LLNL-TR-440491

Detailed Modeling, Design, and Evaluation of a Scalable Multi-level Checkpointing System

A. T. Moody, G. Bronevetsky, K. M. Mohror, B. R. de Supinski

July 8, 2010

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Detailed Modeling, Design, and Evaluation of a Scalable Multi-level Checkpointing System

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

July 20, 2010

Abstract

High-performance computing (HPC) systems are growing more powerful by utilizing more hardware components. As the system mean-time-before-failure correspondingly drops, applications must checkpoint more frequently to make progress. However, as the system memory sizes grow faster than the bandwidth to the parallel file system, the cost of checkpointing begins to dominate application run times.

A potential solution to this problem is to use multi-level checkpointing, which employs multiple types of checkpoints with different costs and different levels of resiliency in a single run. The goal is to design light-weight checkpoints to handle the most common failure modes and rely on more expensive checkpoints for less common, but more severe failures. While this approach is theoretically promising, it has not been fully evaluated in a large-scale, production system context.

To this end we have designed a system, called the Scalable Checkpoint/Restart (SCR) library, that writes checkpoints to storage on the compute nodes utilizing RAM, Flash, or disk, in addition to the parallel file system. We present the performance and reliability properties of SCR as well as a probabilistic Markov model that predicts its performance on current and future systems. We show that multi-level checkpointing improves efficiency on existing large-scale systems and that this benefit increases as the system size grows. In particular, we developed low-cost checkpoint schemes that are 100x-1000x faster than the parallel file system and effective against 85% of our system failures. This leads to a gain in machine efficiency of up to 35%, and it reduces the the load on the parallel file system by a factor of two on current and future systems.

1 Introduction

Although supercomputing systems use high-quality components, the systems become less reliable at larger scales because increased component counts increase overall fault rates. Applications running on HPC systems can encounter mean times between failures on the order 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 cache bit error every 8 hours [3] and a hard failure every 7-10 days. Exascale systems are projected to fail every 3-26 minutes [4, 5]. Commonly, applications tolerate failures by periodically saving their state to *checkpoint* files. They write these checkpoints to reliable storage, typically a parallel file system. Upon failure, an application can restart from a prior state by reading in a checkpoint file.

Checkpointing to a parallel file system is expensive at large scale, where a single checkpoint can take on the order of tens of minutes [6, 7]. Further, computational capabilities of large-scale facilities have increased more quickly than I/O bandwidths. For example, the rule of thumb for a well-balanced system is 1 GB/s of I/O bandwidth per 1 TeraFLOP of computational capability [8], but BlueGene/L at LLNL and BlueGene/P at Argonne National Laboratory (ANL) achieve less than a tenth of that rate [6, 7]. Typically, the limited bandwidth to parallel file systems is due to system design choices that optimize for system maintainability and availability as well as the need to share storage across multiple machines.

As computing systems increase in scale, increasing failure rates require more frequent checkpoints, but increased system imbalance makes them more expensive. Checkpointing will become both more critical and less practical.

Thus, PetaFLOP-scale applications will either spend most of their time writing checkpoints or use alternative fault tolerance mechanisms such as process-replication approaches that have overheads of over 100% [9].

Multi-level checkpointing [10, 11] is a promising approach for addressing this problem, which uses multiple types of checkpoints that have different levels of resiliency and cost in a single application run. The slowest but most resilient level writes to the parallel file system, which can withstand a failure of an entire machine. Faster but less resilient checkpoint levels utilize node-local storage, such as RAM, Flash or disk, and apply cross-node redundancy schemes. In our experience, as documented in Section 4, most failures only disable one or two nodes at a time, and multi-node failures often disable nodes in a predictable pattern. Thus, an application can usually recover from a less resilient checkpoint level, given carefully chosen redundancy schemes. Multi-level 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.

In this paper we evaluate the effectiveness of multi-level checkpointing in large-scale high-performance systems by designing and implementing a multi-level checkpointing system and developing a probabilistic Markov model that quantifies its benefits. The major contributions of this paper are:

- the design and implementation of SCR, our multi-level checkpointing system [12];
- a detailed failure analysis for several large systems;
- a Markov model of multi-level checkpointing;
- an exploration of modeled multi-level checkpointing performance on today's and future systems;
- and an empirical evaluation of multi-level checkpointing with a real application.

Overall, our results demonstrate that multi-level checkpointing is a critical augmentation to current fault tolerance methods. We show that SCR can increase system efficiency significantly, with gains of as much as 35% while reducing the load on the parallel file system by a factor of two on current and future systems.

The rest of this paper is organized as follows. Section 2 presents related work. In Section 3, we describe and evaluate SCR. Section 4 describes the types of failures we encounter on several large systems. We detail our model of multi-level checkpointing in Section 5, and in Section 6, we apply it to explore the use of multi-level checkpointing in the context of current and future systems. In Section 7, we apply our model to study the trade-offs associated with multi-level checkpointing on disk-less systems, and we conclude with Section 8.

2 Related Work

Many models exist to describe checkpointing systems [13, 14, 15, 16]. However, few have modeled multi-level checkpointing systems. Vaidya developed a Markov model for a two-level checkpoint system that combines global checkpointing with sender-based message logging [17]. He used this model to predict average checkpointing overhead and concluded that there are conditions where it is beneficial to use multi-level checkpointing. We extend Vaidya's model to account for an arbitrary number of levels, each with its own checkpoint cost, recovery cost, and failure rate, and our model also allows for sequential failures within a given computation interval.

Panda and Das extended Vaidya's two-level model to predict the probability of task completion assuming that the system has a fixed number of spare resources and no repair [18]. In our model, we assume the system has an infinite pool of spare resources. This assumption is valid so long as a job does not use all system resources, and so long as the rate of failure does not outpace the rate of repair.

Gelenbe presented a Markov model for multi-level checkpointing in a transactional computing system [10]. He used the Markov model steady state equations to arrive at a formula providing the efficiency of the system. However, he noted that an analytical solution to find the optimum efficiency was too difficult. Without an analytical solution, one must explore the parameter space numerically. With this in mind, we derive expressions for efficiency using a recursive method. This approach nicely lends itself to dynamic programming, which reduces the time required to explore the parameter space.

Several researchers have worked to lower the overheads of writing checkpoints. Oliner et al. present cooperative checkpointing, which reduces overheads by only writing checkpoints that are predicted to be useful, e.g., when a failure in the near future is likely [19]. Plank et al. eliminate the overhead of writing checkpoints to disk with diskless checkpointing [20]. They keep checkpoint data in memory using mirroring and parity methods for redundancy. Chen et al. present an experimental evaluation of an implementation of diskless checkpointing targeted towards floating point applications on high performance computing systems [21]. Nowoczynski et al. lower the overheads of writing checkpoints by increasing the write bandwidth of the file system [22]. Plank and Li write compressed checkpoints asynchronously to lower overheads [23]. Incremental checkpointing reduces the number of full checkpoints taken by periodically saving changes in the application data between full checkpoints [24, 25, 26]. These approaches are orthogonal to multi-level checkpointing and can be used in combination with our work.

Researchers have combined two checkpointing methods in an effort to lower overheads while maintaining resiliency. Silva and Silva combined disk checkpointing with mirror and parity checkpointing [27]. They concluded that using two levels of checkpoints was advantageous. Plank and Li combine incremental and parity checkpointing in a diskless checkpointing approach [28].

To the best of our knowledge, we provide the first implementation and evaluation of multi-level checkpointing on large-scale, production systems. We also provide real failure data from three production systems that indicate that multi-level checkpointing on these systems has potential. Finally, our new performance model enables one to analyze and optimize a multi-level checkpointing system to improve utilization of current and future systems.

3 The Scalable Checkpoint/Restart (SCR) Library

3.1 SCR Description

The Scalable Checkpoint/Restart (SCR) library enables MPI applications to use storage distributed on a system's compute nodes to attain high checkpoint and restart I/O bandwidth. LLNL has used our multi-level checkpoint system since late 2007 with RAM disk on Linux/x86-64/Infiniband clusters. Production runs currently also use solid-state drives (SSDs) on the same cluster architecture. SCR is available under a BSD license [12].

We derive SCR's 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 checkpoint. Second, a typical failure disables a small portion of the system, but it otherwise leaves most of the system intact. As we discuss in Section 4, 85% of failures disable at most one compute node on the clusters on which we currently use SCR.

Our SCR design leverages these two properties 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 checkpoint. SCR can also apply a redundancy scheme to the cache, so it can recover checkpoints after a failure disables a small portion of the system. SCR periodically copies (flushes) a cached checkpoint to the parallel file system in order to withstand failures that disable larger portions of the system. However, a well-chosen redundancy scheme allows checkpoints to be flushed infrequently.

SCR relies on an external service (e.g., the resource manager or MPI library) to cancel a job that experiences a failure. It also requires the resource manager to allow subsequent jobs to be run within the allocation despite the failure. After a failure, SCR attempts to recover the most recent checkpoint from cache. Assuming it does, SCR can either copy the checkpoint to the parallel file system and stop the job, or it can restart the job directly from the cached checkpoint. In this paper, we focus on the latter option, which assumes a job's resource allocation has sufficient nodes to continue. If SCR fails to recover a checkpoint from cache, it restarts the job after it fetches the most recent checkpoint from the parallel file system.

When requesting an allocation from the resource manager, jobs can request extra nodes to serve as spares in case any nodes fail. In practice, we find that failures occur infrequently enough that one or two spare nodes per thousand active nodes is sufficient to cover a job duration between 12 and 24 hours. Longer runs provide sufficient time to repair nodes and return them to the allocation. Thus, we assume an infinite pool of spare nodes.

SCR is designed for globally-coordinated checkpoints that are written primarily as a file per MPI process. Before an application initiates a new checkpoint, it queries the library to determine the directory path it should use to write each checkpoint file. At the end of the checkpoint, the application notifies the library that it has completed writing its files, and then SCR applies a redundancy scheme to those files.

The path that SCR specifies and the redundancy scheme that SCR applies can be different with each checkpoint.

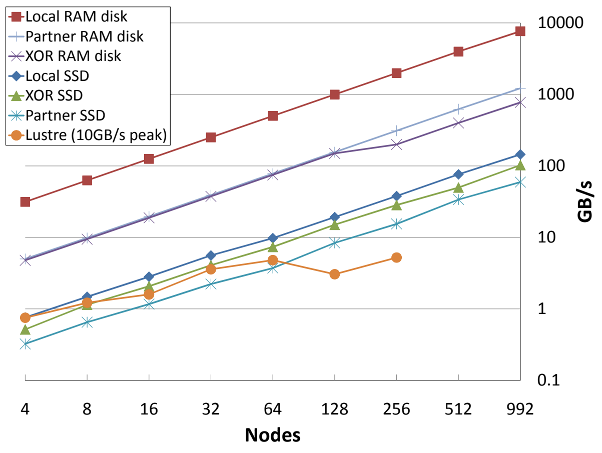


Figure 1: Aggregate write bandwidth to RAM disk, SSD, and Lustre

The path could point to any node-local storage device available on a compute node. For example, it could specify a directory on a RAM disk, a magnetic hard-drive, or an SSD depending on what is available. Additionally, SCR supports three redundancy schemes: `LOCAL`, `PARTNER`, and `XOR`. With `LOCAL`, SCR writes each file to storage local to the node; it does not store redundancy data. With `PARTNER`, it writes each file to local storage and to storage on another *partner* node. SCR can recover a copy of the file, provided a node and its partner do not simultaneously fail. With `XOR`, SCR writes each file to local storage and small sets of nodes collectively compute and store parity redundancy data, similar to RAID-5 [29, 30]. This scheme withstands node failures so long as two or more nodes from the same set do not fail at the same time.

3.2 SCR Performance

To illustrate the performance of SCR, we ran a benchmark to measure the checkpointing bandwidth on the Coastal cluster at LLNL. Coastal consists of 1,152 nodes of which 1,104 are compute nodes. Each compute node has two quad-core 2.4GHz Intel Xeon E5530 processors, 24GB of main memory, and a 32GB Intel X-25E SSD. DDR Infiniband connects the nodes.

Our benchmark calls the SCR library and writes one file per process per checkpoint. It takes several checkpoints, computes the average cost, and reports the aggregate bandwidth. We measured `LOCAL`, `PARTNER`, and `XOR` using both RAM disk and SSDs. We ran the benchmark with 8 MPI ranks per node and scaled up to 992 nodes. We ran a similar bandwidth test using the parallel file system, which is a Lustre system designed to deliver a peak bandwidth of 10GB/s. Figure 1 shows the results.

SCR is clearly scalable, with its mechanisms all following a linear trend. Lustre also scales linearly up to 64 nodes, at which point it levels off near its peak performance of 10GB/s. We observe three general groupings of SCR curves. Including the parallel file system as a fourth, SCR provides up to four checkpoint cost levels on Coastal.

The SCR node-local mechanisms substantially outperform the parallel file system as more nodes are used. SCR's slowest node-local mechanisms use the SSDs, which reach 100GB/s at 992 nodes. This is still 10 times faster than the parallel file system. When using `PARTNER` or `XOR` with RAM disk, SCR reaches 1,000GB/s, which is 100 times faster. And finally, `LOCAL` with RAM disk achieves 10,000GB/s – 1,000 times faster than Lustre.

`PARTNER` performs as well as or better than `XOR` when using RAM disk, however, it is slower with SSDs. When using SSDs, the bottleneck is the SSD write speed. In this case, `XOR` is faster because it writes one full copy plus a small fraction of each file, whereas `PARTNER` writes two full copies of each file. When using RAM disk, the bottleneck is the speed of the network connection. In this case, `PARTNER` and `XOR` are similar, because they both require the same amount of data to be transferred between nodes. `XOR` with RAM disk scales linearly until 256 nodes, where network contention reduces its performance by a constant rate. `PARTNER` uses a different communication pattern that happens to avoid this contention. Network topology could be considered when assigning nodes to `XOR` sets in

Table 1: pF3D checkpoint performance

Cluster Nodes Data	PFS name time BW	Cache type time BW	Speedup
Hera 256 nodes 2.07 TB	lscratchc 300 s 7.1 GB/s	XOR on RAM disk 15.4 s 138 GB/s	19x
Atlas 512 nodes 2.06 TB	lscratcha 439 s 4.8 GB/s	XOR on RAM disk 9.1 s 233 GB/s	48x
Coastal 1024 nodes 2.14 TB	lscratchb 1051 s 2.1 GB/s	XOR on RAM disk 4.5 s 483 GB/s	234x
Coastal 1024 nodes 10.27 TB	lscratch4 2500 s 4.2 GB/s	XOR on RAM disk 180.0 s 603 GB/s	14x

Table 2: pF3D failures on three different clusters

Clusters	Coastal	Hera	Atlas	Total
Time span	Oct 09 - Mar 10	Nov 08 - Nov 09	May 08 - Oct 09	
Number of jobs	135	455	281	871
Node hours	2,830,803	1,428,547	1,370,583	5,629,933
Total failures	24	87	80	191
`LOCAL` required	2 (08%)	36 (41%)	21 (26%)	59 (31%)
`PARTNER/XOR` required	18 (75%)	32 (37%)	54 (68%)	104 (54%)
`Lustre` required	4 (17%)	19 (22%)	5 (06%)	28 (15%)

order to avoid contention, but we leave this to future work.

