Detailed Modeling and Evaluation of a Scalable Multilevel Checkpointing System

Kathryn Mohror, Adam Moody, Greg Bronevetsky, and Bronis R. de Supinski
Lawrence Livermore National Laboratory
{kathryn, moody20, bronevetsky, bronis}@llnl.gov

Abstract—High-performance computing (HPC) systems are growing more powerful by utilizing more components. As the system mean time before failure correspondingly drops, applications must checkpoint frequently to make progress. However, at scale, the cost of checkpointing becomes prohibitive. A solution to this problem is multilevel checkpointing, which employs multiple types of checkpoints in a single run. Lightweight checkpoints can handle the most common failure modes, while more expensive checkpoints can handle severe failures. We designed a multilevel checkpointing library, the Scalable Checkpoint/Restart (SCR) library, that writes lightweight checkpoints to node-local storage in addition to the parallel file system. We present probabilistic Markov models of SCR's performance. We show that on future large-scale systems, SCR can lead to a gain in machine efficiency of up to 35%, and reduce the load on the parallel file system by a factor of two. Additionally, we predict that checkpoint scavenging, or only writing checkpoints to the parallel file system on application termination, can reduce the load on the parallel file system by 20× on today's systems and still maintain high application efficiency.

1 Introduction

Although supercomputing systems use high quality components, they become less reliable at larger scales because increased component counts increase overall fault rates. HPC applications can encounter mean times between failures (MTBFs) of hours or days due to hardware breakdowns [1] and soft errors [2]. For example, the 100,000 node BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) experiences an L1 parity error every 8 hours [3] and a hard failure every 7-10 days. Exascale systems are projected to fail on the order of minutes or hours [4], [5], [6]. Most applications tolerate failures by periodically saving their state to reliable storage *checkpoint* files. Upon failure, an application can restart from a prior state by reading in a checkpoint.

Checkpointing to a parallel file system is expensive at large scale. A single checkpoint can take tens of minutes [7], [8]. Further, large-scale computational capabilities have increased more quickly than I/O bandwidths. Typically, the limited bandwidth results from system design choices that optimize for system maintainability and availability.

Increasing failure rates due to increases in system scale require more frequent checkpoints. Increased system imbalance makes them more expensive. So, checkpointing is both more critical and less practical. Thus, large-scale applications require either more efficient checkpoint mechanisms or alternatives such as process replication, which have overheads over 100% [9].

Multilevel checkpointing [10], [11] uses multiple types of checkpoints that have different levels of resiliency and cost in a single application run to address this problem. The slowest but most resilient level writes to the parallel file system, which can withstand an entire system failure. Faster but less resilient levels use node-local storage, such as RAM, Flash or disk, and apply cross-node redundancy schemes. Most failures only disable one or two nodes, and multinode failures often disable nodes in a predictable pattern [12]. Thus, an application can usually recover from a less resilient checkpoint level, given carefully chosen redundancy schemes. Multilevel checkpointing allows applications to take frequent inexpensive checkpoints and less frequent, more resilient checkpoints, resulting in better efficiency and reduced load on the parallel file system.

We evaluate multilevel checkpointing in large-scale systems through a probabilistic Markov model. Our major contributions over our prior work [12] are:

- Details of our Markov model of multilevel checkpointing;
- An extension of our model for checkpointing to the parallel file system only upon job termination (checkpoint scavenging);
- An evaluation of the viability of checkpoint scavenging.

. Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-JRNL-564721-DRAFT.

Overall, our results demonstrate that multilevel checkpointing significantly improves current methods. We show that it can increase system efficiency significantly, with gains up to 35% while reducing the load on the parallel file system by a factor of two.

The rest of this paper is organized as follows. Section 2 presents related work. In Section 3, we describe SCR, our Scalable Checkpoint/Restart library. Section 4 details our multilevel checkpoint model, Section 5 uses it to evaluate multilevel checkpointing on current and future systems. In Section 6, we extend our model to study the possibility of only writing checkpoints to the parallel file system when absolutely necessary.

2 Related Work

Many models describe checkpoint systems [13], [14], [15], [16]. However, few have modeled multilevel checkpointing. Vaidya developed a Markov model for a two-level checkpoint system [17]. We extend Vaidya's model to an arbitrary number of levels, each with its own checkpoint and recovery costs and failure rate. Our model also allows for sequential failures within a given computation interval. Panda and Das extended Vaidya's model to predict task completion probability. They assume a fixed number of spare resources and no repair [18]. In our model, we assume the system has an infinite pool of spare resources through repair of failed ones. Gelenbe presented a Markov model for multilevel checkpointing [10]. He derived a formula for system efficiency from the Markov model steady state equations. However, he noted that an analytical solution for the optimum efficiency was intractable. Instead, we derive expressions for efficiency using a recursive method.

Checkpoint-on-failure (CoF) [19] is similar to checkpoint scavenging. CoF only writes checkpoints after a failure occurs. CoF is suitable for applications that implement algorithmic based fault tolerance (ABFT), and can restart from checkpoints without data from failed processes. In contrast, our approach works for algorithms without ABFT and applies redundancy schemes for node-local checkpoints.

Researchers have combined checkpointing methods to lower overheads while maintaining resiliency [20], [21], [22]. However, to the best of our knowledge, SCR was the first implementation of multilevel checkpointing on large-scale, production systems.

3 The SCR Library

SCR enables MPI applications to use node-local storage to attain high checkpoint and restart I/O bandwidth [12]. We derive its approach from two key observations. First, a job only needs its most recent checkpoint. As soon as it writes the next checkpoint, we can discard the previous one. Second, a typical failure only disables a small portion of the system.

Our SCR design leverages these observations by caching checkpoint files in storage local to the compute nodes instead of the parallel file system. SCR caches only the most recent checkpoints, discarding an older checkpoint with each newly saved one. SCR can apply a redundancy scheme to the cache, so it can recover checkpoints after a failure disables some of the system. SCR periodically copies (flushes) a cached checkpoint to the parallel file system in order to protect against wider failures. However, a well-chosen redundancy scheme allows infrequent flushing of checkpoints.

