Multilevel Diskless Checkpointing

Doug Hakkarinen, *Student Member*, *IEEE*, and Zizhong Chen, *Senior Member*, *IEEE*

Abstract—Extreme scale systems available before the end of this decade are expected to have 100 million to 1 billion CPU cores. The probability that a failure occurs during an application execution is expected to be much higher than today's systems. Counteracting this higher failure rate may require a combination of disk-based checkpointing, diskless checkpointing, and algorithmic fault tolerance. Diskless checkpointing is an efficient technique to tolerate a small number of process failures in large parallel and distributed systems. In the literature, a simultaneous failure of no more than N processes is often tolerated by using a one-level Reed-Solomon checkpointing scheme for N simultaneous process failures, whose overhead often increases quickly as N increases. We introduce an N-level diskless checkpointing scheme that reduces the overhead for tolerating a simultaneous failure of up to N processes. Each level is a diskless checkpointing scheme for a simultaneous failure of i processes, where $i = 1, 2, \ldots, N$. Simulation results indicate the proposed N-level diskless checkpointing scheme achieves lower fault tolerance overhead than the one-level Reed-Solomon checkpointing scheme for N simultaneous processor failures.

Index Terms—Extreme scale systems, high-performance computing, fault tolerance, checkpoint, diskless checkpointing

1 Introduction

Fault tolerance is becoming a critical part of high-performance computing. As processor speeds are no longer increasing at previously observed rates, manufacturers of computers and high-performance computers are using more components and processors to continue improving computational speeds. Using the well-accepted heuristic that a computer has a constant failure rate during most of its operation, failure rates increase linearly with regards to the number of components. If a failure occurs during a computation without a system in place to recover such a failure, unfortunately, the only course of action is to completely restart the computation.

The prevalence of computers in new fields has created greater amounts of data to analyze. This need for computation over larger data sets has resulted in longer computations. With increased time of computation, the penalty for a failure resulting in loss grows. Furthermore, as the computation requires more time with more data, the likelihood of a failure during a specific execution of a program also increases. Due to the higher failure rate and data loads, methods of avoiding and mitigating failures are critical. Specifically, methods to enable computations to recover from a failure are needed.

One way to address fault tolerance is through the use of checkpoints. In disk-based checkpointing, checkpoints are regularly backed up on stable storage during computation. To back up the processor state, each processor generally communicates its state to a storage system. Checkpoints can either be scheduled periodically by the system or scheduled by the application developer. Checkpoints introduce additional overhead, but reduce the amount of recomputation in the case of a failure.

One complexity of checkpointing is ensuring that the processor state is consistent, or that messages actively being sent between processors are not lost during a failure. Strategies to ensure consistency depend on whether the checkpoints are uncoordinated or coordinated. Uncoordinated checkpoints require additional effort (e.g., rollback-dependency graphs and rollback propagation algorithm) to ensure that checkpoints are not *useless*, or unable to be part of a consistent state [25]. Coordinated checkpointing requires orchestration of checkpoints to form a consistent global state. The consistency difficulty applies for checkpoint schedules determined by the system as well as for checkpoint schedules that are determined by the application developer. In application level checkpointing the application developer is responsible for ensuring that the processor state is consistent before calling for a checkpoint, sometimes performing checkpoints at a barrier. Tools such as found in [23] assist application developers to schedule checkpoints. We focus on coordinated checkpoints, specifically aimed at application level checkpoints. However, even for application level checkpoints, schedules are important because the schedule factors into the expected runtime of a program.

There are other methods of providing fault tolerance, including redundant execution [16] and algorithmic-based fault tolerance (ABFT) [2], [3], [4], [5], [6], [10]. Redundant execution runs multiple executions in order to provide fault tolerance, but may require a large number of additional processors. ABFT explores using redundant data available within specific algorithms for use in fault tolerance. ABFT has low overhead, but is limited to specific algorithms rather than being a general strategy. As such, checkpointing remains a leading strategy.

The simultaneous access of the stable storage by a large number of processors is potentially a bottleneck [12]. With larger data sets, more data must be checkpointed. Furthermore, as failure rates increase, the optimal frequency of checkpoints generally increases. These two factors require

- *D. Hakkarinen is with the Department of Electrical Engineering and Computer Science, Colorado School of Mines, Golden, CO 80401. E-mail: dhakkari@mines.edu.*
- *Z. Chen is with the Department of Computer Science and Engineering, University of California, Riverside. E-mail: chen@cs.ucr.edu.*

Manuscript received 7 Mar. 2011; revised 25 Nov. 2011; accepted 17 Dec. 2011; published online 4 Jan. 2012.
Recommended for acceptance by N. Ranganathan.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-03-0154.
Digital Object Identifier no. 10.1109/TC.2012.17.

that disk-based checkpointing systems must provide ever increasing bandwidth to avoid a bottleneck. Research continues in methods to increase the bandwidth available to disk systems for checkpoints, such as through parallel I/O. If the necessary bandwidth is available, disk-based checkpointing often provides excellent performance. If the bandwidth to disk is limited, diskless checkpointing may be a viable option to provide or augment checkpointing.

Diskless checkpointing [12] provides redundancy by having all processors routinely calculate a Reed-Solomon encoding of the processor states and store the encoding on a dedicated checkpoint node. Assuming nodes have the same amount of memory, recovering from n failures will require an additional n nodes, as derived from [24]. If a processor fails, then the original state of that processor at the last checkpoint can be reconstructed using the redundant information on the checkpoint processors and the states of all the other surviving processors. The checkpoint and recovery can be performed through a reduction operation. In cases where memory is limited or the message size is large, using a pipelined approach to this reduction can reduce the overhead of these operations. One method for performing checkpoints efficiently is shown in [7]. The original state of a processor can be reconstructed using the memory content of other processors. Specifically, the state that was last transmitted is recoverable. Diskless checkpointing can only recover up to a specific number of failures before recomputation is necessary. The number of failures is determined by the diskless strategy used and number of dedicated checkpoint processors. In general, the capability of recovering more failures requires more redundant processors and increased overhead per checkpoint.

The vision of a fault-tolerance system for exascale computational environments may involve multiple strategies. ABFT may be used for computations where ABFT capable algorithms dominate the execution time [2], [3], [4], [5], [6], [10]. Disk-based checkpointing can be used when bandwidth allows. Diskless checkpointing can augment disk-based checkpointing through layering diskless and disk-based checkpointing. In this paper, we explore the benefits of layering diskless checkpoint schemes within diskless checkpoint schemes. This work is useful for both pure diskless checkpointing systems or improved layered diskless with disk-based systems.

Previous research in diskless checkpointing includes diskless checkpointing implemented to assist with specific algorithms on clusters of workstations [13], application of diskless checkpointing to superscalar architectures [9], methodologies for increasing the performance of diskless checkpointing using incremental checkpoints [15], and the use of floating-point coding in conjunction with diskless checkpointing to improve performance and portability [8].

Diskless checkpointing has also been used to tolerate a simultaneous failure of no more than N processors by using a Reed-Solomon encoding scheme for N simultaneous processor failures. However, the amount of time to recover after a failure and the amount of time to perform a checkpoint usually increase quickly as N increases. Using the results from [7], the overhead and recovery times for a diskless checkpoint scheme increase linearly with the number of failures the scheme is capable of handling.

This paper proposes an N-level diskless checkpointing scheme to reduce the overhead for tolerating a simultaneous failure of no more than N processors by layering the diskless checkpointing schemes for a simultaneous failure of i processors, where $i = 1, 2, \ldots, N$. While the N-level diskless checkpointing scheme is able to tolerate a simultaneous failure of N processors, it tolerates the more frequent case of a simultaneous failure of less than N processors with lower overhead than the one-level scheme for N simultaneous processor failures. The N-level diskless checkpointing scheme can reduce the total fault tolerance increase in runtime significantly compared with a one-level scheme. We focus on developing and verifying an analytical model for N-Level diskless checkpointing under the assumption that checkpoints are evenly spaced. We then examine how to determine the number of checkpoints to use by level. We verify the analysis through simulation.