The general performance trends observed with our benchmark also extend to production applications that use SCR. One particular application, the `pF3D` laser-plasma interaction code [31], has used SCR since late 2007 on the Hera, Atlas and Coastal systems at LLNL. Hera and Atlas are architecturally similar to Coastal, except that they have somewhat different node configurations and lack SSDs. Table 1 shows the checkpoint costs and bandwidths achieved by `pF3D` during four different large-scale runs each configured to use two checkpoint levels: one type of node-local cache and the parallel file system. Note that the cached checkpoints increase `pF3D` checkpoint bandwidths by factors ranging from 14 to 234 when compared to the parallel file system.

4 System Reliability

We analyzed the job logs of 871 runs of `pF3D` that aggregate to over 5 million node-hours on three different clusters to obtain its failure rates on these platforms. Table 2 categorizes each failure according to the checkpoint level that was required for recovery. A `LOCAL` checkpoint could handle 31% of the observed failures, while `PARTNER` or `XOR` could handle another 54%. We had to restart using a checkpoint from the parallel file system for only 15% of the observed failures.

The most common type of failure (54%) consisted of node failures tripped by a bad power supply, a failed network interface card, or an unexplained reboot. Each of these failures disabled a single node, except on Coastal, where two nodes share a single power supply. Based on the system architecture, we can configure SCR to avoid assigning two nodes that share a common power supply as partners or members of the same `XOR` set. Thus, `PARTNER` and `XOR` are sufficient to recover from these failures.

Failed writes to the parallel file system were also common (34%). We classify a write failure as temporary (22%)

if the write succeeded in a subsequent job that was restarted within the same resource allocation. Otherwise, we label the failure as persistent (12%). A `LOCAL` checkpoint is sufficient to restart a job after a temporary write failure. Persistent write failures consumed the rest of a job's resource allocation and forced a restart from `Lustre` in another allocation.

We also observed ten job hangs (5%), which is a condition where the job stopped progressing but did not fail. At LLNL, we use an in-house tool, called `io-watchdog`, to detect and cancel such jobs so that a subsequent job can run. After a hanging job has been canceled, it can be restarted from a `LOCAL` checkpoint. We also found that seven jobs failed (4%) due to a floating-point exception or a memory segmentation violation as the result of a transient processor fault. We can also use a `LOCAL` checkpoint to recover from these faults.

A final set of infrequent hardware failures (3%) required restarting from the parallel file system. In two such failures, multiple nodes were disabled when a power breaker on the wall switched off. We could configure `PARTNER` or `XOR` to account for a failed power breaker. However, this event occurs so infrequently that we just rely on the parallel file system to recover from it. Three failures were due to persistently bad hardware such as a bad processor or memory DIMM that caused jobs to hang repeatedly until their resource allocation expired.

Of the failures that require a restart from the parallel file system, persistent write failures are especially expensive. Although the job often has a valid checkpoint cached on the cluster, the persistent write failure prevents SCR from flushing this checkpoint. System design should reduce persistent write failures by either improving the parallel file system reliability or by providing backup parallel file systems. If we allow for temporary write failures but exclude persistent write failures, only 6 of 169 failures (3.6%) would have required a restart from the parallel file system. Thus, we could have restarted 96.4% of the failed jobs from checkpoints cached on the cluster. Nonetheless, we could restart 85% of the failed jobs from cached checkpoints even with the current rate of persistent write failures.

5 Multi-level Checkpoint Model

A multi-level checkpointing system can store the same checkpoint data using several mechanisms, each of which may have a different cost and level of resilience. 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 k or higher for recovery. A *level k recovery* refers to the process of restoring an application using a checkpoint saved at level k. A multi-level checkpointing system alternates between different types of checkpoints to minimize the overall application running time. Since more severe failures happen less frequently, the system records zero or more level k checkpoints for every level $k+1$ checkpoint.

Several factors determine the performance of a multi-level checkpointing system: the length of the compute interval between checkpoints, the checkpoint and recovery costs at each level, and the failure rates. One must minimize checkpoint frequency at each level to reduce overhead. At the same time, one must consider the expected failure frequency of each level, which determines the lost compute time in the event of a failure. Overall, these factors present a complex optimization problem that requires an explicit multi-level checkpoint model to determine the optimal number of levels and checkpoint frequencies.

We provide a novel probabilistic model of multi-level checkpointing that can predict the behavior of SCR given the factors that can affect its performance. This model can guide general use of multi-level checkpoint systems for current and future systems and motivate system designs that provide adequate overall reliability and efficiency. We can use the model to optimize performance of a given multi-level checkpointing implementation on a specific HPC system, as follows. For a given implementation and system, we measure the performance and failure probability of each checkpoint level, as we did for SCR and LLNL clusters in Sections 3 and 4. This data is then provided to the model as input, along with the length of the compute interval, the set of checkpoint levels to be used, and the checkpoint frequency for each level. The model then predicts the overhead of a particular configuration, accounting for delays due to checkpoints, failures, and restarts. By parameterizing the model with different input values, we can identify the configuration that provides the optimal performance on a given system.

5.1 Assumptions and Error

Our model captures multi-level checkpointing systems with some simplifying assumptions. Of course, any assumptions made when designing a model may introduce errors into its predictions. However, we believe that the errors we introduce are relatively small. Here we discuss our assumptions and the potential impact they have.

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 with one another. 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. Also, it accounts for shared components in the system architecture when configuring its redundancy schemes. For example, processes on the same node are not selected to be partner processes.

We assume that checkpoints are taken at regular intervals throughout the job. This may not always be the case. However, the application we study here, `pF3D`, does checkpoint at regular intervals.

We also assume costs to read and write checkpoints are constant throughout the job. Of course, read and write times vary, especially if shared resources such as the network or parallel file system are used. This introduces error into the results, particularly for checkpoints on the parallel file system.

When a failure occurs, we assume the application rolls back to the most recent checkpoint capable of recovering from the failure. We do not model possible savings from using an older checkpoint that is also sufficient for recovery but stored at a lower level. Such a checkpoint requires more work to be re-computed, but its faster recovery time may lead to better overall efficiency. The effect here is that we may underestimate the performance of multi-level checkpointing.

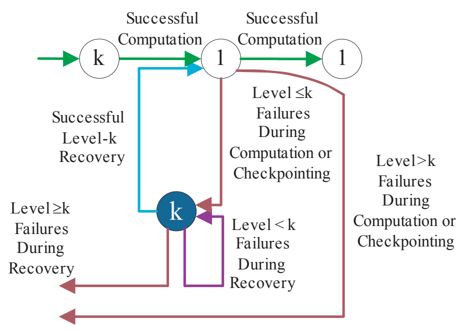
We assume an infinite pool of spare nodes, as discussed in Section 3. When using SCR in practice, users often request extra nodes in their job allocation. Generally speaking, the failure rate is less than the repair rate, so this assumption typically holds. In the case that there are no spare nodes, SCR copies the most recent checkpoint to the parallel file system and terminates the job; however, we do not model this capability. Similarly, the model does not account for allocation time limits that batch systems impose. 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 will lead the model to overestimate performance in cases where SCR is forced to copy checkpoints to the parallel file system when it otherwise would not.

5.2 Model Overview

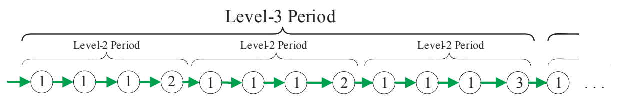
We model a multi-level checkpointing system as a Markov Model (MM). An MM is a directed graph in which nodes represent application states and edges represent the transitions between states. We annotate each edge with the probability that the application will transition from the source state to the destination state. MMs are history-less: they assume transition probabilities only depend on the current state. We also annotate transitions 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 2(a) presents the basic structure of our model. The white states in the top row are computation states, and the single blue state at the bottom is a recovery state. Each computation state is labeled by the checkpoint level it terminates with, while the recovery state is labeled by the checkpoint level it uses to restore the application.

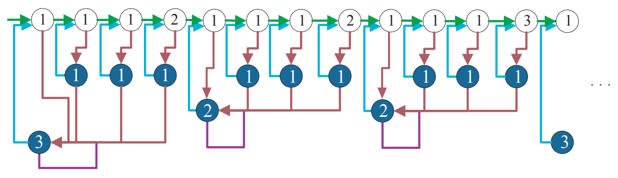
If no failures occur during application execution or checkpointing, the application transitions from one computation state to the next (horizontal transitions between computation states). If a failure occurs, the application transitions to the recovery state corresponding to the most recent checkpoint capable of recovering from the failure (downward transitions from computation states). For example, if a failure at level k or less occurs while in the middle computation state in Figure 2(a), the system transitions to recovery state k, which restores the application using the checkpoint written at the end of the previous computation state. However, if a failure occurs at a level greater than k, the system must transition to a recovery state corresponding to an older checkpoint saved at a higher level.


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

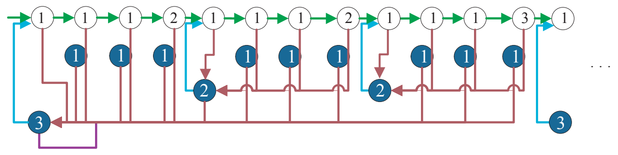
(a) Basic structure of multi-level Markov model



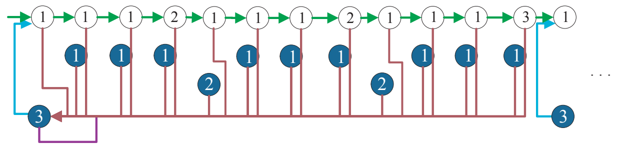
(b) No failures



(c) Level 1 failures and recoveries

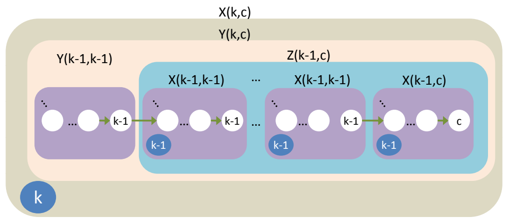


(d) Level 2 failures and recoveries

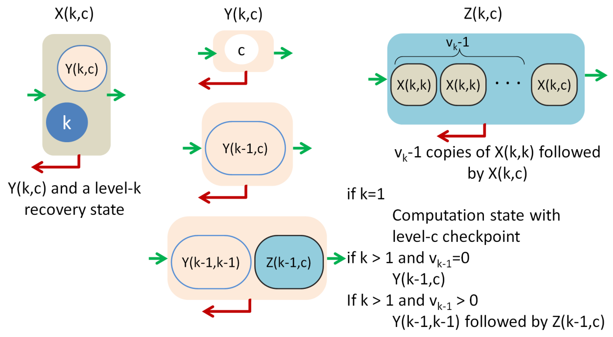


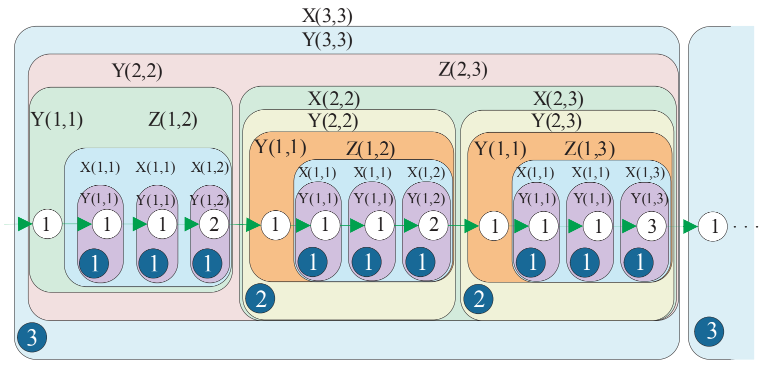
(e) Level 3 failures and recoveries

Figure 2: Structure and example of the multi-Level Markov model



(a) General hierarchical structure


X(k,c)
Y(k,c)
k
Y(k,c) and a level-k recovery state
Y(k,c)
c
Y(k-1,c)
Y(k-1,k-1)
Z(k-1,c)
if k=1
Computation state with level-c checkpoint
if k > 1 and v_{k-1}=0
Y(k-1,c)
If k > 1 and v_{k-1} > 0
Y(k-1,k-1) followed by Z(k-1,c)
Z(k,c)
v_k-1
X(k,k)
X(k,c)
v_k-1 copies of X(k,k) followed by X(k,c)
(b) Structure of basic blocks



(c) Example hierarchical structure

Figure 3: Hierarchical structure of Markov model

If no failures occur during recovery, the application transitions to the computation state that follows the checkpoint used for recovery (upward transition from recovery state). If a failure occurs while in a level k recovery, and if that failure is at a level less than k, we assume the current recovery state must be restarted (loop transition back to recovery state). However, if the failure is at level k or greater, we assume the application must transition to a higher-level recovery state (downward transition from recovery state). We assume a recovery state at level L, the highest level, can be restarted to recover from a failure at any level.

Figures 2(b)–(e) show a full MM for an example three level checkpointing scheme. Figure 2(b) shows the portion that corresponds to failure-free execution, containing only computation states. Checkpoints are taken hierarchically, with three level 1 checkpoints during each level 2 period and two level 2 checkpoints during each level 3 period. Figure 2(c) shows the portion that models the effect of level 1 failures, while Figures 2(d)–(e) show the transitions corresponding to level 2 and level 3 failures respectively.

Our model has a recursive structure, which we take advantage of to develop recurrence equations to efficiently solve for the expected run time. As illustrated in Figure 3(a), a full model can be built by recursively composing three basic blocks, which we label $X(k,c)$, $Y(k,c)$, and $Z(k,c)$, where $k, c \in 1, 2, \cdots, L$. We show the structure of these blocks in Figure 3(b). An $X(k,c)$ block consists of a $Y(k,c)$ block and a recovery state at level 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 denotes the checkpoint level taken by the last computation state in a block. The parameter k denotes the level of a block. For $X(k,c)$ and $Z(k,c)$ blocks in particular, k denotes the checkpoint level that precedes the first computation state in a block. In Figure 3(c), we show how these blocks are composed to describe the example from Figure 2.