The `pF3D` laser-plasma interaction code [23] has used SCR since late 2007. We base our multilevel checkpoint model on SCR's implementation.

4 Multilevel Checkpoint Model

Our novel probabilistic model of multilevel checkpointing can predict the behavior of SCR given the factors that can affect its performance. We evaluate SCR's performance using the model with parameters that represent SCR usage by `pF3D`. This model can guide general use of multilevel checkpoint systems for current and future systems and motivate system designs that provide adequate overall reliability and efficiency. To model multilevel checkpointing systems, we make some simplifying assumptions that naturally introduce errors into the model's predictions. However, these errors are relatively small. We now discuss our assumptions and their potential impact.

We assume that failures are independent. Thus, a failure within a job does not increase the probability of another failure within that job or future jobs. In reality, some failures are correlated. However, SCR is designed to mitigate effects of correlated failures. For example, it can avoid using failed nodes in a job allocation as those nodes may be likely to fail again.

We assume that checkpoints are globally coordinated and taken at regular intervals throughout the job. While not always true, SCR requires globally coordinated checkpoints and `pF3D` does checkpoint at regular intervals. We also assume costs to read and write checkpoints are constant throughout the job. However, read and write times actually vary, particularly when shared resources such as the parallel file system are used, which leads to some error in our model. We assume that the application recovers from the most recent viable checkpoint when a failure occurs. We do not model possible savings from using an older checkpoint that is also sufficient for recovery but available from faster storage. Thus, we may underestimate the possible performance of multilevel checkpointing.

We assume an infinite pool of spare nodes. When using SCR in practice, users often request extra nodes in their job allocation; upon node failure SCR restarts the job using the extra nodes, ignoring those that failed. In general, the failure rate is less than the repair rate, so this assumption typically holds. In the absence of spare nodes, SCR copies the most recent checkpoint to the parallel file system and terminates the job; we model this capability in Section 6. Similarly, the model does not account for batch system allocation time limits. We assume a single level L checkpoint period completes within the allocation time limit. In practice, SCR handles batch limits by copying the most recent checkpoint to the parallel file system before the allocation expires. These assumptions cause our model to overestimate performance if SCR must copy checkpoints to the parallel file system.

4.1 Model Overview

In a multilevel checkpointing system, each of L checkpointing mechanisms is a *level*, for which level 1 checkpoints are the least expensive and resilient, while level L checkpoints are the most expensive and resilient. In our model, we assume that a checkpoint at level k can be used to recover from a superset of the failure modes that are recoverable using checkpoints at levels less than k. A *level k failure* refers to a failure severe enough that we require a checkpoint at level $i \geq k$ for recovery. A *level k recovery* restores an application using a checkpoint saved at level k. A multilevel checkpointing system alternates between different types of checkpoints. Since more severe failures happen less frequently, the system records zero or more level k checkpoints for every level $k+1$ checkpoint.

In our Markov model of a multilevel checkpointing system, nodes represent application states and edges represent the transitions between states. We annotate each edge with the probability that the application will transition from the source state to the destination state and with cost information such as the time spent in the source state given that the transition is taken. Our model has *computation* and *recovery* states. Computation states represent periods of application computation followed by a checkpoint. Recovery states represent the process of restoring an application from a checkpoint saved previously.

Figure 1 presents our model's basic structure. The white states in the top row are computation states, and the single blue state at the bottom is a recovery state. We label each computation state by the checkpoint level with which it terminates and the recovery state by the checkpoint level that it uses to restore the application. If no failures occur during application execution or checkpointing, the application transitions from one computation state to the next. If a failure occurs, the application transitions to the recovery state corresponding to the most recent checkpoint capable of recovering from the failure. For example, if a failure at level i and $i \leq k$ while in the middle computation state in Figure 1, the system transitions to recovery state k, which restores the application using the checkpoint that was written at the end of the previous computation state. However, if $i \geq k$, the system must transition to a recovery state that corresponds to an older checkpoint saved at a higher level.

If no failures occur during recovery, the application transitions to the computation state that follows the checkpoint used for recovery. If a failure at level $i < k$ occurs while in a level k recovery, we assume the current recovery state must be restarted. However, if $i \geq k$, the application must transition to a higher-level recovery state. We assume a level L recovery can be restarted to recover after a failure at any level.

We exploit the recursive structure of our model to develop recurrence equations that we efficiently solve for the expected run time. As Figure 2 shows, we can build a full model by recursively composing three basic blocks, which we label $X(k,c)$, $Y(k,c)$, and $Z(k,c)$,

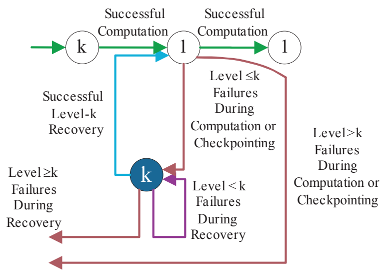
Successful Computation
Successful Computation
k
1
1
Successful Level-k Recovery
Level ≤k Failures During Computation or Checkpointing
Level>k Failures During Computation or Checkpointing
Level ≥k Failures During Recovery
k
Level < k Failures During Recovery

Fig. 1. Basic structure of multilevel Markov model

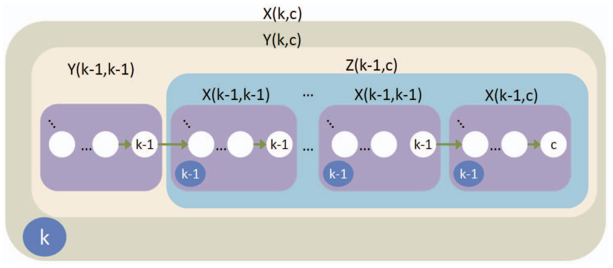

Fig. 2. Hierarchical structure of Markov model

Symbol	Definition
L	Number of checkpoint levels modeled
v_k	Number of level k checkpoints within each level $k+1$ period
t	Length of compute interval before the application initiates a checkpoint
c_k	Time to record a level k checkpoint
r_k	Time to complete a level k recovery
λ_k	Average rate of level k failures assuming Poisson distributions

TABLE 1
Model parameters

where $k, c \in 1, 2, \cdots, L$. An $X(k,c)$ block consists of a $Y(k,c)$ block and a base state for recovery at level k, R_k. A $Z(k,c)$ block consists of a series of $X(k,k)$ blocks and a terminating $X(k,c)$ block. When $k > 1$, a $Y(k,c)$ block consists of either a single $Y(k-1,c)$ block or a $Y(k-1,k-1)$ block followed by a $Z(k-1,c)$ block. Finally, when $k = 1$, a $Y(k = 1, c)$ block is a base state corresponding to a computation state that terminates with a checkpoint at level c. The parameter c is the checkpoint level taken by the last compute state in a block and k is the level of a block. An instance of $X(L,L)$ represents a level L interval.