2 Related Work

Plank and Li present a system that uses an additional processor to help perform diskless checkpointing and outlines how to expand the process to tolerate multiple failures [14]. Plank's work also extended this technique for a high-performance network of workstations and evaluated several diskless checkpointing variants [12]. Other work in the field includes the comparison of multiple types of diskless checkpointing [19]. Techniques have been developed to handle multiple failures for specific matrix operations [11]. Finally, [30] uses diskless checkpointing in a configuration called double checkpointing to provide fault tolerance for Charm++ parallel programming. Double checkpointing provides a single layer of diskless checkpointing capable of multiple failure recovery. In our work we demonstrate the benefit of multiple layers of diskless checkpoints.

The layering of diskless and standard checkpoints was studied in [18], [20], [21]. Layering checkpoints that can only detect inconsistent states with checkpoints that can only recover inconsistent states has been explored [22]. Layering checkpoints has also been used for creating a two layer recovering scheme for shared disk architectures [1] and application specific multilevel fault tolerance schemes [17].

Our work requires consistent and coordinated checkpoints at approximately equal intervals. Forced checkpoints [26] are one method of addressing the difficulties of creating consistent checkpoints. Using forced checkpoints increases the flexibility to produce consistent states, but at the cost of the additional checkpoints.

Layering different diskless checkpointing schemes to tolerate multiple simultaneous failures appears to be novel.

3 Diskless Checkpointing Review

We review the methods of diskless checkpointing to aid understanding of our scheme. We review requirements to coordinate, store, and encode checkpoints.

Like any checkpoint scheme, diskless checkpoints must ensure a consistent state is maintained. Coordinated checkpoints can accomplish this through several methods [25]. Blocking checkpoint coordination requires an orchestrating process to initiate a checkpoint. The coordinator creates a checkpoint and then notifies other processors to take a checkpoint. Other processes then flush their communication channels, take a *tentative* checkpoint, and then acknowledge the coordinator. Once the coordinator receives all acknowledgments, it broadcasts a commit message that finishes the

```
/* Multi-level diskless checkpointing. */

1 : Determine when to perform which level of
    the checkpoints;
2 : Coordinate to obtain a consistent state;
3 : Take the local in memory checkpoints;
4 : Encode local in memory checkpoints to
    dedicated checkpoint processes
    using the Reed-Solomon error correcting code;
```

Fig. 1. N-level diskless checkpointing algorithm.

checkpoint. Another method is nonblocking checkpoint coordination, where a checkpoint request is placed within the communication channels (assuming reliable FIFO channels). This strategy eliminates the problem of undelivered messages that cause an inconsistent state. A third method is to use loosely synchronized clocks to enable checkpoint coordination. This functions by taking local checkpoints at a specified time and then waiting the maximum time for messages or failures to occur to consider the checkpoint consistent [27], [28], [29]. No matter which method is used, diskless checkpoints require a consistent state to act.

As mentioned in the introduction, checkpoints may be scheduled by either the system or the application developer. We assume that the application developer will be scheduling the checkpoints and will ensure that these are done at a point where the state is consistent. One common way to ensure consistency is to perform checkpoints at a synchronization point in a program. Tools such as [23] aid developers to schedule consistent checkpoints.

In many applications of diskless checkpointing capable of handling one failure, a dedicated checkpoint processor is maintained. The storage overhead is one copy of the local checkpoint. The copy of the local checkpoint is equivalent to the size of the processor state on one of the other processors. For checkpoint methods that require more than one failure redundancy, one dedicated checkpoint processor is needed per number of failures that must be simultaneously recovered. Additionally, the checkpoint storage overhead increases correspondingly.

For our method, we use Reed Solomon encoding. Performing the entire Reed-Solomon encoding through a binomial tree approach could have high overhead due to the size of messages and memory needed. Chen and Dongarra [7] describe a method of reducing the overhead of this process in cases where the message size is large. In the case of using diskless checkpointing, the message size is the processor state, i.e., large. This method involves pipelining, sending, and encoding. Pipelining functions by dividing the message on each processor into segments of equal size. The checkpoint for a segment is calculated by repeated transmissions and partial encoding linearly across processors. After the segment clears the last noncheckpoint processor, it can be sent to the checkpoint processor. The next segment is started once the previous segment clears the second processor. This process is repeated for each segment. After every segment is complete, the entire checkpoint is complete. The runtime is dependent on the latency α, the bandwidth β, the time to compute the sum of two bytes within arrays γ, the number of processors p, and the message size m. The runtime of performing a checkpoint for a k-failure encoding decreases from $\approx 2 \cdot \lceil \log p \rceil \cdot k(\beta + 2\gamma)m$ for a binomial encoding to $\approx (1 + O(\frac{\sqrt{p}}{\sqrt{m}}))^2 \cdot k \cdot (\beta + 2\gamma)m$ for the pipelined encoding [7]. The additional memory needed is on the order of the size of a segment.

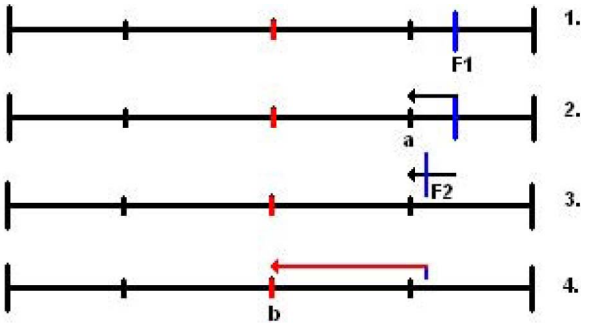

Fig. 2. Flow of a two-failure recovery. 1) Failure occurs at time F1. 2) System attempts to recover to most recent one failure checkpoint (a). 3) Failure occurs while attempting recovery at F2. 4) System recovers to most recent two-failure checkpoint (b).

4 N-Level Diskless Checkpointing

4.1 Checkpointing Algorithm

To use N-level diskless checkpointing, a schedule for the diskless checkpoints must be developed for each level of recovery supported. In other words, checkpoints that can recover from one, two, and up to N simultaneous failures must be scheduled. When a checkpoint is scheduled, each processor takes specific steps (see Fig. 1). The coordination of checkpoints by processors is important to ensure the consistency of the checkpoint. After a local checkpoint, the processors communicate their states to each other. Reed-Solomon encoding of these combined states is performed and stored to ensure that no matter which processors fail, the remaining processors are capable of reconstructing the state of the failed processors. Communication and encoding can be pipelined together to improve performance and avoid bottlenecks. For any diskless checkpoint there is both communication and calculation overhead. An N-failure checkpoint can recover any number of failures from $1, 2, \ldots, N$.

4.2 Recovery Algorithm

Upon a detected failure, an N-Level Diskless Checkpointing system first attempts to use the most recent recovery checkpoint (usually 1-failure) to recover the state of the failed processor. Should an additional failure occur during this recovery period, the system then uses the most recent two-failure recovery checkpoint to recover both failed processor states. We consider any failures that occur before recovery has completed to be simultaneous. In general, after m failures have occurred without any successful recovery, the system returns to the most recent recovery checkpoint that is capable of handling m failures and uses that checkpoint to attempt to recover the failed processors. If the number of processor failures exceeds the number that is supported, the system needs to restart the computation. Fig. 2 shows the flow of a typical two-failure case.

Multiple layers allow for additional flexibility in the use of diskless checkpointing. Previously, most diskless checkpointing schemes only allow for a single failure. The interval between diskless checkpoints in a single failure scheme is optimizable using similar techniques to those

used for disk-based checkpointing. Diskless checkpointing and disk-based checkpointing differ when a failure during a recovery occurs. In disk-based checkpointing, the system simply restarts from the same checkpoint. In diskless checkpointing, the system may need to restart from an earlier checkpoint. In this respect, N-level diskless checkpointing has a disadvantage against disk-based checkpointing; however, the gains in not using stable storage make the development valuable. The possibility of falling through a checkpoint increases the importance of understanding the expected execution time for using a layered diskless system.