Upon transitioning into the first computation state of an $X(k,c)$, $Y(k,c)$, or $Z(k,c)$ block, the system will

Symbol	Definition
L	Number of checkpoint levels being modeled
v_k	Number of level k checkpoints within each level $k+1$ period
t	Length of compute interval before application initiates a checkpoint
c_k	Time to record a level k checkpoint
r_k	Time to complete a level k recovery
$f_k(t)$	Probability of suffering a level k failure at time t (the probability density function)
$F_k(T)$	Probability of suffering a level k failure during the time period $[0..T]$ (the cumulative distribution function: $F_k(T) = \int_0^T f_k(t)\,dt$)
λ_k	Average rate of level k failures assuming Poisson distributions

Table 3: Model parameters

eventually either transition out of the block to the first computation state of the next block, or it will transition to an external recovery state at some level $k \in 1, 2, \cdots, L$. We represent the probabilities and expected run times for each of these transitions in vectors $\vec{\boldsymbol{p}}$ and $\vec{\boldsymbol{t}}$, which we compute for each block. Each vector has $L+1$ elements where the i-th element, for $i \in 0, 1, \cdots, L$, is labeled p_i and t_i, respectively. Element p_0 represents the probability that a transition is made from a block to the first computation state of the next block, and t_0 represents the expected run time spent within the block given such a transition. Element p_k, for $k \in 1, 2, \cdots, L$ represents the probability that a transition is made from a block to an external recovery state due to a failure scenario requiring a recovery at level k, and element t_k represents the expected run time spent in the block given such a transition.

We define $\vec{\boldsymbol{p}}$ and $\vec{\boldsymbol{t}}$ similarly for the base recovery states. In this case, p_0 and t_0 represent the probability and expected run time of completing the recovery process without failure and transitioning to the computation state that follows the checkpoint used for recovery. Element p_k represents the probability of encountering a level k failure while in recovery, and t_k represents the expected run time before encountering such a failure.

Given the constituent components that a block is built from, along with the $\vec{\boldsymbol{p}}$ and $\vec{\boldsymbol{t}}$ vectors for each of those components, we can compute the $\vec{\boldsymbol{p}}$ and $\vec{\boldsymbol{t}}$ vectors for the block. Starting from the base computation and recovery states, we can compute the probability and expected run time vectors for all blocks in a given model structure.

5.3 Model Parameters

We use the definitions listed in Table 3 to parameterize our multi-level checkpoint model. The number of checkpoint levels is represented by L. The number of level k checkpoints taken within each level $k+1$ period is represented by v_k, for $k \in 1, 2, \cdots, L-1$, We assume the application checkpoints (to some level) after completing regular intervals of computation. We represent the length of this compute interval by t. The parameter c_k represents the time required to write a level k checkpoint, and r_k represents the time required to restore an application using a level k checkpoint.

Functions $f_k(t)$ and $F_k(T)$ determine the reliability of the system. The probability that a level k failure occurs at time t is given by $f_k(t)$. The probability that a level k failure occurs at some point during the time period from $t = 0$ to $t = T$ is given by $F_k(T)$. In other words, $f_k(t)$ is the probability density function for level k failures, and $F_k(T)$ is the corresponding cumulative distribution function. In this work, we assume failures at each level follow a Poisson distribution, where λ_k represents the average failure rate at level k.

5.4 Base States

For the base computation and recovery states, p_0 is simply the probability that the application executes for a certain period of time without encountering a failure, and t_0 is just the length of this period. Further, p_k for $k \in 1, 2, \cdots, L$, is simply the probability that a level k failure occurs before any other failure during this period, and t_k is the expected run time before encountering such a failure. Given $f_k(t)$ and $F_k(T)$, we can directly compute the elements of $\vec{\boldsymbol{p}}$ and $\vec{\boldsymbol{t}}$ for the base states.

Since failures at different levels are assumed to be independent, the probability that there are no failures at any level during the time interval $t = 0$ to $t = T$ is given by

$$p_0(T) = (1 - F_1(T)) \cdot (1 - F_2(T)) \cdots (1 - F_L(T)),$$

and the expected run time given that no failures occur during that interval is simply

$$t_0(T) = T.$$

Furthermore, for each level $k \in 1, 2, \cdots, L$, the probability that a level k failure will occur *before* a failure occurs at any other level during the time interval $t = 0$ to $t = T$ is given by

$$p_k(T) = \int_0^T (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt.$$

The integrand above expresses the probability that a level k failure will occur during some infinitesimally small interval of width dt starting at time t,

$$f_k(t)\, dt,$$

multiplied by the probability that a failure at another level has not already occurred by time t,

$$(1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t)).$$

The integral then sums the probabilities of each of these small intervals for all values of t between 0 and T to derive the total probability that a level k failure will occur before a failure occurs at any other level during the full time interval from $t = 0$ to $t = T$.

Similarly, when $p_k(T) > 0$, the expected run time given that a level k failure occurs before a failure occurs at any other level during the time interval $t = 0$ to $t = T$ is given by

$$t_k(T) = \frac{\int_0^T t \cdot (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt}{p_k(T)}.$$

Now, assuming failures at each level follow a Poisson distribution, with failures at level k occurring at an average rate of λ_k, the above expressions for $p_0(T)$, $t_0(T)$, $p_k(T)$, and $t_k(T)$ evaluate to

$$p_0(T) = e^{-\lambda T}, \tag{1}$$
$$t_0(T) = T, \tag{2}$$

and for $k \in 1, 2, \cdots, L$,

$$p_k(T) = \frac{\lambda_k}{\lambda}(1 - e^{-\lambda T}), \tag{3}$$
$$t_k(T) = \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}, \tag{4}$$

where

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

The derivation for the above formulas is provided in Section 9.2.

5.4.1 Computation state: a $Y(k = 1, c)$ block

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. We denote the probability and expected run time vectors for this state as $\vec{\boldsymbol{p_Y}}$ and $\vec{\boldsymbol{t_Y}}$, respectively. Using Formulas 1–4 (after substituting i for k as a subscript label) and setting $T = t + c_c$, we find that

$$p_{Y0} = p_0(t + c_c) = e^{-\lambda\cdot(t+c_c)},$$
$$t_{Y0} = t_0(t + c_c) = t + c_c,$$

and for $i \in 1, 2, \cdots, L$,

$$p_{Yi} = p_i(t + c_c) = \frac{\lambda_i}{\lambda}(1 - e^{-\lambda\cdot(t+c_c)}),$$

and when $p_{Yi} > 0$,

$$t_{Yi} = t_i(t + c_c) = \frac{1 - (\lambda \cdot (t + c_c) + 1) \cdot e^{-\lambda\cdot(t+c_c)}}{\lambda \cdot (1 - e^{-\lambda\cdot(t+c_c)})}.$$

5.4.2 Recovery state at level k

While in a recovery 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 denote the probability and expected run time vectors for this state as $\vec{\boldsymbol{p_R}}$ and $\vec{\boldsymbol{t_R}}$. Using Formulas 1–4 (after substituting i for k as a subscript label) and setting $T = r_k$, we find that

$$p_{R0} = p_0(r_k) = e^{-\lambda\cdot r_k},$$
$$t_{R0} = t_0(r_k) = r_k,$$

and for $i \in 1, 2, \cdots, L$,

$$p_{Ri} = p_i(r_k) = \frac{\lambda_i}{\lambda}(1 - e^{-\lambda\cdot r_k}),$$

and when $p_{Ri} > 0$,

$$t_{Ri} = t_i(r_k) = \frac{1 - (\lambda \cdot r_k + 1) \cdot e^{-\lambda\cdot r_k}}{\lambda \cdot (1 - e^{-\lambda\cdot r_k})}.$$

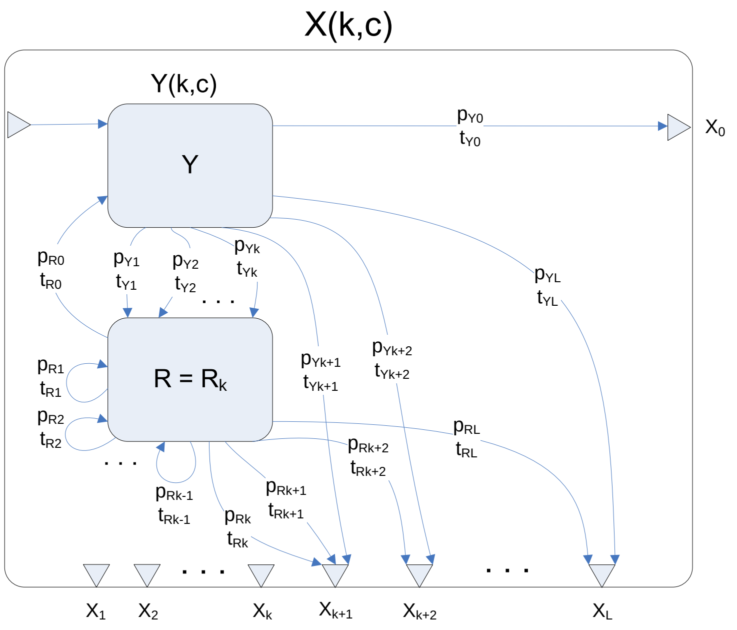


Figure 4: Diagram of $X(k, c)$

5.5 The $X(k, c)$ block

An $X(k, c)$ block internally consists of a $Y(k, c)$ block and a recovery state at level k, which we denote by R_k. Let us refer to these three components as simply X, Y, and R. A diagram of an $X(k, c)$ block is shown in Figure 4. Here we compute the probability and expected run time vectors for X, which we denote as $\vec{\boldsymbol{p_X}}$ and $\vec{\boldsymbol{t_X}}$, assuming that we are given the vectors for Y, as $\vec{\boldsymbol{p_Y}}$ and $\vec{\boldsymbol{t_Y}}$, and R, as $\vec{\boldsymbol{p_R}}$, and $\vec{\boldsymbol{t_R}}$. To simplify the final expressions, we merge groups of related transitions into single transitions. (See Section 9.3 for details on merging transitions.)

First, the Y block has one or more edges to the R state. To be precise, 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 from Y to R into a single transition having probability of P_{YR} and an expected run time of T_{YR} as shown in the top portion of Figure 5. Using formulas 7 and 8 from Section 9.3, we get

$$P_{YR} = \sum_{i=1}^{k} p_{Yi}$$

and, when $P_{YR} > 0$,

$$T_{YR} = \frac{\sum_{i=1}^{k} p_{Yi} \cdot t_{Yi}}{P_{YR}}.$$

Second, the R state has zero or more edges that loop back to itself, as failures at a number of different levels cause the recovery to be restarted. To be precise, a recovery state at the maximum level $k = L$ is restarted upon the occurrence of a failure at any level, while a recovery state at a level $k < L$ is restarted upon the occurrence of a failure at any level less than k. Again using formulas 7 and 8 from Section 9.3, we merge each of these possible

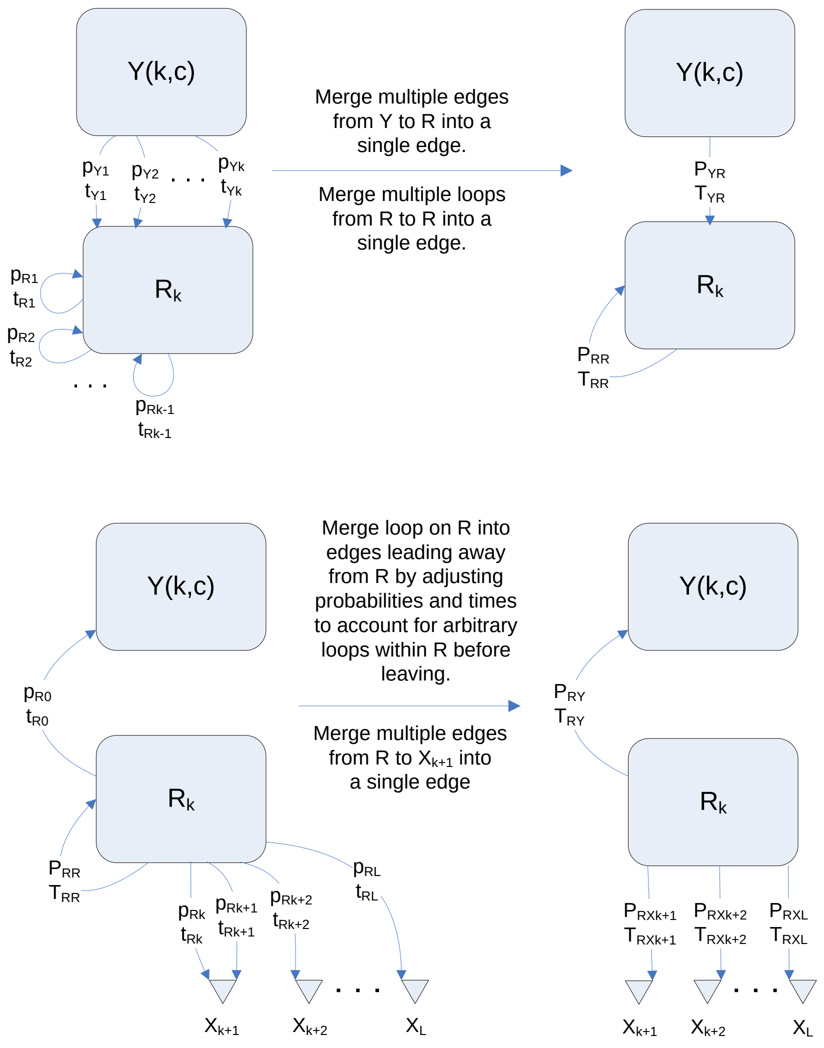

Y(k,c)
p_Y1 t_Y1
p_Y2 t_Y2
p_Yk t_Yk
R_k
p_R1 t_R1
p_R2 t_R2
p_Rk-1 t_Rk-1
Merge multiple edges from Y to R into a single edge.
Merge multiple loops from R to R into a single edge.
P_YR T_YR
P_RR T_RR
p_R0 t_R0
P_RR T_RR
p_Rk t_Rk
p_Rk+1 t_Rk+1
p_Rk+2 t_Rk+2
p_RL t_RL
X_k+1
X_k+2
X_L
Merge loop on R into edges leading away from R by adjusting probabilities and times to account for arbitrary loops within R before leaving.
Merge multiple edges from R to X_k+1 into a single edge
P_RY T_RY
P_RXk+1 T_RXk+1
P_RXk+2 T_RXk+2
P_RXL T_RXL

Figure 5: Merging edges in $X(k, c)$

transitions from R to R into a single transition having probability P_{RR} and an expected run time T_{RR} as shown in the top portion of Figure 5:

$$P_{RR} = \begin{cases} 0 & \text{for } k = 1, \\ \sum_{i=1}^{k-1} p_{Ri} & \text{for } 1 < k < L, \\ \sum_{i=1}^{k} p_{Ri} & \text{for } k = L \end{cases}$$

and, when $P_{RR} > 0$,

$$T_{RR} = \begin{cases} \frac{\sum_{i=1}^{k-1} p_{Ri} \cdot t_{Ri}}{P_{RR}} & \text{for } 1 < k < L, \\ \frac{\sum_{i=1}^{k} p_{Ri} \cdot t_{Ri}}{P_{RR}} & \text{for } k = L. \end{cases}$$

The last substitution we make is to collapse the single loop transition from R to R into the transitions leading away from R. Upon entering R, a transition away from R eventually happens, provided $P_{RR} < 1$. However, one or more loops back to R may occur before transitioning away. It is possible to adjust the probabilities and expected run times of the transitions leading away from R to account for an arbitrary number of loops back to R before making the transition away. Derivation of the formulas to make these adjustments is provided in Section 9.3.