We use the definitions listed in Table 1 to parameterize our multilevel checkpoint model. The supplemental document for this paper includes complete derivations of the results in Sections 4.2 to 4.5. In the interest of space, we only report a high level description here.

4.2 Base States

For the base computation and recovery states, p_0 is the probability that the application executes for some time, t_0, without encountering a failure. For $k \in 1, 2, \cdots, L$, the probability that the first failure during this period occurs at level k is p_k and t_k is the expected run time before encountering that failure. With T representing the time for spent in the state before exiting, and assuming an exponential distribution, the expressions for $p_0(T)$ and $t_0(T)$ evaluate to $p_0(T) = e^{-\lambda T}$ and $t_0(T) = T$, and for $k \in 1, 2, \cdots, L$, $p_k(T)$ and $t_k(T)$ evaluate to

$$p_k(T) = \frac{\lambda_k}{\lambda}(1 - e^{-\lambda T}), \quad t_k(T) = \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})},$$

where $\lambda = \lambda_1 + \lambda_2 + \cdots + \lambda_L$.

A $Y(k = 1, c)$ block is a base computation state in which the application executes for an interval of length t and then writes a checkpoint at level c, which requires a time of c_c. From the formulas above, $p_{Y0} = p_0(t + c_c)$ and $t_{Y0} = t_0(t + c_c)$, and for $i \in 1, 2, \cdots, L$, $p_{Yi} = p_i(t + c_c)$, and when $p_{Yi} > 0$, $t_{Yi} = t_i(t + c_c)$.

While in a recovery base state at level k, the system is recovering from a failure using a checkpoint saved at level k, which requires a time of r_k. We find that the probability of exiting with no failures is $p_{R0} = p_0(r_k)$ and the time to exit with no failures is $t_{R0} = t_0(r_k)$. For $i \in 1, 2, \cdots, L$, the probability of exiting on a failure at level i is $p_{Ri} = p_i(r_k)$, and when $p_{Ri} > 0$, the time before exiting on failure at level i is $t_{Ri} = t_i(r_k)$.

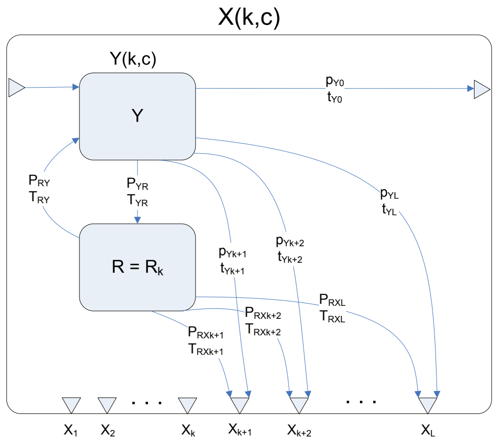

Fig. 3. Simplified diagram of $X(k,c)$

4.3 The $X(k,c)$ block

An $X = X(k,c)$ block internally consists of a $Y = Y(k,c)$ block and a recovery state at level k, $R = R_k$. To simplify the final expressions, we merge groups of related transitions into single transitions. We show the merged transitions in Figure 3.

Y transitions to the recovery state R for any failure scenario that requires a recovery level at k or less. We merge each of these transitions into a single transition that has probability of P_{YR} and an expected run time of T_{YR}. Once in R, a transition away from R eventually happens, provided that $\sum_{i=0}^{k} p_{Ri} < 1$. However, one or more loops back to R may occur before transitioning away. We merge the transitions from R to Y; its probability is P_{RY} and its expected run time is T_{RY} as shown in Figure 3.

Failures at levels $i \geq k$ cause a transistion out of R_k to a higher level recovery state. The transitions from R have probability P_{RX_i} and expected run time T_{RX_i}. However, if $k = L$, then recovery is restarted upon failure at any level so for each $i \in 1, 2, \cdots, L$, $P_{RX_i} = 0$. While in a recovery state at level $k < L$, the system transitions to a recovery state at level $k+1$ if a level k or level $k+1$ failure occurs. Otherwise, for the occurrence

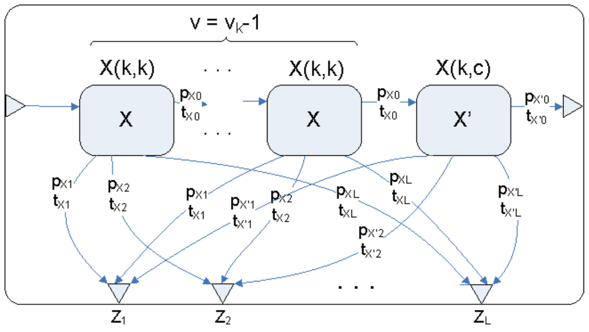

Fig. 4. The $Z(k, c)$ State

of a failure at level i, where $i > k + 1$, a transition is made to a recovery state at level i.

4.4 The $Z(k, c)$ block

A $Z = Z(k, c)$ block only exists when $v_k > 0$. It consists of a chain of $X = X(k, k)$ blocks of length $v_k - 1$ followed by a $X' = X(k, c)$ block, as Figure 4 shows. We define $v = v_k - 1$.

The probability of successfully transitioning from the Z block to the first computation state of the next block is the probability that v consecutive successful transitions from X blocks are followed by one successful transition from the X' block, $p_{Z0} = (p_{X0})^v \cdot p_{X'0}$. When $p_{Z0} > 0$, the expected time to make this transition is $t_{Z0} = v \cdot t_{X0} + t_{X'0}$. The total probability to leave Z for a recovery state at level i is the sum of the probabilities corresponding to each of the possible paths from the substates of Z.

