the type of the most recent failure into account when estimating the ETTF are limited, since the two most common types of failures recorded at LANL have a similar impact on the expected time until the next failure. For other systems with larger differences between the main categories of failures, failure-type specific methods might be more effective.

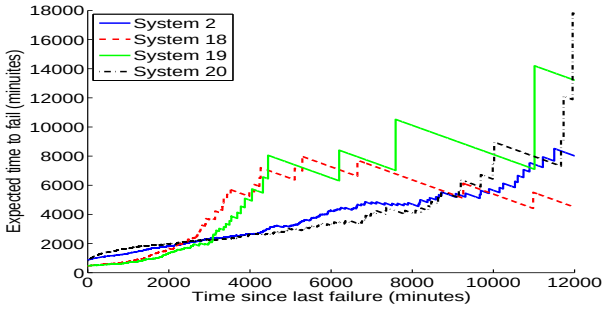

Fig. 5. Expected remaining time to fail given time since last failure.

B. Decreasing hazard rates

A number of previous studies [5], [6], [12] suggest placing checkpoints dynamically based on the statistical distribution of the time between failures, rather than using a fixed checkpoint interval. The motivation is that, unlike the exponential distribution, empirical distributions often exhibit decreasing hazard rates (as indicated by a shape parameter less than 1 in a Weibull distribution; see column "Weibull shape" in Table I for the shape parameter of the Weibull fit for LANL data). A decreasing hazard rate function predicts that if a long time has elapsed since the last failure then the expected remaining time until the next failure is long. The intuition is that in a system with decreasing hazard rates one can reduce the checkpoint frequency if a long time has passed without seeing any failures (as a long time without failures implies a longer expected time until the next failure).

Note that the implementation of such a method in practice is more involved than that of the methods we have previously considered. In all previous methods, the checkpoint interval was fixed at the beginning of an application run, and updated only in the case of a failure, when the application was restarted. Methods that take decreasing hazard rates into account require that an application be able to adapt its checkpoint interval while running, and not just at start/restart time. Furthermore, methods based on hazard rates require knowledge of the distribution of time between failures, rather than just the mean of the distribution. For example, previous work typically assumed a Weibull model of the underlying failure distribution is available.

We are mainly interested in exploring the general potential of hazard-rate based methods, without having to worry about issues due to the potential loss of accuracy when fitting a theoretical distribution to the empirical data, as required by previous approaches. We therefore rely directly on the failure trace data to determine for each system how exactly the expected time to the next failure depends on the elapsed time since the last failure. Figure 5 plots the expected remaining time to the next failure as a function of the time since the last failure for systems 2, 18, 19, and 20; i.e. datapoint (x, y) means that after running without failures for x minutes the expected remaining time until the next failure (on top of the x minutes already completed) is y minutes. Not surprisingly, given the parameters of the fitted distribution in Table II, we observe an increasing trend in all curves.

We use the curves in Figure 5 to implement an adaptive checkpointing method, called *Adaptive*. Any time a new checkpoint interval calculation is made (i.e. after a failure or a checkpoint), the curves in Figure 5 are used to determine the expected time until the next failure as a function of how much time has elapsed since the last failure. This estimate is then plugged into Young's formula to determine the length of the next checkpoint interval.

The column labeled "%wasted (Adaptive)" in Table II shows the results obtained from running *Adaptive* on the LANL traces, compared to our old Young(stat) from Section II. Overall, we observe that improvements are marginal. We identify as the main reason that while failure rates do change as a function of the elapsed time since the last failure, this change happens too slowly to have a large impact. For example, in the case of systems 2 and 20 it takes 1000 minutes of failure free execution before the expected time until the next failure doubles from initially around 800 minutes to 1600 minutes (leading to an increase of 1.4X in the checkpoint interval). However, only 23% of all failure intervals are longer than 1000 minutes, so the number of opportunities where this knowledge can be brought to bear is limited.

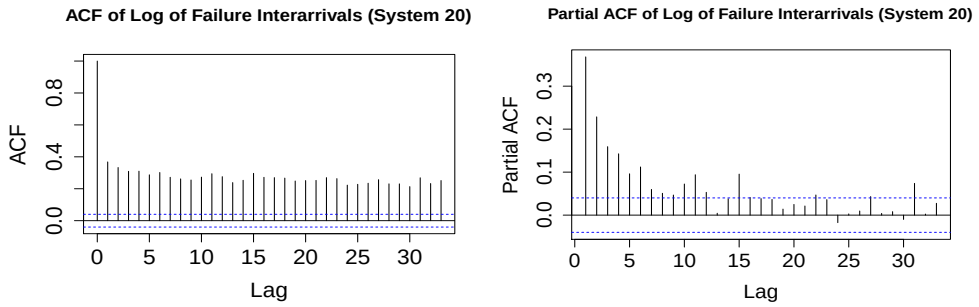

Fig. 6. The Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) of failure inter-arrival times for LANL system 20.