First, consider the transition from R to Y. Using formulas 9 and 10 from Section 9.3, we collapse the loop into this transition and redefine its probability to be P_{RY} and expected run time to be T_{RY} as shown the bottom portion of Figure 5, where

$$P_{RY} = \begin{cases} \frac{p_{R0}}{1-P_{RR}} & \text{for } P_{RR} < 1, \\ 0 & \text{for } P_{RR} = 1, \end{cases}$$

and, when $P_{RY} > 0$,

$$T_{RY} = t_{R0} + \frac{P_{RR}}{1 - P_{RR}} \cdot T_{RR}.$$

Now, consider the transitions from R to external recovery states. These transitions contribute directly to the elements of the X vectors. For the i-th element of an X vector, where $i \in 1, 2, \cdots, L$, we first merge any edges from R that contribute to this element, and then we collapse the loop into the merged edge to define a new edge with probability P_{RX_i} and expected run time T_{RX_i}, as shown in the bottom portion of Figure 5.

Recall under our model that a recovery state at the maximum level $k = L$ is restarted upon the occurrence of a failure at any level. Such a state only contains loop-back transitions. In this case, there is zero probability that R transitions to an external recovery state, so when $k = L$, we have for each $i \in 1, 2, \cdots, L$

$$P_{RX_i} = 0.$$

While in a recovery state at level k, where $k < L$, the system transitions to a recovery state at level $k + 1$ if either a level k failure or a level $k + 1$ failure occurs. Otherwise, for the occurrence of a failure at level i, where $i > k + 1$, a transition is made to a recovery state at level i. Thus, when $k < L$, we have for each $i \in 1, 2, \cdots, L$

$$P_{RX_i} = \begin{cases} 0 & \text{for } 1 \le i \le k \text{ or } P_{RR} = 1, \\ \frac{p_{Rk} + p_{R(k+1)}}{1-P_{RR}} & \text{for } i = k+1 \text{ and } P_{RR} < 1, \\ \frac{p_{Ri}}{1-P_{RR}} & \text{for } i > k+1 \text{ and } P_{RR} < 1, \end{cases}$$

and, when $P_{RX_i} > 0$,

$$T_{RX_i} = \begin{cases} \frac{p_{Rk} \cdot t_{Rk} + p_{R(k+1)} \cdot t_{R(k+1)}}{p_{Rk} + p_{R(k+1)}} + \frac{P_{RR}}{1-P_{RR}} \cdot T_{RR} & \text{for } i = k+1, \\ t_{Ri} + \frac{P_{RR}}{1-P_{RR}} \cdot T_{RR} & \text{for } i > k+1. \end{cases}$$

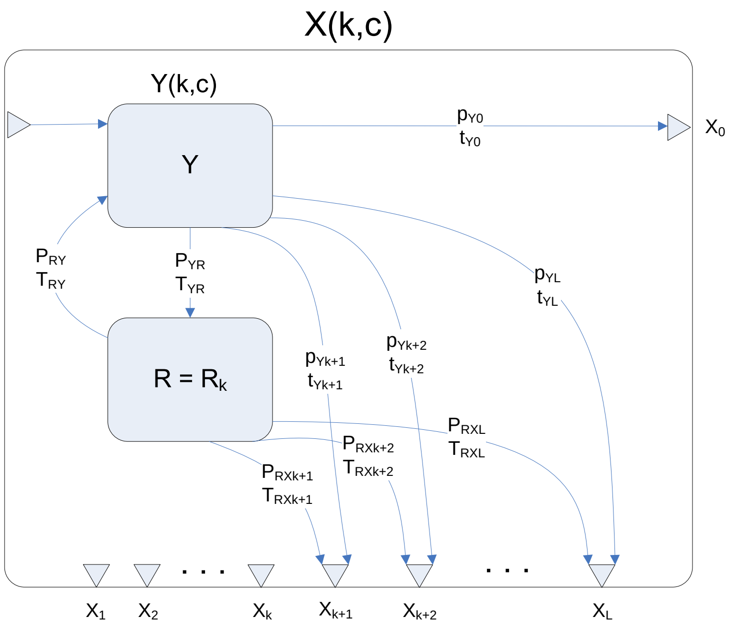


Figure 6: Simplified diagram of $X(k, c)$

After making all of these transformations, the simplified $X(k, c)$ block is shown in Figure 6. Finally, we derive the total probabilities and expected run times to transition out of the X block as

$$p_{X0} = \begin{cases} \frac{p_{Y0}}{1-P_{YR}\cdot P_{RY}} & \text{for } P_{YR}\cdot P_{RY} < 1, \\ 0 & \text{for } P_{YR}\cdot P_{RY} = 1, \end{cases}$$

and, when $p_{X0} > 0$,

$$t_{X0} = t_{Y0} + \frac{P_{YR}\cdot P_{RY}}{1 - P_{YR}\cdot P_{RY}} \cdot (T_{YR} + T_{RY}).$$

Also, for each level $i \in 1, 2, \cdots, L$

$$p_{Xi} = \begin{cases} 0 & \text{for } 1 \le i \le k \text{ or } P_{YR}\cdot P_{RR} = 1, \\ \frac{p_{Yi}+P_{YR}\cdot P_{RX_i}}{1-P_{YR}\cdot P_{RY}} & \text{for } i > k \text{ and } P_{YR}\cdot P_{RR} < 1, \end{cases}$$

and, when $p_{Xi} > 0$,

$$t_{Xi} = \frac{p_{Yi}\cdot t_{Yi} + P_{YR}\cdot P_{RX_i}\cdot (T_{YR} + T_{RX_i})}{p_{Yi} + P_{YR}\cdot P_{RX_i}} + \frac{P_{YR}\cdot P_{RY}}{1 - P_{YR}\cdot P_{RY}} \cdot (T_{YR} + T_{RY}).$$

Note that the internal recovery state at level k contained within the $X(k, c)$ block handles internal failures such that $X(k, c)$ never transitions to an external recovery state at level k or lower.

5.6 The $Z(k, c)$ block

A $Z(k, c)$ block only exists when $v_k > 0$, and it internally consists of a chain of $X(k, k)$ blocks of length $v_k - 1$ followed by a single $X(k, c)$ block. Let us refer to these blocks as simply Z, X, and X', where $Z = Z(k, c)$,

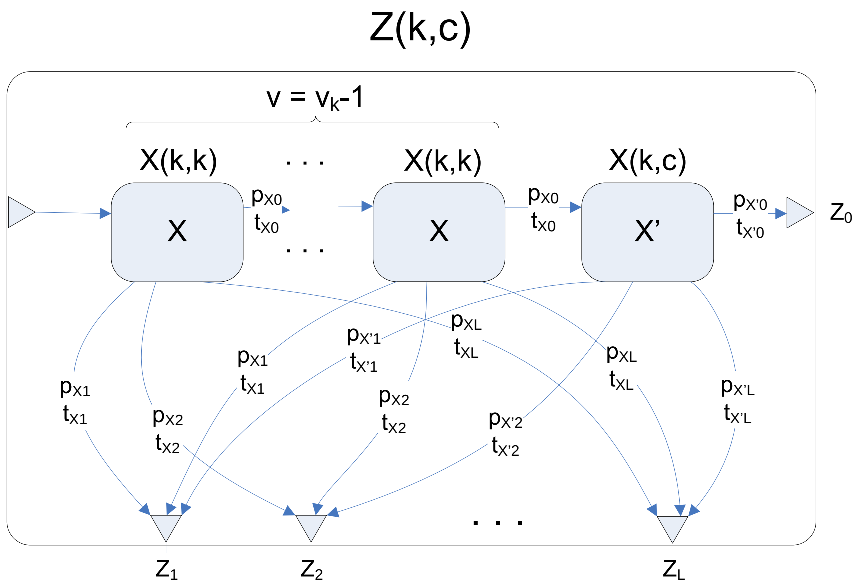


Figure 7: Diagram of $Z(k, c)$

$X = X(k, k)$, and $X' = X(k, c)$. Also, we define v such that $v = v_k - 1$. A diagram of a $Z(k, c)$ block is shown in Figure 7. Here we compute the probability and expected run time vectors for Z, which we denote as $\vec{p_Z}$ and $\vec{t_Z}$, assuming that we are given the vectors for X, as $\vec{p_X}$ and $\vec{t_X}$, and X', as $\vec{p_{X'}}$ and $\vec{t_{X'}}$.

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 out of X blocks are followed by one successful transition out the X' block. The probability of this sequence occurring is

$$p_{Z0} = (p_{X0})^v \cdot p_{X'0}$$

and when $p_{Z0} > 0$, the expected time to make this transition is

$$t_{Z0} = v \cdot t_{X0} + t_{X'0}.$$

For each failure level $i \in 1, 2, \cdots, L$, there are multiple paths through Z which require a transition to a recovery state at level i. The first X block may transition to a recovery state at level i. Or, that block may transition successfully to the next X block, which in turn may transition to a recovery state at level i. Or, the first two X blocks may transition successfully on to the the third X block, which may transition to a recovery state at level i, and so on up to and including the final X' block. The total probability to leave Z for a recovery state at level i is the sum of the probabilities corresponding to each of these possible paths:

$$\begin{aligned} p_{Zi} &= p_{Xi} + p_{X0} \cdot p_{Xi} + (p_{X0})^2 \cdot p_{Xi} + \cdots + (p_{X0})^{v-1} \cdot p_{Xi} + (p_{X0})^v \cdot p_{X'i} \\ &= (1 + p_{X0} + (p_{X0})^2 + \cdots + (p_{X0})^{v-1}) \cdot p_{Xi} + (p_{X0})^v \cdot p_{X'i}. \end{aligned}$$

Using equation 5 from Section 9.1 and substituting $x = p_{X0}$ and $N = v - 1$, and knowing that if $p_{X0} = 1$ then $p_{Xi} = 0$, we can simplify p_{Zi} to

$$p_{Zi} = \begin{cases} \frac{1-(p_{X0})^v}{1-p_{X0}} \cdot p_{Xi} + (p_{X0})^v \cdot p_{X'i} & \text{for } p_{X0} < 1, \\ p_{X'i} & \text{for } p_{X0} = 1. \end{cases}$$

Similarly, we compute the expected run time to transition from Z to a recovery state at level i by considering the probability of taking each possible path along with the expected time to take each path. When $p_{Zi} > 0$, we have

$$t_{Zi} = \frac{A_i}{p_{Zi}}$$

where

$$\begin{aligned} A_i &= p_{Xi} \cdot t_{Xi} + (p_{X0})^1 \cdot p_{Xi} \cdot (1 \cdot t_{X0} + t_{Xi}) \\ &\quad + (p_{X0})^2 \cdot p_{Xi} \cdot (2 \cdot t_{XO} + t_{Xi}) + \cdots + (p_{X0})^{v-1} \cdot p_{Xi} \cdot ((v-1) \cdot t_{X0} + t_{Xi}) \\ &\quad + (p_{X0})^v \cdot p_{X'i} \cdot (v \cdot t_{X0} + t_{X'i}) \\ &= B_i + (p_{X0})^v \cdot p_{X'i} \cdot (v \cdot t_{X0} + t_{X'i}), \end{aligned}$$

where

$$\begin{aligned} B_i &= p_{Xi} \cdot t_{Xi} + (p_{X0})^1 \cdot p_{Xi} \cdot (1 \cdot t_{X0} + t_{Xi}) \\ &\quad + (p_{X0})^2 \cdot p_{Xi} \cdot (2 \cdot t_{XO} + t_{Xi}) + \cdots + (p_{X0})^{v-1} \cdot p_{Xi} \cdot ((v-1) \cdot t_{X0} + t_{Xi}) \\ &= (p_{Xi} \cdot t_{Xi} + (p_{X0})^1 \cdot p_{Xi} \cdot t_{Xi} + (p_{X0})^2 \cdot p_{Xi} \cdot t_{Xi} + \cdots + (p_{X0})^{v-1} \cdot p_{Xi} \cdot t_{Xi}) \\ &\quad + ((p_{X0})^1 \cdot p_{Xi} \cdot 1 \cdot t_{X0} + (p_{X0})^2 \cdot p_{Xi} \cdot 2 \cdot t_{X0} + \cdots + (p_{X0})^{v-1} \cdot p_{Xi} \cdot (v-1) \cdot t_{X0}) \\ &= (1 + (p_{X0})^1 + (p_{X0})^2 + \cdots + (p_{X0})^{v-1}) \cdot p_{Xi} \cdot t_{Xi} \\ &\quad + (1 \cdot (p_{X0})^1 + 2 \cdot (p_{X0})^2 + \cdots + (v-1) \cdot (p_{X0})^{v-1}) \cdot p_{Xi} \cdot t_{X0}. \end{aligned}$$

Using equations 5 and 6 from Section 9.1, substituting $x = p_{X0}$ and $N = v - 1$, and knowing that if $p_{X0} = 1$ then $p_{Xi} = 0$, we can simplify B_i to

$$B_i = \begin{cases} \frac{1-(p_{X0})^v}{1-p_{X0}} \cdot p_{Xi} \cdot t_{Xi} + \frac{p_{X0} - v \cdot (p_{X0})^v + (v-1) \cdot (p_{X0})^{v+1}}{(1-p_{X0})^2} \cdot p_{Xi} \cdot t_{X0} & \text{for } p_{X0} < 1, \\ 0 & \text{for } p_{X0} = 1. \end{cases}$$

5.7 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 5.4.1.