4.5 The $Y(k, c)$ block

A $Y(k, c)$ block is built using three different constructions depending on the values of k and (when $k > 1$) v_{k-1}. If $k = 1$, then $Y(k, c) = Y(k = 1, c)$, which is a base computation state. The probability and expected run time vectors for this state can be directly computed as described in Section 4.2.

If $k > 1$ and $v_{k-1} = 0$, then $Y = Y(k, c)$ consists of a single $Y' = Y(k - 1, c)$ block. We compute the probabilities and expected run times to transition from Y given the probabilities and expected run times to transition from Y' as $p_{Y0} = p_{Y'0}$ and $t_{Y0} = t_{Y'0}$, and, for each level $i \in 1, 2, \cdots, L$, $p_{Yi} = p_{Y'i}$ and $t_{Yi} = t_{Y'i}$.

If $k > 1$ and $v_{k-1} > 0$, then $Y = Y(k, c)$ consists of a starting $Y' = Y(k - 1, k - 1)$ block followed by a $Z = Z(k-1, c)$ block, as Figure 5 shows. The probability that a successful transition from Y occurs is the probability that both Y' and Z transition successfully, $p_{Y0} = p_{Y'0} \cdot p_{Z0}$ and the expected time for this transition is $t_{Y0} = t_{Y'0} + t_{Z0}$. The total probability to leave Y for a recovery state at level i is the sum of the probabilities for each path.

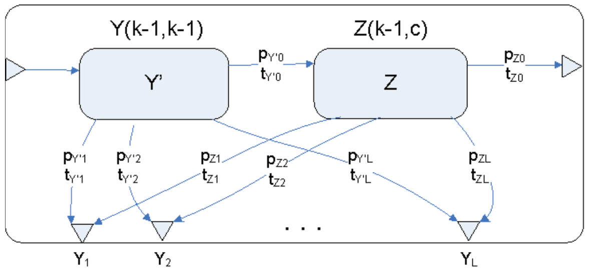

Fig. 5. The $Y(k, c)$ State for $k > 1$ and $v_{k-1} > 0$

4.6 Model Metrics

We consider two key metrics: *efficiency* and *parallel file system load*. We define *efficiency* as the ratio of *idealTime* to *expectedTime*, where $idealTime$ is the minimum run time assuming the application spends no time checkpointing and encounters no failures, while $expectedTime$ is the expected run time that the model predicts for a set of parameters. This metric indicates how much time is lost to checkpointing, including recovery from failures.

To compute efficiency, we parameterize the model with a set of checkpoint levels including their checkpoint and recovery costs, failure rates, and time between checkpoints. We then compute the expected time to complete a level L period. The value of t_{X0} for the $X(L, L)$ state is the $expectedTime$ to complete a level L period. The $idealTime$ is the total number of compute intervals multiplied by the length of each interval.

To judge the impact on the parallel file system for a particular model configuration, we consider the expected time between writing consecutive checkpoints to the parallel file system. We define the *load* on the parallel file system to be the inverse of $expectedTime$.

5 Model Exploration

We used our model to explore the behavior of SCR under varying conditions. We show the benefits of multilevel over single-level checkpointing as failure rates and parallel file system characteristics change.

We show predictions of `pF3D` efficiency to those observed in real runs on Coastal and Atlas in Table 2. The data show that the model's predictions are within a few percent of our observations. Despite limitations due to the significant time required to gather data, these results demonstrate that our model is accurate.

System	Expected Efficiency	Observed Efficiency	Duration of Observation
Coastal	95.2%	94.68%	716,613 node-hours
Atlas	96.7%	92.39%	553,829 node-hours

TABLE 2
Expected and observed efficiency

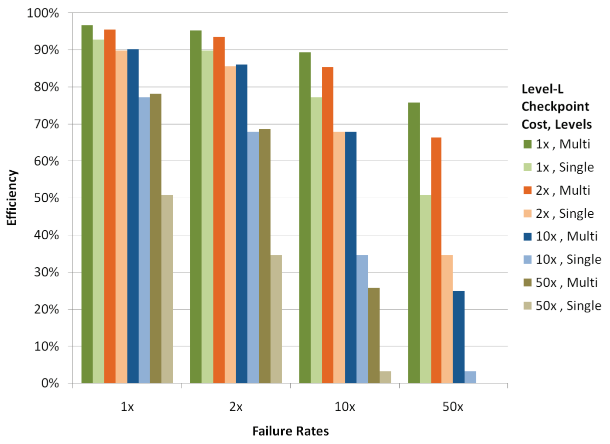

Fig. 6. Optimal efficiency for single- and multilevel checkpointing

We now use the model to explore multilevel checkpointing in a more general context. For checkpoint costs, we use the times observed for checkpointing `pF3D` on Coastal using three different checkpointing levels, with costs of 0.5 seconds, 4.5 seconds, and 1052 seconds, with recovery costs equal to checkpoint costs. Using collected failure data for `pF3D` on Coastal, we use rates in failures per job-second of $2 \cdot 10^{-7}$ for level 1, $1.8 \cdot 10^{-6}$ for level 2, and $4 \cdot 10^{-7}$ for level 3.

As future systems become larger, failure rates are expected to increase, and as the system memory size grows faster than the performance of the parallel file system, the cost of accessing the parallel file system is expected to increase. To explore these effects, we increase the base failure rates and the level L checkpoint costs by factors of 2, 10, and 50. We do not adjust the costs of lower-level checkpoints, since the performance of node-local storage is expected to scale with system size. For each combination, we identified the compute interval and the level 1 and level 2 checkpoint counts that provide the highest efficiency. We compare this to single-level checkpointing, for systems with only a parallel file system available.

Figure 6 presents the efficiency achieved for each configuration. We label the results for the multilevel system as "Multi" and those for the single-level system as "Single." The groupings of bars along the x-axis correspond to failure rates that are 1, 2, 10, or 50 times the base values. Within each grouping, we increase the cost of the level L checkpoint by 1, 2, 10, and 50 times the base value.