We experimented with synthetically generated failure traces using smaller Weibull shape parameters than the shape parameters observed for the LANL data. We do find that improvements increase for smaller shape parameters. For example, for a shape parameter of 0.3 we observe an average improvement of 10% for *Adaptive* over Young.

It is worth noting that the best method previously reported in the literature for dynamically placing checkpoints (by Bougeret et al. in [5]) reports improvements over other methods, in particular for small shape parameters. However, that method is in the best cases only able to match that of Young(stat), something that we find our much simpler moving averages methods able to accomplish.

Summary: For the LANL systems that our data comes from, improvements from placing checkpoints dynamically based on the hazard rate function are negligible and hence hardly justify the associated overhead and complexity. We observe improvements only in synthetic experiments with shape parameters much smaller than those observed in practice.

C. Autocorrelation

In this section, we propose to take information about the burstiness of the failure process into account when making checkpointing decisions. To quantify the burstiness and degree of correlation between failures in LANL's systems we plot in Figure 6 both the auto-correlation and partial auto-correlation functions of failure inter-arrivals in one of LANL's systems (system 20). We observe strong positive auto-correlations between failure inter-arrivals in this system. When repeating this analysis for the rest of LANL's systems we found similar trends.

These observations motivate us to use autoregression (AR) to model the time between failures. We fit an AR model to the observed sequence of failure inter-arrivals for each LANL system and then use the fitted model to predict the time to next failure each time a checkpoint scheduling decision is to be made; i.e., after a system failure occurs. The new checkpointing interval is determined by plugging the estimate from the AR model into Young's formula. The results from this method are shown in the column labeled "%wasted AR" in Table II.

We observe that in most cases AR provides the lowest level of wasted work among all policies. (For better readability, we have marked in each row in Table II the lowest observed level of wasted work in bold font). The improvements are largest for systems 6, 11, and 13, with levels of wasted work that are 10-15% lower under AR than under Opt (the optimal interval identified through exhaustive search). However, in most cases improvements are more modest with an average improvement of AR over Young(stat) of 4% and over Opt of 7%.

Summary: Among all methods, using autoregression to estimate the expected time until failure performs best. However, the improvements are significant (in the 10-15% range) only in a few cases, and quite modest (in the 2–6% range) in most cases. Hence the typical practitioner is likely to prefer using moving averages (e.g. Young(SMA)), due to their simplicity.

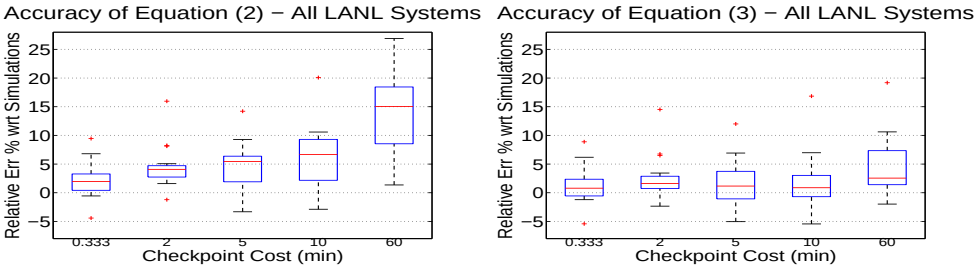

Fig. 7. Errors in estimation of wasted work using equations with respect to the wasted work under Young when using trace-based simulations.

V. System performance for varying configurations

A. A Back-of-the envelope formula for wasted work

So far in this work we have relied on simulations based on actual failure logs to determine the fraction of wasted time. Often it is useful to have a simple back-of-the envelope estimate of wasted work available without having to run simulations. Examples include situations where no failure logs are available for a system, or one wants to experiment with parameters (e.g. the MTTF) that differ from the real system. This allows answering questions such as "How much does the fraction of wasted time drop if I could reduce the checkpoint overhead by a factor of two" or "How many processors can I run on while still keeping the wasted time below some threshold". We next explore simple approaches to estimate the fraction of wasted work in a system.

Consider the fraction of time that is wasted in an HPC system performing periodic checkpoints with an interval Δ. The first component of wasted work is due to the fact that once every Δ time units a checkpoint needs to be written which takes time C. Hence, the system spends C/Δ fraction of its time checkpointing. Secondly, every time a failure happens (i.e. on average once every MTTF time units), some work is lost that needs to be recomputed. The amount of lost work is equal to the time since the last checkpoint. If failures are equally likely to happen anywhere in a checkpointing interval, the expected amount of lost work for each failure would be roughly $\Delta/2$. That means on average the fraction of time spent redoing lost work is $(\Delta/2)/MTTF$. (Note that we do not include the time needed to restart the application to the state of the most recent checkpoint, as this time does not depend on the checkpointing interval and hence will be the same for any checkpointing policy). Combining these two sources of wasted work, means we want to choose Δ to minimize the following function W:

$$W(\Delta) = \frac{C}{\Delta} + \frac{\Delta}{2 \cdot MTTF} \tag{2}$$

It turns out that this function is minimized by choosing Δ according to Young's formula. However, the derivation makes a number of assumptions that are clearly not true in practice, such as the assumption that failures do not happen during checkpoints (recall the discussion in Section II).