5 Performance Analysis

5.1 Assumptions for Model

We assume that the system requires that all processors are functional in order to proceed with computation. This assumption is reasonable as it reflects the default mode among many implementations of Message Passing Interface (MPI). We also assume that the version of MPI is capable of allowing for recovery from a fail-stop failure, as provided by FT-MPI. It is also assumed that a system fails with a constant failure rate, leading to an exponential distribution of failures.

Let S_i denote a scheme that is capable of recovering from i failures. With only $S_1, S_2, \ldots S_n$, only a maximum of n failures are recoverable. Note that n must be less than the total number of processors, otherwise the entire job must be restarted. It is possible to have some form of nondiskless checkpointing scheme set up rather than restart completely, but that is beyond the scope of this research.

We initially analyze the case where up to two failures can be recovered. The checkpointing scheme consists of N_2 two-failure checkpoints that can recover successfully from two failures at a delay of t_2. Each two-failure checkpoint requires t_{c2} to set up. In each time period started or ended by a two-failure checkpoint, there are N_1 one-failure checkpoints, each is capable of recovering from one failure with a delay of t_1 and requires t_{c1} to setup. Furthermore, if a restart is required, we assume that an additional time t_3 occurs. Finally, before the program can complete, an additional time t_{c3} is waited. By adding t_3 and t_{c3} the modeled system acts as if it were inside a disk-based checkpointing system. To use this scheme in conjunction with disk-based checkpoints, the disk-based checkpoint setup time should be used in place of t_{c3} and recovery time should replace t_3.

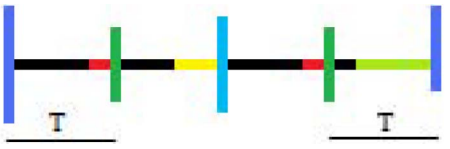

Fig. 3. Demonstration of assumptions. Each interval T is constant, but the amount of "useful" work done in an interval changes (shown in black). The other colors show the respective amounts of time spent checkpointing for the checkpoint after them.

In an actual program, choosing when a checkpoint occurs may not be possible due to the need for consistent states, however frequently a suitable approximation is available and can be determined by the application programmer. We assume that the time that a checkpoint begins is not limited by this factor. We realize that this is a concern that would impact the performance of a layered diskless checkpointing system. The impact of variation from the proposed schedule may be examined in future work.

We propose a schedule where each checkpoint occurs at a distinct time from all other checkpoints. We use the convention that the amount of time from the end of one checkpoint to the end of the next checkpoint, or interval, will be a constant value, T. Since different levels of checkpoints require different amounts of time to perform, progress on executing the program varies from one interval to the next. This is shown in Fig. 3, where the black line represents time spent executing program code, and colors represent time spent setting up checkpoints. Fig. 4 shows an alternative view of the checkpoint scheduling indicating the ordering of the layered checkpoints. Based on the construction there will be a total of $K = (N_2 + 1) \cdot (N_1 + 1)$ intervals of length T. Of these, one ends the program, N_2 intervals end with a second-level checkpoint, and $N_1(N_2 + 1)$ end with a first-level checkpoint. Fig. 5 shows how the checkpoints are scheduled.

5.2 Expected Runtime for a Two-Level Scheme

We seek a closed-form analysis for the expected runtime to ease the determination of when a multilevel diskless checkpointing should be used. As there is the possibility

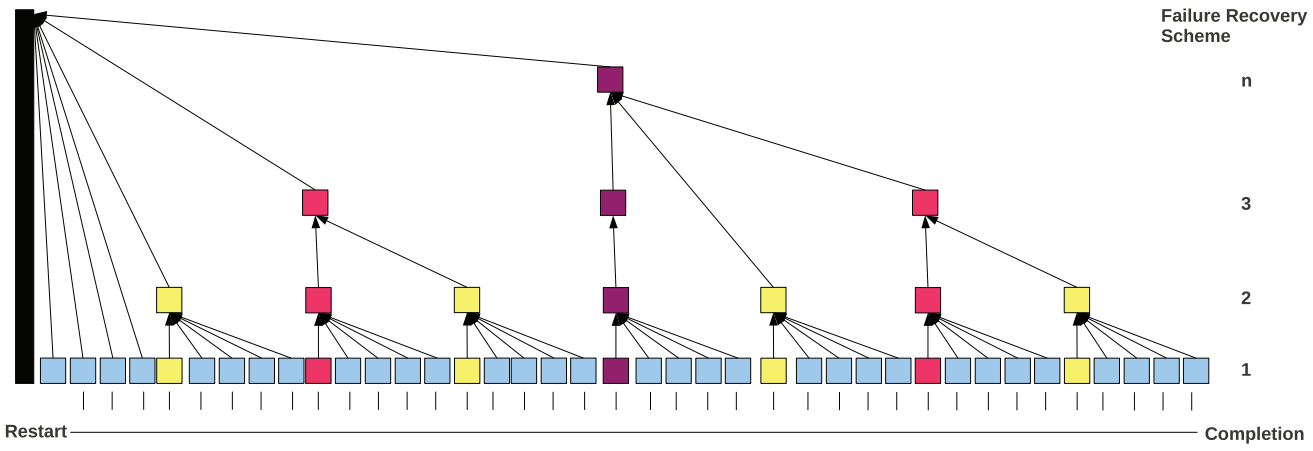

Fig. 4. Structure of the checkpointing schemes. Failures will attempt to recover to the previous diskless checkpoint first. If additional failures cause this to be impossible, the recovery falls to the next level. N-Level diskless checkpointing can only recover up to N failures before restart is necessary.

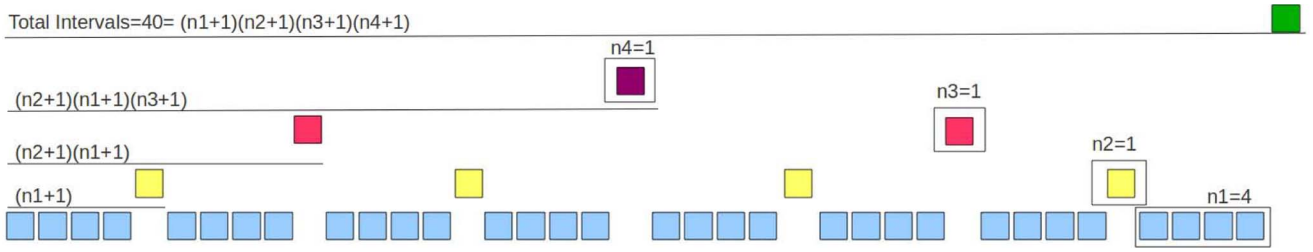

Fig. 5. Alternative view demonstrating the checkpoint schedule. The total number of intervals is calculated as $(N_1+1)(N_2+1)\ldots(N_n+1)$.

of fall-through of lower levels of checkpoints, a closed-form analysis is needed for the analysis of the tradeoffs of using this scheme. One tradeoff is that falling through a checkpoint level increases the time for that particular execution. However, this increased time may be overcome by reduced checkpoint overhead. Finding the expected runtime for a system requires analyzing the expected time for each interval between checkpoints individually. Unlike disk-based checkpoints, once a checkpoint is reached it is still possible, given a certain number of failures, to have to reperform work before that checkpoint. The use of intervals will simplify this analysis. One key question is what length of time should be set as an interval. This can be solved as a function of the checkpoint times and the initial faultless, uncheckpointed time as shown in (1). With the value of T determined, the checkpoint schedule can be determined by subtracting the length of each checkpoint setup from checkpoint end. However, this does not affect the approach in this paper and thus will not be explored further.

$$\begin{aligned} t_{init} &= (T - t_{c3}) + \sum_{j=1}^{N_2}(T - t_{c2}) + \sum_{j=0}^{N_2}\sum_{i=1}^{N_1}(T - t_{c1}), \\ T &= \frac{t_{init} + t_{c3} + N_2(t_{c2}) + t_{c1}(N_2+1)N_1}{(1 + N_2 + (N_2+1)N_1)}. \end{aligned} \tag{1}$$

Overall, the objective is to determine the expected value of the runtime of a system. Since the intervals are disjoint pieces of time that contain the entire runtime, linearity of expectations can used to show that the expected runtime of the system is the sum of the expected times to complete each interval.