In all cases, the multilevel system results in higher efficiencies and increases the time between checkpoints to the parallel file system. Moreover, both advantages increase with either increasing failure rates or higher parallel file system costs. The gain in machine efficiency ranges from a few percent up to 35%, and, although not shown here, the load on the parallel file system is reduced by a factor ranging from 2x-4x. Thus, compared to single-level checkpointing, multilevel checkpointing simultaneously increases efficiency while reducing load on the parallel file system. These results highlight the benefits of multilevel checkpointing on current and future systems.

Overall, we find that multilevel checkpointing is essential for future systems. Even with systems that are $50\times$ less reliable, a three level checkpointing system achieves efficiencies over 75%, as long as we maintain relative parallel file system performance. On the other hand, we find that we cannot tolerate higher failure rates if the cost to access the parallel file system also increases. In particular, if systems become $50\times$ less reliable and the cost of saving application state to the parallel file system rises by $10\times$, a three level checkpointing system only achieves 26% efficiency.

6 Scavenging of Last Checkpoint

We now extend our model to the scenario in which we only write level L checkpoints upon job termination, and not periodically during the run. This scenario could be advantageous on systems that have high costs for level L checkpoints. Also, some systems do not support application restart on its current allocation when a failure occurs; instead, the job must be restarted in a new allocation. In this case, upon application termination, a multilevel checkpoint system should write the last complete checkpoint to the parallel file system. This approach only incurs the high overhead of level L checkpoints when strictly necessary. However, this carries the risk of the level L checkpoint failing because the lower-level checkpoints were corrupted.

We refer to the process of pushing a checkpoint set to the parallel file system upon allocation termination as *scavenging a checkpoint*. In our model, scavenging only occurs on application failure, and a successfully terminating job that experiences no failures during execution writes its last checkpoint to the parallel file system for use in a subsequent allocation. Level $k < L$ checkpoints are taken at regular intervals during the application run and cached on the compute nodes.

We make several assumptions in our model. In the event of any failure, the application does not restart; instead, the scavenge process begins. On a level k failure, we attempt to scavenge the most recent checkpoint at level $i \geq k$. If a failure occurs at level $i \geq k$ while scavenging, we attempt to scavenge the most recent checkpoint at level j or greater, where $j > i$. If a level L failure occurs while scavenging, the scavenge operation fails. When the application is restarted in a new allocation, it must roll back to the last level L checkpoint taken before the prior allocation. Finally, we estimate the time to scavenge the level k checkpoint as the time for a level L checkpoint. In reality, we may incur a small additional overhead to rebuild the level k checkpoint depending upon the failure mode.

We evaluate the effectiveness of scavenging by computing the expected efficiency of the job, as defined in Section 4.6. However, we compute it differently, because the job will terminate on any failure and not attempt restarts. We define *efficiency* as the ratio of the expected amount of work done by the application before its expected termination time either on success, w_{X0}, on scavenge, w_{XS}, or on failure without a successful scavenge, w_{Xi}. In our model, the work done by the application is the checkpoint interval, t, multiplied by the number of intervals completed successfully; work excludes the time for any checkpointing activities. The expected termination time is the time to exit on success, t_{X0}, on scavenge, t_{XS}, or on failure without successful scavenge, t_{Xi}, and includes the work time, level $k \leq L$ checkpointing time, and scavenge time.

$$\textit{efficiency} = \frac{p_{X0} \cdot w_{X0} + p_{XS} \cdot w_{XS} + p_{Xi} \cdot w_{Xi}}{p_{X0} \cdot t_{X0} + p_{XS} \cdot t_{XS} + p_{Xi} \cdot t_{Xi}}$$

We base our revised efficiency computation on the amount of work performed in each state. On failure, if scavenging succeeds, we compute the expected amount of work completed before the failure. If scavenging fails, we compute the expected amount of work completed in failure transitions only. A transition from state on success means that no failures occurred and the amount of work completed is the work accumulated in any substates of the current state.

If a level k failure occurs and the current state contains a level $i \geq k$ checkpoint, we can scavenge a checkpoint. We call this *transitioning on scavenge*. In this case, the amount of work completed is the amount of work accumulated in substates before the failure. A transition on failure occurs if the current state does not contain a level $i \geq k$ checkpoint or a level $i \geq k$ failure occurs while scavenging. On failure transitions at level $i > k$, the amount of work completed in the current state is zero. In the extreme case, if no completed checkpoints can be scavenged and the application must roll back to the last level L checkpoint, the total expected amount of work completed is zero.

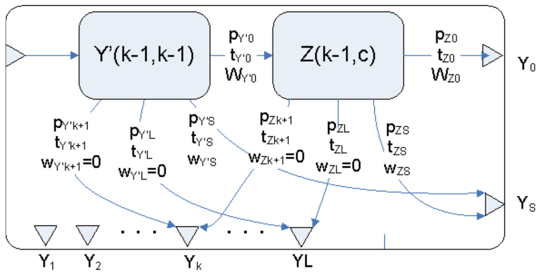
Y'(k-1,k-1)
Z(k-1,c)
$p_{Y'0}$ $t_{Y'0}$ $W_{Y'0}$
p_{Z0} t_{Z0} W_{Z0}
Y_0
$p_{Y'k+1}$ $t_{Y'k+1}$ $w_{Y'k+1}=0$
$p_{Y'L}$ $t_{Y'L}$ $w_{Y'L}=0$
$p_{Y'S}$ $t_{Y'S}$ $w_{Y'S}$
p_{Zk+1} t_{Zk+1} $w_{Zk+1}=0$
p_{ZL} t_{ZL} $w_{ZL}=0$
p_{ZS} t_{ZS} w_{ZS}
Y_S
Y_1
Y_2
Y_k
YL

Fig. 7. The $Y(k,c)$ State for $v_{k-1} > 0$

We make several changes to our model. Recovery states within $X(k,c)$ are now scavenge states. Transitions from each compute state at failure levels $i <= k$ are *scavenge transitions*. Instead of maintaining the probability, expected total time, and expected amount of work at each failure level for scavenge transitions, we compute the overall expected values. Unless otherwise specified, the probabilities and times for exiting states on failure and success are the same as in Sections 4.2 to 4.5.