We further refine Equation (2) by taking into account that checkpoints take place only during the fraction of failure intervals that are larger than Δ, which we estimate by assuming an exponential distribution (an approximation we make for simplicity, as in theory a Weibull distribution is a better fit). We also take into account that checkpoints are only written once every $C + \Delta$ time units. In combination these modifications lead to the following formula for wasted work W:

$$W(\Delta) = e^{-\frac{\Delta}{MTTF}} \cdot \frac{C}{C + \Delta} + \frac{\Delta}{2 \cdot MTTF} \tag{3}$$

Figure 7 studies the accuracy of the estimations from the two equations versus simulations for all LANL systems. The boxplots show the percentage of error in the estimation of wasted work by the equations, with respect to the wasted work that results under Young when using trace-based simulations. (The actual estimations can be found in the last two columns in Table II).

We observe that Equation (2) results are accurate for smaller checkpoint costs, usually within 5% of the simulation results, but can deviate quite a bit for larger checkpoint costs, in particular for cases where the wasted work is large. For example, the equation overestimates the wasted work by 22% for system 18 for a checkpoint cost of 60 min. While the numbers are still in the same ballpark, it might be desirable to have higher

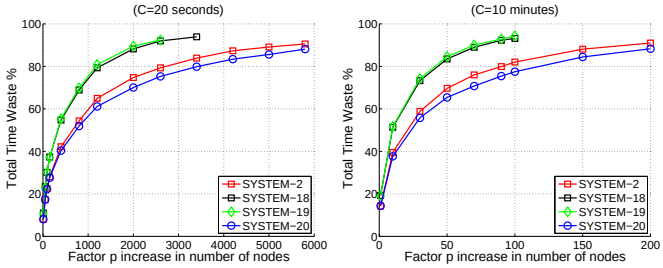

Fig. 8. Wasted time as systems scale out.

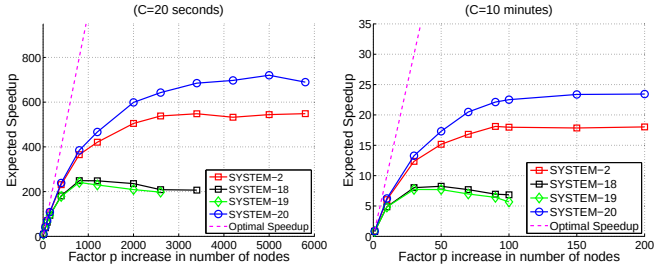

Fig. 9. Speed-up as systems scale out.

accuracy even for those cases. However, we observe highly accurate estimates for Equation (3), with an average error of 3.7% (compared to 6.7% under Equation (2)). Even in the more difficult cases, where wasted work is high, the error is always below 15%. We also experimented with other refinements, addressing some of the other simplifications that the derivation of Equation (2) relied on, but did not find that they lead to significant improvements.

Summary: The fraction of wasted work can be approximated with simple back-of-the envelope formulas based only on the MTTF and checkpointing cost C.

B. Wasted time as systems scale out

As the projected increase in the speed of individual components is limited, performance increases in future systems will have to come from an increase in the number of components. To continue past growth trends for FLOPs of leadership applications component counts in future systems will have to increase by several orders of magnitudes. In this section, we take a look at the wasted work as systems scale out. We assume that the failure rate of individual components will remain the same and that the cost for a node to write a checkpoint does not depend on the total number of nodes involved in the computation (which is consistent with observations in [15] and [16]), but that when increasing the number of components by a factor of p, failure rates will increase by a factor of p.

Figure 8 plots the wasted time for our four different systems, when scaling up the number of nodes, and assuming a hero run of an application which utilizes all available nodes. We observe that in many cases running applications on all nodes in a system becomes infeasible. For example, for p=100X the fraction of wasted time ranges from 65% to 85% for checkpoint costs of C=5min or C=10min, and approaches 100% for C=60min.

The results indicate that some drastic changes need to take place for checkpointing to stay viable. As our work shows that little can be gained from further optimizations to the checkpoint interval and it is unlikely that per-component reliability improves, the only factor in the equation that can make up for increasing component counts is the checkpoint cost C. In particular, in order to keep the fraction of wasted time the same as now, when the number of processors increases by a factor of p the checkpoint cost needs to be reduced by a factor of p. Various suggestions have been made in the past to speed up checkpoints. Below we evaluate their potential to deliver the required reduction in checkpoint costs.