The expected time to step from one interval, say the $(k-1)$th to the kth is inspected, denoted by E$(k-1,k)$. Using the total law of probability, this can be viewed as the sum of expectations given disjoint events weighted by the probabilities of those events. This analysis uses the disjoint events of differing numbers of failures. Equation (2) shows the general form of this equation where f is the number of failures and n is the maximum number of failures tolerated before restart must take place. In the case examined in detail later in this paper, n would be two.

$$\begin{aligned} E(k-1,k) = {} & E(k-1,k|f=0)P(f=0) \\ & + E(k-1,k|f=1)P(f=1) \\ & \cdots \\ & E(k-1,k|f>n)P(f>n). \end{aligned} \tag{2}$$

The expected amount of time for a given number of failures can be determined. If there are no failures, the expected time for an interval is just the length of the interval, T. If there is one failure, the last checkpoint will be able to recover it. This means that it would only need to use a one-failure recovery scheme and restart the interval. This will involve losing any amount of time spent working on the interval before failure, the time to recover from one failure t_1, and the expected amount of time to successfully complete the interval. For two failures, the expected value should be the amount of time lost before a failure, the time lost trying to recover using a one-failure checkpoint, the time to recover from two failures t_2, and the expected time to complete the interval from the last two failure checkpoint. This last term depends on how many intervals have successfully completed since the last two-failure checkpoint. If more than two failures occur, then the program will need to start over after a delay t_3. To complete the interval from the beginning requires the completion of all previous intervals as well as the interval in question.

In order to aid in the modeling of these expected values, let a function *z(k)* be defined by (3). *z(k)* is the interval where the last two-failure checkpoint occurred in the process of attempting to complete interval k. Subsequently, if two failures should occur, the execution will drop back to start performing the $(z(k)+1)$ interval of work.

$$z(k) = (\lfloor (k-1)/(N_1+1) \rfloor \cdot (N_1+1)). \tag{3}$$

Since a constant failure rate is assumed, the failure distribution is the exponential distribution and is subsequently memoryless. Thus, the probability of a failure occurring only depends on the amount of time that passes. Equation (4) shows the probability density function and cumulative density function for a system with a constant failure rate of λ. Our later simulations govern the failure probabilities through the manipulation of λ.

$$\begin{aligned} f(t) &= \lambda e^{-\lambda t}, \\ F(t) &= 1 - e^{-\lambda t}. \end{aligned} \tag{4}$$

The probability of an interval being completed with no failure is 1-F(T). For a system to have only one failure, it must fail in the first T amount of time and then not fail for the next t_1 time, as it would have failed twice in that case. For a system to have only two failures, it must fail in the first T amount of time, then fail again in the next t_1 amount of time, and finally not fail in the next t_2 amount of time. To have three or more failures it must fail in the first T, then again in the next t_1, and again in the next t_2. With all of the

probabilities and expected values specified, (5) shows the expected time to complete the kth interval.

$$\begin{aligned}E_1(k-1,k) &= T\cdot(e^{-T\lambda})\\ &+\int_0^T(\tau_1+R_1)e^{-t_1\lambda}\lambda e^{-\lambda\tau_1}d\tau_1\\ &+\int_0^T\int_0^{t_1}(\tau_2+\tau_1+R_2)e^{-t_2\lambda}\lambda e^{-\lambda\tau_2}d\tau_2\lambda e^{-\lambda\tau_1}d\tau_1\\ &+\int_0^T\int_0^{t_1}\int_0^{t_2}(\tau_3+\tau_2+\tau_1+R_3)\lambda^3e^{-\lambda(\tau_3+\tau_2+\tau_1)}d\tau_3d\tau_2d\tau_1\\ &with\\ R_1 &= t_1+E_1(k-1,k),\\ R_2 &= t_2+\sum_{j=z(k)+1}^{k}E_1(j-1,j),\\ R_3 &= t_3+\sum_{j=1}^{k}E(j-1,j).\end{aligned}\tag{5}$$

Equation (6) shows this result of this integration, with the derivation of this result available in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2012.17. α_T, β_T, and γ_T are functions of the recovery times and checkpoint times.

$$\begin{aligned}E_b(k-1,k) &= \alpha_T\sum_{j=1}^{z(k)}E(j-1,j)+\gamma_T\\ &= \theta_TE_b(k-(N_1+1)-1,k-(N_1+1))\\ &with\\ \theta_T &= \left(1+\alpha_T\left(\sum_{i=0}^{N_1}(1+\alpha_T+\beta_T)^i\right)\right).\end{aligned}\tag{6}$$

The expected time of all the intervals up to and including the first two-failure checkpoint can be expressed as a function of E(0,1). Each interval immediately after any second-level checkpoint can be expressed as a function of all the checkpoints before. Each interval that is not immediately after a two-failure checkpoint (that is not the very first checkpoint) can be expressed as a function of the interval before it. Thus, by induction, every interval can be expressed as a function of E(0,1). The total expected running time, E_{tot}, can be represented by (7), where K is the total number of intervals of length T. It should be noted that $N_2 = K/(N_1+1)-1$.

$$\begin{aligned}E_{tot} &= \sum_{j=1}^{K}E(j-1,j)\\ &= E(0,1)\\ &+\sum_{i=1}^{K/(N_2+1)-1}E_b(i*(N_2+1),i*(N_2+1)+1)\\ &+\sum_{i=0}^{K/(N_2+1)-1}\sum_{j=1}^{N_1}E_a(i(N_2+1)+j,i(N_2+1)+j+1).\end{aligned}\tag{7}$$

The overall expected time shown in (7) can be expressed as a function of T and the expected time of the first interval. This is shown in (8), with the derivation available in the Appendix, which is available in the online supplemental material. Thus, for this two-level checkpoint schedule, a closed form expected execution time exists.

$$\begin{aligned}E_{tot} &= \sum_{j=1}^{K}E(j-1,j)\\ &= \sum_{i=0}^{K/(N_1+1)-1}\sum_{j=1}^{N_1+1}E(i(N_1+1)+j-1,i(N_1+1)+j)\\ &= \sum_{i=0}^{K/(N_1+1)-1}\phi_TE(i(N_1+1),i(N_1+1)+1)\\ &= \phi_TE(0,1)\sum_{j=0}^{K/(N_1+1)-1}\theta_T^j\\ &= \phi_T\gamma_T\sum_{j=0}^{N_2}\theta_T^j.\end{aligned}\tag{8}$$

5.3 Finding Good Checkpoint Intervals

To find the best possible runtime for a given set of recovery times and checkpoint setup times, optimal values for N_1 and N_2 must be found. If N_2 is zero, the expected runtime is shown in (9). A bound on N_2 and N_1 can thus be formed based on (9). Equation (10) shows this case.

$$E_{tot|N2=0} = \phi_T\gamma_T,\tag{9}$$

$$\begin{aligned}\phi_T\gamma_T &\geq t_{init}+t_{c3}+N_1(t_{c1})+N_2(t_{c2})\\ N_2 &\leq \frac{\phi_T\gamma_T-t_{c3}-N_1(t_{c1})}{t_{c2}}.\end{aligned}\tag{10}$$