6.1 Y States

The simplest case is a $Y(k=1,c)$ block, which contains a single compute interval of duration t followed by a level c checkpoint. Upon successful transition from this state, the expected amount of work completed is t. Upon transition from Y on failure for any level $i \in 1, 2, \cdots, L$, $w_{Yi} = 0$ because no work interval and checkpoint completed successfully. We cannot transition from Y on scavenge, because a $Y(k=1,c)$ state does not contain a scavenge state. Thus, on transtioning from Y, $p_S = 0$, $t_S = 0$, and $w_S = 0$.

If $v_{k-1} = 0$ and $k > 1$, Y consists of a single enclosed $Y(k-1,c)$ state, Y'. The work completed for a transition from Y on success or failure is $w_{Y0} = w_{Y'0}$ and $w_{Yi} = w_{Y'i} = 0$. For a scavenge transition, we have $p_{YS} = p_{Y'S}$, $t_{YS} = t_{Y'S}$, and $w_{YS} = w_{Y'S}$.

If $v_{k-1} > 0$ and $k > 1$, Y consists of a $Y' = Y(k-1,k-1)$ state and a $Z = Z(k-1,c)$ state (Figure 7). On successful transition from Y, w_{Y0} is the sum of the work done in Y' and Z, $w_{Y0} = w_{Y'0} + w_{Z0}$. Upon a transition from Y on failure at levels $i \in 1 \cdots L$, $w_{Yi} = 0$. The probability of transitioning on scavenge p_{YS} is the probability that the job exits on scavenge in Y' or completes Y' successfully and exits on scavenge from Z. Similarly, the time and work completed, t_{YS} and

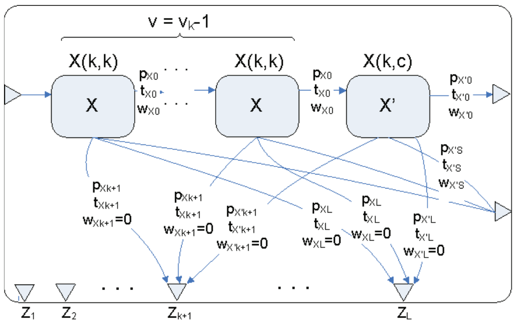

Fig. 8. $Z(k,c)$ State

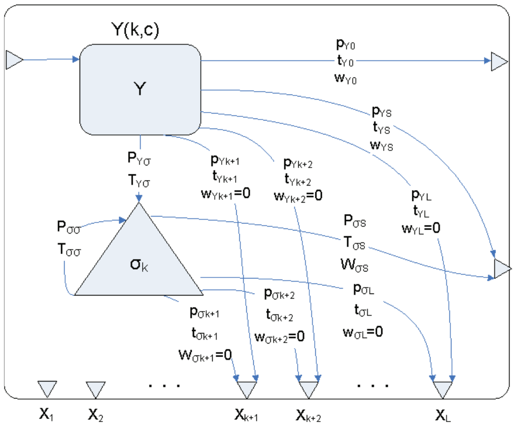

Fig. 9. $X(k,c)$ State

w_{YS} depend on the probabilities of transitioning from Y' and Z on success or scavenge.

6.2 Z State

A $Z = Z(k,c)$ state consists of $v = v_k - 1$ consecutive $X = X(k,k)$ states, followed by a $X' = X(k,c)$ state (Figure 8). The expected amount of work that is performed in Z on successful transition from the state is $w_{Z0} = w_{X0} \cdot v + w_{X'0}$. Upon transition from Z on failure for $i \in 1 \cdots L$, the amount of work completed is $w_{Xi} = 0$.

6.3 X State

An $X = X(k,c)$ state consists of a $Y = Y(k,c)$ state and a scavenge state σ_k. A failure at level $i \leq k$ in Y causes a transition from Y to σ_k. For any failures at levels $i < k$ that occur in σ_k, the scavenge operation restarts. Failures at levels $i \geq k$ in Y or $i \geq k$ in σ_k, cause a transition from X on failure.

The probability and time for a successful transition from X are $p_{X0} = p_{Y0}$ and $t_{X0} = t_{Y0}$. The expected amount of work that is performed in X on successful transition from the state is the amount of work that was performed in Y, $w_{X0} = w_{Y0}$.

A failure at level $i \leq k$ in Y causes a transition to the scavenge state σ_k. The amount of work done on transition from Y to σ_k is $w_{Y\sigma_k} = 0$. As stated previously, we estimate the time for a successful scavenge operation as the time for a level L checkpoint, c_L. Failures at levels $i < k$ restart the scavenge operation. The probability and time before a restart of the scavenge operation are $P_{\sigma\sigma}$ and $T_{\sigma\sigma}$. No work is done in σ_k, so $w_{\sigma i} = 0$.

The probability and time for leaving σ_k on success or failure for $i \in 0, 1, \cdots, L$ depend on the probabilities for exiting σ_k successfully after a number of restarts. The amount of work accomplished is $W_{\sigma i} = w_{\sigma 0}$. However, if $P_{\sigma\sigma} = 1$ then $P_{\sigma i}$, $T_{\sigma i}$, and $W_{\sigma i}$ are zero.

When transitioning from X on failure at level $i = k$, the probability, time, and work only depend on the scavenge state. However, when transitioning from X on failure at level $i > k$, the values depend on the Y and scavenge states.

6.4 Model Predictions of Benefits of Scavenge

We now use our model to predict the performance of scavenging. In our experiments, we use the overhead and failure rates for the Coastal cluster at LLNL, given in Section 5. We first explore the relationship between application efficiency and the compute interval. We vary the failure rates and overhead of writing to the parallel file system by 1×, 2×, and 10× as described previously in Section 5. The trend for today's failure rates is a broad range of compute intervals with near-optimal efficiencies (89%, 95%, and 96%) that level off near 80%, 90%, and 92% as overheads increase by 1×, 2×, and 10× and checkpoint intervals increase up to 10,000 seconds. At failure rates that are 2× greater than today's, the ranges of near-optimal efficiencies are shorter, peak at 83%, 93%, and 95%, and level off at 74%, 83%, and 87% efficiencies for increasing checkpoint overhead and checkpoint interval up to 10,000 seconds. However, at failure rates of 10×, the curves reach lower optimal efficiencies (55%, 81%, and 88%) and drop off more sharply with increasing compute interval to 46%, 67, and 70% efficiencies.