Checkpoint compression: One suggestion is to reduce the amount of data that needs to be checkpointed via data compression techniques. Previous work shows that one can achieve compression factors in the range of 5–10X. [10]. While these factors are impressive, they will not be sufficient to achieve the level of reduction in checkpoint cost that is required.

Checkpointing to flash: Another suggestion is to use local flash drives to store a checkpoint and then slowly in the background drain the checkpoint data to the disk-based parallel file system, while the application continues execution [1]. Flash drives provide significantly faster latency in particular for random reads, compared to traditional hard disk drives. However, the checkpoint cost depends on the sustained write bandwidth of the storage device. Based on vendor specifications the sustained write bandwidth of hard disk drives is in the 80-150 MB/sec, while typical numbers for SSDs are a few hundred MB/sec, with some high-end drives quoting up to GB/sec. That means the expected speed-up in the optimal case will be in the order of 10X, again by itself not sufficient to make up for the increased failure rates.

Incremental checkpoints: Other work investigates the possibility of replacing some full checkpoints with (smaller) incremental checkpoints, where only data that has changed since the most recent checkpoint is being stored [14], [17]. Wang et al. [17] experiment with incremental checkpointing for a number of applications and report incremental checkpoint sizes that are 10-20% smaller than the size of a full checkpoint, again an improvement that is not sufficient by itself.

Alternatives to coordinated blocking checkpoints: An alternative is a deviation from traditional coordinated checkpoints. Most methods for coordination free checkpointing suffer from other overheads, as some form of message logging is usually involved. Our work might renew interests in methods, such as [3], which minimize the degree of coordination.

Summary: The necessary reductions in checkpoint cost in next generation HPC systems will likely require a whole array of new techniques. Improvements from each individual previously proposed technique are on the order of at most 10X, while overall improvements of several orders of magnitude will be necessary. An alternative is a deviation from traditional coordinated blocking checkpoints. Uncoordinated checkpointing will require some form of message logging, which for tightly coupled parallel applications will likely have unacceptabe overheads. However, various methods to make checkpoints non-blocking, such as [3], might become interesting alternatives.

C. Application speed-up with increasing number of nodes

In an ideal world, a parallel application would speed up linearly with the number of nodes it is running on, e.g. a problem that takes 1 hour to solve on one processor should require $1/p$ hours on p processors. In reality such a perfect speed-up is rarely achieved, due to the communication and synchronization overheads.

In this section we explore a second factor limiting the speed-up of a parallel application, which has received less attention: how do failures and the need for fault tolerance affect the speed-up? While running an application on a large number of nodes increases available computational cycles, it also increases failure rates, and hence the amount of time required to write checkpoints and recover lost work.

Figure 9 plots the speed-up for a range of p values, where p indicates the factor by which the number of processors in the original system is increased. In order to isolate the effect of failures on speed-up, we assume that the computational part of an application scales perfectly, e.g. in a world without failures an application achieves speed-up p when using p times as many processors.

We observe that even for relatively small factors increase in the number of nodes (compared to the increases in the number of nodes necessary for future exascale systems), the achieved speed-up is far from the optimal (linear) speed-up. Moreover, after a certain point the overheads due to increased failure rates completely negate the additional compute cycles gained when adding nodes to the system. The point where adding nodes to the system does not result in an increase in speed-up is quite consistently reached when the system reaches around 80% wasted time (compare with Figure 8). After some point, increasing system size actually results in a *decrease* in speed-up. This point is typically reached when the fraction of wasted time reaches around 90%.

Summary: Overheads due to fault tolerance can severely impact the speed-up a parallel application can achieve. Even at a checkpoint cost as small as 20 seconds, the increase in the number of nodes necessary for future systems results in a speed-up that is far from the optimal (linear) speed-up.

VI. Related Work

As a testament to the importance of the problem, a large body of work exists on optimizing the checkpoint interval. Section II provided a brief summary of Young's formula [18] and follow-up work that strives to further improve upon Young. Our work differs from the above in that our goal is not a further refinement of existing approaches. Instead, our focus is on the careful evaluation of different approaches (some previously proposed in the literature, some new approaches, which we propose in this paper) on real world logs, and on our observation that for real world traces very simple methods perform near optimal.

Our work is also the first to look at practical implementation issues of checkpoint interval computation. Previous work typically assumes a-priori knowledge of the underlying statistical failure process (e.g. the distribution of time between failures) and then evaluates the proposed method on (usually synthetically generated) data following these statistical properties. On the other hand, we show that checkpointing based on Young's formula

is robust to errors in the parameter estimates and that near optimal performance can be achieved by learning the parameters on the fly, based only on a short time window of recent failure history for a system. Finally, we show that an adaptation of formulas used in the derivation of Young's formula can be used for accurate back-of-the envelope estimates of the wasted work under different system parameters.