As the minimum value for N_1 is 0, this leads to the bound for N_2 as shown below:

$$\begin{aligned}N_2 &\leq \frac{\phi_T\gamma_T-t_{c3}}{t_{c2}}\\ &= \frac{\gamma_T-t_{c3}}{t_{c2}}\\ &with\\ T &= \frac{t_{init}+t_{c3}+N_2(t_{c2})}{(1+N_2)}.\end{aligned}\tag{11}$$

5.4 Extending to N-Level Checkpointing

Extension from a two-level to an N-level checkpointing scheme requires additional consideration as the complexity of the problem increases. In order to look at the expected time, a few additional parameters must be defined. Specifically, n is the number of checkpoint levels. $N_1, N_2, \ldots N_n$ are the numbers of checkpoints of each level between those one level higher. Note that this means there will be a total of N_n checkpoints that can recover n failures, $(N_n+1)\cdot N_{n-1}$ that can recover n-1 failures and so on. t_i is the recovery time for a scheme that can recover up to i failures, with t_0 being the size of an interval (previously T). t_{ci} is the time required to set up a single checkpoint of a

scheme that can recover from i failures. Equation (12) shows the expected time of an interval in terms of the previous intervals. We use the L function to estimate the expected time to recover to the current interval given that a specified number of failures occur. The H function is an expression of the probability of having at least a specific number of failures in an interval.

$$\begin{aligned} &E(k-1,k;n) \\ &\quad = e^{\lambda t_0}\left(\sum_{i=1}^{n-1} L(i+1,k;n)H(i)e^{-\lambda t_{i+1}}\right) \\ &\qquad + e^{\lambda t_0}\left(\sum_{i=1}^{k-1} E(j-1,j)H(n)\right) \\ &\qquad + e^{\lambda t_0}\left(\left(t_{n+1}+\frac{n+1}{\lambda}\right)H(n)\right) \\ &\qquad + e^{\lambda t_0}\left(\frac{1}{\lambda}\cdot\sum_{i=0}^{n-1}(i+1)(e^{-\lambda t_{i+1}})H(i)\right). \end{aligned} \tag{12}$$

The H function is defined in (13) and represents the probability of having a specific number of failures within an interval. H is a function of the number of failures as well as the times for recovery which are nonrandom parameters of the system:

$$H(x) = \prod_{j=0}^{x}(1-e^{-\lambda t_j}). \tag{13}$$

The L function represents the time it takes to get back to an interval given that a failure checkpoint needed to be used. This equation sums the expected values of the intervals since a checkpoint capable of recovering the failures occurred.

$$L(x,k;n) = \sum_{j=Z(x,k)+1}^{k-1} E(j-1,j;n). \tag{14}$$

The Z function calculates at what interval the last checkpoint for a given number of failures, x, occurred from a given interval, k. The Z function is nonrandom.

$$Z(x,k) = \left\lfloor \frac{k-1}{\prod_{j=1}^{x-1} N_j + 1} \right\rfloor \cdot \prod_{j=1}^{x-1} N_j + 1. \tag{15}$$

Equation (16) shows the length of the interval assuming all intervals are even.

$$t_0 = \frac{T_{init} + t_{c.(n+1)} + \sum_{i=1}^{n} t_{c.i}\cdot N_i \cdot \prod_{j=1}^{i-1}(N_j+1)}{\sum_{i=1}^{n} N_i \cdot \prod_{j=1}^{i-1}(N_j+1)}. \tag{16}$$

The calculation of the total number of intervals is defined as:

$$K = \prod_{1}^{n}(N_i+1). \tag{17}$$

The total execution time for a program of length T_{init} with specified n, $N_1..N_n$,$t_1..t_{n+1}$, $t_{c.1}..t_{c.(n+1)}$ can be formed by plugging in t_0 from (16), $K$from (17), and the expected values into:

$$E(1,K;n) = \sum_{i=1}^{K} E(i-1,i;n). \tag{18}$$

This provides a recursive definition that can be solved numerically if the expected time for the first interval is calculated. The closed form allows a potential user of a layered diskless system using this checkpoint schedule to determine its expected effectiveness. We next verify this formulation using simulations. We then use this closed form to demonstrate the improvement of layered diskless checkpointing over 1-level diskless checkpointing.

6 Experimental Results

6.1 Verifying Analytic Model By Simulation

To verify the results of the analysis, a simulation is used to evaluate its accuracy. We chose to evaluate using a simulation approach to verify the analysis is correct for the assumptions. Future work will include fault injection studies to more closely match real-world systems. The simulation calculates the expected value of each interval in order from first interval to last interval. Each interval is run a number of times. The simulated times for completion of the interval are averaged to estimate the expected completion time. This estimate is then used as the expected time for that interval for all subsequent intervals in the simulation. After all intervals are simulated, the expected times for the intervals are summed to give the total expected simulated time. For a more detailed description of the simulation strategy, see Section 6.2. The simulated program has a faultless, uncheckpointed execution time of 10 days.

The simulation starts with determining near-optimal numbers of checkpoints by level using the analytical formula. The process of this selection is discussed in further detail later. After the checkpoint schedule is determined, the performance of that level of checkpointing is examined. Initially, the total simulation time to complete is determined by taking the uncheckpointed time and adding the time required for each of the checkpoints. Then, the total simulation time is split into intervals that consist of the time between any two adjacent checkpoints, including the time to complete the checkpoint at the end of the interval. As this checkpointing scheme often requires reseting to previous intervals when one or more failures occurs, the variance of the execution time increases greatly. To counteract this effect in the estimation of expected value, an interval by interval estimation scheme is used. Specifically, each interval, starting with the first interval, has its expected time estimated. After the interval is estimated, if the interval is needed again for a later interval, the simulation will use the expected execution time previously calculated rather than resimulate the interval. While this functions well for calculating the expected time, it should be noted that this simulation as written cannot immediately estimate the variance of the total execution time.

For each interval, a sample of an exponential distribution is taken according to the λ parameter. The sampled value indicates whether the interval is passed (if the sample is larger than the interval length), or at what time a failure occurred during the interval (if the sample is smaller than the interval length). If no failure occurs, then the runtime for that trial of the interval is the length of the interval. If a failure occurs, the runtime is incremented by the time until

TABLE 1
Parameter Values for Analysis Verification Simulation

Param	Low	High	Increment	Unit
λ	.001	.01	.003	Failures/Hour
t_{c1}	.5	3	.5	Hours
t_{c2}	t_{c1}	3	.5	Hours
t_{c3}	.5	3	.5	Hours
t_{r1}	.5	3	.5	Hours
t_{r2}	t_1+.5	3	.5	Hours
t_{r3}	t_2+.5	3	.5	Hours
N_1	1	5	1	Ckpts/(N_2+1)
N_2	1	5	1	Ckpts

the failure occurred. Another exponential sample is then taken to determine the time of the next failure. If the next failure occurs during the time to recover from the first failure, then the second failure, considered a simultaneous failure, must be handled by the next level of checkpointing. At this point, the expected times for previous intervals are added to estimate the time to return to the beginning of the current interval. Once execution returns to the beginning of the interval, the process begins again except with its runtime reflecting the time penalties that the failure caused. Eventually, the trial of the interval completes and provides a sample of the execution time of that interval. Each interval runs many times, with an average of the runtimes being used as the expected execution time of that interval. For the results shown, each interval is estimated by the average of 1,000 executions of the interval. Once all the intervals are estimated, the total expected execution time is determined by summing the expected values of the intervals.

The parameters for simulation were set up according to Table 1. In this table, t_{cx} is the time to set up a checkpoint of level x and t_{rx} is the time to recover from a failure of x simultaneous failures (e.g., t_x in the analysis). $N1$ and $N2$ are the number of one-failure and two-failure checkpoints used, respectively. The simulation was run four separate times with the number of interval repetitions being increased from 5,000 to 20,000 at 5,000 step increments. In each case, each result was compared with the analytic result in terms of percentage difference relative to the simulation time. The maximum percentage difference was then taken for each of the four numbers of repetitions. This result is charted in Fig. 6, showing that the simulation is converging toward the analysis.