We predict the probability that an application will exit with its last checkpoint written to the parallel file system, either with no failures or on scavenge. For this experiment, we fix the probability of exiting the scavenging operation ($P_{\sigma s}$ in Figure 9) at 80%, based on observed successful scavenge rates with `pF3D`. This is because we observe that failures of the parallel file system are more likely when it has previously failed

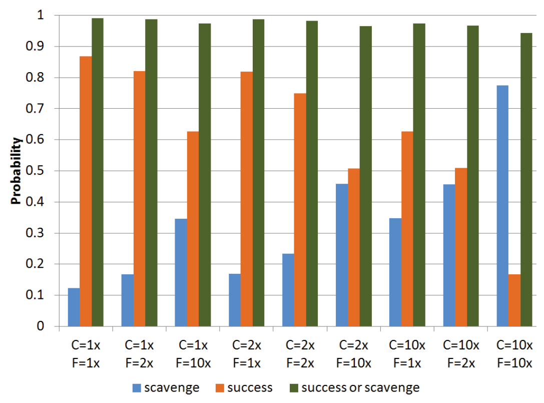

Fig. 10. Application success or scavenge probabilities

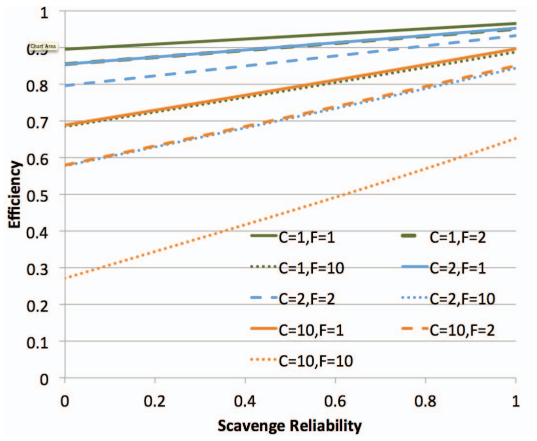

Fig. 11. Reliability of scavenge operation and efficiency

Efficiency	Scavenge or Success	PFS Failure Increase
92.9%	99.8%	1×
92.9%	99.8%	2×
92.7%	99.7%	10×
92.5%	99.4%	100×

TABLE 3
Efficiency of Scavenge from Simulation

during the execution. We show the results for varying parallel file system overhead and failure rates in Figure 10. The probability of exiting with no failures starts at 86% and decreases with both increasing parallel file system and failure rates to a low of 17% when the overhead and failure rates are increased to $10\times$. In contrast, the probabilities of exiting on scavenge increase with increasing overhead and failure rates, from 12% at today's overhead and rates to 77% when they are increased $10\times$. The combined probability of exiting with no failures or on scavenge is 99% on today's systems and falls slightly with increasing overhead and failure rates to a low of 94% when the overhead and rates are at $10\times$. The remaining 1-6% are the probabilities that the application exits without being able to transfer the final checkpoint to the parallel file system.

We use our model to predict the reduction in load on the parallel file system when using scavenging compared to single-level checkpointing. Here, the load is the expected time between writes to the parallel file system. Scavenging greatly reduces the load on today's systems, by $20\times$. As parallel file system overhead increases, the benefit of scavenging decreases; however the lowest benefit in our experiments is still $10\times$ reduction in load. In general, the benefit increases with increasing failure rate, reaching a maximum reduction of $60\times$ for failure rates at ten times today's values.

In Figure 11, we explore the relationship between the reliability of the scavenge operation and application efficiency. We fix the probability of completing the scavenge operation to a given value instead of computing the probability as given in Section 6.3. Generally, job efficiencies are high for systems with today's failure rates, because the probabilities of exiting the job successfully without needing to scavenge are relatively high. However, with increasing failure rates and parallel file system overheads, the reliability of scavenging affects job efficiency more drastically. We find that on future systems with higher failure rates, a highly reliable scavenge operation can increase job efficiency by as much as $2.4\times$.

Because we had not implemented the scavenge operation in SCR at the time we observed the pF3D runs, we verify our model with simulation of 100,000 executions of pF3D on Coastal. Table 3 shows the average application efficiencies and probabilities of getting the last checkpoint to the parallel file system, either by scavenging or by a failure-free execution. The results from the simulation support those from our model. The model and simulation agree that at today's checkpointing costs and failure rates application efficiency will stay close to 90% (Figure 11). They also both predict that the probability of getting the last checkpoint to the parallel file system will remain high, even with increasing parallel file system failures (Figure 10). Please see the supplemental document to this paper for details of the simulation.

Overall, our results show that a scavenge mechanism can dramatically extend the range of systems for which checkpoint/restart remains a viable resilience strategy.

7 Conclusions

We presented models of multilevel checkpointing with SCR, which we validated against results from executions and simulations. Our novel, hierarchical Markov models predict the performance of multilevel checkpointing systems based on system reliability and checkpoint cost. Our analysis demonstrates that multilevel checkpointing significantly improves system efficiency, particularly as failure rates and relative parallel file system checkpoint costs increase. We find that we can still achieve 85% efficiency even if systems become $50\times$ less reliable. Further, multilevel checkpointing simultaneously reduces the load on the parallel file system by more than a factor of two.

We explored the impact of checkpoint scavenging, a key extension to multilevel checkpointing. We found that scavenging results in high efficiencies even when overheads of the parallel file system are increased by $2\times$ and $10\times$. Further, the extremely high probability that the final checkpoint of the job can be transferred to the parallel file system means that work completed is not lost and the job can be restarted in a new allocation. Scavenging also has the benefit of only writing to the parallel file system when absolutely necessary. Our model predicts that scavenging can reduce the load on the parallel file system by as much as $20\times$ on today's systems, and up to $60\times$ on future systems. We also found that a highly reliable scavenge operation can increase job efficiencies by up to 2.4× on systems with higher failure rates and parallel file system overheads.