VII. Conclusion

Despite a myriad of papers on the optimization of checkpoint-restart protocols, practitioners still rely on very crude ad-hoc rules of thumb to choose the key parameter in these protocols, the checkpoint interval. The goal of this paper is to remedy this situation by identifying solutions that are *practical*, and at the same time achieve *good performance* in terms of the associated wasted work (due to lost work after failures and time spent writing checkpoints).

Going back 40 years in checkpointing research, we find that one of the oldest and simplest formulas, often criticized for relying on too many unrealistic assumptions, achieves near optimal performance across all 20 failure traces we experiment with. We find that more complex methods that try to correct inaccuracies provide no tangible improvements in the amount of wasted work, even for input scenarios that significantly deviate from those assumptions.

We also look at a number of practical implementation issues and show that Young's formula can easily be adapted for use in practice. We show that all required parameters can be estimated online through a combination of simple window-based methods for MTTF estimation, achieving performance comparable to that under the optimal checkpoint interval (obtained through offline analysis) for a trace.

We investigated a number of more advanced methods which dynamically change the checkpoint interval. One of these is based on a previously proposed idea (using the hazard rate function of a system) and two are new methods we propose. The best performing of these methods is a new method based on MTTF predictions using *autoregression*. However, even for that method improvements over Young are significant (in the 10% range) for only a small subset of the systems and parameters. On average, the improvements of all advanced methods are not large enough to justify the added complexity that comes with them.

While the above results might be disappointing from a theoretical point of view, they are good news for practitioners as it means that they can rely on simple practical methods without sacrificing performance.

We show that a simple back-of-the envelope formula can be used to accurately estimate the wasted work in a system, based only on the MTTF and checkpoint cost C. Such a formula is useful for tuning an application (for example to decide on the number of processors to run on or necessary reduction in the checkpoint cost) or planning future systems.

We perform a number of projections on the impact of checkpoint/restart overheads as the number of nodes in a system increases. We observe that even under optimal checkpoint placement, the limits of traditional coordinated checkpoint/restart might be reached soon, and that likely a combination of techniques will be necessary to keep checkpoint/restart viable. Our findings might encourage more future work on reducing coordination needs for checkpoints, along the line of work by Agbaria et al [3].

References

[1] ExaScale FSIO. Can we get there? Can we afford to? http://institute.lanl.gov/hec-fsio/workshops/2010/presentations/day1/Grider-HECFSIO-2010-ExascaleEconomics.pdf.

[2] Operational Data to Support and Enable Computer Science Research, Los Alamos National Laboratory. http://institute.lanl.gov/data/fdata/.

[3] A. Agbaria and W. H. Sanders. Application-driven coordination-free distributed checkpointing. In *Proc. of ICDCS'05*, Washington, DC, USA.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proc. of SC'09*, USA.

[5] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proc. of SC'11*, NY, USA.

[6] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent. A flexible checkpoint/restart model in distributed systems. In *Proc. of PPAM'09*, Berlin, Heidelberg.

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

[8] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Proc. of DSN'13*, Budapest, Hungary.

[9] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proc, of SC'11*, NY, USA.

[10] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges. On the viability of checkpoint compression for extreme scale fault tolerance. In *Proc. of Euro-Par'11*, Berlin, Heidelberg.

[11] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun. Optimizing HPC fault-tolerant environment: An analytical approach. In *Proc. of ICPP'10*, San Diego, CA.

[12] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott. A reliability-aware approach for an optimal checkpoint/restart model in HPC environments. In *Proc. of CLUSTER'07*, Texas, USA.

[13] P. Mallikarjuna Shastry and K. Venkatesh. Analysis of dependencies of checkpoint cost and checkpoint interval of fault tolerant MPI applications. *Intl. Journal on Computer Science and Engineering (IJCSE)*, 2(8):2690–2697, 2010.

[14] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. Reliability-aware approach: An incremental checkpoint/restart model in HPC environments. In *Proc. of CCGRID'08*, Lyon, France.

[15] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of DSN'06*, Edinburgh, UK.

[16] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, pages 12–22. IOP Publishing, 2007.

[17] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid checkpointing for MPI jobs in HPC environments. In *Proc. of ICPADS'10*, Shanghai, China.

[18] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9), Sept. 1974.