If $k > 1$ and $v_{k-1} = 0$, then $Y(k,c)$ consists of a single $Y(k-1,c)$ block. In this case, let us refer to these blocks as Y and Y', where $Y = Y(k,c)$ and $Y' = Y(k-1,c)$. Because Y consists solely of Y', the probability and expected run time vectors to transition out of Y, which we denote as $\vec{p_Y}$ and $\vec{t_Y}$, are trivially computed given the vectors for Y', as $\vec{p_{Y'}}$ and $\vec{t_{Y'}}$:

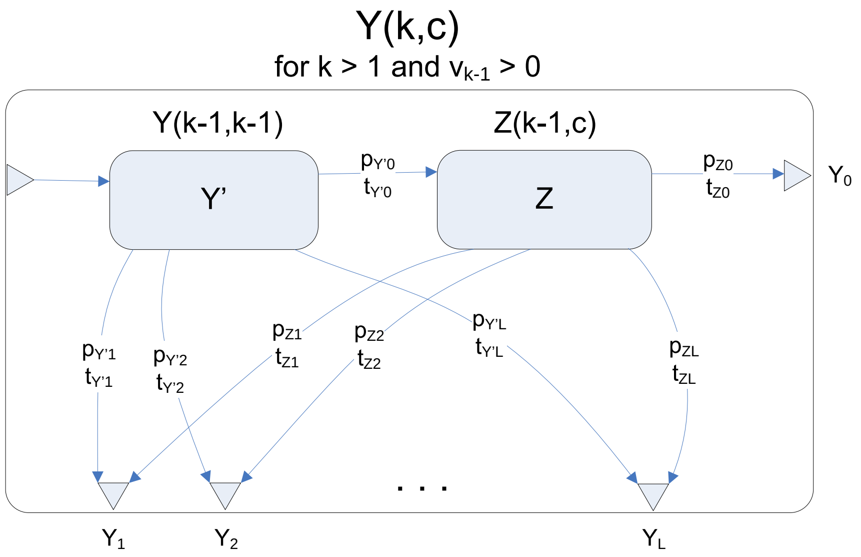


Figure 8: Diagram of $Y(k,c)$

$$p_{Y0} = p_{Y'0},$$
$$t_{Y0} = t_{Y'0}$$

and, for each level $i \in 1, 2, \cdots, L$,

$$p_{Yi} = p_{Y'i},$$
$$t_{Yi} = t_{Y'i}.$$

If $k > 1$ and $v_{k-1} > 0$, then $Y(k,c)$ consists of a starting $Y(k-1,k-1)$ block followed by a $Z(k-1,c)$ block, as shown in Figure 8. Let us refer to these states as Y, Y', and Z, where $Y = Y(k,c)$, $Y' = Y(k-1,k-1)$, and $Z = Z(k-1,c)$. Here we compute the probability and expected run time vectors to transition out of Y, which we denote as $\vec{p_Y}$ and $\vec{t_Y}$, assuming that we are given the vectors for Y', as $\vec{p_{Y'}}$ and $\vec{t_{Y'}}$, and Z, as $\vec{p_Z}$ and $\vec{t_Z}$.

The probability that a successful transition out of Y occurs is the probability that both Y' and Z transition successfully. The probability of such a sequence occurring is

$$p_{Y0} = p_{Y'0} \cdot p_{Z0}$$

and the expected time to make this transition is

$$t_{Y0} = t_{Y'0} + t_{Z0}.$$

For each failure level $i \in 1, 2, \cdots, L$, there are two possible paths through Y that can cause a transition to a recovery state at level i. The Y' block may transition immediately to a recovery state at level i, or the Y' block may transition successfully to Z, which in turn may transition to a recovery state at level i. The total probability to leave Y for a recovery state at level i is the sum of the probabilities corresponding to each path

$$p_{Yi} = p_{Y'i} + p_{Y'0} \cdot p_{Zi}.$$

When $p_{Yi} > 0$, the expected run time of this transition is

$$t_{Yi} = \frac{p_{Y'i} \cdot t_{Y'i} + p_{Y'0} \cdot p_{Zi} \cdot (t_{Y'0} + t_{Zi})}{p_{Yi}}.$$

5.8 Model Metrics

In this work, we are interested in two key metrics: *efficiency* and *parallel file system load.* We define efficiency as

$$\mathit{efficiency} = \frac{\mathit{idealTime}}{\mathit{expectedTime}}.$$

Here, *idealTime* is the minimum run time assuming the application spends no time checkpointing and encounters no failures, while *expectedTime* is the expected run time as predicted by the model for a given set of parameters. This metric indicates how much time is lost to checkpointing activities, including recovery from failures.

To compute the efficiency, we consider a single level L period. After parameterizing the model with a set of checkpoint levels, checkpoint costs, recovery costs, failure rates, and a time interval between checkpoints, we can compute the expected time required to complete a single level L period. Namely, we compute $\vec{p}$ and $\vec{t}$ for $X(L, L)$. The value of element t_0 then provides the *expectedTime* to complete a level L period, which is well-defined if $p_0 = 1$. The *idealTime* is simply the number of compute intervals during this period multiplied by the length of each interval. To be precise, this is computed as

$$\begin{aligned} \mathit{idealTime} &= (v_1 + 1) \cdot (v_2 + 1) \cdot (v_{L-1} + 1) \cdot t \\ &= t \cdot \prod_{k=1}^{L-1} (v_k + 1) \end{aligned}$$

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 parallel file system load to be the inverse of this time. Checkpoints are written to the parallel file system at the end of each level L period, so the expected time between checkpoints is the same as the *expectedTime* as defined above. Hence, we define the parallel file system load as

$$\mathit{load} = \frac{1}{\mathit{expectedTime}}.$$

We use these metrics in the following sections to evaluate the benefits of different model configurations.

6 Model Exploration

We used our model to explore the behavior of SCR under varying conditions. We show the benefits of multi-level checkpointing over single-level checkpointing as failure rates and parallel file system characteristics change. Also, we illustrate how to use the model to select parameter values that maximize system utilization.

First, we compare the model's predictions of `pf3D` efficiency to that observed in real runs on Coastal and Atlas in Table 4. The data show that the model's predictions are within a few percent of observed reality for this application on real systems. Albeit limited due to the many hours required to gather data, these results provide some assurance that our model is accurate.

Table 4: Expected and observed efficiency

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

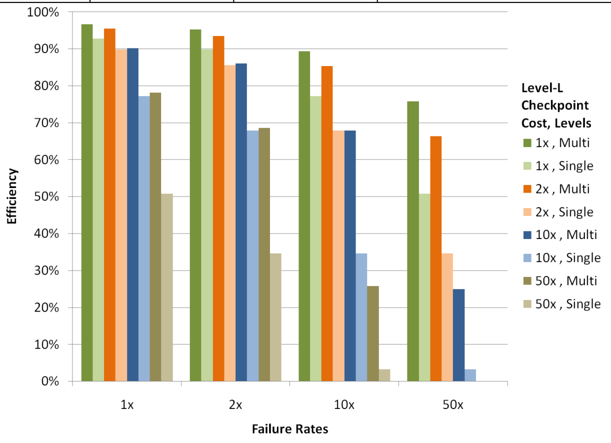


Figure 9: Optimal efficiency for single- and multi-level checkpointing

We now use the model to explore multi-level checkpointing in a more general context. In the following experiments, we simulated a three level checkpointing system ($L = 3$) and varied the length of the compute interval, the number of level 1 and level 2 checkpoints per level 3 period, the failure rates, and the cost of level 3 checkpoints.

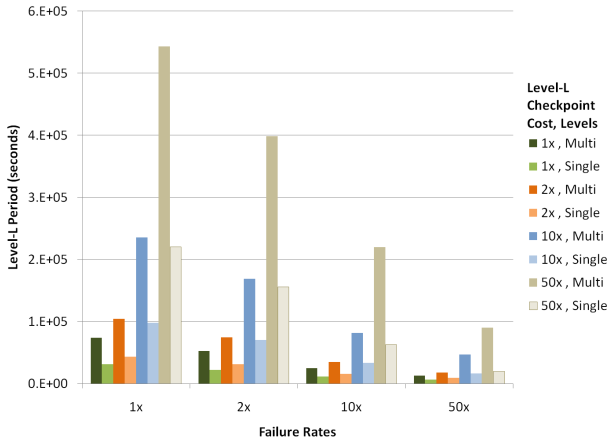
For checkpoint costs, we use the times recorded in Table 1 for checkpointing `pF3D` on Coastal using `LOCAL` on RAM, `XOR` on RAM, and Lustre, which gives us costs of 0.5 seconds, 4.5 seconds, and 1052 seconds, respectively. We set recovery costs to be the same as checkpoint costs. Using the failure data for `pF3D` on Coastal in Table 2, we express the failure rates in units of failures per job-second, i.e., average number of failures at a given level per node-hour, multiplied by the number of nodes used in the job, divided by 3,600 seconds per hour. This leads to failure rates 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. For comparison, we performed the same experiment for single-level checkpointing, assuming only the parallel file system is available.

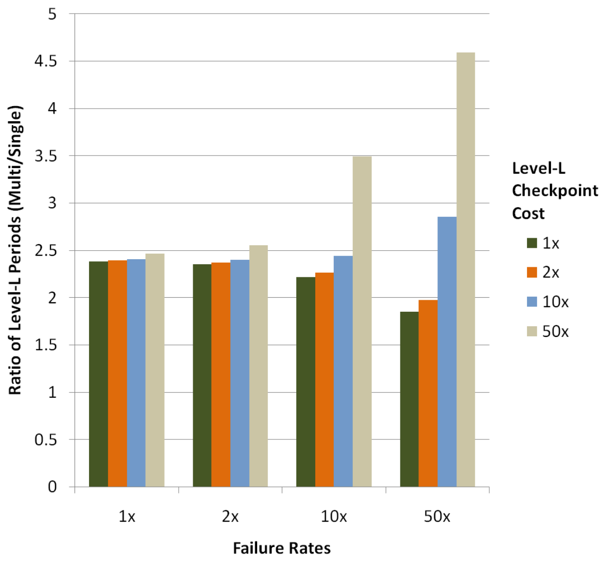
Figure 9 presents the efficiency achieved for each configuration, and Figure 10(a) shows the time between level L checkpoints. We label the results for the multi-level system as "Multi" and those for the single-level system as "Single." The groupings of bars along the x-axis correspond to failure rates that are one, two, ten, or fifty times the base values. Within each grouping, we increase the cost of the level L checkpoint by one, two, ten, and fifty times the base value.

In all cases, the multi-level system results in higher efficiencies, and it 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, as can be seen in Figure 10(b), the load on the parallel file system is reduced by a factor ranging from 2x-4x. Thus, compared to single-level checkpointing, multi-level checkpointing simultaneously increases efficiency while reducing load on the parallel file system. These results highlight the benefits of multi-level checkpointing on current and future systems.

Overall, we find that multi-level checkpointing is essential for future systems. Even with systems that are 50× less reliable, a three level checkpointing system achieves efficiencies over 75%, so long as we maintain relative



(a) Optimal level-L periods



(b) Factor by which load is reduced

Figure 10: Checkpoints to the parallel file system

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× less reliable and the cost of saving application state to the parallel file system rises by 10×, a three level checkpointing system only achieves 26% efficiency.

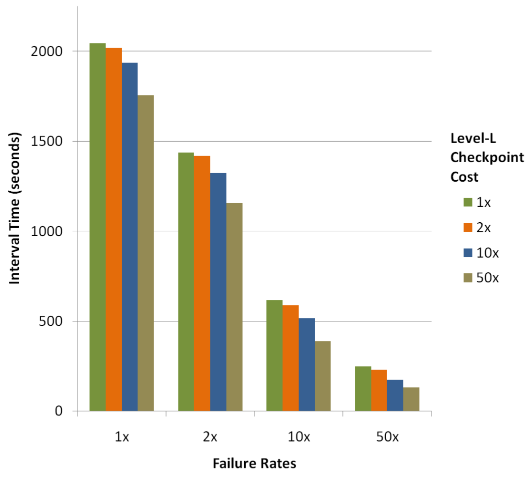
Figure 11 provides insight into multi-level checkpointing by showing the compute intervals and level 2 checkpoint counts that provide the optimal efficiency for each failure rate and level L checkpoint cost combination. We do not graph the optimal level 1 checkpoint counts, as they were zero in all cases. The low number of optimal level 1 checkpoint counts is due to the relatively low number of failures from which we can recover using level 1 checkpoints.

The optimal compute interval decreases with either increasing failure rate or increasing level L checkpoint cost. For the level 2 checkpoint counts, we found that the optimal counts increased dramatically with increasing failure rate. The trend for increasing level L checkpoint cost is not as clear, with higher optimal values for costs that are twice or 50× greater than for the costs that were one and 10× higher. These results indicate that applications running on systems with higher failure rates or higher parallel file system overheads must increase the ratio of lower-level checkpoints to parallel file system checkpoints to maximize efficiency.

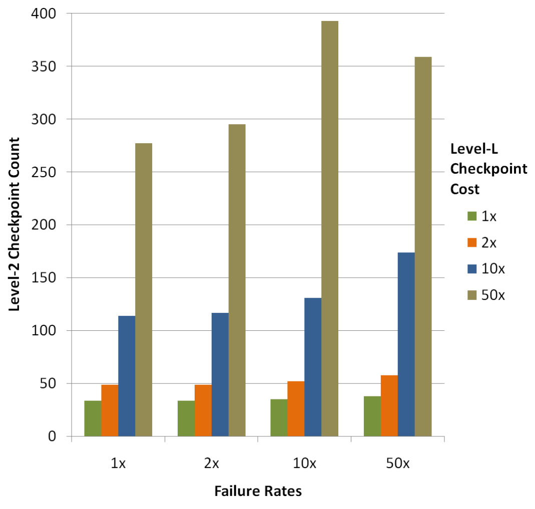
We investigated the relationship between our model parameters and the expected efficiency of the multi-level model. Figure 12 shows the expected efficiency when failure rates and the level L checkpoint cost are 1×, 2×, and 10× their base values. The plots were produced by setting the level 1 count to zero, setting the level 2 count to five, and varying the compute interval from 0 to $2 \cdot 10^5$ seconds.

In general, efficiencies first increase sharply with increasing compute intervals and then decrease relatively slowly after reaching the optimum. The optimal efficiencies are lower and the declines in efficiency past the peak are sharper for higher parallel file system costs and failure rates. Further, the compute interval has a complex relationship with failure rates and checkpoint costs. The efficiencies produced by short compute intervals are highly sensitive to checkpoint cost and insensitive to failure rates. However, this effect is reversed for large compute intervals, where efficiencies are sensitive to failure rates but insensitive to checkpoint costs.

Overall, we observe a broad range of compute intervals that result in near-optimal efficiencies for level 1 and level 2 checkpoint counts that are below or near optimal values. The effect of increasing level 1 counts is lower efficiencies and narrower near-optimal ranges. The effect of increasing level 2 counts is relatively higher efficiencies and wider near-optimal ranges up to a point beyond the optimal level 2 count, after which the benefits of adding level 2 checkpoints declines. These results show that we must carefully choose parameters when using multi-level



(a) Optimal compute interval



(b) Optimal level 2 checkpoint counts

Figure 11: Optimal multi-level checkpointing parameter values

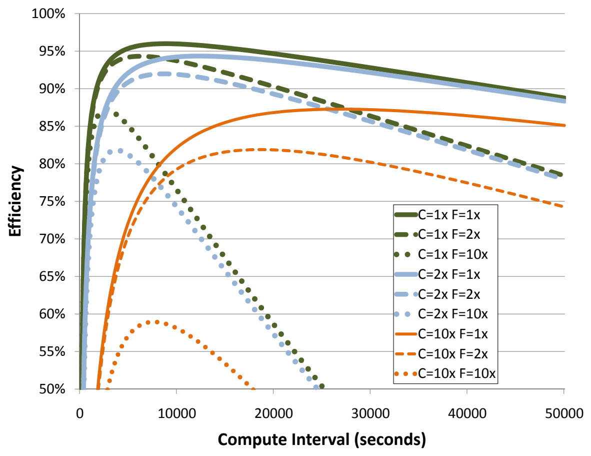


Figure 12: Efficiency versus compute interval

checkpointing as failure rates and parallel file system costs increase. However, we can utilize our model to guide these decisions.