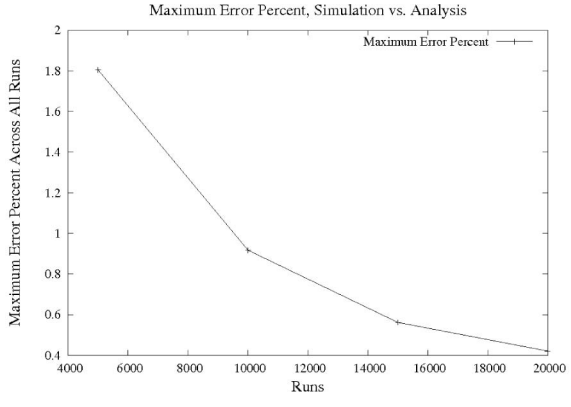

Fig. 6. Maximum difference (in percent of analysis) observed between analysis and simulation over parameters described in table 1.

TABLE 2
Parameter Values for Comparison of
1-Level, 2-Level, and 3-Level Schemes

Param	Values	Unit
λ	[1/24 , 1/12, 1/6, 1, 2, 4]	Failures/Hour
t_c	[1,10]	Minutes
t_{cN}	$t_{c1} \cdot N$	Minutes
t_r	[1,10]	Minutes
t_{rN}	$t_{r1} \cdot N$	Minutes
$FaultlessRuntime$	24	Hours

6.2 N-Level versus 1-Level Diskless Checkpointing

To demonstrate the improvement of N-level checkpointing relative to 1-level checkpointing, both the analytical formula and experimental simulation are compared on varying parameters of lambda, times to recover a one-level failure, and time to set up a one-level checkpoint. A 1-level scheme, a 2-level scheme, and a 3-level scheme are compared. The checkpoints in the 1-level scheme are capable of recovering from a single simultaneous failure. The 2-level scheme has two types of checkpoints, being able to recover from only one simultaneous failure, and up to two simultaneous failures, respectively. The 3-level scheme has the types of checkpoints in the 2-level scheme with an additional type of checkpoint capable of recovering from three failures.

Table 2 shows the parameter values used for comparing the three schemes. The simulated program has a faultless, uncheckpointed execution time of one day. We assume that the time for a two-failure recovery is twice the time for a one-failure recovery. We assume that the time for a three-failure recovery is three times the time for a one-failure recovery. These values are based on the work in [7]. Similarly, the setup time for a two-failure checkpoint is assumed to be twice the setup time for a one-failure checkpoint. For a three-failure checkpoint, the checkpoint setup time is three times the setup time for a one-failure checkpoint. In general, we assume that the time for an N failure recovery time is N times that of recovering from one failure, and that setting up an Nth-level checkpoint is the same as N times that of a one-failure checkpoint. According to these assumptions, establishing the time to recover from a one failure and the setup time of a one-failure checkpoint is sufficient to specify recovery times and checkpoint setup times for any number of failures and any level of checkpoint.

For the simulation, a near-optimal number of checkpoints were determined through empirical search for each level. In the case of one-level checkpointing, the optimal was found by increasing the number of checkpoints until the expected performance became worse with additional checkpoints as determined by the analytical formula. The multilevel checkpoints (two, and three) require finding the near-optimal number of one-failure checkpoints, two-failure checkpoints, and three-failure checkpoints. Unfortunately, finding the optimal number of checkpoints for more than one level is computationally costly. The near-optimal multilevel checkpoints were determined, also using the analytical formula, by trying combinations of numbers of checkpoints on each level up to the number of checkpoints determined to be optimal for the one-level checkpoint while only trying a small number of lower level checkpoints. For some parameter values, the near-optimal number of

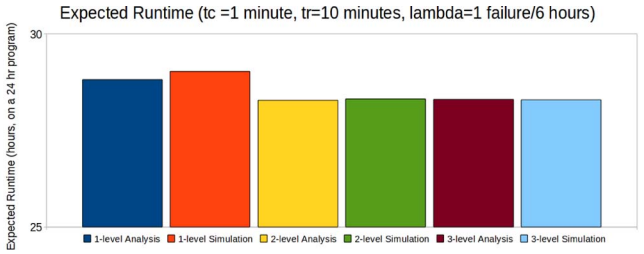

Fig. 7. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 1$ min, $\mathrm{tr} = 10$ min, and $\lambda = 1$ failure/6 hours.

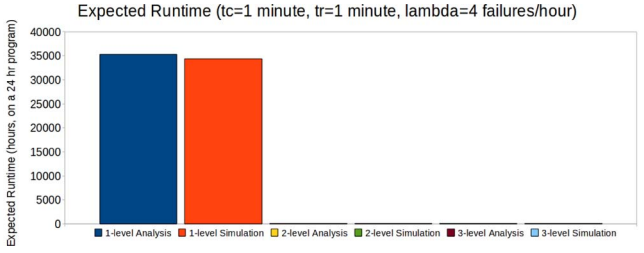

Fig. 8. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 1$ min, $\mathrm{tr} = 1$ min, and $\lambda = 4$ failures/hour.

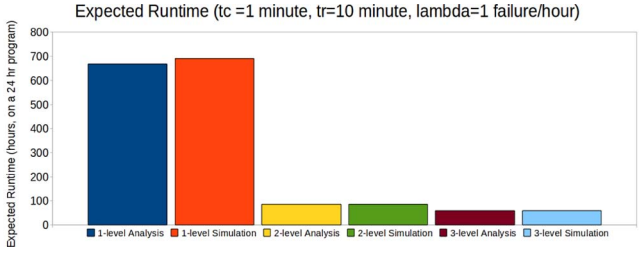

Fig. 9. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 1$ min, $\mathrm{tr} = 10$ min, and $\lambda = 1$ failure/hour.

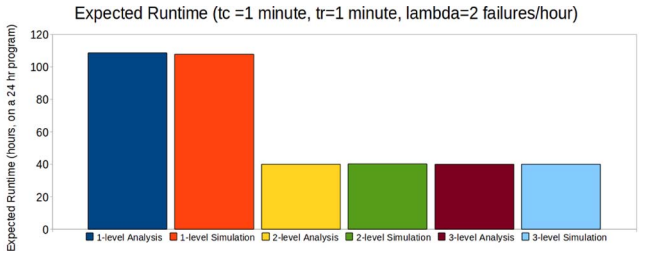

Fig. 10. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 1$ min, $\mathrm{tr} = 1$ min, and $\lambda = 2$ failures/hour.

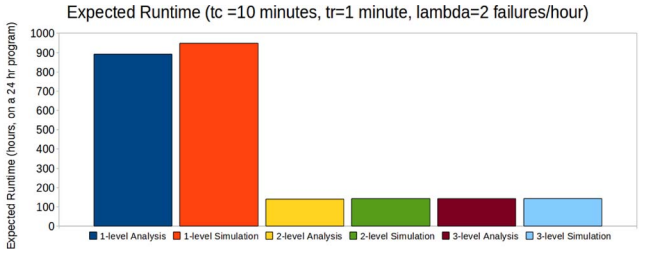

Fig. 11. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 10$ min, $\mathrm{tr} = 1$ min, and $\lambda = 2$ failures/hour.

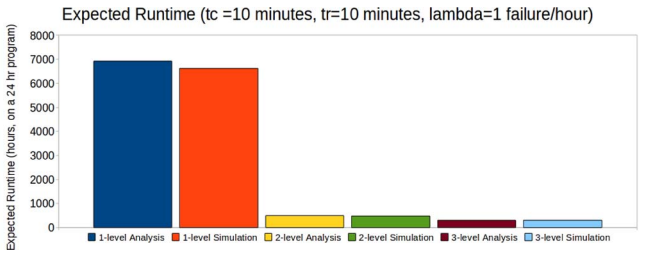

Fig. 12. Comparison of analysis and simulation expected runtimes for the 1-level, 2-level, and 3-level schemes with $\mathrm{tc} = 10$ min, $\mathrm{tr} = 10$ min, and $\lambda = 1$ failure/hour.

checkpoints for the 2-level and 3-level schemes only consisted of one-failure checkpoints. Under these circumstances the three schemes are effectively the same (Fig. 7 compares the performance of a case where the 2-level and 3-level have the same set of checkpoints).