APPENDIX

System ID	MTTF (min)	Weibull (shape)	C (min)	Static			Dynamic (Moving Averages)			Advanced Methods			Equations	
				% wasted Opt(stat)	% wasted Young(stat)	% wasted Daly	% wasted SMA(w=30)	% wasted WMA(w=30)	% wasted EMA(w=30)	% wasted AR	% wasted (Adaptive)	% wasted ETTF(Type)	% wasted Equation (2)	% wasted Equation (3)
2	840.9	0.739	0.333	2.737	2.7807	2.774	2.7021	2.7089	2.7119	**2.6791**	2.7581	2.796	2.8156	2.7575
			2	6.596	6.5647	6.6097	6.4768	6.4628	6.4553	**6.4019**	6.6524	6.7089	6.8968	6.560
			5	10.263	10.3	10.299	10.064	10.051	10.051	**9.9908**	10.285	10.369	10.905	10.089
			10	14.238	14.275	14.265	13.996	14.018	14.051	**13.82**	14.34	14.475	15.422	13.847
			60	31.762	31.996	31.733	31.203	31.31	31.191	**31.188**	32.023	32.373	37.775	29.777
3	3546.2	0.823	0.333	1.334	1.3784	1.3784	1.3317	1.3349	1.3626	1.322	1.3684	**1.316**	1.3711	1.357
			2	3.205	3.2544	3.3404	**3.1644**	3.259	3.2762	3.1721	3.2136	3.319	3.3585	3.276
			5	4.908	5.2461	5.1238	5.1763	5.1522	5.1006	**4.8167**	5.1912	5.16	5.3103	5.108
			10	7.031	7.1231	7.391	7.437	7.1706	7.2192	**6.7791**	7.3324	7.321	7.5099	7.112
			60	16.362	16.658	16.634	16.111	16.837	16.235	**15.936**	17.011	17.179	18.395	16.205
4	3363.4	0.827	0.333	1.363	1.4062	1.4404	1.4246	1.3785	1.3952	1.3667	**1.3542**	1.388	1.4079	1.393
			2	3.243	3.3935	3.3912	3.4476	3.3112	3.3435	**3.2509**	3.426	3.342	3.4486	3.362
			5	5.096	5.4381	5.1906	5.2713	5.2466	5.2903	**5.0086**	5.1207	5.484	5.4527	5.240
			10	7.076	7.3355	7.357	7.2815	7.2111	7.211	**6.9724**	7.2577	7.688	7.7113	7.293
			60	16.660	17.029	16.88	16.818	17.081	16.977	**16.423**	17.478	19.121	18.889	16.588
5	3272.9	0.873	0.333	1.390	1.3905	1.4246	1.4183	1.3913	1.4116	**1.3803**	1.4028	1.449	1.4272	1.412
			2	3.349	**3.3552**	3.3643	3.433	3.4547	3.471	3.4549	3.5174	3.613	3.496	3.407
			5	5.139	5.3036	5.4161	5.2825	5.3405	5.4058	**5.1736**	5.5251	5.591	5.5276	5.309
			10	7.332	7.6851	7.57	7.5888	7.4957	7.4593	**7.2692**	7.4555	8.146	7.8172	7.387
			60	17.039	17.802	17.829	17.67	**17.098**	17.511	17.326	17.869	20.283	19.148	16.789
6	16530	1.025	0.333	0.578	0.62784	0.62685	0.60852	0.60784	0.63347	**0.50624**	0.66722	0.66	0.63506	0.632
			2	1.382	1.5293	1.4989	1.4798	1.5097	1.6164	**1.2299**	1.6346	1.933	1.5556	1.538
			5	2.237	2.3973	2.3053	2.4912	2.3742	2.365	**2**	2.3075	3.461	2.4596	2.415
			10	3.187	3.4335	3.3493	3.5136	3.6032	3.4445	**2.8341**	3.3231	5.665	3.4784	3.390
			60	7.655	7.9	8.2629	8.2598	8.8165	8.4049	**6.8413**	8.3467	16.654	8.5203	8.013
8	5044.8	0.716	0.333	1.106	1.112	1.1298	1.1779	1.1996	1.1596	**1.0608**	1.1179	1.133	1.1496	1.140
			2	2.678	2.7353	2.7306	2.945	2.9689	2.8037	**2.5612**	2.7754	2.867	2.8159	2.758
			5	4.196	4.1981	4.3393	4.5056	4.5293	4.4499	**4.1444**	4.2332	4.806	4.4523	4.309
			10	5.822	6.0805	6.0587	6.434	6.4142	6.3252	**5.7993**	6.1616	6.978	6.2964	6.014
			60	13.632	14.12	14.125	14.535	14.72	14.541	**13.547**	13.98	17.83	15.423	13.848
9	3503.8	0.546	0.333	1.302	1.2921	1.3074	1.2228	1.2148	1.1816	**1.1373**	1.3041	1.224	1.3794	1.365