7 SCR on Diskless Systems

Clusters often use diskless compute nodes. However, Linux supports a RAM disk, which is a file system maintained in main memory. We can configure SCR to use RAM disk as its node-local storage, but we then need sufficient memory to store at least one checkpoint in memory along with the application working set. Simulations that require all available memory on diskless clusters cannot use SCR. However, often sufficient memory can be made available by running the same simulation on more compute nodes to spread the working set among more processes. We then must determine if it is worthwhile to use more compute resources in order to use SCR. Our multi-level checkpoint model can guide this choice by predicting an expected savings in node hours required to complete the job.

First, we derive the required application scalability to benefit by using more nodes as a function of machine efficiencies with and without SCR. Assume the original problem requires N_1 nodes and N_2 nodes are required when using SCR, with $N_2 > N_1$. Excluding checkpoints and assuming no failures, let the time to complete the job be T_{S1}

when using N_1 nodes and T_{S2} when using N_2 nodes. Then, the scalability factor, α, of the application is determined by the difference in speedup normalized to T_{S1} divided by the difference in the number of nodes normalized to N_1, such that

$$\alpha = \frac{\frac{T_{S1}}{T_{S2}} - 1}{\frac{N_2}{N_1} - 1}.$$

Solving for T_{S2}, we get

$$\begin{aligned}
\frac{\frac{T_{S1}}{T_{S2}} - 1}{\frac{N_2}{N_1} - 1} &= \alpha \\
\frac{T_{S1}}{T_{S2}} - 1 &= \alpha \cdot \frac{N_2 - N_1}{N_1} \\
\frac{T_{S1}}{T_{S2}} &= \frac{\alpha \cdot (N_2 - N_1) + N_1}{N_1} \\
\frac{1}{T_{S2}} &= \frac{\alpha \cdot (N_2 - N_1) + N_1}{N_1 \cdot T_{S1}} \\
T_{S2} &= \frac{N_1 \cdot T_{S1}}{N_1 \cdot (1 - \alpha) + N_2 \cdot \alpha}.
\end{aligned}$$

Let e_1 represent the machine efficiency when using N_1 nodes without SCR, and let e_2 be the efficiency when using using N_2 nodes with SCR. Then, the total time spent on the machine to complete the job in each case is

$$\begin{aligned}
T_1 &= \frac{T_{S1}}{e_1}, \\
T_2 &= \frac{T_{S2}}{e_2} = \frac{N_1 \cdot T_{S1}}{e_2 \cdot N_1 \cdot (1 - \alpha) + e_2 \cdot N_2 \cdot \alpha}.
\end{aligned}$$

The node hours required complete the job in each case are

$$\begin{aligned}
H_1 &= N_1 \cdot T_1 = \frac{N_1 \cdot T_{S1}}{e_1}, \\
H_2 &= N_2 \cdot T_2 = \frac{N_1 \cdot T_{S1}}{e_2 \cdot \alpha + e_2 \cdot (1 - \alpha) \cdot \frac{N_1}{N_2}}.
\end{aligned}$$

Finally, the ratio of node hours when using N_2 nodes to the node hours when using N_1 nodes is

$$H_2 \,/\, H_1 = \frac{e_1}{e_2 \cdot (\alpha + (1 - \alpha) \cdot \frac{N_1}{N_2})}.$$

It is beneficial to use SCR by allocating more nodes when $H_2 < H_1$, that is, when $H_2/H_1 < 1$, which occurs when the scalability factor satisfies the following condition:

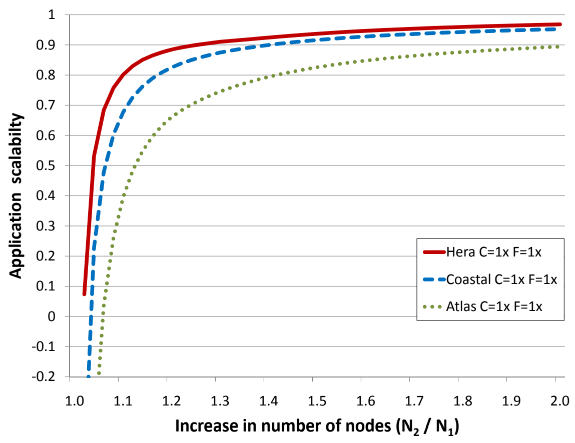


Figure 13: Required scalability to benefit by allocating more nodes

$$\frac{e_1}{e_2 \cdot (\alpha + (1-\alpha) \cdot \frac{N_1}{N_2})} < 1$$

$$e_1 < e_2 \cdot \alpha + e_2 \cdot \frac{N_1}{N_2} - e_2 \cdot \alpha \cdot \frac{N_1}{N_2}$$

$$e_1 - e_2 \cdot \frac{N_1}{N_2} < \alpha \cdot e_2 \cdot (1 - \frac{N_1}{N_2})$$

$$e_1 \cdot \frac{N_2}{N_1} - e_2 < \alpha \cdot e_2 \cdot (\frac{N_2}{N_1} - 1)$$

$$\alpha > \frac{e_1 \cdot \frac{N_2}{N_1} - e_2}{e_2 \cdot (\frac{N_2}{N_1} - 1)}.$$

This is a general expression for all applications. Now to answer the question for a particular application, we use our model to estimate the machine efficiency with and without SCR. For example, consider the pF3D runs from Table 1. Assume the original problem requires N_1 nodes and using SCR requires $N_2 > N_1$ nodes. We must first adjust the failure rates and checkpoint costs in order to compute the expected efficiency with additional nodes.

The mean-time-before-failure for the application decreases when it uses more nodes since it is exposed to more hardware. The amount of hardware used scales linearly with the number of nodes. Since we assume multiple failures within a class are independent, we scale the failure rate of each failure class linearly with the number of nodes.

Also, the total size of the data set remains fixed regardless of the number of nodes. Thus, we assume the cost to checkpoint to the parallel file system remains constant. However, the amount of data stored per node decreases linearly with the number of nodes since we spread the data evenly among the nodes. Thus, the cost to checkpoint to RAM disk decreases linearly with the number of nodes.

With these adjustments, we apply the model to compute the conditions under which using SCR by allocating more nodes decreases total node hours. We express the results by plotting the required application scalability as a function of the ratio of the number of nodes, N_2/N_1. If an application meets or exceeds the required scalability at a particular N_2/N_1 ratio, then it will benefit by allocating N_2/N_1 more nodes to use SCR. Using the pF3D checkpoint costs from the top three rows in Table 1 and the failure rates from Table 2, we show the required application scalability for Hera, Atlas, and Coastal in Figure 13.

The three clusters have three distinct curves. Coastal is the most reliable system, but its parallel file system is the slowest. Atlas and Hera have much higher failure rates than Coastal, but they have faster parallel file systems. Atlas and Hera have similar failure rates but the parallel file system on Hera is faster than on Atlas. These varying

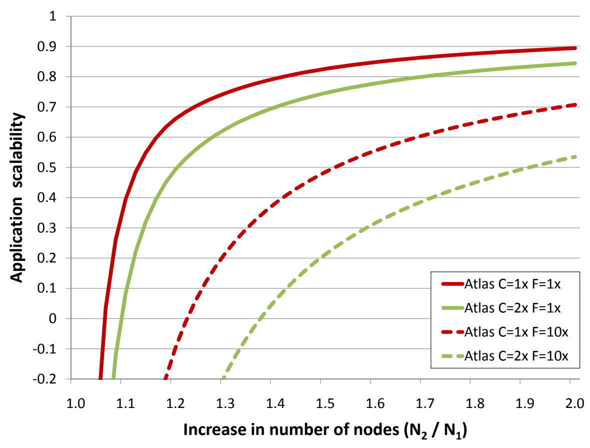


Figure 14: Required scalability for higher overhead and failure rates

properties produce three different curves. Thus, the same application may benefit from requesting additional nodes on one system but not another. Generally, more applications will benefit from using more nodes on Atlas than on Coastal, and more on Coastal than on Hera.

We observe ranges for which allocating more nodes is beneficial for all values of N_2/N_1 plotted. Applications can benefit even if we must double the number of nodes to use SCR. However, at this limit, the application must scale very well, requiring a scalability factor between 0.90 and 0.97 depending on the platform. We require less scalability when SCR needs fewer nodes. For example, on Atlas, if only 20% more nodes are required, the application only needs a scalability factor of at least 0.65 to benefit. Finally, the application can benefit even with zero or negative scalability if it needs less than 10% additional nodes. Thus, the application may compute more slowly when using more nodes but still complete the problem in fewer node hours due to the increased efficiency provided by SCR.

In order to understand the potential impact of larger future systems, Figure 14 plots the required scalability for Atlas if we increase the parallel file system cost and the failure rate. More applications benefit by allocating more nodes to use SCR when either the cost to checkpoint to the parallel file system or the failure rate increases. If the checkpoint cost doubles and the failure rate increases by a factor of ten, an application with zero scalability could allocate up to 40% more nodes and still complete in fewer node hours. While not shown, the machine efficiency at this extreme drops to as low as 62% without SCR, but it stays above 85% with SCR. For all cases, we reduce the checkpoint frequency to the parallel file system by a factor between two and four when using SCR.

8 Conclusions

We presented a novel multi-level checkpointing implementation, the Scalable Checkpoint/Restart (SCR) library. SCR combines checkpointing to stable storage with lower-overhead, less-resilient checkpoint types, e.g., copying checkpoints to memory on other nodes. SCR performance far exceeds that of the parallel file system, as much as $1,000\times$ faster when using `LOCAL` checkpoints.

We also presented a detailed failure analysis for several large HPC systems. We found that applications could restart from a large majority of system failures using low-overhead checkpoints cached on the cluster. Only 15% of the observed failures required applications to restart from the parallel file system. Further, improving the reliability of the parallel file system would enable applications to recover from up to 96.4% of all failures using checkpoints cached on the cluster.

Finally, we presented a novel, hierarchical Markov model that predicts the performance of multi-level checkpointing systems based on system reliability and checkpoint cost. This model can guide users in selecting the best checkpointing parameters for their application. Our analysis with this model demonstrates that multi-level 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.

Our model also demonstrates that many applications will complete in fewer processor hours by using more nodes in order to provide sufficient memory to use SCR. Finally, SCR achieves these gains in efficiency while simultaneously reducing the load on the parallel file system by more than a factor of two.

For future work, we will extend our model to account for additional features of SCR. For example, we do not currently model SCR's ability to drain lower level checkpoints to the parallel file system. We will extend our model to determine how this feature can help reduce writes to the parallel file system. We will also extend SCR to copy checkpoints to the parallel file system asynchronously.

References

[1] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *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] B. Schroeder and G. Gibson, "Understanding Failure in Petascale Computers," *Journal of Physics Conference Series: SciDAC*, vol. 78, p. 012022, June 2007.

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

[6] 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.

[7] 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.

[8] R. Hedges, B. Loewe, T. McLarty, and C. Morrone, "Parallel File System Testing for the Lunatic Fringe: The Care and Feeding of Restless I/O Power Users," in *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2005, pp. 3–17.

[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] "Scalable Checkpoint/Restart Library." [Online]. Available: http://sourceforge.net/projects/scalablecr/

[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] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," in *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, 2006, pp. 14–23.

[20] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, October 1998.

[21] 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," in *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 213–223.

[22] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest Checkpoint Storage System for Large Supercomputers," in *3rd Petascale Data Storage Workshop (PDSW)*, Nov. 2008.

[23] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, no. 2, pp. 62–67, 1994.

[24] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, 2004, pp. 277–286.

[25] S. I. Feldman and C. B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, 1988, pp. 112–123.

[26] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2008, pp. 783–788.

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

[28] 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.

[29] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, 1988.

[30] W. Gropp, R. Ross, and N. Miller, "Providing Efficient I/O Redundancy in MPI Environments," in *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting*, 2004.

[31] 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.

9 Appendix A: Mathematical Derivations

In this section, we detail several mathematical derivations for our model described in Section 5. In Section 9.1, we show expressions for relevant geometric sums. We present derivations of probabilities and expected run times for base states when assuming failures follow Poisson distributions in Section 9.2. In Section 9.3, we derive formulas useful for merging edges in our Markov model.

9.1 Geometric sums

The following two formulas compute the sums of geometric series for when $x \neq 1$:

$$\sum_{i=0}^{N} x^i = \frac{1 - x^{(N+1)}}{1 - x}, \tag{5}$$

$$\sum_{i=1}^{N} i \cdot x^i = \frac{x - (N+1) \cdot x^{(N+1)} + N \cdot x^{(N+2)}}{(1-x)^2}. \tag{6}$$

For completeness, we derive these two formulas below. First, we show the derivation for Formula 5. When $x \neq 1$, let $A = \sum_{i=0}^{N} x^i$, then

$$\begin{aligned}
A - A \cdot x &= (\sum_{i=0}^{N} x^i) - (\sum_{i=0}^{N} x^i) \cdot x \\
&= (x^0 + x^1 + \cdots + x^N) - (x^0 + x^1 + \cdots + x^N) \cdot x \\
&= (x^0 + x^1 + \cdots + x^N) - (x^1 + x^2 + \cdots + x^{N+1}) \\
&= x^0 + x^1 + \cdots + x^N - x^1 - x^2 - \cdots - x^{N+1} \\
&= x^0 + (x^1 - x^1) + (x^2 - x^2) + \cdots + (x^N - x^N) - x^{N+1} \\
&= x^0 - x^{N+1} \\
A \cdot (1 - x) &= 1 - x^{N+1} \\
A &= \frac{1 - x^{N+1}}{1 - x} \\
\sum_{i=0}^{N} x^i &= \frac{1 - x^{N+1}}{1 - x}.
\end{aligned}$$