The simulation results are compared with the analytical results. Figs. 7, 8, 9, 10, 11, and 12 show the results for several of the runs with the equivalent analytical times comparing the different schemes. These particular data points demonstrate several different behaviors of the schemes over a variety of tc, tr, and λ values.

In all of these figures, the analytic result tightly matches the simulation result for each of the schemes. Additionally, the results show that there is often a sizable reduction of expected execution time when using 2-level or 3-level schemes in comparison with the 1-level scheme. This reduction can be seen in Fig. 9, and more so in Fig. 8 where the 1-level execution time dwarfs the expected execution time of both 2-level and 3-level schemes to the extent that they barely appear on the chart. The 1-level does not perform as well as the 2-level or 3-level schemes. The intuitive reason for this difference is that the failure rate happens to have a high probability of two failures, but that the probability of an additional failure during the recovery period is low. In other words, there are a sufficient number of 1-failures that the cumulative time of 1-failure recovery periods grows sufficiently and additional failure is likely. However, the number of those recovery periods is not sufficiently high to require a 3-level scheme. As such, the 2-level and 3-level schemes perform similarly. At higher failure rates the 2-level scheme will also degrade at some point. The selection of the number of levels should be based on the failure rate.

The 3-level scheme often provides an additional improvement over even the 2-level scheme, as can be seen in Figs. 9 and 12. However, this is not always the case as can be seen in Figs. 10 and 11, where the 3-level scheme only has two-failure and one-failure checkpoints.

6.3 Sensitivity to Checkpoint Schedule Fidelity

Up to this point, we have assumed that the checkpoints are able to be taken exactly when they are scheduled. Section 6.2 demonstrates potential benefits to multiple levels of diskless checkpointing under that assumption. We now explore how important the assumption of equal intervals is to the analysis. Even in cases where the application or system is able to schedule approximately equal intervals for checkpoints, deviations from the precise schedule are still likely. We therefore explored the sensitivity of the solution to perturbations of the checkpoint schedule. Any particular combination of checkpoint coordination algorithm and application will result in different deviations from a

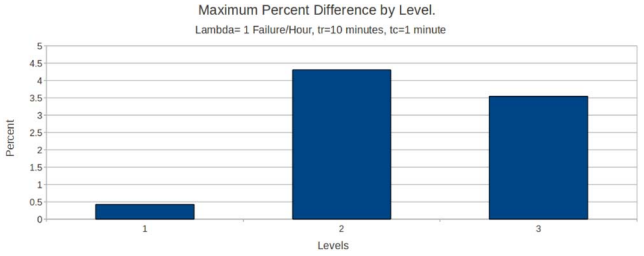

Fig. 13. The impact of inexact checkpoint schedules on the expected program execution time.

checkpoint schedule. We instead pick a particular modification to checkpoint schedules to demonstrate that 1) the analysis does not immediately break with unequal intervals, and 2) that some potential checkpoint coordination behaviors are covered by this analysis. In particular, we explore the impact of a checkpoint coordination system that on average has the desired exact intervals but deviates from each checkpoint according to a normal distribution.

To evaluate this impact, we simulated both the unperturbed and a set of perturbed schedules. The perturbation replaced each checkpoint completion time with a new time selected from a Gaussian distribution centered at the original checkpoint completion time with a standard deviation of 5 percent of an interval.

For the comparison, tests were run on the best checkpoint schedules found through the previous exhaustive search (see Section 6.2). We simulated 1,000 independently selected modified checkpoint schedules. The average of 10,000 complete runtime simulations was calculated for each modified checkpoint schedule (instead of stepping from 5,000 to 20,000 as in Section 6.2). We then computed the *maximum* percent difference between the expected runtimes of the unmodified schedule versus each of the 1,000 modified schedules.

Fig. 13 shows an example sensitivity analysis for the case of $\lambda = 1.0$, $\mathrm{tr} = 10$ minutes, and $\mathrm{tc} = 1$ minute. Fig. 13 indicates that the worst difference of all the trials of optimal numbers of equal interval checkpoints was 5.5 percent ($\lambda = 4.0,\ \mathrm{tr} = 10\ \mathrm{minutes}, \mathrm{tc}\ = 1\ \mathrm{minute}, 3\ \mathrm{levels}$). The average difference was 0.3 percent, which indicates that the impact of the inexact checkpoint schedules on the expected program execution time is often negligible if checkpoint intervals are not too far away from equal.

7 Discussion

The experimental results show that 2-level and 3-level diskless checkpointing outperforms the one-failure checkpointing scheme in many cases, and performs equally well in other cases. Sometimes the various schemes use the same checkpointing schedule. A single level, but multifailure scheme (e.g., only having checkpoints that can recover from up to 2 failures) may perform as well as the 2-level or 3-level schemes for some cases where the one-failure, 1-level scheme performed poorly. In such a case that scheme may not perform as well as the multilevel schemes where they comparatively matched the performance of the one-failure scheme. As such, the use of a multilevel scheme is more useful than any 1-level scheme with the same types of checkpoints available.

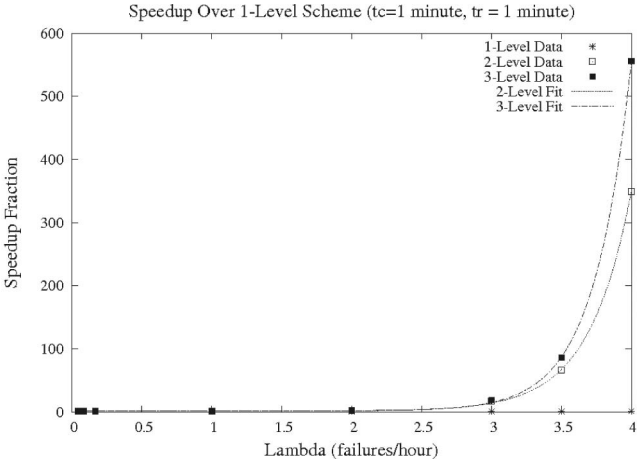

Fig. 14. Speedup fraction of 2-Level and 3-Level schemes over the 1-Level scheme with increasing λ. The tc and tr parameters are held constant at 1 minute.

As can be seen in Fig. 14, increasing the failure rate increases the benefit of multilevel diskless checkpointing. The expected runtime for 1-level checkpointing increases on a faster exponential curve than 2-level or 3-level checkpointing. Additional levels of checkpointing should be considered when the cost of checkpointing and recovery for those levels does not exceed the benefit provided. The cost of checkpointing and recovery for additional levels should be considered based upon the system being investigated. The assumptions about the recovery time and checkpoint setup times (e.g., that the recovery time from N failures is N times the recovery time for one failure) may or may not hold in real-world systems, and as such these times should be evaluated based on the system. Fortunately, the analysis technique presented in this paper is independent of these assumptions and is still valid regardless of the checkpoint setup and failure recovery times. Of course, changing these parameters would change the results of the evaluation.

The current model assumes that failures are independent; however, this is not likely in real systems. One way to address this is through a sample implementation in the system to verify the expected performance approximately matches the true performance. The weakness of this approach is that the degree to which failures are correlated depends on the software, hardware, and configuration of the system. Another way to address this would be to study failure patterns of the real-world system and attempt to form a failure dependency graph. Upon creation of this graph, along with the failure probabilities that are derivable from this paper, it would be possible to ascertain whether it is sufficiently independent to use the model of this paper using the Lovasz local lemma. Upon verifying that the failures are sufficiently independent, the results of this paper could be used to determine what kind of diskless checkpointing method should be used for the system.