			2	3.106	3.2379	3.235	2.8146	**2.801**	2.8771	2.8854	3.2916	2.948	3.3788	3.296
			5	4.709	4.9291	4.8465	4.4457	4.4347	**4.3498**	4.7322	4.8142	4.984	5.3424	5.138
			10	6.700	6.873	6.8538	6.1589	5.9046	**5.8766**	6.6944	6.8926	7.103	7.5552	7.153
			60	15.281	15.892	16.078	14.191	**13.609**	13.979	17.192	15.864	18.537	18.506	16.292
10	4103	0.545	0.333	1.169	1.2012	1.2083	1.1742	**1.1442**	1.1466	1.1587	1.2013	1.192	1.2747	1.263
			2	2.807	2.8877	2.9054	2.8764	2.7157	2.6952	**2.6495**	2.8142	2.884	3.1223	3.051
			5	4.363	4.639	4.7285	4.4723	4.4739	4.2483	**4.2214**	4.371	4.616	4.9368	4.761
			10	6.071	6.3135	6.2078	6.1359	6.0069	6.1566	**5.8082**	6.3467	7.076	6.9817	6.637
			60	14.044	14.638	14.468	13.987	13.826	**13.472**	13.553	13.89	17.77	17.102	15.19
11	3618.8	0.564	0.333	1.278	1.3264	1.3405	1.2499	1.2176	1.2154	**1.1662**	1.3559	1.233	1.3573	1.344
			2	3.039	3.1725	3.093	2.8746	2.8118	2.8223	**2.6712**	3.2182	2.981	3.3247	3.244
			5	4.745	4.9438	4.7575	4.5412	4.5949	4.5198	**4.1027**	4.9071	4.744	5.2567	5.058
			10	6.620	6.9612	6.8536	6.2551	6.4282	6.4723	**5.7603**	6.9131	7.054	7.4342	7.044
			60	15.092	15.386	15.488	14.876	15.144	15.239	**13.293**	15.199	17.326	18.21	16.061
12	3824.8	0.615	0.333	1.213	1.3031	1.2886	1.1826	1.1969	1.2009	**1.084**	1.2728	1.256	1.3202	1.307
			2	2.887	3.1095	3.0847	2.8978	2.8796	2.8507	**2.7687**	3.18	3.112	3.2339	3.158
			5	4.580	4.8733	4.6545	4.4447	4.4494	**4.2104**	4.7207	4.8258	4.804	5.1133	4.925
			10	6.443	6.7878	6.9175	6.4911	6.0773	**5.9235**	6.5195	6.7509	7.073	7.2312	6.862
			60	14.862	15.951	15.345	14.299	**14.21**	14.491	16.375	15.786	17.862	17.713	15.672
13	4955.3	0.504	0.333	1.066	1.086	1.0809	1.0398	1.0549	1.0341	**0.91997**	1.0976	1.091	1.1599	1.150
			2	2.477	2.6246	2.7177	2.4682	2.4105	2.4288	**2.1734**	2.6277	2.683	2.8412	2.782
			5	3.994	4.1111	4.1273	3.8404	3.8005	3.8475	**3.5019**	4.1479	4.486	4.4923	4.346
			10	5.542	5.7855	5.6222	5.1827	5.3735	5.1697	**4.7498**	5.8213	6.263	6.353	6.066
			60	12.857	13.128	13.313	12.511	11.908	12.629	**11.212**	12.622	16.137	15.562	13.960
14	6171.4	0.405	0.333	0.899	0.94964	0.98178	0.92814	0.88763	0.88239	**0.80583**	0.95477	0.952	1.0394	1.031
			2	2.138	2.1955	2.2244	2.2653	2.1435	2.1492	**2.1058**	2.1611	2.872	2.5459	2.498
			5	3.324	3.5246	3.5902	3.3664	3.2901	3.0893	**2.9353**	3.5474	5.429	4.0254	3.908
			10	4.562	4.7409	5.004	4.5693	4.4179	**4.3886**	4.3896	4.9067	8.599	5.6928	5.461
			60	10.329	10.988	10.976	10.69	11.377	11.2	**10.337**	10.878	15.953	13.944	12.642
15	7329	0.878	0.333	0.871	0.95433	0.99989	0.97766	0.95639	**0.9237**	0.94785	0.92944	0.993	0.95375	0.947
			2	2.137	2.3648	2.2489	2.3116	2.3188	**2.2071**	2.3738	2.2121	2.786	2.3362	2.296
			5	3.264	3.8206	3.8309	3.5734	**3.296**	3.3935	3.7031	3.553	5.027	3.6938	3.595
			10	4.887	5.3786	5.8569	5.1086	**4.8425**	5.0286	5.1984	5.2196	8.021	5.2239	5.028
			60	11.379	11.878	11.665	11.609	12.58	11.837	**11.138**	12.606	16.648	12.796	11.689
16	1260	0.722	0.333	2.244	2.2574	2.2681	2.2454	2.275	2.2562	**2.1866**	2.2793	2.254	2.3003	2.261