References

[1] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006, pp. 249–258.

[2] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, September 2005.

[3] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz, "Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, 2007, pp. 1–11.

[4] K. Bergman *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.6676

[5] E. Vivek Sarkar, Ed., *ExaScale Software Study: Software Challenges in Exascale Systems*, 2009.

[6] J. Daly *et al.*, "Inter-Agency Workshop on HPC Resilience at Extreme Scale," February 2012. [Online]. Available: http://institutes.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf

[7] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-Forwarding Infrastructure for Petascale Architectures," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 153–162.

[8] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer, "Parallel I/O on the IBM Blue Gene/L System," Blue Gene/L Consortium Quarterly Newsletter, Tech. Rep., First Quarter, 2006.

[9] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[10] E. Gelenbe, "A Model of Roll-back Recovery with Multiple Checkpoints," in *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, 1976, pp. 251–255.

[11] N. H. Vaidya, "A Case for Multi-Level Distributed Recovery Schemes," Texas A&M University, Tech. Rep. 94-043, May 1994.

[12] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, November 2010, pp. 1 –11.

[13] J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[14] A. Duda, "The Effects of Checkpointing on Program Execution Time," *Information Processing Letters*, vol. 16, no. 5, pp. 221–229, 1983.

[15] J. S. Plank and M. G. Thomason, "Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems," *Journal of Parallel Distributed Computing*, vol. 61, no. 11, pp. 1570–1590, 2001.

[16] J. Daly, "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303 – 312, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-4F490KH-6/2/6ebfa65591e5d0eb09e2ae5ae3b2ed44

[17] N. H. Vaidya, "A Case for Two-Level Distributed Recovery Schemes," in *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, 1995, pp. 64–73.

[18] B. S. Panda and S. K. Das, "Performance Evaluation of a Two Level Error Recovery Scheme for Distributed Systems," in *4th International Workshop on Distributed Computing, Mobile and Wireless Computing (IWDC)*, 2002, pp. 88–97.

[19] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A Checkpoint-on-Failure Protocol for Algorithm-Based Recovery in Standard MPI," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Eds. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 477–488.

[20] L. Silva and J. Silva, "Using Two-Level Stable Storage for Efficient Checkpointing," *IEEE Proceedings - Software*, vol. 145, no. 6, pp. 198–202, Dec 1998.

[21] J. S. Plank and K. Li, "Faster Checkpointing with N+1 Parity," in *Twenty-Fourth International Symposium on Fault-Tolerant Computing (FTCS), Digest of Papers*, Jun 1994, pp. 288 –297.

[22] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*, 2011.

[23] R. L. Berger, C. H. Still, E. A. Williams, and A. B. Langdon, "On the Dominant and Subdominant Behavior of Stimulated Raman and Brillouin Scattering Driven by Nonuniform Laser Beams," *Physics of Plasmas*, vol. 5, p. 4337, 1998.

Kathryn Mohror is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). Kathryn's research on high-end computing systems is currently focused on scalable fault tolerant computing with the Scalable Checkpoint/Restart Library (SCR), a multi-level checkpointing library that has been shown to significantly reduce checkpointing overhead. Her other research interests include scalable automated performance analysis and tuning, parallel file systems, and parallel programming paradigms. She is currently the lead of the Tools Working Group for the MPI Forum.

Kathryn received her Ph.D. in Computer Science in 2010, an M.S. in Computer Science in 2004, and a B.S. in Chemistry in 1999 from Portland State University (PSU) in Portland, OR. Kathryn has been working at LLNL since 2010.

Adam Moody is a Computer Scientist within Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL). His role includes research, development, and support of system software on LC's large-scale supercomputers with a focus on communication and fault tolerance libraries. He has participated in the Message Passing Interface (MPI) Forum since 2008, in which he served as a chapter author for the 2.1 and 2.2 standards. He leads development of the Scalable Checkpoint/Restart (SCR) Library.

Adam attended The Ohio State University where he earned an M.S. in Computer Science in 2003, an M.S. in Electrical Engineering in 2003, a B.S. in Computer Science and Engineering in 2001, and a B.S. in Engineering Physics in 2001. He joined LLNL in 2004.

Greg Bronevetsky is a Computer Scientist at Lawrence Livermore National Laboratory (LLNL). He received a BS from The College of New Jersey in 1999 and a PhD from Cornell University in 2006, both in Computer Science. He has worked at LLNL since 2006, first as a Lawrence post-doctral Fellow and currently as a Department of Energy Early Career Fellow. His research interests include performance modeling and analysis, symbolic compiler analysis, statistical modeling, as well as scalable resilience techniques, including algorithmic resilience.

Bronis R. de Supinski is the Chief Technology Officer (CTO) for Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL). In this role, he is responsible for formulating LLNL's large-scale computing strategy and overseeing its implementation. His position requires frequent interaction with high performance computing (HPC) leaders and he oversees several collaborations with the HPC industry as well as academia.

Prior to becoming CTO for LC, Bronis led several research projects in LLNL's Center for Applied Scientific Computing. Most recently, he led the Exascale Computing Technologies (ExaCT) project and co-led the Advanced Scientific Computing (ASC) program's Application Development Environment and Performance Team (ADEPT). ADEPT is responsible for the development environment, including compilers, tools and run time systems, on LLNL's large-scale systems. ExaCT explored several critical directions related to programming models, algorithms, performance, code correctness and resilience for future large scale systems. He currently continues his interests in these topics, particularly programming models, and serves as the Chair of the OpenMP Language Committee.

Bronis earned his Ph.D. in Computer Science from the University of Virginia in 1998 and he joined CASC in July 1998. In addition to his work with LLNL, Bronis is also a Professor of Exascale Computing at Queen's University of Belfast and an Adjunct Associate Professor in the Department of Computer Science and Engineering at Texas A&M University. Throughout his career, Bronis has won several awards, including the prestigious Gordon Bell Prize in 2005 and 2006, as well as an R&D 100 for his leadership of a team that developed a novel scalable debugging tool. He is a member of the ACM and the IEEE Computer Society.