Now, we show the derivation for Formula 6. When $x \neq 1$, let $A = \sum_{i=1}^{N} i \cdot x^i$, then

$$
\begin{aligned}
A - A \cdot x &= (\sum_{i=1}^{N} i \cdot x^i) - (\sum_{i=1}^{N} i \cdot x^i) \cdot x \\
&= (1 \cdot x^1 + 2 \cdot x^2 + \cdots + N \cdot x^N) - (1 \cdot x^1 + 2 \cdot x^2 + \cdots + N \cdot x^N) \cdot x \\
&= (1 \cdot x^1 + 2 \cdot x^2 + \cdots + N \cdot x^N) - (1 \cdot x^2 + 2 \cdot x^3 + \cdots + N \cdot x^{N+1}) \\
&= 1 \cdot x^1 + 2 \cdot x^2 + \cdots + N \cdot x^N - 1 \cdot x^2 - 2 \cdot x^3 - \cdots - N \cdot x^{N+1} \\
&= x^1 + (2 \cdot x^2 - 1 \cdot x^2) + (3 \cdot x^3 - 2 \cdot x^3) + \cdots + (N \cdot x^N - (N-1) \cdot x^N) - N \cdot x^{N+1} \\
&= x^1 + x^2 + x^3 + \cdots + x^N - N \cdot x^{N+1} \\
&= (x^0 + x^1 + x^2 + x^3 + \cdots + x^N - x^0) - N \cdot x^{N+1} \\
&= ((\sum_{i=0}^{N} x^i) - 1) - N \cdot x^{N+1} \\
&= (\frac{1 - x^{N+1}}{1-x}) - 1 - N \cdot x^{N+1} \\
&= \frac{1 - x^{N+1} - 1 \cdot (1-x) - N \cdot x^{N+1} \cdot (1-x)}{1-x} \\
&= \frac{1 - x^{N+1} - 1 + x - N \cdot x^{N+1} + N \cdot x^{N+2}}{1-x} \\
&= \frac{x - (N+1) \cdot x^{N+1} + N \cdot x^{N+2}}{1-x} \\
A \cdot (1-x) &= \frac{x - (N+1) \cdot x^{N+1} + N \cdot x^{N+2}}{1-x} \\
A &= \frac{x - (N+1) \cdot x^{N+1} + N \cdot x^{N+2}}{(1-x)^2} \\
\sum_{i=0}^{N} i \cdot x^i &= \frac{x - (N+1) \cdot x^{N+1} + N \cdot x^{N+2}}{(1-x)^2}.
\end{aligned}
$$

9.2 Poisson distributions

In Section 5.4, the general formulas to compute the probability and expected run time vectors for base computation and recovery states are determined to be

$$
\begin{aligned}
p_0(T) &= (1 - F_1(T)) \cdot (1 - F_2(T)) \cdots (1 - F_L(T)), \\
t_0(T) &= T,
\end{aligned}
$$

and, for $k \in 1, 2, \cdots, L$,

$$
p_k(T) = \int_0^T (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt,
$$

and when $p_k(T) > 0$,

$$t_k(T) = \frac{\int_0^T t \cdot (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt}{p_k(T)}.$$

That section then claims that, for failures that follow Poisson distributions with failures at level k occuring at an average rate of λ_k, the above expressions evaluate to:

$$\begin{aligned} p_0(T) &= e^{-\lambda T}, \\ t_0(T) &= T, \end{aligned}$$

and for $k \in 1, 2, \cdots, L$,

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

where

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

Here, we derive these results. Given that failures at each level follow a Poisson distribution, with failures at level k occuring at an average rate of λ_k, the probability density function and cumulative distribution function are the following:

$$\begin{aligned} f_k(t) &= \lambda_k \cdot e^{-\lambda_k t}, \\ F_k(t) &= 1 - e^{-\lambda_k t}. \end{aligned}$$

The probability that there are no failures during the time interval from $t = 0$ to $t = T$ evaluates to

$$\begin{aligned} p_0(T) &= (1 - F_1(T)) \cdot (1 - F_2(T)) \cdots (1 - F_L(T)) \\ &= (1 - (1 - e^{-\lambda_1 T})) \cdot (1 - (1 - e^{-\lambda_2 T})) \cdots (1 - (1 - e^{-\lambda_L T})) \\ &= (e^{-\lambda_1 T}) \cdot (e^{-\lambda_2 T}) \cdots (e^{-\lambda_L T}) \\ &= e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_L) \cdot T} \\ &= e^{-\lambda T} \end{aligned}$$

where

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

The expected run time given that no failures occur is simply

$$t_0(T) = T.$$

The probability that a failure at level k occurs before a failure occurs at any other level during the time interval $t = 0$ to $t = T$ evaluates to

$$\begin{aligned}
p_k(T) &= \int_0^T (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt \\
&= \int_0^T (1 - (1 - e^{-\lambda_1 t})) \cdot (1 - (1 - e^{-\lambda_2 t})) \cdots (1 - (1 - e^{-\lambda_{k-1} t})) \cdot \\
&\quad (\lambda_k \cdot e^{-\lambda_k t}) \cdot (1 - (1 - e^{-\lambda_{k+1} t})) \cdots (1 - (1 - e^{-\lambda_L t}))\, dt \\
&= \int_0^T (e^{-\lambda_1 t}) \cdot (e^{-\lambda_2 t}) \cdots (e^{-\lambda_{k-1} t}) \cdot (\lambda_k \cdot e^{-\lambda_k t}) \cdot (e^{-\lambda_{k+1} t}) \cdots (e^{-\lambda_L t})\, dt \\
&= \int_0^T \lambda_k \cdot e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_L) \cdot t}\, dt \\
&= \lambda_k \cdot \int_0^T e^{-\lambda t}\, dt \\
&= -\frac{\lambda_k}{\lambda} e^{-\lambda t} \ \Big|_0^T \\
&= (-\frac{\lambda_k}{\lambda} e^{-\lambda \cdot T}) - (-\frac{\lambda_k}{\lambda} e^{-\lambda \cdot 0}) \\
&= -\frac{\lambda_k}{\lambda} e^{-\lambda T} + \frac{\lambda_k}{\lambda} \\
&= \frac{\lambda_k}{\lambda} (1 - e^{-\lambda T}).
\end{aligned}$$

The expected run time given that a failure at level k occurs before a failure occurs at any other level during the time interval $t = 0$ to $t = T$ evaluates to

$$t_k(T) = \frac{\int_0^T t \cdot (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt}{p_k(T)}.$$

Simplifying the numerator of $t_k(T)$, we get

$$\begin{aligned}
&\int_0^T t \cdot (1 - F_1(t)) \cdot (1 - F_2(t)) \cdots (1 - F_{k-1}(t)) \cdot f_k(t) \cdot (1 - F_{k+1}(t)) \cdots (1 - F_L(t))\, dt \\
&= \int_0^T t \cdot (e^{-\lambda_1 t}) \cdot (e^{-\lambda_2 t}) \cdots (e^{-\lambda_{k-1} t}) \cdot (\lambda_k \cdot e^{-\lambda_k t}) \cdot (e^{-\lambda_{k+1} t}) \cdots (e^{-\lambda_L t})\, dt \\
&= \int_0^T t \cdot \lambda_k \cdot e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_L) \cdot t}\, dt \\
&= \lambda_k \cdot \int_0^T t \cdot e^{-\lambda t}\, dt.
\end{aligned}$$

Since

$$\int_a^b x \cdot e^{cx}\, dx = \frac{1}{c^2} \cdot (cx - 1) \cdot e^{cx} \ \Big|_a^b,$$

we find that

$$\begin{aligned}
\lambda_k \cdot \int_0^T t \cdot e^{-\lambda t}\, dt &= \frac{\lambda_k}{\lambda^2} \cdot (-\lambda t - 1) \cdot e^{-\lambda t} \ \Big|_0^T \\
&= (\frac{\lambda_k}{\lambda^2} \cdot (-\lambda \cdot T - 1) \cdot e^{-\lambda \cdot T}) - (\frac{\lambda_k}{\lambda^2} \cdot (-\lambda \cdot 0 - 1) \cdot e^{-\lambda \cdot 0}) \\
&= \frac{\lambda_k}{\lambda^2} \cdot (-\lambda T - 1) \cdot e^{-\lambda T} - \frac{\lambda_k}{\lambda^2} \cdot (0 - 1) \cdot 1 \\
&= -\frac{\lambda_k}{\lambda^2} \cdot (\lambda T + 1) \cdot e^{-\lambda T} + \frac{\lambda_k}{\lambda^2} \\
&= \frac{\lambda_k}{\lambda^2} (1 - (\lambda T + 1) \cdot e^{-\lambda T}).
\end{aligned}$$

Substituting back into $t_k(T)$, we get

$$\begin{aligned}
t_k(T) &= \frac{\frac{\lambda_k}{\lambda^2}(1 - (\lambda T + 1) \cdot e^{-\lambda T})}{p_k(T)} \\
&= \frac{\frac{\lambda_k}{\lambda^2}(1 - (\lambda T + 1) \cdot e^{-\lambda T})}{\frac{\lambda_k}{\lambda}(1 - e^{-\lambda T})} \\
&= \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}.
\end{aligned}$$

9.3 Merging Edges

9.3.1 Multiple edges between two blocks

When a block has multiple edges that transition to the same destination block, we apply the following forumlas to merge those edges into a single logical edge.

Assume a block has N edges that all transition to the same destination block, and assume that a transition along edge $i \in 1, 2, \cdots, N$ is taken with probability p_i and expected cost t_i. Then, these edges can be combined into a single logical edge having probability P and expected cost T computed as

$$P = \sum_{i=1}^{N} p_i \tag{7}$$

and when $P > 0$,

$$T = \frac{\sum_{i=1}^{N} p_i \cdot t_i}{P}. \tag{8}$$

The total probability P is simply the sum of the probabilities of the edges, and the expected cost T is the average of the expected costs of the edges weighted by their respective probabilities.

9.3.2 Loop-back edges

When a block has an edge that loops back to itself, often what is of interest is the total probability and expected cost of transitioning to another block, allowing for an arbitrary number of loops before doing so. We apply the following formulas to merge the effects of a loop-back edge into the probabilities and expected costs of the edges that lead away from a block.

Assume a block has one or more edges leading away as well as a loop-back edge. Assume a transition along the loop-back edge is taken with probability p_{loop} and expected cost t_{loop}. Now, consider an arbitrary away edge, and assume it is taken with probability p and expected cost t. Then, the total probability P and expected cost T that a transition is made away from the block via this particular away edge, accounting for an arbitrary number of loops before doing so is given by

$$P = \begin{cases} \frac{p}{1-p_{loop}} & \text{for } p_{loop} < 1, \\ 0 & \text{for } p_{loop} = 1 \end{cases} \tag{9}$$

and when $P > 0$,

$$T = t + \frac{p_{loop}}{1 - p_{loop}} \cdot t_{loop}. \tag{10}$$

Here we derive the above forumlas. Consider a block having one or more edges that lead away as well as one edge that loops back to itself with probability p_{loop} and expected cost t_{loop}. Given that the system starts in this block and given that $p_{loop} < 1$, a transition away from the block must eventually happen. However, one or more transitions along the loop-back edge may occur before transitioning away.

Now consider an arbitrary away edge of this block, and assume for a single transition step, there is a probability p and associated expected cost t of taking this edge. Given that the system starts in this block, our goal is to compute P, which we define to be the total probability that a transition is eventually made along this away edge, allowing for an arbitrary number of loop transitions before doing so.

The system may transition away from the block along the away edge as the first transition with probability p. Or, it may take the loop transition first and then take the away edge as the next transition with probability $p_{loop} \cdot p$. Or, it may take two consecutive loop transtions and then take the away edge as the third transition with probability $p_{loop} \cdot p_{loop} \cdot p$. This pattern can be continued an infinite number of times. To get the total probability P, we sum the probabilities of each of these paths that all terminate with taking the away edge being considered. Thus,

$$P = p + p_{loop} \cdot p + (p_{loop})^2 \cdot p + \cdots$$

This sum can be written succinctly by taking the limit of formula 5 as $N \to \infty$ given that $0 \leq p_{loop} < 1$.

$$\begin{aligned} P &= p + p_{loop} \cdot p + (p_{loop})^2 \cdot p + \cdots \\ &= p \cdot (1 + p_{loop} + (p_{loop})^2 + \cdots) \\ &= p \cdot \sum_{i=0}^{\infty} (p_{loop})^i \\ &= p \cdot (\frac{1}{1 - p_{loop}}) \\ &= \frac{p}{1 - p_{loop}}. \end{aligned}$$

Of course if $p_{loop} = 1$, then $p = 0$ since $p + p_{loop} \leq 1$. In this case, $P = 0$ as each term in the infinite sum is multiplied by zero, so we arrive at the following for P as listed in 9:

$$P = \begin{cases} \frac{p}{1-p_{loop}} & \text{for } p_{loop} < 1, \\ 0 & \text{for } p_{loop} = 1. \end{cases}$$

When $P > 0$, one may follow a similar method to compute the expected cost of eventually taking the considered away edge. However, here we use a recursive method instead. This recursive method can be applied since the model is memory-less – the probability and expected cost of taking a particular edge from a particular block is the same regardless of the path taken to arrive at the block. Thus, upon entering the block, we assume the total expected cost to transition away from the block along the considered away edge is known to be T. From above, we also know that the total probability of taking this edge is P.

Now, given that the system starts in this block, consider the first transition step. This step may transition along the away edge with probability p and expected cost t, or it may transition along the loop-back edge and then eventually transition along the away edge. After taking the loop and re-entering the block, we know the total probability of leaving the block via the considered away edge is P and the expected cost is T. Thus, the probability of taking the away edge after first taking the loop is $p_{loop} \cdot P$, and the expected cost of this path is $t_{loop} + T$. When $P > 0$, we solve for T by computing the expected cost of taking either of these two paths

$$\begin{aligned} T &= \frac{p \cdot t + p_{loop} \cdot P \cdot (t_{loop} + T)}{p + p_{loop} \cdot P} \\ T \cdot (p + p_{loop} \cdot P) &= p \cdot t + p_{loop} \cdot P \cdot t_{loop} + p_{loop} \cdot P \cdot T \\ T \cdot p + T \cdot p_{loop} \cdot P - p_{loop} \cdot P \cdot T &= p \cdot t + p_{loop} \cdot P \cdot t_{loop} \\ T \cdot p &= p \cdot t + p_{loop} \cdot P \cdot t_{loop} \\ T &= t + \frac{p_{loop} \cdot P \cdot t_{loop}}{p} \\ T &= t + \frac{p_{loop} \cdot (\frac{p}{1-p_{loop}}) \cdot t_{loop}}{p} \\ T &= t + \frac{p_{loop}}{1 - p_{loop}} \cdot t_{loop}. \end{aligned}$$

This expression matches the one given in 10.

10 Appendix B: Python Implementation of Multi-level Checkpoint Model

In this section, we present an implementation of our multi-level checkpointing model described in Section 5. We use this code to compute the results shown in Sections 6 and 7. The code is written in Python version 2.6.

10.1 Comments

For each simulation run, there are several parameters that can be set that correspond to variables in the mathematical derivation of our model:

- Number of checkpoint levels: L
- Compute interval: t
- A list of the times to write checkpoints at each level: c_c
- A list of the times to recover from checkpoints at each level: r_k
- A list of the failure rates at each level: λ_i
- Sum of the failure rates: λ
- A list of the number of level-k checkpoints for each level-$k+1$ checkpoint: v_k

In an effort to make our code follow the mathematical derivation of our model as closely as possible, we begin counting list elements at index number 1. This means that each Python list will have a dummy value of zero as the first value. The values for level-1 will occur in each list at index 1.