			2	5.357	5.4337	5.4838	5.4	5.4386	5.4236	**5.3206**	5.4854	5.334	5.6346	5.407
			5	8.313	8.4639	8.4391	8.4557	8.4261	8.3871	**8.2875**	8.447	8.369	8.909	8.356
			10	11.663	11.794	11.816	11.759	11.873	11.741	**11.594**	11.682	11.85	12.599	11.524
			60	26.227	26.262	26.537	26.548	26.452	26.267	**25.856**	26.553	27.158	30.862	25.249
18	449	0.817	0.333	3.794	3.779	3.7776	3.7907	3.7981	3.7733	**3.7514**	3.8077	3.8218	3.8534	3.746
			2	9.063	9.0716	9.1506	9.1589	9.0766	9.1578	**8.988**	9.1353	9.0717	9.4389	8.820
			5	14.008	14.128	14.064	14.004	**13.836**	13.986	13.962	14.146	13.974	14.924	13.443
			10	19.233	19.283	19.385	19.334	19.215	19.292	**19.009**	19.406	19.305	21.106	18.282
			60	41.592	42.283	41.803	42.021	41.734	41.974	**41.687**	41.94	42.63	51.699	38.098
19	470.3	0.889	0.333	3.742	3.74	3.7249	3.7166	3.7164	3.734	**3.709**	3.7511	3.7165	3.7649	3.662
			2	8.898	8.995	8.9243	8.8809	8.9641	8.8849	**8.8224**	8.9897	8.8767	9.222	8.630
			5	13.740	13.789	13.74	13.771	13.781	13.781	13.785	13.812	**13.711**	14.581	13.164
			10	19.001	19.065	19.178	19.037	19.064	**18.998**	19.028	19.195	19.122	20.621	17.916
			60	41.527	42.026	**41.594**	41.898	41.651	41.696	41.853	41.918	42.188	50.511	37.423
20	815.4	0.646	0.333	2.788	2.8009	2.8177	2.6685	**2.632**	2.6618	2.6556	2.7829	2.7089	2.8594	2.800
			2	6.658	6.6932	6.681	6.2902	**6.2899**	6.3113	6.3592	6.7144	6.4369	7.0041	6.657
			5	10.235	10.273	10.332	**9.7618**	9.7753	9.7983	9.8406	10.292	9.8903	11.075	10.234
			10	14.164	14.35	14.263	13.735	**13.606**	13.785	13.727	14.315	13.768	15.662	14.040
			60	30.760	30.994	31.315	30.147	30.124	**30.061**	30.331	30.941	30.509	38.363	30.148
21	1437.6	0.685	0.333	2.100	2.2529	2.0573	2.0358	1.9441	2.0091	**1.7863**	2.1417	2.049	2.1535	2.119
			2	4.817	5.0516	5.0356	4.9803	4.8634	4.8031	**4.4285**	4.7894	4.754	5.2749	5.075
			5	7.782	8.0971	8.0082	7.9533	7.5754	7.6675	**6.8231**	8.2176	7.482	8.3404	7.853
			10	11.037	11.818	11.472	10.511	10.366	10.232	**9.7075**	11.293	10.687	11.795	10.847
			60	24.254	25.431	25.46	23.964	22.916	23.197	**21.531**	24.344	24.418	28.892	23.901
23	8740	0.735	0.333	0.848	0.87829	0.88462	0.98185	0.96799	0.93377	**0.81556**	0.86959	0.846	0.87338	0.868
			2	2.065	2.1057	2.1518	2.3379	2.35	2.2872	**2.013**	2.0775	2.211	2.1393	2.106
			5	3.242	3.3781	3.3374	3.7333	3.7439	3.518	**3.1701**	3.3398	3.728	3.3826	3.299
			10	4.505	4.6612	4.6919	5.1912	5.1643	5.0098	**4.5793**	4.6064	5.426	4.7837	4.619
			60	10.618	11.054	10.871	11.974	11.97	11.652	**10.312**	11.096	14.478	11.718	10.781
24	24133	0.716	0.333	0.503	0.52094	0.5248	0.51615	0.5111	**0.49673**	0.50762	0.53759	0.535	0.52559	0.524
			2	1.220	1.2367	1.2438	1.2476	1.2452	1.208	**1.2021**	1.2955	1.597	1.2874	1.275
			5	1.897	2.0143	1.9559	1.9777	1.9246	**1.8754**	1.9268	2.0498	2.888	2.0356	2.005
			10	2.748	2.9262	2.8001	2.8866	2.6974	2.7379	**2.6244**	2.7079	4.753	2.8788	2.818
			60	6.515	6.9571	6.7611	6.464	6.7574	6.3707	**6.3615**	6.4429	8.417	7.0515	6.7

TABLE II

COMPARISON OF THE FRACTION OF TIME WASTED WHEN USING DIFFERENT CHECKPOINTING METHODS, FOR ALL LANL SYSTEMS.