Coordination of checkpoints is a critical premise of this work. The more obvious case where this method will be effective is iterative methods (e.g., Conjugate Gradient method, Generalized Minimal Residual method, and Biconjugate Gradient Stabilized method) where each iteration takes approximately the same amount of time and a

consistent checkpoint can be obtained automatically at the end of any iteration. While the number of iterations may not result in an exact match of the equal interval schedule, the simulations in Section 6.3 indicate that the analysis is still useful when the intervals are approximately equal. In applications where checkpoint coordination is more difficult, the use of roughly equal interval checkpoints combined with forced checkpoints to ensure consistency is an option. This approach would require additional overhead for forced checkpoints that is dependent on the application. The analysis shown here would apply if the additional overhead can be factored into the total execution time in a fashion that preserves roughly equal checkpoint intervals. Ultimately, in some cases the ability to construct approximately equal checkpoint intervals may be infeasible. Layered diskless checkpoints may still be useful; however, the analysis shown here is not applicable. A separate analysis should be considered for such cases.

8 Conclusion

We provide a method for deriving a pure analytic model for the expected time to complete a program using two-level diskless checkpointing, as well as an analytic model that can be used in conjunction with numerical methods for determining the expected time to complete a program for a generic N-level diskless checkpointing system. Additionally, we develop a simulation framework for N-level diskless checkpointing, and compare the analysis and simulation. Furthermore, we suggest a method to search for near-optimal numbers of checkpoints and levels that provide a reasonable starting point when exhaustive search is too expensive. The results of the model indicate that N-level diskless checkpointing is a highly promising system technique for improving the expected runtimes of high-performance computing programs for a number of system configurations in comparison with current methods. With increased processors and subsequent increased failure rates, this method will help improve expected execution times.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions to improve the quality of this paper. This research is partly supported by the US National Science Foundation, under grants #CNS-1304969, #CCF-1305622, and #OCI-1305624.

References

[1] P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, "Distributed Multi-Level Recovery in Main-Memory Databases," *Proc. Fourth Int'l Conf. Parallel and Distributed Information Systems,* pp. 44-55, 1996.

[2] Z. Chen and J. Dongarra, "Algorithm-Based Fault Tolerance for Fail-Stop Failures," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 12, pp. 1628-1641, Dec. 2008.

[3] Z. Chen, "Optimal Real Number Codes for Fault Tolerant Matrix Operations," *Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC '09),* pp. 14-20, Nov. 2009.

[4] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing," *Proc. 25th ACM Int'l Conf. Supercomputing (ICS '11),* May-June 2011.

[5] D. Hakkarinen and Z. Chen, "Algorithmic Cholesky Factorization Fault Recovery," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '10),* pp. 19-23, Apr. 2010.

[6] Z. Chen, "Algorithm-Based Recovery for Iterative Methods without Checkpointing," *Proc. 20th ACM Int'l Symp. High-Performance Parallel and Distributed Computing (HPDC '11),* pp. 8-11, June 2011.

[7] Z. Chen and J. Dongarra, "Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing," *IEEE Trans. Computers,* vol. 58, no. 11, pp. 1512-1524, Nov. 2009.

[8] Z. Chen, G.E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault Tolerant High Performance Computing by a Coding Approach," *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '05),* pp. 213-223, 2005.

[9] C. Engelmann and A. Geist, "A Diskless Checkpointing Algorithm for Super-Scale Architectures Applied to the Fast Fourier Transform," *Proc. First Int'l Workshop Challenges of Large Applications in Distributed Environments (CLADE '03),* p. 47, 2003.

[10] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* vol. 33, no. 6, pp. 518-528, June 1984.

[11] Y. Kim, J.S. Plank, and J.J. Dongarra, "Fault Tolerant Matrix Operations for Networks of Workstations Using Multiple Checkpointing," *Proc. High-Performance Computing on the Information Superhighway (HPC-Asia '97),* p. 460, 1997.

[12] J. Plank, K. Li, and M. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 10, pp. 972-986, Oct. 1998.

[13] J.S Plank, Y. Kim, and J. Dongarra, "Fault Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 125-138, June 1997.

[14] J.S. Plank and K. Li, "Faster Checkpointing with n + 1 Parity," technical report, Knoxville, TN, 1993.

[15] J.S. Plank, J. Xu, and R.H.B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," Technical Report CS-95-302, Univ. of Tennessee, Aug. 1995.

[16] Y. Shang, Y. Jin, and B. Wu, "Fault-Tolerant Mechanism of the Distributed Cluster Computers," *Tsinghua Science and Technology,* vol. 12, Supplement 1, pp. 186-191, 2007.

[17] Y. Shang, B. Wu, T. Li, and S. Fang, "Fault-Tolerant Technique in the Cluster Computation of the Digital Watershed Model," *Tsinghua Science and Technology,* vol. 12, Supplement 1, pp. 162-168, 2007.

[18] L. Silva and J. Silva, "Using Two-Level Stable Storage for Efficient Checkpointing," *IEE Proc. Software,* vol. 145, no. 6, pp. 198-202, 1998.

[19] L.M Silva and J.G Silva, "An Experimental Study about Diskless Checkpointing," *Proc. EUROMICRO Conf.,* vol. 1, pp. 395-402, 1998.

[20] N.H Vaidya, "Another Two-Level Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency," technical report, 1994.

[21] N.H Vaidya, "A Case for Two-Level Distributed Recovery Schemes," *Proc. ACM SIGMETRICS Joint Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 64-73, 1995.

[22] A. Ziv, "Analysis and Performance Optimization of Checkpointing Schemes with Task Duplication," PhD thesis, Stanford, CA, 1996.

[23] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated Application-Level Checkpointing of MPI Programs," *ACM SIGPLAN Notices,* vol. 38, no. 10, pp. 84-94, June 2003.

[24] J.S Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Software—Practice and Experience,* vol. 27, no. 9, pp. 995-1012, Sept. 1997.

[25] E.N Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys,* vol. 34, no. 3, pp. 375-408, Sept. 2002.

[26] J.F Chiu and G. Chiu, "Placing Forced Checkpoints in Distributed Real-Time Embedded Systems," *Computing and Control Eng. J.,* vol. 13, no. 4, pp. 197-205, Aug. 2002.

[27] K. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, Feb. 1985.

[28] T. Lai and T. Yang, "On Distributed Snapshots," *Information Processing Letters,* vol. 24, no. 3, pp. 153-158, 1987.

[29] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems,* pp. 39-47, Oct. 1992.
[30] G. Zheng, L. Shi, and L.V. Kale, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," *Proc. IEEE Int'l Conf. Cluster Computing,* pp. 93-103, Sept. 2004.

Doug Hakkarinen received the BS degree in computer science and the BA degree in molecular, cellular, developmental biology from the University of Colorado in 2003. He received the MS degree in computer science from the Colorado School of Mines in 2009. He is currently working toward the PhD degree in computer science from the Colorado School of Mines. His research interests include high performance computing, fault tolerance and reliability, optimization of parallel algorithms for geophysics, and computational science and engineering. He is a student member of the IEEE.

Zizhong Chen received his BS degree in mathematics from Beijing Normal University, China, in 1997, and his MS and PhD degrees in computer science from the University of Tennessee, Knoxville, in 2003 and 2006, respectively. He is currently an assistant professor of computer science at the University of California, Riverside. His research interests include high performance computing, fault tolerance and checkpointing, power-aware algorithms and software, real number error/erasure correcting codes, numerical linear algebra algorithms and software, and computational science and engineering. During the past six years, he has taught more than 20 classes for over 10 different courses, reviewed papers for more than 15 journals, and served as a member of technical program committee for more than 20 international conferences and workshops. He received the Best Paper Award from the 19th International Supercomputer Conference in 2004, the Distinguished Research Award from Jacksonville State University in 2008, the Outstanding Faculty Award from the Colorado School of Mines in 2010, and the CAREER Award from National Science Foundation in 2012. He is a senior member of the IEEE.