10.2 Code

```
# This is a container class used to hold the parameters for each run of the model.
# The values are set at initialization time.
# Here, we have simply hard-coded in example values. In the real code, the values are
# set by command line options.
class options:
    def __init__(self):
        # L is the number of levels
        self.L = 2
        # t is the compute interval in seconds
        self.t = 600
        # checkWriteTimes is a list of the checkpoint costs at each level in seconds
        self.checkWriteTimes = [0, 15, 1835]
        # checkRecoverTimes is a list of the recovery costs at each level in seconds
        self.checkRecoverTimes = [0, 15, 1835]
        # lambdas is a list of the failure rates at each level
        self.lambdas = [0, 8.54e-7, 2.01e-7]
        # lam is the sum of the failure rates
        self.lam = 1.06e-6
        # V is a list of the counts of each level-k checkpoint per level=k+1 checkpoint
        self.V = [0, 100, 1]
        self.debug = False
        # epsilon is an upper bound on accumulated error
        self.epsilon = 1e-7
        # We precompute the values for R-base states (recoveryVals),
        # because they don’t change over the course of the execution. This variable
        # is set at the beginning of a run with a call to computeRecoveryVals()
        self.recoveryVals = None

 # driver code for running the model

 def computeRecoveryVals(opts):
        R = [None]
        for l in range(1,opts.L+1):
           r = RbaseCase(l, opts)
           R.append(r)
        return R

 def RunModel(userParameters):
     # initialize an instance of the options class with user input,
     # assumed to be stored in variable userParameters
     opts = setupOptions(userParameters)

     # precompute the values for the R base states
     opts.recoveryVals = computeRecovery(opts)

     # begin the recursive computation of the model
     Xr = X(opts.L, opts.L, opts)

     # compute the efficiency of the simulated run
     l = len(V)-1
     count = 1.0
     efficiency = 0.0
     for i in range(1,l):
        count *= (V[i] + 1)
     count *= V[l]
     saved = count * opts.t
     if(Xvals.tX0 != 0):
        efficiency = saved/Xvals.tX0

 # the model code begins here

 #Y(1, _, V) . base case
 #X(L, C, V) . Y(L, C, V)
 #Y(L, C, V) . if V[L-1] > 0:
 #                Y(L-1, L-1, V) + Z(L-1, C, V)
 #             else:
 #                Y(L-1, C, V)
 #Z(L, C, V) . (V[L]-1)*X(L, L, V) + X(L, C, V)

 # container classes for holding the values of states, mainly for ease of debug printing
 def valDebugPrint(val, p0, t0, pis, tis):
       str = "%s: p%s0=%4f t%s0=%4f \np%sis=[" % (val,val,p0,val, t0,val)
       for i in range(len(pis)):
         str += "%4f " % (pis[i])
       str += "]\nt%sis=[" % val
       for i in range(len(tis)):
         str += "%4f " % ( tis[i])
       str += "]"
       print str

 class XValues:
     def __init__(self,pX0, tX0, pXis, tXis):
       self.pX0 = pX0
       self.tX0 = tX0
       self.pXis = pXis
       self.tXis = tXis
     def debugPrint(self):
       valDebugPrint("X", self.pX0, self.tX0, self.pXis, self.tXis)

 class YValues:
     def __init__(self,pY0, tY0, pYis, tYis):
       self.pY0 = pY0
       self.tY0 = tY0
       self.pYis = pYis
       self.tYis = tYis
     def debugPrint(self):
       valDebugPrint("Y", self.pY0, self.tY0, self.pYis, self.tYis)

 class ZValues:
     def __init__(self,pZ0, tZ0, pZis, tZis):
       self.pZ0 = pZ0
       self.tZ0 = tZ0
       self.pZis = pZis
       self.tZis = tZis
     def debugPrint(self):
       valDebugPrint("Z", self.pZ0, self.tZ0, self.pZis, self.tZis)

 class RValues:
     def __init__(self,pR0, tR0, pRis, tRis):
       self.pR0 = pR0
       self.tR0 = tR0
       self.pRis = pRis
       self.tRis = tRis
     def debugPrint(self):
       valDebugPrint("R", self.pR0, self.tR0, self.pRis, self.tRis)

 def p0(T, opts):
     lam = opts.lam
     return  exp(-lam * T)

 def t0(T, opts):
     return T

 def pi(T, i, opts):
     lam = opts.lam
     lami = opts.lambdas[i]
     a =  (1.0 - exp(-lam * T))
     return ((lami*a ))/lam
```

```
def ti(T, opts):
    lam = opts.lam
    top = 1.0 - (lam * T + 1.0)* exp(-lam * T)
    bottom = lam * (1.0 - exp(-lam * T))
    return  top/bottom

def RbaseCase(k, opts):
    # compute probabilities of exiting R  with no failures
    rs = opts.checkRecoverTimes
    rK = rs[k]
    L = opts.L
    pR0 = p0(rK,opts)
    tR0 = t0(rK, opts)

    # compute probabilities of exiting R  at each failure level
    pRis = [0] # put a dummy in for index 0
    tRis = [0]
    for i in range(1,L+1):
        pRi = pi(rK, i, opts)
        tRi = ti(rK, opts)
        pRis.append(pRi)
        tRis.append(tRi)

    if opts.debug:
       total = (sum([pR0] + pRis))
       assert (fabs(total - 1.0) < opts.epsilon), ("%4f %4f" % (total, total - 1.0))
    ret = RValues(pR0,tR0,pRis,tRis)
    return ret

def YbaseCase(c, opts):
    t = opts.t
    cws = opts.checkWriteTimes
    cC = cws[c]
    L = opts.L

    # compute probability  and time of exiting Y with no failures
    pY0 = p0(t + cC, opts)
    tY0 = t0(t + cC, opts)

    # compute probabilities of exiting Y at each failure level
    pYis = [0]# put a dummy in for index 0
    tYis = [0]
    for i in range(1,L+1):
        pYi = pi(t + cC, i, opts)
        tYi = ti(t + cC, opts)
        pYis.append(pYi)
        tYis.append(tYi)

    if pY0 == 0.0 :
       return YValues(pY0,tY0,pYis,tYis)

    if opts.debug:
        # the sum of all probabilities leaving a state should be 1.0
        total = (sum([pY0] + pYis))
        assert (fabs(total - 1.0) < opts.epsilon), ("%4f %4f" % (total, total - 1.0))
        # each individual ti should be less than t0
        for i in range(1,L+1):
            assert tY0 > tYis[i]

    ret = YValues(pY0,tY0,pYis,tYis)
    return ret

#Z(k, c) . (V[k]-1)*X(k, k) + X(k, c)
def Z(k, c, opts):
    if opts.debug:
          print "ENTER Z(%s, %s)" % (k, c)
    v = opts.V[k]

    L = opts.L
    # initialize to 0's for all values
    X1r = XValues(0,0,[0 for i in range(L+1)],[0 for i in range(L+1)])
    # if there is more than one X state in this Z, first compute
    # the values for X(k, k)
    if (v - 1 > 0):
       X1r = X(k, k, opts)
       if opts.debug:
          X1r.debugPrint()
    # compute the values for X(k, c)
    X2r = X(k, c, opts)
    if opts.debug:
       X2r.debugPrint()

    # compute the probability and time of exiting Z with no failures
    pZ0 = pow(X1r.pX0,v-1) * X2r.pX0
    tZ0 = (v-1) * X1r.tX0 + X2r.tX0

    # compute the probabilities and times of exiting Z at each failure level
    pZis = [0]
    tZis = [0]

    for i in range(1,L+1): # index starts with 1
        pZi = 0.0
        tZi = 0.0
        if i > k:
           pZi = ((1.0 - pow(X1r.pX0,v-1))/(1.0 - X1r.pX0))
                    *(X1r.pXis[i]) + pow(X1r.pX0,v-1) * X2r.pXis[i]

           B1 = (1.0 - pow(X1r.pX0,v-1))/(1 - X1r.pX0) *X1r.pXis[i]*X1r.tXis[i]
           B2 = (X1r.pX0 - (v-1) * pow(X1r.pX0,v-1) + (v-2) * pow(X1r.pX0,v))
           B2 = B2/(pow((1-X1r.pX0),2)) * X1r.pXis[i] * X1r.tX0
           B = B1 + B2
           A = B + pow(X1r.pX0,v-1) * X2r.pXis[i]* ((v-1) * X1r.tX0 + X2r.tXis[i])
           tZi = A/pZi
        pZis.append(pZi)
```

```
         tZis.append(tZi)

     if pZ0 == 0.0:
        return ZValues(pZ0, tZ0, pZis, tZis)
     if opts.debug:
        # the sum of all probabilities leaving a state should be 1.0
        assert fabs(sum([pZ0] + pZis) - 1.0) < opts.epsilon
        # each individual ti should be less than t0
        for i in range(1,L+1):
            assert tZ0 > tZis[i]

     if opts.debug:
        print "LEAVE Z(%s, %s)" % (k, c)
     ret = ZValues(pZ0, tZ0, pZis, tZis)
     return ret

#Y(k, c) . if V[k-1] > 0:
#                Y(k-1, k-1) + Z(k-1, c)
#             else:
#                Y(k-1, c)

def Y(k, c, opts):
    if opts.debug:
         print "ENTER Y(%s, %s)" % (k, c)
    # if k == 1, we compute the values for the base case
    if(k == 1):
       Yr = YbaseCase(c, opts)
       if opts.debug:
           Yr.debugPrint()
           print "LEAVE Y(%s, %s)" % (k, c)
       return  Yr
    # if there are no recovery states at level k-1, there is no Z state
    if(opts.V[k-1] == 0):
       Yr = Y(k-1, c, opts)
       if opts.debug:
          Yr.debugPrint()
          print "LEAVE Y(%s, %s)" % (k, c)
       return Yr
    # otherwise, compute values for a Y and Z state
    L = opts.L

    Yr = Y(k-1, k-1, opts)
    if opts.debug:
       Yr.debugPrint()
    Zr = Z(k-1, c, opts)
    if opts.debug:
       Zr.debugPrint()

    # compute the probability and time of exiting Y with no failures
    pY0 = Yr.pY0 * Zr.pZ0
    tY0 = Yr.tY0 + Zr.tZ0
    # compute the probabilities and times of exiting Y at each failure level
    pYis = [0]
    tYis = [0]
    for i in range(1,L+1):
        pYi = Yr.pYis[i] + Yr.pY0 * Zr.pZis[i]
        top = (Yr.pYis[i]*Yr.tYis[i] + Yr.pY0*Zr.pZis[i]*(Yr.tY0 + Zr.tZis[i]))
        tYi = top/pYi
        pYis.append(pYi)
        tYis.append(tYi)

    #if pY0 == 0.0:
    if pY0  < opts.epsilon:
       return YValues(pY0, tY0, pYis, tYis)

    if opts.debug:
       # the sum of all probabilities leaving a state should be 1.0
       assert fabs(sum([pY0] + pYis) - 1.0) < opts.epsilon
       # each individual ti should be less than t0
       for i in range(1,L+1):
           assert tY0 > tYis[i]

    if opts.debug:
        print "LEAVE Y(%s, %s)" % (k, c)
    ret = YValues(pY0, tY0, pYis, tYis)
    return ret

#X(k, c) . Y(k, c)
def X(k, c, opts):
    if opts.debug:
       print "ENTER X(%s, %s)" % (k, c)

    L = opts.L
    # has Y and recovery
    Yr = Y(k, c, opts)
    if opts.debug:
       Yr.debugPrint()

    r = opts.recoveryVals
    pRis = r[k].pRis
    pR0 = r[k].pR0
    tRis = r[k].tRis
    tR0 = r[k].tR0
    pYR = 0.0
    tYR = 0.0
    pRR = 0.0
    tRR = 0.0
    for i in range(1,k+1): # range(1,5+1) makes 1,2,3,4,5
      pYR += Yr.pYis[i]
      tYR += Yr.pYis[i] * Yr.tYis[i]

    tYR = tYR/pYR
```

```
M = k-1
if k == opts.L:
  M = k
for i in range(1,M+1): # range(1,5+1) makes 1,2,3,4,5
  pRR += pRis[i]
  tRR += pRis[i] * tRis[i]

if (k != 1):
   tRR = tRR/pRR

# if too close to 0, just return 0
if (pRR - 1.0 > opts.epsilon):
   pX0 = 0.0
   tX0 = 0.0
   pXis = [0 for i in range(1,L+1)]
   tXis = [0 for i in range(1,L+1)]
   return XValues(pX0, tX0, pXis, tXis)

pRY = pR0/(1.0-pRR)
tRY = tR0 + (pRR/(1.0-pRR)) * tRR

PRis = [0]
TRis = [0]
for i in range(1,L+1):
   pRi = 0.0
   tRi = 0.0
   if i == k + 1:
      pRi = (pRis[k] + pRis[i])/(1.0 - pRR)
      tRi = (pRis[k]*tRis[k] + pRis[i]*tRis[i])/(pRis[k]+pRis[i])
                + pRR/(1.0-pRR) * tRR
   elif i > k + 1:
      pRi = pRis[i]/(1.0 - pRR)
      tRi = tRis[i] + pRR/(1.0 - pRR) * tRR
   PRis.append(pRi)
   TRis.append(tRi)

# error condition, returns  all zeros
if (pYR - 1.0 > opts.epsilon) or (pYR+Yr.pY0 - 1.0 > opts.epsilon)
     or (1.0 - pYR*pRY < opts.epsilon) :
   pX0 = 0.0
   tX0 = 0.0
   pXis = [0 for i in range(1,L+1)]
   tXis = [0 for i in range(1,L+1)]
   return XValues(pX0, tX0, pXis, tXis)

pX0 = Yr.pY0/(1.0 - pYR*pRY)
tX0 = Yr.tY0 + ((pYR*pRY)/(1.0 - pYR*pRY)) * (tYR + tRY)

pXis = [0]
tXis = [0]
for i in range(1,L+1):
   pXi = 0.0
   tXi = 0.0
   if i > k:
       pXi = (Yr.pYis[i] + pYR * PRis[i])/(1.0 - pYR*pRY)
       tXi_1 = Yr.pYis[i]*Yr.tYis[i] + pYR*PRis[i] * (tYR + TRis[i])
       tXi_1 = tXi_1/(Yr.pYis[i] + pYR*PRis[i])
       tXi_2 = (pYR*pRY)/(1.0 - pYR*pRY) * (tYR + tRY)
       tXi = tXi_1 + tXi_2
   pXis.append(pXi)
   tXis.append(tXi)

#print "pX0: %s" % pX0
if pX0 < opts.epsilon:
   return XValues(pX0, tX0, pXis, tXis)

if opts.debug:
   # the sum of all probabilities leaving a state should be 1.0
   total = fabs(sum([pX0] + pXis))
   assert ((total - 1.0) < opts.epsilon), ("%4f %4f" % (total, total - 1.0))
   # each individual ti should be less than t0
   for i in range(1,L+1):
       assert tX0 > tXis[i]

if opts.debug:
    print "LEAVE X(%s, %s)" % (k, c)
ret = XValues(pX0, tX0, pXis, tXis)
return ret
```